Datapath system for multiple electron beam lithography systems using image compression

Jeehong Yang
Serap A. Savari
H. Rusty Harris

Datapath system for multiple electron beam lithography systems using image compression

Jeehong Yang
Qualcomm Technologies Inc.
5775 Morehouse Drive
San Diego, California 92121
E-mail: jeehongy@qti.qualcomm.com

Serap A. Savari
H. Rusty Harris
Texas A&M University
Department of Electrical and Computer Engineering
3128 TAMU
College Station, Texas 77843

Abstract. The datapath throughput of electron beam lithography systems can be improved by applying lossless image compression to the layout images and using an electron beam writer that contains a decoding circuit packed in single silicon to decode the compressed image on-the-fly. In our past research, we had introduced Corner2, a lossless layout image compression algorithm that achieved significantly better performance in compression ratio, encoding/decoding speed, and decoder memory requirement than Block C4. However, it assumed a somewhat different writing strategy from those currently suggested by multiple electron beam (MEB) system designers. The Corner2 algorithm is modified so that it can support the writing strategy of an MEB system. *© 2013 Society of Photo-Optical Instrumentation Engineers (SPIE)* [DOI: 10.1117/1.JMM.12.3.033018]

Subject terms: data compression; image compression; multiple electron beam lithography; maskless direct write lithography; datapath.

Paper 12064 received Jun. 18, 2012; revised manuscript received Jun. 4, 2013; accepted for publication Jul. 25, 2013; published online Sep. 9, 2013.

1 Introduction

As the semiconductor technologies develop, circuit elements are becoming smaller and are thus harder to fabricate using conventional photolithography systems. Scientists have been seeking various ways to solve this problem using alternative lithographical methods. Among them, electron beam lithography (EBL) has been considered as a useful candidate for next-generation lithography.[1]

EBL consists of three parts as shown in Fig. 1: (1) a digital layout image that is stored at the storage system, (2) a data path through which the layout image is transmitted, and (3) an electron beam writer that writes the transmitted layout image on the photoresist using an electron beam. Instead of masking the layout image patterns from the light source, an EBL system writes the layout image digitally pixel-by-pixel using electron beams.

EBL systems have a number of advantages over conventional photolithography systems:

- EBL systems are well-known to obtain very high resolutions. By using the electron beam as its light source, EBL can offer far better resolution than conventional photolithography systems.
- EBL systems do not require physical masks. Conventional photolithography requires high-quality physical masks, which are very expensive to fabricate[2] and maintain.[1] Since many current applications can require 20 to 30 masks, the cost of physical masks for conventional photolithography is enormous. However, EBL systems do not require masks. Instead, the software-controlled e-beam writer writes the mask pattern directly to the photoresist layer.
- EBL systems are suitable for rapid prototyping. Because EBL can write any patterns as long as the layout images are provided, EBL systems not only enable fast modifications, but also allow multiple chips to be written on the same wafer.

However, EBL systems have a drawback over physical mask lithography systems—they are slow.[3] Because EBL systems write layout images pixel by pixel, the throughput of EBL is extremely low and is hence not suitable for the mass production of circuits. Moreover, considering the enormous data requirement—Dai[4] suggested 735 Tera pixels are required for a 300-mm wafer using 45-nm technology—to cover the entire wafer, this is unsustainable without massively parallel electron beams. Over the decades, scientists have been trying to solve this problem and are recently attacking it by applying multiple electron beam (MEB) writers to the system.[2,4,5,6] By writing multiple pixels at a time, it is possible to decrease the writing time and increase the throughput. Furthermore, by carefully selecting the number of electron beam writers of the EBL systems, it is possible to match the throughput of conventional photolithography systems.

Many innovations in MEB lithography are being developed for various applications, such as mask writing, prototyping, writing critical layers in high-volume manufacturing (HVM), and writing all layers in HVM. Recently, Lin[1] suggested that MEB direct-write (DW) systems writing all layers in HVM are the most economical option for next-generation lithography technology, especially once the wafer size increases to 450 mm.

However, there are still a few outstanding problems to address before MEB DW systems can replace conventional lithography systems, and one of these is the datapath issue. For MEB DW systems to maintain sufficient throughput, many bits must be simultaneously transmitted to the array of electron beam writers. This raises a question of how to provide the massive layout image data (which is typically several hundred terabits per wafer[4]) to MEB DW systems. Because of a bandwidth shortage between the storage where the layer images are deposited and the MEB DW system,

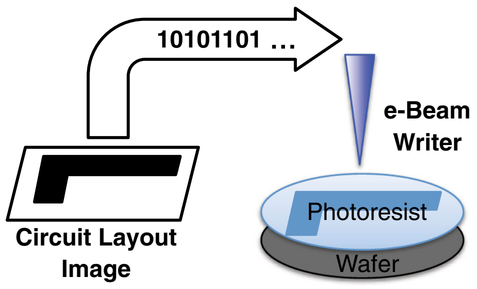


Fig. 1 Electron beam lithography systems.

obtaining competitive throughput using an MEB DW system is not possible with conventional datapath methods.

Dai and Zakhor[3,7] proposed a datapath system with a lossless image compression component. As shown in Fig. 2, they cache compressed layout images in storage disks and send this compressed data to the processor board memory. Then the MEB systems can have higher throughput if the decoder embedded within the array of electron beam writers can quickly recover the original images from the compressed files. Since the decoder cannot store an entire compressed layout image, each compressed layout image is repeatedly transferred from the storage to the decoder as it is written multiple times on a wafer. Cramer et al.[8] later improved and tailored the algorithm of Ref. 7 to operate on a particular MEB system called reflective electron beam lithography (REBL).[5]

Based on the datapath system introduced by Dai and Zakhor,[3] Yang and Savari developed a lossless compression algorithm, Corner[9] and Corner2,[10] that has better compression performance than Block C4 (BC4) (Ref. 7) for both regular and irregular circuits. The Corner2 algorithm utilizes dictionary-based compression to handle repeated circuit components and applies a transform that is specifically tailored for layout images to deal with irregular circuit components. The transformation used in Corner2 utilizes the fact that most polygons in the layout images are Manhattan, i.e., they have right angle corners. This transformation as well as the dictionary-based compression are tailored for the application so that the decoder can reconstruct the layout image with only a small cache requirement.

Krecinic et al.[11] independently introduced a vertex-based circuit layout image representation format, which can be viewed as a variant of the Corner2 transformation along with a version of run length encoding (RLE) to compress circuit layout images. However, they did not account for circuit regularity or use more advanced entropy encoding techniques to further compress the representation as was done in Ref. 10.

Yang and Savari[12] recently improved the frequent pattern discovery algorithm of Corner2 by using isolated polygons, i.e., polygons that are separated from each other, as candidate patterns and by solving an integer programming problem. The result shows that the Corner2 algorithm obtains high compression ratios and fast encoding/decoding times while requiring limited decoder cache on the decoder hardware.[12] Moreover, the entire decompression is simple so that it could be implemented as a hardware add-on to the electron beam writer.[13]

However, the Corner2 algorithm was not optimized for the MEB systems that are currently under development. Corner2 assumes the decoder can write in a row-by-row fashion with a raster order, i.e., from top to bottom and from left to right, but neither MAPPER,[6] IMS,[14] nor REBL[5] utilizes raster writing. In fact, REBL writes a 4096×248 block at a time to produce a large pixel with multilevel electron beam dosage, while MAPPER and IMS have electron beam writers positioned in a lattice formation, allowing each electron beam writer to write a designated block in a zig-zag order.

In this paper we introduce Corner2-MEB (C2-MEB), which is a modified Corner2 algorithm suitable for MEB systems that have writing strategies similar to MAPPER. We redesigned the algorithm to support both block processing and a zig-zag writing order. The experimental results show that for MAPPER-like systems, there are performance deteriorations because of block processing, but the C2-MEB algorithm still attains high performance. The rest of this paper consists of four parts. In Secs. 2 and 3 we describe the C2-MEB encoding and decoding processes. We show experimental results in Sec. 4 and conclusions are given in Sec. 5.

Throughout the paper we assume binary (i.e., black and white) circuit layout images because the control signals of MAPPER-like systems are turning the electron beams on or off. In Refs. 10 and 12 we assumed binary circuit layout images as did Krecinic et al. in Ref. 11. However, there are MEB systems like REBL[5] that use multilevel electron beam dosage, and the research works (Refs. 8, 3, 7, and 15) consider multilevel (gray) circuit layout images. Our earlier work[15] adapts Corner2 to gray circuit layout images. We also assume an ideal e-beam writing setting as on-the-fly corrections to recover the tool dose/alignment deviations are beyond the scope of this work.

2 Compression Algorithm

Figure 3 shows an overview of the C2-MEB compression algorithm. First, the layout image is separated into blocks so that each block is written by a single electron beam writer. Second, we detect the frequent patterns within the blocks. In order to do that, we first extract the frequent patterns from the graphic data system II (GDSII)[16] description of the entire layer image as in Ref. 10, and generate candidate patterns from the individual image blocks and choose the optimized frequent pattern list from all of the candidate patterns as in Ref. 12. Third, each block goes through a forward transformation process that replaces the frequent patterns from the image blocks and applies the corner transform to the

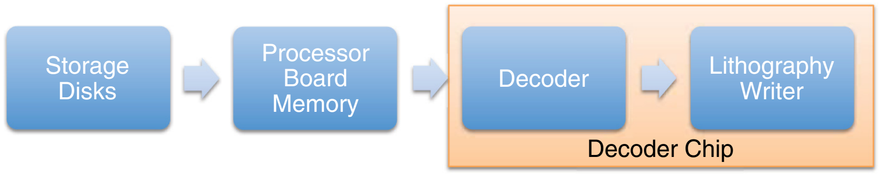
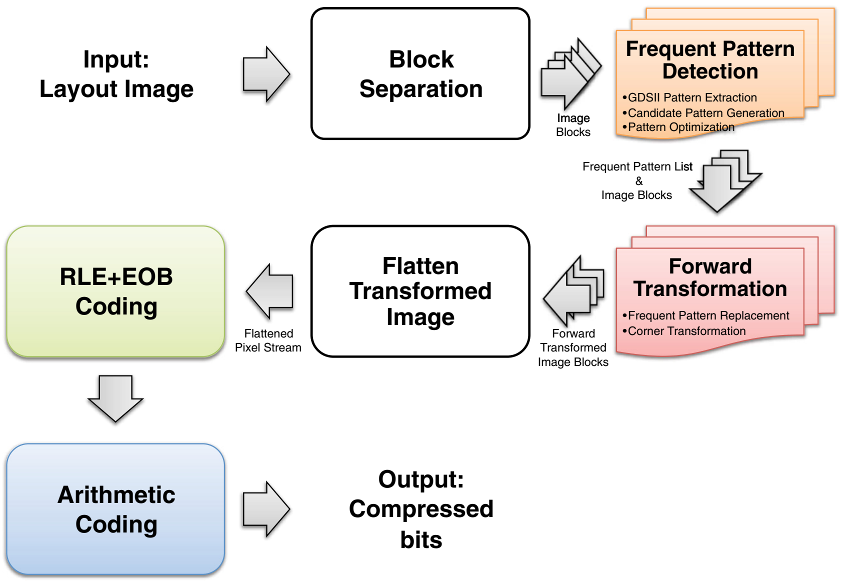


Fig. 2 Datapath for an MEB system.



Input: Layout Image
Block Separation
Image Blocks
Frequent Pattern Detection
•GDSII Pattern Extraction
•Candidate Pattern Generation
•Pattern Optimization
Frequent Pattern List & Image Blocks
Forward Transformation
•Frequent Pattern Replacement
•Corner Transformation
Forward Transformed Image Blocks
Flatten Transformed Image
Flattened Pixel Stream
RLE+EOB Coding
Arithmetic Coding
Output: Compressed bits

Fig. 3 Corner2-MEB compression algorithm overview.

unmatched parts. Note this process is similar to the frequent pattern replacement (FPR) and corner transform process in the original Corner2 algorithm,[10] but has been revised so that each image block is later reconstructed by the decoder in a zig-zag order. Fourth, the pixels of encoded image blocks are flattened to form a pixel stream so that the decoder can at each time simultaneously reconstruct a pixel for each block. Finally, the flattened pixel stream is compressed to a bit stream using entropy coding technology—RLE, end-of-block (EOB) coding, and arithmetic coding—as in the original Corner2 algorithm. In the following subsections, we will explain each step in detail.

In Fig. 3, the frequent pattern detection and forward transformation algorithms refer to processes that are applied to individual image blocks as opposed to the full layout image. Similarly, the input/output of those two components is an image block and not the full image. For example, the forward transformation process is applied independently to each image block to produce the corresponding transformed image block.

2.1 *Block Separation*

Since each electron beam writer has a limited writing region, we need to make sure the image that we pass to each electron beam writer corresponds to its writing region, which is a block. Therefore, we segment the layout image into blocks. Figure 4 shows the MAPPER lithography system writing strategy.[6] In the right part of the figure, the green pillars show the electron beam positions. While the electron beam are moving in a horizontal zig-zag order and turned on and off depending on the control signal, the stage where the wafer is fixed is moving vertically so that the electron beams write a tall rectangular stripe region on the wafer. Each electron beam writes a region 2 μm wide and 33 mm tall in a horizontal zig-zag order. In the HVM setting, once the electron beams write 33-mm tall blocks, they continuously repeat writing the blocks until they reach the end of wafer. That is, each electron beam writes a 2-μm wide stripe with the height of the wafer region that it covers with multiple copies of the image it writes.

The left part of the figure shows how the movement of the stage affects the writing strategy on the wafer. By moving the stage in a vertical zig-zag order, the MAPPER system with 13,000 electron beams writes a 26-mm ($13{,}000 \times 2$ μm) wide stripe during a single vertical scan. By writing these 26-mm wide stripes on the wafer in a vertical zig-zag order, we are actually printing upside down copies of the circuit layout image for the stripes written from the bottom to the top. This could be a problem if a circuit layout image has to

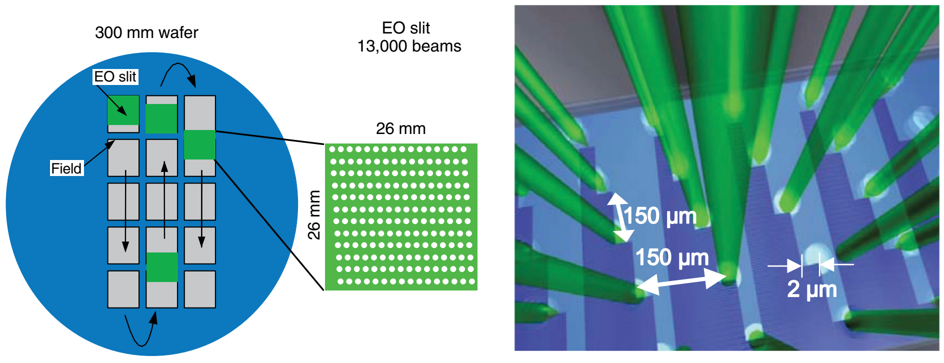


Fig. 4 The MAPPER writing strategy. Each electron beam writes a tall rectangular region in a horizontal zig-zag order.

be covered by multiple stripes. However, the 26-mm width of a stripe is wide enough to cover most circuits; for example, the previous generation 45 nm Intel Core i7-920 process had a die size of 263 mm^2.[17] In our MAPPER-like writing strategy, we assume one or multiple copies of the circuit are covered by this 26-mm wide stripe so that we do not have to consider processing different images for each 26-mm wide stripe.

Observe that since MAPPER staggers its writing regions in the vertical direction by 150 μm as shown in the right part of Fig. 4, we cannot simply partition the image into blocks and pass each block to the corresponding electron beam writer. The writing strategy of the electron beam writers to create the circuit layout image on the wafer is shown in Fig. 5. Here, the starting position of each electron beam in each writing is marked with a black square and the writing

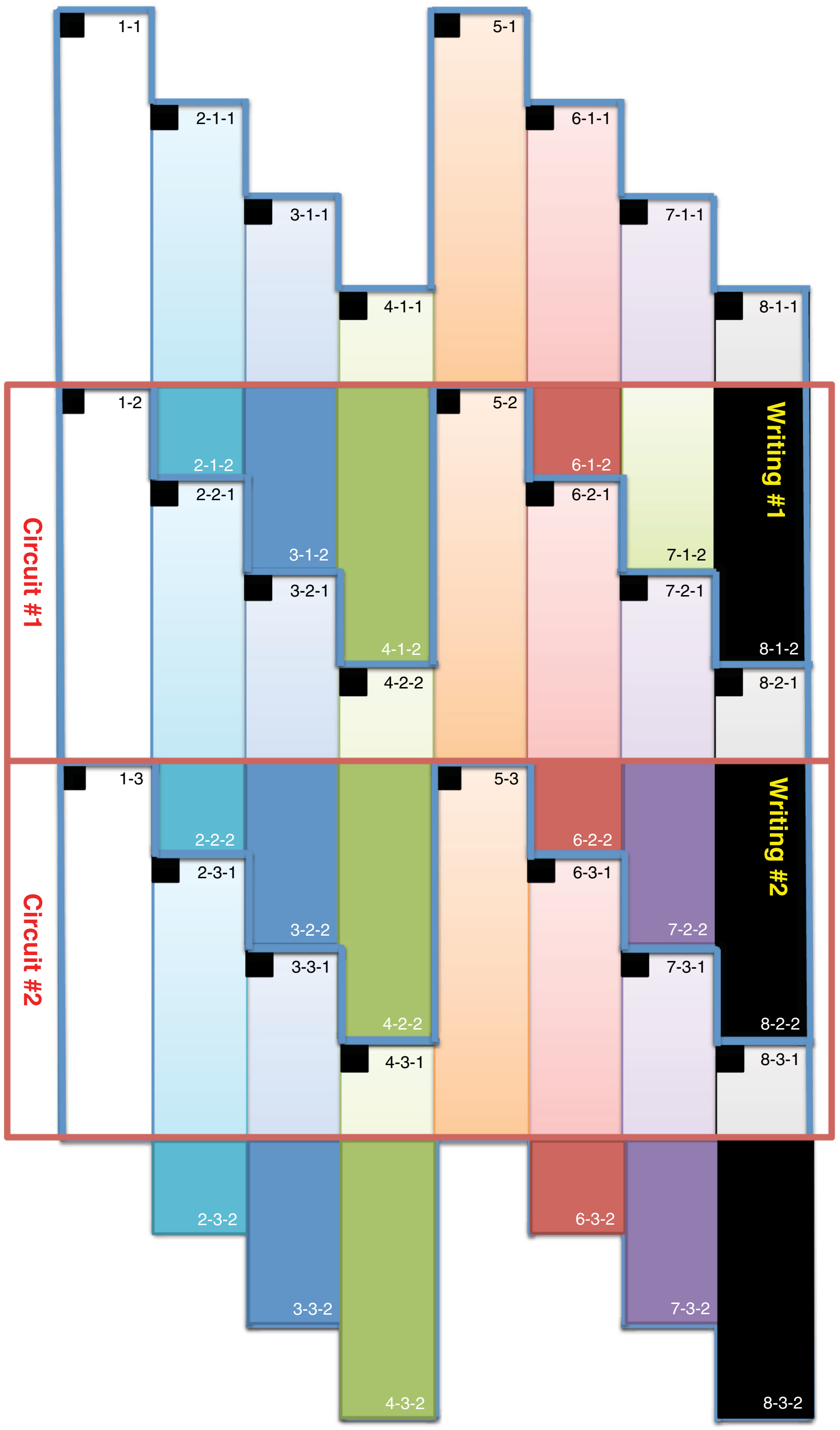


Fig. 5 The application of the MAPPER writing region to the wafer. The black squares represent the starting positions of the electron beams during each writing, and each tall rectangle within a circuit region illustrates the writing region of the corresponding electron beam writer.

region of each electron beam writer is illustrated as a tall rectangular block where the rows are successively written from the top row to the bottom row in a horizontal zig-zag order. In the HVM setting, whenever an electron beam writer finishes writing a block, it repeats writing the same block until it reaches the end of the wafer region. In the example of Fig. 5, each electron beam writer writes three copies of the blocks (represented by writings #1 to #3), and two copies of the circuit layout image (represented by circuits #1 and #2) are printed on the wafer.

In order to emphasize that some blocks can contribute to two different circuit layout image copies, we used different coloring (light and dark) for the corresponding blocks. For example, block #2 of writing #2 consists of two parts block 2-2-1 and block 2-2-2. While block 2-2-1 is part of circuit #1 along with the second part of block #2 of writing #1 (block 2-1-2), block 2-2-2 is part of circuit #2 along with the first part of block #2 of writing #3 (block 2-3-1). Finally, note that in Fig. 5 all blocks with the same color are identical. That is, blocks 2-1-1, 2-2-1, and 2-3-1 are identical and blocks 2-1-2, 2-2-2, and 2-3-2 are identical.

Therefore, the block separation process reinterprets the circuit layout image (or circuit) in the order of the writings of Fig. 5. Figure 6 shows how this process changes the circuit layout image. The top of Figure 6 shows a circuit of Fig. 5, which is a circuit layout image. To obtain the block separated image, we first partition the circuit layout image. Then, we apply circular shifts to the blocks that contribute to two different circuit layout image copies in Fig. 5—the corresponding blocks are blocks #2 to #4 and #6 to #8—so that each block has its electron beam starting point at the top-left corner of the block, as in the second part of Fig. 6. Note that the second part of Fig. 6 corresponds to a writing of Fig. 5. Throughout the discussion of the processes in Fig. 3, when we refer to image blocks we are referring to the blocks in the second part of Fig. 6. The reason that we concentrate on a writing instead of a circuit in the lower part of Fig. 6 is to emphasize the flow of decompressed data to the electron beams.

The frequent pattern discovery algorithm, which is described in Sec. 2.2.1, searches each block to discover the frequent patterns used for all blocks. Then each block is a separate input to the forward transformation process. Since it is important to understand the FPR process in order to illustrate the frequent pattern discovery process, in the next subsection we begin by explaining the forward transformation process.

2.2 *Forward Transformation*

Once the blocks are separated and the frequent patterns are discovered, each block separately goes through the forward transformation process. During the forward transformation process, we first search the image block looking for embeddings of predetermined frequent patterns. Whenever there exists an embedding of a frequent pattern, we replace the embedding with a simple representation. After all of the frequent pattern embeddings are replaced with simple representations, we apply corner transformation to the remaining image where no frequent pattern embeddings were found. Through this process, we handle the regular circuit parts with FPR and the irregular circuit parts with corner transformation.

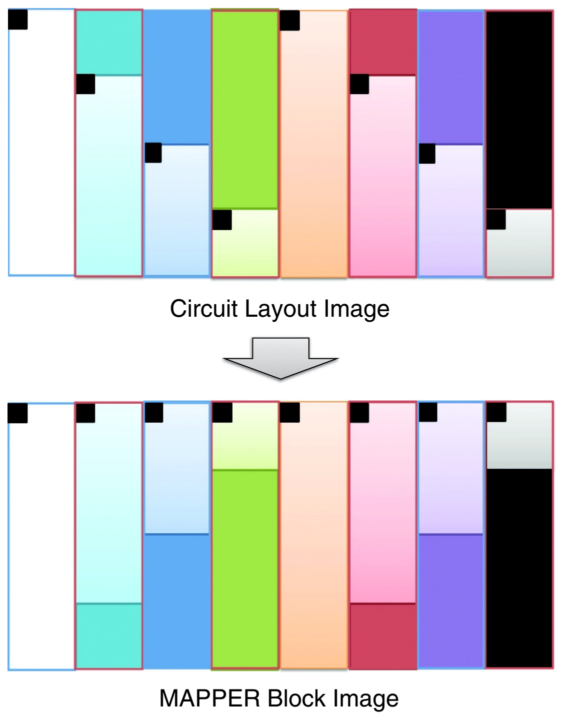


Fig. 6 The effects of block separation. The circuit layout image is partitioned and some of the partitioned blocks are rearranged.

In the following subsections, we will explain in detail the FPR and corner transformation processes.

2.2.1 *Frequent pattern replacement*

Figure 7 offers an overview of FPR. The inputs to the FPR encoder are an image block and the frequent pattern list, and the FPR encoder outputs a matched pattern image and a binary residue image. The FPR process is applied to each image block with the frequent pattern list fixed for all image blocks. The FPR encoder seeks the patterns within the block. Whenever a pattern is matched within the image block, the encoder will replace the first point of the pattern embedding with a pattern symbol and will replace the rest of filled pixels that have been matched with 0's (or empty). Note that because of the zig-zag writing, the first point of the pattern embedding could be either the top-left corner or the top-right corner of the pattern depending respectively on whether the first row of the pattern is odd or even. For the example in Fig. 7, the first 3×3 square pattern begins at the first row of the image. Since the first row is written from left to right, the top-left corner of the pattern is replaced with the pattern symbol (gray pixel). Similarly, the second 3×3 square pattern is found at the second row of the image, and since the second row is written from right to left, the top-right corner of the pattern is replaced with the pattern symbol.

Note the output matched pattern can have symbols from the set $\{0, S_1, S_2, \ldots, S_P\}$, where S_i is the symbol used to represent frequent pattern i and P is the size of the frequent pattern list. Finally, note that when the FPR encoder seeks a pattern, it more precisely searches for the pattern surrounded by empty rows and columns so that the pattern embedding is isolated, i.e., not connected to other polygons. This is necessary to avoid interference with corner transformation and

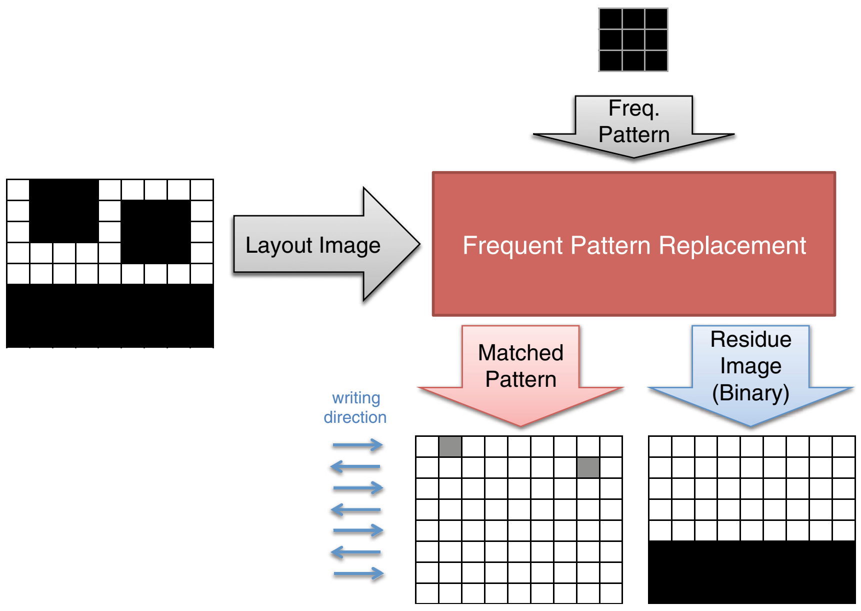


Fig. 7 Frequent pattern replacement.

to prevent partial pattern matching, which could result in performance deterioration. Otherwise, the 3×9 rectangle in the bottom of Fig. 7 can be represented by three 3×3 squares, which is not desirable.

Once the FPR process is applied, the residue image block is passed to the corner transformation process.

2.2.2 *Corner transformation*

The input to the corner transformation process is the residue image block from the FPR encoder, and the output is the corner transformed image block. As we indicated earlier, like the FPR encoder, the corner transformation process is only processing a block of the layout image at a time.

Figure 8 illustrates the corner transformation process. First, we expand the input image block by introducing two empty columns (shown as gray grids in the figure) to surround the input image block in order to handle the zig-zag writing order. Second, we apply horizontal and vertical bitransitional encoding on the expanded image block. This bitransitional encoding marks the pixels (black) where the current pixel value is different from the previous pixel read in the encoding direction, i.e., left to right for horizontal encoding and top to bottom for vertical encoding. Next, we left-shift the even-numbered rows, i.e., the rows that are written from right to left. During this left-shift, the pixels from the surrounding right column could get inside the image area.

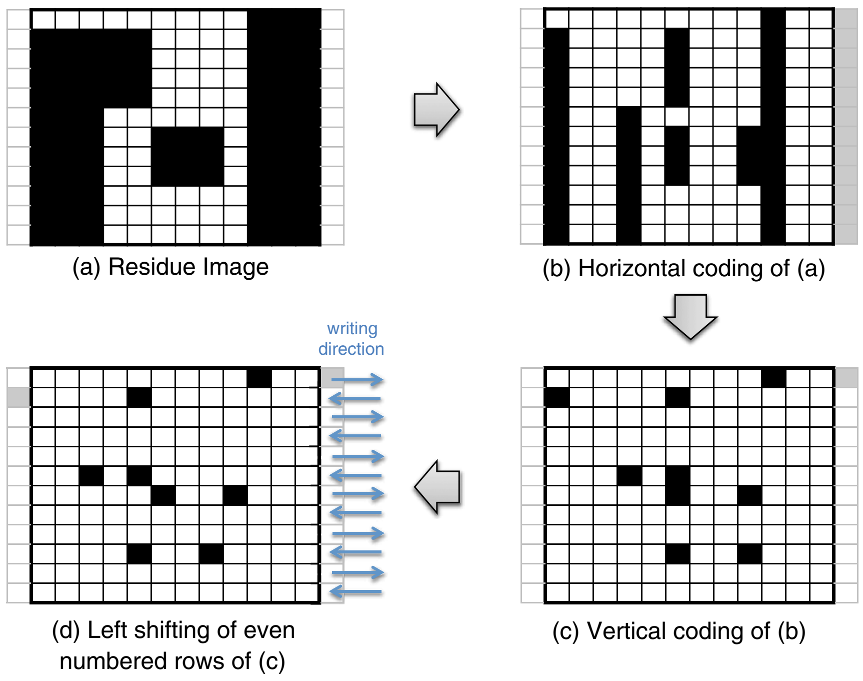

(a) Residue Image
(b) Horizontal coding of (a)
writing direction
(d) Left shifting of even numbered rows of (c)
(c) Vertical coding of (b)

Fig. 8 Corner transformation process of Corner2. The new step (d) accounts for the zig-zag writing strategy.

Finally, we discard the pixels in the expanded columns. By applying this corner transformation, we represent the residue image block using its transitional corners. We call these points transitional corners because they are similar to polygon corners, but they are extracted by transitional coding. Similar to the FPR process, the corner transformation is separately applied to each image block.

The algorithm is summarized in Algorithm 1. In the algorithm, is $x \in [1, \ldots , C]$ the column index of the image block, $y \in [1, \ldots , R]$ is the row index of the image block, and we will assume all odd rows are written from left to right and all even rows are written from right to left. Note that $R \times C$ is the dimension of the block and is predefined by the MEB DW system.

In Ref. 10, we introduced a one-step corner transformation algorithm where pixel (x, y) is processed as a function of the input pixels $(x - 1, y)$, $(x, y - 1)$, and $(x - 1, y - 1)$. We modified that algorithm so that it applies a left-shift for the even-numbered rows in lines 12 to 16 of Algorithm 1.

Observe that the transitional corners can only appear at a polygon corner, right of a polygon corner, left of a polygon corner, below the polygon corner, to the bottom-right of a polygon corner, and to the bottom-left of a polygon corner. By matching isolated patterns during FPR, we guarantee that the transitional corners produced by Algorithm 1 do not overlap with any of the pattern symbols in the matched pattern image block. We sum the pattern matched image block and the corner transformed image block to form the forward transformed image block. By choosing the frequent pattern replacement symbols (S_i) to not overlap with corner symbols

Algorithm 1 Corner transformation algorithm.

```
Input: Layer image $\mathrm{IN} \in \{0,1\}^{R \cdot C}$
Output: Corner image $\mathrm{OUT} \in \{0,1\}^{R \cdot C}$
1: Initialize $\mathrm{OUT}(x,y) = 0, \quad \forall\, x, y.$
2: for $y = 1$ to $R$ do
3:   for $x = 1$ to $C$ do
4:     if $y$ is odd then
5:       if $\mathrm{IN}(x-1, y-1) = \mathrm{IN}(x, y-1)$ and $\mathrm{IN}(x-1, y) \neq \mathrm{IN}(x, y)$ then
6:         $\mathrm{OUT}(x, y) = 1$
7:       end if
8:       if $\mathrm{IN}(x-1, y-1) \neq \mathrm{IN}(x, y-1)$ and $\mathrm{IN}(x-1, y) = \mathrm{IN}(x, y)$ then
9:         $\mathrm{OUT}(x, y) = 1$
10:      end if
11:    else
12:      if $\mathrm{IN}(x-1, y-1) = \mathrm{IN}(x, y-1)$ and $\mathrm{IN}(x-1, y) \neq \mathrm{IN}(x, y)$ then
13:        $\mathrm{OUT}(x-1, y) = 1$
14:      end if
15:      if $\mathrm{IN}(x-1, y-1) \neq \mathrm{IN}(x, y-1)$ and $\mathrm{IN}(x-1, y) = \mathrm{IN}(x, y)$ then
16:        $\mathrm{OUT}(x-1, y) = 1$
17:      end if
18:    end if
19:  end for
20: end for
```

$\{0, 1\}$, we can always separate the matched pattern image block and the corner transformed image block from the forward transformed image block.

Finally, after all blocks have been forward transformed, they are input to the flatten pixel stream process, which will be explained in Sec. 2.4, to produce a one-dimensional (encoded) pixel stream.

2.3 *Frequent Pattern Discovery*

We next discuss the generation of the frequent pattern list. This process contains three subprocesses: (1) pattern extraction from the GDSII layout description, (2) candidate pattern generation, and (3) selection of the optimized pattern list. In the following subsections we will offer a detailed description of these procedures.

2.3.1 *GDSII pattern extraction*

In Ref. 10 we extracted patterns that are frequently used in the entire layout image by seeking the frequently used substructures in the original GDSII layout descriptions since the layout image is rasterized from the GDSII representation. This procedure, which can be effective, is illustrated in Fig. 9.[10]

However, as illustrated in Fig. 10, this method has shortfalls because some substructures could result in different image patterns depending on the rasterization grid. In Fig. 10, two 5 nm × 5 nm squares are defined in the layout description (GDSII). When we rasterize the image in a 4-nm grid (Fig. 10, right), we are actually quantizing a 4×4 block from the 1-nm grid as a single pixel (Fig. 10, left), and we fill the pixel if the number of filled pixels in the original block is at least 8. The rasterized image shown in Fig. 10 (right) has two different polygons, but they came from the same layout description.

Furthermore, since we have partitioned the image into blocks, the patterns extracted from the GDSII representation may not have matches. We therefore need to search for frequent patterns within the block images.

2.3.2 *Candidate pattern generation*

In order to extract patterns that are frequently used for entire image blocks, we search the blocks. We then generate a list of candidate patterns and later determine which among these should be included in the frequent pattern list. In this section, we will discuss the candidate pattern discovery algorithm first described in Ref. 12. As we explained in Sec. 2.2.1, during the FPR process, we match isolated patterns, i.e., we first surround the pattern with empty rows on the top and bottom and empty columns to the left and right. We require that the frequent patterns satisfy the following conditions: (1) the patterns should be defined in a rectangular region, (2) the patterns should never overlap with other patterns, and (3) the patterns should be isolated.

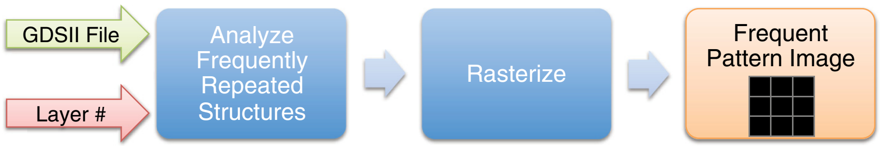


Fig. 9 Frequent pattern discovery from GDSII layout description.

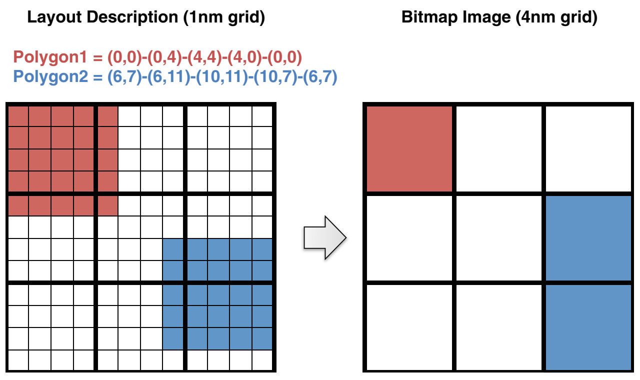

Layout Description (1nm grid)
Bitmap Image (4nm grid)
Polygon1 = (0,0)-(0,4)-(4,4)-(4,0)-(0,0)
Polygon2 = (6,7)-(6,11)-(10,11)-(10,7)-(6,7)

Fig. 10 Example of pattern mismatch due to rasterization.

First, we will explain how the candidate patterns are generated. Later, we determine which ones will be included in the frequent pattern dictionary. The candidate pattern generating algorithm is shown in Algorithm 2. In the algorithm, $x \in [1, \ldots, C]$ is the column index of the image block and $y \in [1, \ldots, R]$ is the row index of the image block. Once again note that $R \times C$ is the block dimension, which is predetermined by the MEB DW system.

The algorithm starts by picking a pixel from the image block in raster order, i.e., from top to bottom and then from left to right. If the pixel is filled (1), then we define a rectangular region $(x_0, y_0) - (x_1, y_1)$ so that all of the filled pixels that are connected to (x, y) are covered by it (line 5). We then make the rectangular region as the candidate pattern `Pattern` (line 6) and search the list of candidate patterns `PatternList` (line 7) to see whether the pattern was already in the list or not. If `Pattern` was already in the `PatternList`, we increase its frequency by 1 (line 9). Otherwise, we put `Pattern` into the `PatternList` and initialize its frequency to 1 (line 11). Finally, we make sure that the region is not searched again by marking the region in `checked` (lines 13 to 17), and this prevents the patterns from overlapping.

Algorithm 2 Candidate pattern generation.

```
Input: Layout image IN ∈ {0,1}^{R·C}
Output: List of patterns PatternList
Intermediate: List of patterns checked ∈ {0,1}^{R·C}
1: Initialize Checked(x, y) = 0,   ∀ x, y
2: for y = 1 to R do
3: for y = 1 to C do
4: if Checked(x, y) = 0 and IN(x, y) = 1 then
5: (x_0, y_0, x_1, y_1) = DefinePatternRegion(x, y)
6: Pattern = MakePattern(x_0, y_0, x_1, y_1)
7: P = PatternList.Find(Pattern)
8: if P ≠ NOTFOUND  then
9: PatternList[P].frequency+ = 1
10: else
11: PatternList.Insert(Pattern, 1)
12: end if
13: for yy = y_0 to y_1 do
14: for xx = x_0 to x_1 do
15: Checked(xx, yy) = 1
16: end for
17: end for
18: end if
19: end for
20: end for
```

Once again, note that this frequent pattern discovery algorithm is applied separately to each image block. By augmenting the discovered `PatternList` after processing each block image, we obtain the final candidate pattern list. Moreover, to incorporate the GDSII extracted patterns in Sec. 2.3.1 with the Algorithm 2 patterns, we first run the frequent pattern replacement algorithm of Sec. 2.2.1 on each image block using only the GDSII extracted patterns. We next apply Algorithm 2 to the residue image blocks, which are the block image regions that have not been matched by the frequent pattern replacement process. By combining both the GDSII extracted patterns and the patterns generated using Algorithm 2, we obtain the final candidate pattern list.

2.3.3 *Pattern optimization*

Once the candidate patterns are discovered, we analyze them in order to decide which patterns to keep and which patterns to discard. The final patterns will be used as the frequent patterns of the FPR encoder for every image block. As discussed in Ref. 12, there are two parameters that we consider to make this decision. The first parameter is *gain*, which provides information on what improvement we should expect by using the pattern for the frequent pattern replacement process. Since compression is related to the corners we remove by patterns as well as to the frequency of patterns, we define the gain of pattern p as

$$\mathrm{Gain}_p = [N(C_p) - 1] \times N(F_p),$$

where $N(C_p)$ is the number of corners of pattern p and is the frequency of pattern p.

The second parameter is *cost*, which shows how much decoder memory is required to keep the pattern in the decoder memory. For the pattern p whose dimension is $w_p \times h_p$, the decoder usually needs $w_p \times h_p$ bits of memory to store the pattern. However, we can reduce the cost when the pattern is fully filled. For this case, since we already know that all pixels are filled, all we need to store are the pattern dimensions. We allocate 16 bits for each dimension and a 1 bit flag to specify whether or not the pattern is fully filled. Therefore, the cost of pattern p with dimension $w_p \times h_p$ is defined as follows:

$$\mathrm{Cost}_p = \begin{cases} 33, & \text{if pattern } p \text{ is fully filled} \\ 33 + w_p \times h_p, & \text{otherwise} \end{cases}.$$

To reduce the complexity of the optimization problem, we initially discard the candidate patterns whose gains were less than a preset threshold *Threshold*.

To choose the frequent patterns, we want to

$$\text{maximize} \sum_p \mathrm{Gain}_p \cdot x_p \quad \text{such that} \sum_p \mathrm{Cost}_p \cdot x_p \leq P_{\mathrm{size}}, \tag{1}$$

where x_p is a binary number indicating whether pattern should be used as a frequent pattern (1) or not (0), and P_{size} is the decoder memory in bits that can be used to store the frequent pattern dictionary.

Equation (1) is an instance of a standard binary integer programming (BIP) problem.[18]

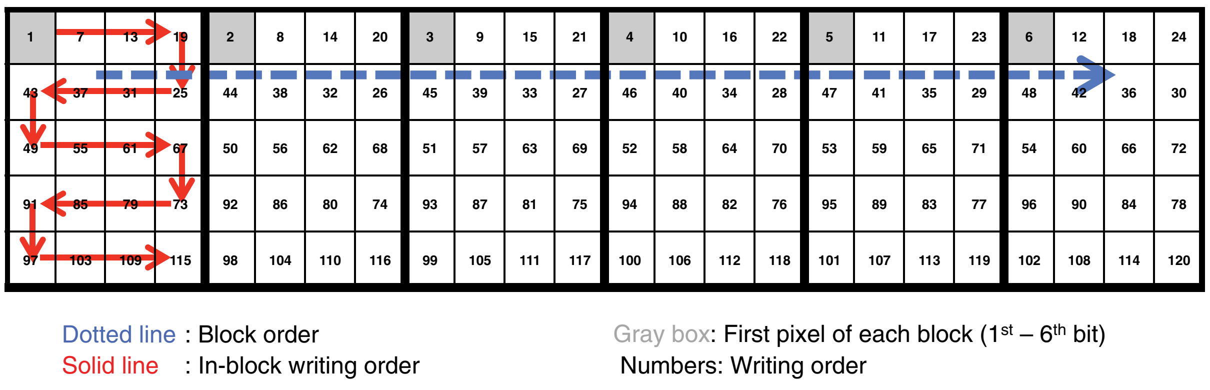


Fig. 11 Permute pixels corresponding to the writing strategy.

$$\text{Maximize } \mathbf{c}^t\mathbf{x} \quad \text{subject to } \mathbf{a}^T\mathbf{x} \le b \quad \text{and } \mathbf{x} \in \{0, 1\} \tag{2}$$

by setting $c = [\text{Gain}_1 \quad \text{Gain}_2 \quad \cdots \quad \text{Gain}_G]$, $b = P_{\text{size}}$, and $\mathbf{a} = [\text{Cost}_1 \quad \text{Cost}_2 \quad \ldots \quad \text{Cost}_G]$.

Here G is the number of candidate patterns. By applying a widely used BIP solver,[19] we are able to choose the optimal frequent patterns from the generated candidate pattern list.

Finally, note that this optimal frequent pattern set is used during the frequent pattern replacement process of every image block. If we instead use a different frequent pattern set for each image block, we can only assign a fraction of P_{size} for the frequent pattern replacement of each image block in order to sustain the same decoder memory requirement. That results in discarding some of the large patterns that are widely used in multiple image blocks and, hence, in deteriorating the compression performance.

2.4 *Flatten Pixel Stream*

Once the forward transformed image blocks are generated, we flatten the forward transformed image blocks into a one-dimensional pixel stream so that the MAPPER system receives an encoded pixel for each block at a time to match the current placement of all of the electron beams. We obtain this stream by gathering the transformed image blocks and reading one pixel from each block in a horizontal zig-zag order.

An example of this permutation is shown in Fig. 11 where each block has adimension of 5×4 and each block is written in zig-zag order. For example, we first read the first (or top-left) pixels of each block in the order of the blocks. Then, we read the second pixels in zig-zag order of each block and continue that until the last pixels in zig-zag order of each block are read. The order in which the pixels are read in this example is marked in the image.

Finally, note that this permutation operation inputs the forward transformed image blocks and outputs a long one-dimensional stream to which the following entropy coding schemes will be applied.

2.5 *Entropy Encoding*

We use the final entropy coding scheme of Refs. 10 and 12 to compress the final flattened stream of pixels. Since the forward transformed images are very sparse, the flattened stream contains long run of zeroes making RLE[20] and EOB[10] coding efficient to compress it. Moreover, we also find long runs of EOBs as the forward transformed images are very sparse. We use an N-ary representation to encode runs of EOBs and an M-ary representation to encode runs of zeroes as in Refs. 10 and 12. Observe that the output of RLE+EOB coding can have up to $P + 1 + M + N$ symbols, where P is the number of frequent patterns after the pattern optimization process in Sec. 2.3.3 and one symbol is needed for representing the transitional corners shown in Sec. 2.2.2.

This $P + 1 + M + N$ symbol string is compressed using arithmetic coding[21,22] for further compression. We followed the implementation of arithmetic coding provided by Witten et al.[22] For the implementation, the decoder requires four bytes per alphabet symbol, and since we used $P + 1 + M + N$ symbols, $32(P + 1 + M + N)$ bits were required for arithmetic decoding.

3 Decompression Algorithm

The C2-MEB decoder decompresses the compressed bit stream to write the circuit layout image using the electron beam arrays. Because of the memory constraints of the decoder, the same compressed bit stream is repeatedly retransmitted to the decoder during each writing of an electron beam.

The C2-MEB decompression process is illustrated in Fig. 12 and consists of four major blocks: arithmetic decoding, run length and EOB decoding, block reconstruction, and an inverse transformation process block for each electron beam that is directly connected to it. The entire decoder is fabricated on the same silicon as the electron beam writer (controller) array.

The first two steps, arithmetic decoding and run length and EOB decoding, reverse the corresponding encoding procedures and output the $(P + 2)$-ary flattened pixel stream of Sec. 2.4. In the block reconstruction process, the input pixel stream is separated into multiple pixel streams, which each correspond to a forward transformed image block. Each such block is converted back into a reconstructed image block, which is passed to an electron beam writer that writes the decoded block onto the wafer as the stage moves. The different image blocks are processed and written in parallel on the wafer by independent inverse transformation processes and the corresponding electron beam writers.

The decoder requires $\lceil \log_2(3) \times \text{width} \rceil$ bits of cache for each inverse transformation process block to keep intermediate information for the row-by-row decoding of each block,

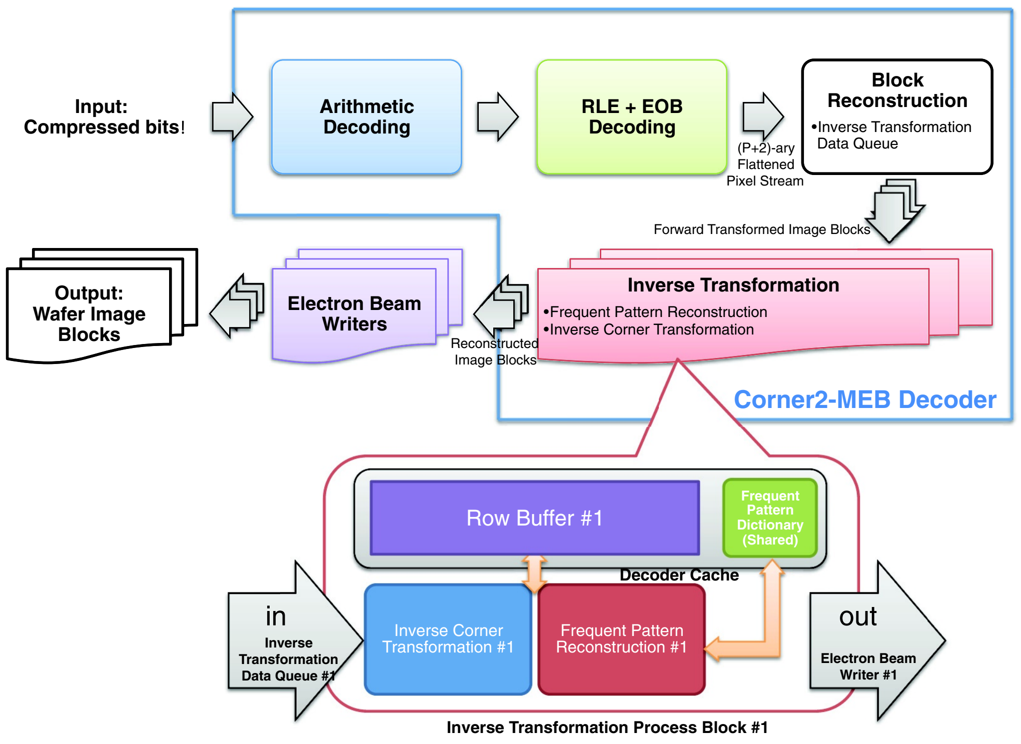

Input: Compressed bits!
Arithmetic Decoding
RLE + EOB Decoding
(P+2)-ary Flattened Pixel Stream
Block Reconstruction
•Inverse Transformation Data Queue
Forward Transformed Image Blocks
Inverse Transformation
•Frequent Pattern Reconstruction
•Inverse Corner Transformation
Reconstructed Image Blocks
Electron Beam Writers
Output: Wafer Image Blocks
Corner2-MEB Decoder
Row Buffer #1
Frequent Pattern Dictionary (Shared)
Decoder Cache
in
Inverse Transformation Data Queue #1
Inverse Corner Transformation #1
Frequent Pattern Reconstruction #1
out
Electron Beam Writer #1
Inverse Transformation Process Block #1

Fig. 12 Corner2-MEB decompression algorithm overview.

P_{size} bits of memory for the frequent pattern dictionary, and $32(P+1+M+N)$ bits for arithmetic decoding, where width is the width of the image block, P is the number of total frequent patterns, and M/N are RLE/EOB parameters.

3.1 *Block Reconstruction*

Since arithmetic and RLE+EOB decoding are well understood in data compression, we will next discuss the reconstruction of the forward transformed image blocks of Sec. 2.2 from the flattened pixel stream of Sec. 2.4. As illustrated in Fig. 13, we can achieve this by wiring each symbol of the flattened pixel stream to the corresponding inverse transformation data queue.

In the example shown in Fig. 13, we are assuming there are six electron beam writers and six inverse transformation processes working simultaneously for the system. The information stored in the inverse transformation data queue is passed onto the inverse transformation process where each row of a forward transformed image block is reconstructed and decoded as a row of the reconstructed image block. Since each inverse transformation process block is connected to the corresponding electron beam writer, each reconstructed row is passed to the electron beam writer to be written on the wafer.

3.2 *Inverse Transformation*

Inverse transformation is applied to each forward transformed image block to reconstruct the original image block. The reconstructed image block is forwarded to the associated electron beam writer. The inverse transformation performs frequent pattern reconstruction and inverse corner transformation. Recall that the frequent patterns are encoded using symbols $S_1, \ldots, S_P$ while the transitional corners are marked by the symbol 1.

The bottom half of Fig. 12 shows the detailed hardware architecture of an inverse transformation block. As illustrated in the figure, both the inverse corner transformation and the frequent pattern reconstruction processes require a row buffer to keep track of the previous row's status. The frequent pattern reconstruction process also requires a frequent pattern dictionary. Note that this frequent pattern dictionary is shared among all of the inverse transformation blocks.

Observe that since the electron beams write a pixel of each image block at the same time, synchronization is needed among the inverse transformation processes. Here we have the inverse transformation processes produce a row of each reconstructed image block in parallel.

We next explain the operation of the inverse corner transformation and the frequent pattern reconstruction processes for an image block.

3.2.1 *Inverse corner transformation*

The inverse corner transformation uses pixels from the previous row and column to decode the current pixel. We designed the decoder to decode the input corner transformed image block in a row-by-row manner instead of in its entirety in order for this process to be compatible with the restricted memory available to the hardware decoder. The inverse corner transformation process is as follows: First, the decoder reads the input corner transformed image block in a zig-zag order. The zig-zag order is the same as raster order except that it reads the even rows from right to left. Second, the decoder processes the current pixel by checking the status of the row buffer (BUFF). The row buffer is used to store the status of the previous (decoded) row. It uses two symbols, 0 and 1, to represent its status, and hence, the buffer requires

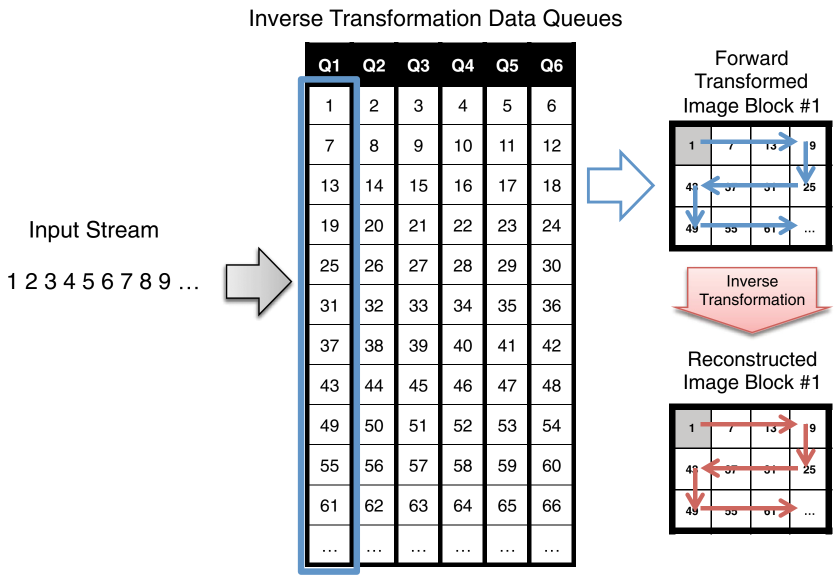

Inverse Transformation Data Queues
Q1 Q2 Q3 Q4 Q5 Q6
Input Stream
1 2 3 4 5 6 7 8 9 ...
Forward Transformed Image Block #1
Inverse Transformation
Reconstructed Image Block #1

Fig. 13 Reconstructing the forward transformed image blocks from the flattened stream. The decoder places each incoming stream symbol into the appropriate inverse transformation data queue. All of the symbols in a queue are rearranged into a block that undergoes the inverse transformation process to produce a decoded image block.

width bits of memory. "0" means no transition while "1" means transition and indicates the starting/ending point of a vertical line. Third, whenever the read symbol is a transitional corner point ("1"), the decoder starts reconstructing a horizontal line by setting the horizontal line fill flag (`Fill`) until it reads another transitional corner point from the input row and resets the horizontal line fill flag. For the even rows, this line is written from right to left, while it is written from left to right for the odd rows. Fourth, for every new horizontal line created by the transitional corners, the row buffer (`BUFF`) is updated so that the decoder can take the status of the current row into account while reconstructing the next row.

Note that because we are dealing with transitional corners, a horizontal line starts from a transitional corner point and ends one pixel before the pairing transitional corner point. In Fig. 8, step (d) is applied to sustain the same decoding rule for the even rows. If the encoder did not left-shift the even rows, then in order to reconstruct the horizontal lines of those rows, the decoder has to start a horizontal line one pixel after a transitional corner point and end it at the pairing transitional corner point. However, there could be a problem reconstructing the horizontal line of the even rows when the first pixel of the row (i.e., the rightmost pixel) has to be filled. Since there is no pixel to the right of the rightmost pixel, the reverse line from right to left cannot be reconstructed using this decoding rule. By inserting an empty right column and allowing the transitional corners to appear there and applying left-shifts to the even rows as in step (d) of Fig. 8, we can always reconstruct horizontal lines by starting from a transitional corner point and ending it one pixel before the corresponding transitional corner point following the row direction.

Because the inverse corner transformation rule is independent of the row direction, we have to make sure that the column index matches the row direction. Algorithm 3 describes the inverse corner transformation process. We assume the inverse corner transformation decoder can randomly access the entire row buffer and the input corner transformed image block is read in a zig-zag order. Note that the $\bigoplus$ operation is a binary `XOR` operation and is only applied to binary summands.

In Algorithm 3, the horizontal fill flag (`Fill`) is initialized for every row (line 4). Then we check the row number and update the column index (lines 6 to 10). If the row is odd numbered, then the column index starts from 1 to C (lines 6 and 7), while the order is reversed (from C to 1) if the row number is even. We update the column index if the row number is even (lines 8 to 10). Lines 11 to 13 process the buffer. If the buffer is filled, i.e., if there is a vertical fill, then the corresponding pixel is filled. If the input pixel (read in zig-zag order) is a transitional corner ("1"), the decoder changes the status of the horizontal fill flag (lines 14 to 16). Finally,

Algorithm 3 Inverse corner transformation.

```
Input: Corner transformed image block $\mathrm{IN} \in \{0,1\}^{R \cdot C}$
Output: Reconstructed image block $\mathrm{OUT} \in \{0,1\}^{R \cdot C}$
Intermediate: Row Buffer $\mathrm{BUFF} \in \{0,1\}^{R}$
1: Initialize $\mathrm{BUFF}(x) = 0, \quad \forall\, x.$
2: Initialize $\mathrm{OUT}(x) = 0, \quad \forall\, x, y.$
3: for $y = 1$ to $R$ do
4:     $\mathrm{Fill} = 0$
5:     for $x = 1$ to $C$ do
6:         if $y$ is odd then
7:             $x' = x$
8:         else
9:             $x' = C + 1 - x$
10:        end if
11:        if $\mathrm{BUFF}(x') = 1$ then
12:            $\mathrm{OUT}(x', y) = 1$
13:        end if
14:        if $\mathrm{IN}(x', y) = 1$ then
15:            $\mathrm{Fill} = \mathrm{Fill} \bigoplus 1$
16:        end if
17:        $\mathrm{OUT}(x', y) = \mathrm{OUT}(x', y) \bigoplus \mathrm{Fill}.$
18:        $\mathrm{BUFF}(x') = \mathrm{BUFF}(x') \bigoplus \mathrm{Fill}.$
19:    end for
20: end for
```

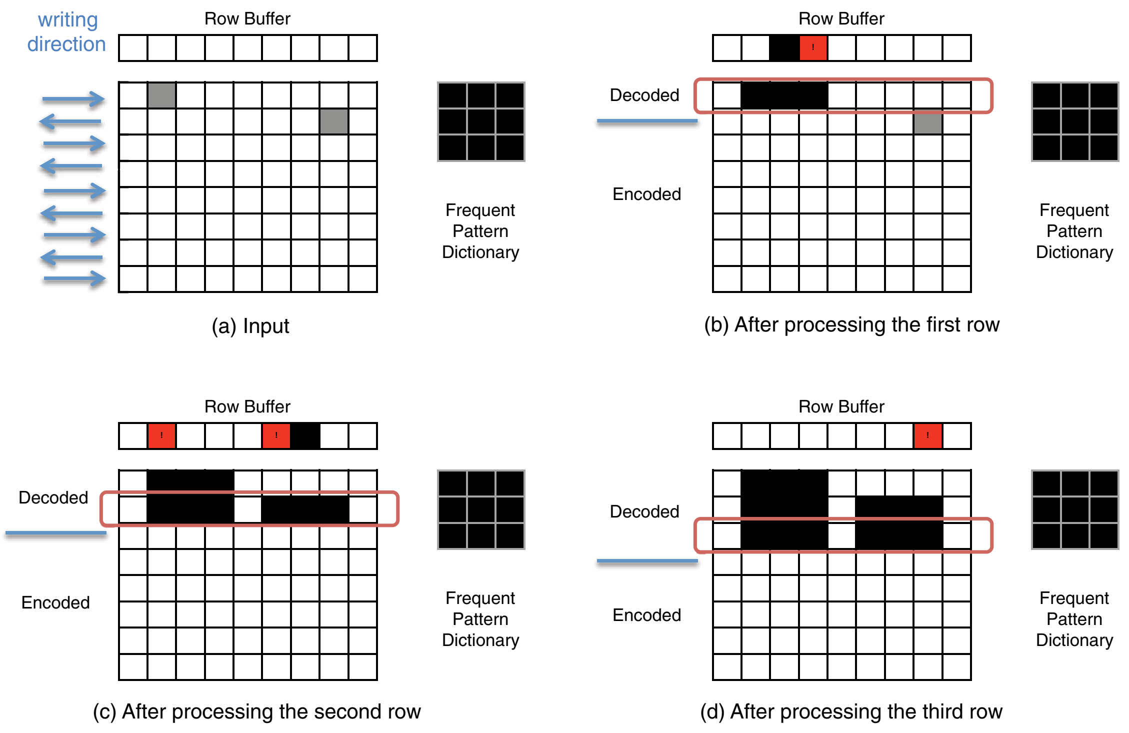

writing direction
Row Buffer
Frequent Pattern Dictionary
(a) Input
Decoded
Encoded
(b) After processing the first row
(c) After processing the second row
(d) After processing the third row

Fig. 14 Frequent pattern reconstruction example.

depending on the horizontal fill flag (Fill), the output pixel (line 17) and the buffer (line 18) are updated if necessary. If the input pixel is "0," the decoder makes no horizontal/vertical changes to the image, but fills the output pixels and updates the buffer according to the fill status.

3.2.2 *Pattern reconstruction*

If the decoder finds symbols $\{S_1, \ldots, S_P\}$ within the forward transformed image block, it starts the pattern reconstruction process. First, it reconstructs the first row of the corresponding pattern according to its writing order. If the decoder is processing an odd row, it will reconstruct the pattern row from left to right and otherwise it will reconstruct the pattern row from right to left. Second, it updates the corresponding buffer with the following string:

[PatternRow].

Here \$ is a special symbol used to indicate the start of frequent pattern p replacement. If pattern of dimension $w_p \times h_p$ is to be reconstructed, then Pattern# is a $\lceil \log_2(P) \rceil$-bit binary representation of p and Row# is a $\lceil \log_2(h_p - 1) \rceil$-bit binary representation of one less than the remaining number of pattern p rows the decoder has to reconstruct. Because Pattern# determines which frequent pattern the decoder needs to reconstruct (and its dimension), and Row# determines which row of the corresponding pattern it has to reconstruct, the decoder can reconstruct the pattern. We could alternatively use base-3 logarithms, but that would complicate the decoding hardware.

Note that this [PatternRow]. stream is updated depending on the writing direction of the next row. An example of the frequent pattern reconstruction process is demonstrated in Fig. 14. When the first row is decoded, we update the

Table 1 Compression ratio (x)—Memory array (block size: 888 × 17,816).

Layer	C2-MEB	C2-BIP	C2-ORG	BC4
1	11,074	22,201	18,658	147.78
2	457	1289	1216	79.80
3	11,074	22,201	18,658	147.78
4	83	151	86	54.12
5	640	920	920	133.22
6	640	920	920	133.22
7	104	84	84	31.58
8	41	90	67	24.40
9	89	171	170	38.17
10	96	114	57	25.31
11	204	442	442	75.70
12	2837	11,059	3169	121.69
13	5971	28,606	9616	141.02
Average	164	261	192	57.36

Table 2 Encoding/decoding time (s)—Memory array.

	Encoding time (s)				Decoding time (s)			
Layer	C2-MEB	C2-BIP	C2-ORG	BC4	C2-MEB	C2-BIP	C2-ORG	BC4
1	23.12	18.89	4.29	1875	2.81	2.40	2.53	55.44
2	30.84	26.48	26.36	1891	3.57	3.16	3.34	54.99
3	23.74	18.36	4.29	1788	2.81	2.42	2.53	52.90
4	43.47	49.02	48.93	1815	3.55	2.96	3.20	53.70
5	15.40	25.26	26.85	1831	3.22	2.95	3.05	53.55
6	15.55	25.28	26.86	1796	3.24	2.90	3.05	54.48
7	25.88	23.07	25.27	1846	3.52	3.21	3.23	54.63
8	92.07	97.32	109.97	1885	3.87	3.21	3.51	54.79
9	32.57	25.37	30.18	1817	3.44	2.79	3.05	54.83
10	68.83	84.16	78.49	1917	3.58	2.98	3.44	54.07
11	64.58	70.68	80.22	1795	3.40	2.78	3.04	56.43
12	15.64	6.23	4.16	1775	3.46	2.95	2.49	51.50
13	17.20	6.52	4.16	1762	3.58	2.94	2.48	51.88
Total	468.87	485.45	470.05	23,793	44.04	37.67	38.93	703.18

row buffer in the reverse direction starting at the fourth column because the next row is written from right to left and the pattern involves the second, third, and fourth columns. The [PatternRow]. stream is represented as [1] in the reverse direction because there is only one frequent pattern defined in the dictionary making $\lceil \log_2(P) \rceil = 0$. (In Fig. 14, the \$'s are represented by red pixels and the 1's are represented by black pixels.) In the next row, the decoder has to process two more rows of the pattern. When the second row is decoded, since the writing order of the next row is from left to right and since we need to write one more row to the first pattern, the buffer of the first pattern is updated to [0] in the forward direction. Similarly, the buffer of the second pattern is updated to [1]. Finally, when the third row is decoded, we reinitialize the buffer of first pattern to [0 0] to indicate that the frequent pattern reconstruction is complete.

In order to combine this frequent pattern reconstruction and the inverse corner transformation, the inverse transformation decoder requires $\lceil \log_2(3) \times C \rceil$ bits for the row buffer for each inverse transformation block and P_{size} bits to store the pattern dictionary. Given that the forward transformed image is a $(P+2)$-ary array with dimension $R \times C$, this buffer requirement is comparatively small.

4 Experimental Results

We tested the algorithm on two custom circuits—a memory circuit and a binary frequency shift keying (BFSK) transmitter circuit used for testing Corner2.[10,12] The memory core was targeting 500-nm lithography technology containing 13 layers of repeated memory cell structure. The custom-designed BFSK transmitter was targeting 250-nm lithography technology containing 19 layers of mostly irregular features. We scaled these images to the point where their minimum features were small rectangles (with less than a dozen pixels) to simulate the MAPPER environment. Most parts of C2-MEB were written in C/C++ with OpenMP support for processing the forward and inverse transformation processes in parallel. For the decoding process, the inverse transformation processes are synchronized so that the same writing row of each was processed in parallel. Because C2-MEB blocks can be processed independently, we applied parallel processing for faster encoding/decoding speed. The binary integer programming part was written in MATLAB using the MATLAB function bintprog.[19] All of the experiments ran on a laptop computer having a 2.53 GHz Intel Core 2 Duo CPU and 4 GB RAM.

We applied C2-MEB and compared the results with Corner2-BIP (C2-BIP),[12] Corner2 (C2-ORG),[10] and BC4.[7] While C2-BIP, Corner2, and BC4 were applied to the entire layout image, C2-MEB was applied in a block fashion. We set the MAPPER block size to be $888 \times$ height because the MAPPER writing region is a width-2 μm stripe on a 2.25-nm grid.[6] Here, height is the height of the circuit layer image, which was 17,816 for the memory circuit and 31,624 for the BFSK circuit.

In the following subsections we show the compression results for both the memory and BFSK circuits on the MAPPER system. In the following subsections, we defined the compression ratios to be

Table 3 Compression ratio (x)—BFSK circuit (block size: $888 \times 31,624$).

Layer	C2-MEB	C2-BIP	C2-ORG	BC4
1	12,126	20,874	9226	151
2	6275	7432	6760	150
3	1016	1596	1186	137
4	2942	3745	3270	147
5	925	1125	1033	59
6	476	609	580	128
7	297	399	390	111
8	122	173	167	81
9	361	474	452	122
10	179,212	460,192	195,365	153
11	139	206	200	86
12	848	1082	1006	138
13	142	209	203	89
14	943	1170	1093	139
15	167	236	230	92
16	1070	1368	1296	141
17	5635	8019	6656	150
18	19,687	26,467	19,579	152
19	12,005	20,773	9173	151
Average	385	537	515	113

$$\frac{\texttt{Input file size}}{\texttt{Compressed file size}}.$$

Note that the total compression ratio is not the average of the preceding compression ratios, but the net average, which is defined as

$$\frac{\texttt{Total input file size}}{\texttt{Total compressed file size}}.$$

To compare the result with C2-BIP, we used the same M, N, P_{size} and settings as in Ref. 12.

4.1 *Memory Circuit*

Tables 1 and 2 show the experimental results for the memory circuit when the MAPPER block separation was applied. Because of the block partitioning, C2-MEB had 37.3 and 14.9% smaller compression ratio than C2-BIP and C2-ORG, respectively, which operated on the full layout image. However, C2-MEB was still 2.9 times better than BC4 as shown in Table 1. The block partitioning resulted in fewer pattern matches and introduced more corners in the block boundaries. Moreover, the flattening process also deteriorated the RLE+EOB encoding by segmenting long run of zeroes into smaller pieces. However, because Corner2-variations have better layout image modeling, C2-MEB still resulted in better compression performance than BC4.

On average, C2-MEB encoding was 3% (or 0.3%) faster than C2-BIP (or C2-ORG) and 50.8 times faster than BC4 as shown in Table 2. While C2-MEB generated more candidate patterns due to block truncation, which slowed down the encoding process, it became faster than C2-BIP and C2-ORG because of parallel processing. However, C2-MEB encoding could be further accelerated by using more parallelization. Since the Intel Core 2 Duo CPU only supports running four threads simultaneously, we were not able to take advantage of processing all blocks in parallel. Distributed computing over the grid or an application of GP-GPU-based parallel processing might enable this.

On average, C2-MEB decoding was 17% (or 13%) slower than C2-BIP (or C2-ORG), while it was 16.0 times faster than BC4 as shown in Table 2. Our decoding time results do not include the time to reverse the flattening process because this procedure would be unnecessary in a hardware implementation. The process was only used to verify that the reconstructed image matches with the input image. Furthermore, similar to the encoding time, we expect the decoding time to be further reduced when full parallelization is applied. Considering this as a compress-once-and-decode-multiple-times application, C2-MEB is more realistic than C2-BIP and C2-ORG in that it takes advantage of multiple electron beams, and it offers better compression than BC4.

4.2 *BFSK Circuit*

Tables 3 and 4 show the experimental results for the BFSK circuit when the MAPPER block separation was applied. Like the memory circuit, C2-MEB had a 28.4% smaller compression ratio than C2-BIP and a 25.2% smaller compression ratio than C2-ORG. This is mainly due to the new corners introduced and the shortened runs of zeroes due to block separation. However, C2-MEB still offered a 3.4 times better compression ratio than BC4. On average, C2-MEB was 58% slower than C2-BIP and 79% slower than C2-ORG, while it was 109.0 times faster than BC4. Compared to the memory circuit result, the improvements in encoding speeds were because more candidate patterns had to be generated even though not so many of them were used. Similarly, C2-MEB was 18% slower than C2-BIP and 14% slower than C2-ORG, while it was 26.1 times faster than BC4. However, this was because we could run only four inverse transformation processes in parallel, which could be further improved with the help of massively parallel computing.

5 Conclusion

In this paper, we have modified Corner2 for MAPPER-like systems. Although its compression performance was inferior to that of C2-BIP[12] due to block processing, it is better suited to MEB DW systems because it incorporated the actual writing strategy and because the inverse transformation process can be parallelized for higher throughput. Moreover, the

Table 4 Encoding/decoding time (s)—BFSK circuit.

	Encoding time (s)				Decoding time (s)			
Layer	C2-MEB	C2-BIP	C2-ORG	BC4	C2-MEB	C2-BIP	C2-ORG	BC4
1	6.29	1.74	2.41	451	1.09	0.80	0.84	29.50
2	38.87	16.33	17.88	3791	9.35	6.77	8.22	242.07
3	45.60	15.47	18.31	3800	9.20	6.63	8.24	237.42
4	35.70	15.11	17.21	3760	9.23	6.57	8.22	237.35
5	41.80	26.62	28.07	12,468	9.76	7.29	8.35	280.67
6	40.45	26.28	26.43	4485	9.66	7.26	8.33	274.15
7	63.57	54.74	37.09	4476	11.37	12.08	9.74	282.03
8	35.40	15.36	17.91	4588	10.91	9.71	9.68	289.53
9	59.26	58.11	35.56	4393	11.34	10.91	9.67	275.56
10	38.57	15.93	20.73	4866	10.47	7.61	9.41	293.04
11	37.33	17.56	17.90	4414	10.80	10.10	9.64	281.57
12	53.19	42.59	33.20	4428	11.05	10.89	9.57	278.34
13	34.73	15.00	17.88	4651	10.79	10.32	9.62	290.37
14	53.20	50.16	34.38	4413	11.11	10.46	9.57	276.35
15	37.22	17.29	17.80	4364	10.81	9.33	9.60	277.88
16	55.64	53.06	38.54	4381	11.26	9.34	9.59	275.98
17	32.80	14.30	15.74	3756	9.08	6.49	8.21	236.50
18	27.90	13.21	14.26	3049	7.24	5.23	6.47	190.71
19	5.90	1.73	2.42	454	1.05	0.79	0.84	27.68
Total	743.41	470.59	413.75	80,987	175.58	148.58	153.81	4576.71

performance of C2-MEB was still far better than that of BC4. Hence handling regular circuit parts using dictionary-based compression and irregular circuit parts using corner-based (or vertex-based) representation as in Corner2 is far more effective than handling regular circuit parts using Lempel-Ziv-based compression and irregular circuit parts using prediction-based compression as is done in BC4.

The decoder needs to be implemented in hardware, and we showed that the Corner2 inverse transformation decoder along with the RLE+EOB decoder required only 2% of a Xilinx Spartan 3E field-programmable gate array board even when using 5 kbytes of decoder cache.[13] The C2-MEB decoder consists of an arithmetic decoder, RLE+EOB decoder, and 13,000 inverse transformation blocks with each inverse transformation block smaller than those we introduced in Ref. 13. Each inverse transformation block consists of simple operations such as comparisons, branches, XORs, and binary arithmetic. Since a GPU core can handle these operations and more advanced operations, we argue that the dedicated inverse transformation core will be much smaller than a GPU core. The row buffer requirement for each inverse transformation block is $\lceil \log_2(3) \cdot B_{\text{width}} \rceil$ bits, where is the block width. Therefore, the entire C2-MEB system supporting 13,000 electron beams will require <3 MB of cache. Since current GPUs have >3000 cores[23] and previous generation Intel CPUs have 8 MB of cache,[17] the entire C2-MEB decoder can be built using current silicon technologies.

We have two concerns regarding this algorithm. First, we do not have an efficient hardware implementation for the arithmetic decoder, making the final arithmetic encoding less suitable for the purpose. We can tackle this issue by using other standard compression techniques and trading off part of the compression ratio for an efficient hardware decoder; since C2-MEB has about three times better compression performance than BC4, there is some room for this trade-off.

Second, adjustments to the algorithm will be necessary to account for e-beam proximity correction (EPC).[24] We expect that EPC layout images will not have linear boundaries. Nevertheless, some combination of frequent pattern replacement with a loose matching and a way to handle errors will be useful. This is the next major direction of this line of research.

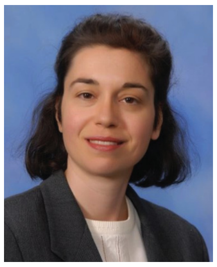
Acknowledgments

The work of the second author was supported in part by the National Science Foundation grants CCF-1017303 and ECCS-1201994.

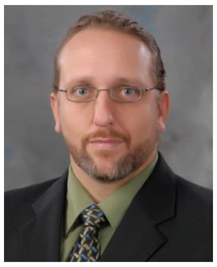
References

1. B. Lin, "Future of multiple-e-beam direct-write systems," *Proc. SPIE* **8323**, 832302 (2012).
2. N. Chokshi, Y. Shroff, and W. G. Oldham, "Maskless extreme ultraviolet lithography," *J. Vac. Sci. Technol. B* **17**(6), 3047–3051 (1999).
3. V. Dai and A. Zakhor, "Lossless compression of VLSI layout image data," *IEEE Trans. Image Process.* **15**(9), 2522–2530 (2006).
4. V. Dai, "Data compression for MEBL systems: architecture, algorithms, and implementation," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley (2008).
5. P. Petric et al., "REBL nanowriter: reflective electron beam lithography," *Proc. SPIE* **7271**, 727107 (2009).
6. E. Slot et al., "MAPPER: high throughput maskless lithography," *Proc. SPIE* **6921**, 69211P (2008).
7. H. Liu et al., "Reduced complexity compression algorithms for direct-write maskless lithography systems," *J. Micro/Nanolithogr. MEMS MOEMS*, **6**(1), 013007 (2007).
8. G. Cramer, H.-I. Liu, and A. Zakhor, "Lossless compression algorithm for REBL direct-write e-beam lithography system," *Proc. SPIE* **7637**, 76371L (2010).
9. J. Yang and S. A. Savari, "A lossless circuit layout image compression algorithm for maskless lithography systems," in *Proc. of the 2010 Data Compression Conf.*, pp. 109–118, IEEE, Snowbird, UT (2010).
10. J. Yang and S. A. Savari, "A lossless circuit layout image compression algorithm for maskless direct write lithography systems," *J. Micro/Nanolithogr. MEMS MOEMS* **10**(4), 043007 (2011).
11. F. Krecinic, S.-J. Lin, and J. J. H. Chen, "Data path development for multiple electron beam maskless lithography," *Proc. SPIE* **7970**, 797010 (2011).
12. J. Yang and S. A. Savari, "Improvements on Corner2, a lossless layout image compression algorithm for maskless lithography systems," *Proc. SPIE* **8352**, 83520K (2012).
13. J. Yang, X. Li, and S. A. Savari, "Hardware implementation of Corner2 lossless compression algorithm for maskless lithography systems," *Proc. SPIE* **8323**, 83232O (2012).
14. E. Platzgummer, C. Klein, and H. Loeschner, "eMET POC: realization of a proof-of-concept 50 keV electron multibeam mask exposure tool," *Proc. SPIE* **8166**, 816622 (2011).
15. J. Yang and S. A. Savari, "Transform-based lossless image compression algorithm for electron beam direct write lithography systems," in *Recent Advances in Nanofabrication Techniques and Applications*, B. Cui, Ed., pp. 95–110, InTech, Rijeka, Croatia (2011).
16. S. M. Rubin, *Computer Aids for VLSI Design*, 2nd ed., Addison-Wesley, Boston (1987).
17. "Intel Core i7-920 processor specifications," http://ark.intel.com/products/37147/Intel-Core-i7-920-Processor-(8M-Cache-2_66-GHz-4_80-GTs-Intel-QPI).
18. L. A. Wolsey, *Integer Programming*, Wiley-Interscience, New York (1998).
19. MathWorks, "bintprog," http://www.mathworks.com/help/toolbox/optim/ug/bintprog.html.
20. S. Golomb, "Run length encodings," *IEEE Trans. Inf. Theory* **12**(3), 399–401 (1966).
21. A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Trans. Inf. Syst.* **16**(3), 256–294 (1998).
22. I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM* **30**(6), 520–540 (1987).
23. Nvidia GeForce GTX 690 GPU Specification http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690/specifications.
24. F. Yesilkoy et al., "Implementation of e-beam proximity effect correction using linear programming techniques for the fabrication of asymmetric bow-tie antennas," *Solid State Electron.* **54**(10), 1211–1215 (2010).

Jeehong Yang received his BS degree in electrical engineering from Yonsei University, Seoul, Korea, in 2003. He received his MS and PhD degrees in electrical engineering and computer science from the University of Michigan-Ann Arbor in 2005 and 2012. He was involved in developing various data compression algorithms (text, audio, image, and video) and parallel processing algorithms. He is currently working at Qualcomm Technologies Inc.



Serap A. Savari is an electrical engineer at Texas A&M University. She received her SB and SM degrees in electrical engineering, her SM degree in operations research, and her PhD degree in electrical engineering and computer science, all from the Massachusetts Institute of Technology. She was with the Computer Sciences Research Center at Bell Laboratories, Lucent Technologies from 1996 to 2003 and with the University of Michigan, Ann Arbor from 2004 to 2007. She was an associate editor for source coding for the *IEEE Transactions on Information Theory* from 2002 to 2005. She was the Bell Labs representative to the DIMACS council from 2001 to 2003 and has served on the program committees for numerous conferences and workshops in data compression and in information theory.



H. Rusty Harris earned his BS in engineering physics, and master's and PhD in electrical engineering at Texas Tech University. He previously worked in technology development in Advanced Micro Devices and was assigned to SEMATECH. He is currently an assistant professor in electrical engineering and physics at Texas A&M University.