Journal of
Micro/Nanolithography, MEMS, and MOEMS

SPIEDigitalLibrary.org/jm3

Lossless circuit layout image compression algorithm for maskless direct write lithography systems

Jeehong Yang
Serap A. Savari

SPIE

Lossless circuit layout image compression algorithm for maskless direct write lithography systems

Jeehong Yang
University of Michigan
Electrical Engineering and Computer Science Department
Ann Arbor, Michigan 48109
E-mail: xosh@eecs.umich.edu

Serap A. Savari
Texas A&M University
Electrical and Computer Engineering Department
College Station, Texas 77843

Abstract. The recent algorithm Corner is a transform-based technique to represent a circuit layout image for maskless direct write lithography systems. We improve the lossless circuit layout compression algorithm Corner so that 1. it requires fewer symbols during the corner transform, 2. it has a simpler and faster decoding process, while 3. it requires a similar amount of memory for the decoding process. [DOI: 10.1117/1.3644620]

Subject terms: data compression; information theory; maskless direct write lithography; integrated circuit fabrication.

Paper 10099RRRRR received Aug. 17, 2010; revised manuscript received Jul. 19, 2011; accepted for publication Sep. 9, 2011; published online Nov. 8, 2011.

1 Introduction

Conventional photolithography systems use physical masks, which are difficult to create and nearly impossible to modify. Maskless direct write lithography systems are an interesting alternative which bypass physical masks.[1] As illustrated in Fig. 1, they instead utilize an array of lithography writers to directly write a mask image on a photoresist coated wafer. There are a number of advantages to maskless lithography systems: First, the flexibility gained by replacing physical masks with electronic images makes maskless lithography systems attractive when rapid prototyping is needed for the chip design process. Second, by removing the mask-making process, the initial cost of chip fabrication is significantly reduced.[1] Third, maskless lithography systems have the potential to be improved by software because the mask images are electronically controlled. This last point will be the focus of this paper.

However, maskless lithography systems have a drawback over physical mask lithography systems: they are slow.[2] This issue can be resolved by massively-parallel lithography writers. In Ref. 3, a maskless lithography system using a bank of 80,000 lithography writers operating in parallel at 24 MHz is illustrated. As suggested by Dai and Zakhor,[2] this lithography system can match the conventional photolithography system throughput, one wafer layer per 1 min, but it raises a question on how to provide the massive layout image data (which is typically several hundred terabits per wafer) to the lithography writer. Because of a bandwidth shortage between the storage where the layer images are deposited and the maskless lithography system, obtaining competitive throughput using a maskless lithography system is not possible with conventional data delivery methods.

Dai and Zakhor[2,4] addressed this problem by designing a data delivery 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 maskless lithography system can have higher throughput if the decoder embedded within the maskless lithography writer can quickly recover the original images from the compressed files.

This type of system has two requirements:[3] 1. the compression ratio should be at least (transfer rate of decoder to writer/transfer rate of memory to decoder), and 2. the decoder circuit has to be simple enough to be implemented within the maskless lithography writer as a small add-on. This second constraint requires the use of a decoder operating with little memory.

Dai and Zakhor[2] found that the layout images of control logic circuits tend to be irregular while the layout images of memory cells tend to have repeated patterns. C4, the first lossless layout image compression algorithm proposed by Dai and Zakhor, applies context prediction and finding repeated regions within an image in an attempt to handle the varying characteristics of layout images. Dai and Zakhor later introduced Block C4,[4] which significantly reduces the encoding complexity.

Our work is based on the framework of Dai and Zakhor.[2] In this paper, we introduce a compression algorithm which has a better compression performance and a faster encoding/decoding process than C4 and Block C4. Because our work provides better compression performance, it can be used to solve the data delivery problem of maskless lithography systems with smaller features. Moreover, since our decoding speed is faster than C4 and Block C4, we can obtain higher throughput.

2 Compression Algorithm

2.1 *Overview*

Circuit layouts are typically stored in GDSII (Ref. 5) or OASIS (Ref. 6) formats. GDSII and OASIS represent circuit features such as polygons and lines, and describe them by their corner points.[5,7] GDSII and OASIS formatted data are far more compact than the uncompressed image of a circuit layer. Therefore it may initially appear that the GDSII and OASIS formats are good candidates for this particular application. However, this is not the case because maskless writers operate directly on pixel bit streams and GDSII and OASIS layout representations must be converted into layout

1932-5150/2011/$25.00 © 2011 SPIE

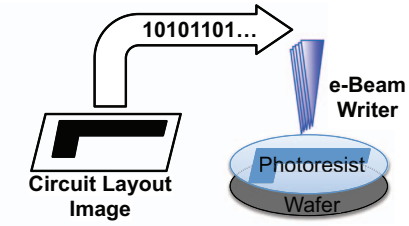

Fig. 1 Maskless lithography.

images before the lithography process begins. In general, this conversion requires 1. eliminating hierarchical structures by replacing all of the copied parts with actual features, 2. arranging the circuit features such as polygons and lines into the corresponding layers of the circuit, and 3. rasterizing (see Fig. 3). As the conversion usually takes hours using a complex computer system with large memory, it cannot be performed within the decoder chip.

Our compression algorithm is inspired by the compactness of the GDSII/OASIS format and is designed to take advantage of ideas such as corner representation and the copying of repeated structures. However, we avoid the complex flattening and rasterizing processes and offer a simple decoding process. We assume throughout that all of the circuit layer images are binary images.

Optical proximity correction (OPC), which is widely used for conventional lithography systems to adjust the shapes of mask features, is not in general necessary for maskless lithography when the application is the direct fabrication of real circuits. OPC is conventionally used to compensate for the image errors due to the diffraction effect,[8] which is not an issue for maskless lithography using electron beams. The only exception is when maskless lithography is used to make precise masks for the fabrication of circuits via photolithography. Since mask making is not a high volume application, we are more interested in the direct fabrication of real circuits, which is the only problem we consider. In this setting we need not worry about OPC.

We instead consider electron beam proximity correction for maskless lithography systems to obtain good quality line edge roughness. This is achieved by applying a multilevel electron beam dosage to each pixel.[9] As shown in Figs. 4(a) and 4(b), uniform electron beam doses result in blurry boundary edges because of the electron beam proximity effect. To compensate for that phenomenon, a higher electron beam dosage was applied to the boundary pixels as in Fig. 4(c) and the proximity effect has been corrected as in Fig. 4(d).

It is possible to represent the proximity corrected layout image using gray images.[9] However, this data eventually has to be reinterpreted as a binary image because the lithography writer does not produce a multilevel electron beam dose

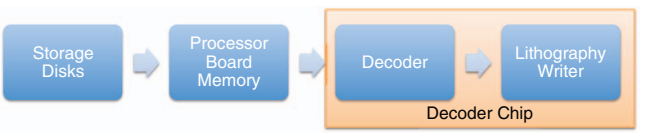

Fig. 2 Data delivery for a maskless lithography system.

during a single write time. Instead, the lithography writer utilizes an electron beam writer to write the corresponding pixel multiple times so that the pixel is exposed with the targeted electron beam dosage; i.e., each electron beam writer uses a proximity-corrected layout image to write a portion of a gray image pixel which corresponds to a block of binary pixels. Our paper considers the "idealized pixel printing model" as in Ref. 3; however, it can be applied to general proximity correction methods by reinterpreting the proximity corrected gray image as a binary image.

Under the idealized pixel printing model, we generated the gray images by the following process as in Ref. 3. First, we start with the GDSII or OASIS layout. Second, as illustrated in Fig. 3, we rasterize the layout image in a 1 nm grid and output a large binary image. Third, this binary image is segmented in blocks and quantized with the appropriate gray level. For example, if we are targeting a 45 nm process technology, the electron beam pixel size chosen would be 22 nm (half the minimum feature size) and a block of 22 $\times$ 22 binary pixels would make up a single gray pixel. To obtain a 1 nm edge placement Dai[3] suggested counting the number of fills in every 22 $\times$ 22 pixel block and quantizing that number to one among 32 levels.

The generated gray image is then reinterpreted as a binary image so that it directly maps to the lithography writer control signal, by changing the grid size. For the previous example, we want a 22 nm pixel to have a 1-nm edge replacement. That means, we need at least 22 dose levels.* Therefore, instead of 32 levels we can choose a quantization of 25 ($=5\times5$) levels. Since every electron beam dose will increase a single level, each 22 nm pixel is written 25 times or, equivalently, that each control signal covers a 4.4 nm ($=22/5$) pixel size. For simplicity, we can recompute the numbers so that each control signal covers a 4 nm ($=22/5.5$) pixel size and is one among 30 levels ($\approx5.5\times5.5$) for each 22 nm pixel. Finally, this new binary representation can be obtained by rasterizing the layout GDSII or OASIS file to the targeted grid (4 nm for example).

A simple illustration of this binary image to gray image and gray image to binary conversion is shown in Fig. 5. Here the grid size of the gray image is set to 4 nm and the grid size of the binary image is set to 2 nm. Fig. 5(a) shows the binary rasterized image at a grid size of 1 nm. By grouping 4×4 blocks of this image as a pixel and quantizing the number of fills to 4 levels (2 bits), we obtain Fig. 5(b), which corresponds to the gray image at a grid size of 4 nm. Because each gray image pixel has 4 levels, we could interpret this as a 4 nm pixel written 4 times. Here a 4 nm gray pixel corresponds to a 2×2 block of binary pixels from a 2 nm binary grid as in Fig. 5(c). Observe that Fig. 5(c) is not generated from Fig. 5(b) but could be generated from Fig. 5(a) by forming the appropriate 2×2 blocks.

Finally, an overview of the compression algorithm is shown in Fig. 6.[10] We begin by seeking frequently occurring patterns. This part roughly matches the patterns that are frequent to improve the compression performance without increasing the encoding complexity. We next transform the remaining image into corner images. We then apply run length encoding (RLE)[11] and end-of-block (EOB) coding

*Note that 32-level was chosen for the previous example so that each pixel can be represented with 5 bits, but 22-level is the requirement.

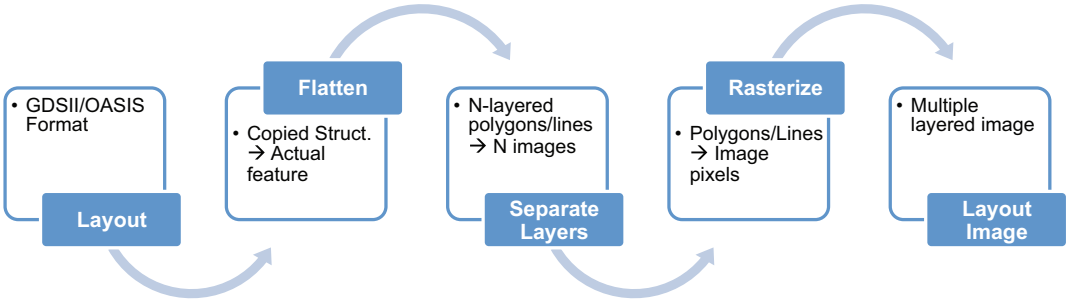

Fig. 3 Preparing layout images from a circuit layout—rasterizing process.

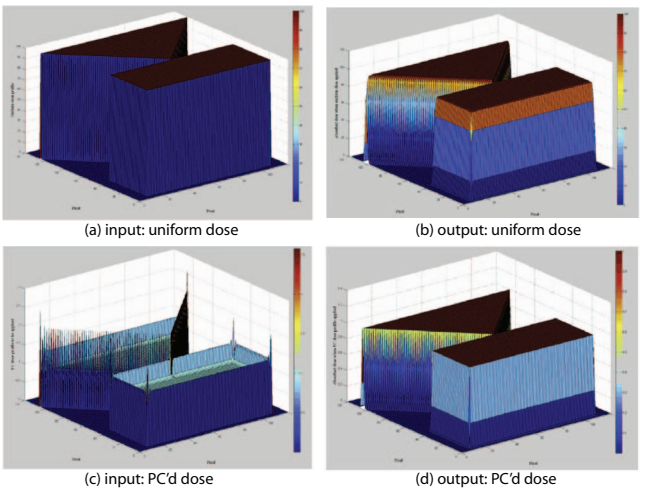

Fig. 4 Proximity correction using gray tone exposure (Ref. 9).

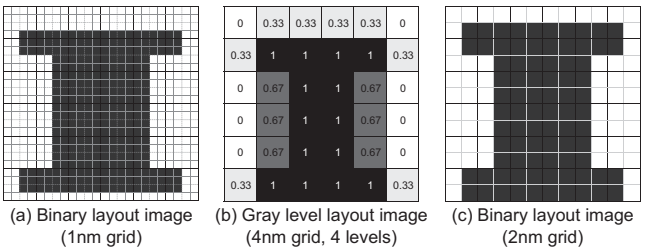

Fig. 5 Binary image versus gray image.

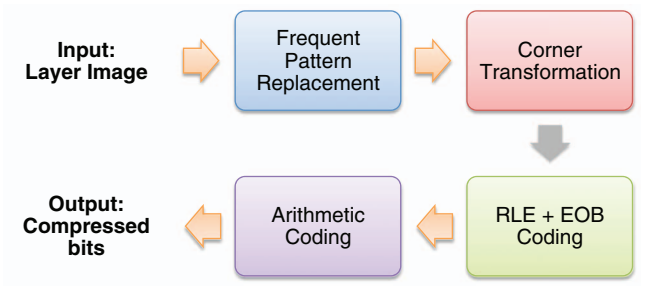

Fig. 6 Compression process overview (Ref. 10).

to further compress the transformed image. We finally compress the resulting data stream with arithmetic coding.[12]

In Secs. 2.2 and 2.3, we will first describe how the frequent pattern replacement and corner transform processes work as separate processes. In Sec. 2.4, we will illustrate how we should tweak them to work in a unified system, and in Sec. 2.5 we will describe the final entropy coding process.

2.2 *Frequent Pattern Replacement*

GDSII/OASIS formats are designed to take advantage of the hierarchical structure presented within circuit layouts. In particular, if a substructure is repeatedly used in a circuit layout, GDSII/OASIS identifies it and then refers to it whenever the substructure occurs. For example, the GDSII/OASIS representation for an 8-bit adder, which is implemented using two 4-bit adders, will consist of a definition of a 4-bit adder and the description of the 8-bit adder in terms of two 4-bit adders as in Fig. 7.

By searching the GDSII/OASIS file and counting the number of references for each definition, we obtain a list of frequent patterns in the mask image. We could alternatively run a complex pattern matching algorithm to detect the frequent patterns, but the outcome would not be significantly different due to the blockwise design of typical circuits. Our approach is not efficient for compression purposes if the input GDSII/OASIS file is unstructured, but this issue can be handled by a preprocessing algorithm proposed by Gu and Zakhor[13] which efficiently restructures input GDSII/OASIS files.

Figure 8 offers an overview of frequent pattern replacement. The input to the procedure is the GDSII/OASIS file and the layer number of the layout image. The first step is to detect all of the substructures that are defined in the GDSII/OASIS representation. The next step is to count the number of references of each substructure extracted in the previous step. We proceed to order the substructures by the number of their occurrences and select the most frequent P of them, where P is chosen so that the representation of the P substructures requires less than P_{SIZE} bytes of memory. Note that we are not fixing the number P, but P_{SIZE}, the memory that is required to store the P patterns. Also, note that because of the decoder memory constraint, P_{SIZE} is very small, and hence, we can only pick a small number of patterns. Each of the P substructures undergoes the rasterizing

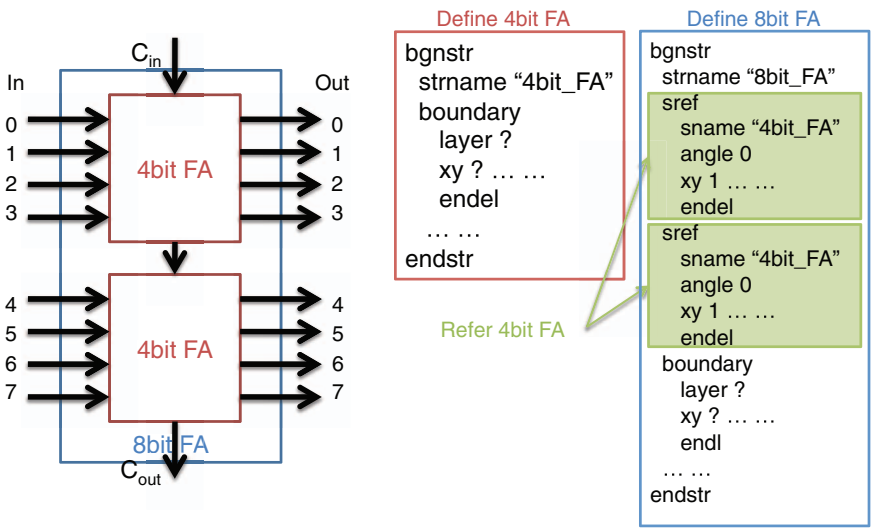

Fig. 7 8-bit adder using two 4-bit adders: GDSII definition (solid line boxes) and references (shaded boxes).

process.[†] During the rasterizing process, we increase both the pattern width and height by 1, so that the pattern has an empty top row and an empty leftmost column as in Fig. 8. The reason why we are adding this empty boundary will be explained in Sec. 2.4. Finally, given the binary layout image the encoder seeks the P patterns within the image. Whenever one of the patterns is matched within the search region, the encoder will replace the top left part of the corresponding part of the image with a string described below and will replace the rest of the filled pixels that have been matched with "0"s (or empty). Note that since the GDSII/OASIS format allows rotated/flipped copies of a substructure, the encoder needs to check all possible rotations/flips during the pattern matching process, and return an encoding accordingly.

In order to restrict the decoder memory, the inside pattern (which excludes the surrounding empty row and column) is encoded using the following symbol string format:

$$\$PPP\ RR\ F.$$

The symbol \$ is a flag for a pattern. The choice of this flag will become clearer in Sec. 2.3. PPP is a length $\lceil \log_2(P) \rceil$ binary string representing the pattern number, RR is a two-bit string representing the rotation of the pattern, and the last bit F indicates whether the pattern is (horizontally) flipped or not. If $P = 1$, then PPP can be omitted. Similarly, if a pattern was never rotated or flipped, then RR and F are omitted for more efficient representation.

Suppose the pattern p dimension is $w_p \times h_p$. Then, in order to make this encoded stream fit in the top row of the image, we assume the original image size $\texttt{min}(w_p, h_p)$ is no shorter than $\lceil \log_2(P) \rceil + 2 \cdot \texttt{is (Rotation)} + \texttt{is (Flip)} + \lceil \texttt{max}[\log_2(w_p - q_{w_p}), \log_2(h_p - q_{h_p})] \rceil$ where $\texttt{is(x)}$ is 1 if x is permitted and 0 otherwise, and q_{w_p} (or q_{h_p}) is one more than the number of empty rightmost columns (or the bottom columns) of the pattern. The final term $\lceil \texttt{max}[\log_2(w_p - q_{w_p}), \log_2(h_p - q_{h_p})] \rceil$ is added because of the decoder memory restriction which will be explained in Sec. 3.2.

In Fig. 8, we illustrate an example where $P = 1$ and the rasterized layer image of the frequent substructure is a 3×3 square. Since the layout image contains two copies of this pattern without the possibility of a rotation or a flip, the encoder outputs \$ (depicted by one "gray" pixel) for the pixel corresponding to the top-left corner of each 3×3 square pattern in the input layout image and "white" pixels for the remaining 8 pixels. For this pattern, $\texttt{min}(w_p, h_p) = 3$ and is greater than $\lceil \log_2 1 \rceil + 2 \cdot 0 + 0 + \lceil \texttt{max}[\log_2(3-1), \log_2(3-1)] \rceil = 1$.

During the generation of binary layout images an image could be truncated if there is a mismatch between the pixel grid and the GDSII grid. As a result, a pattern could be truncated when it is realized as a binary image; i.e., a substructure could be repeated within the GDSII/OASIS domain but the corresponding pattern may not repeat within the binary image domain. Despite this issue, the experimental results indicate that this approach to pattern extraction yields improvements in compression and encoding time over prior algorithms for this problem. To understand why this is the case, observe that the patterns we extract have a limited size and so the mismatch between the two domains is less likely to occur.

The parts of the layout image that have not been matched during the frequent pattern replacement process are handled by the corner transformation step which we will discuss in Sec. 2.3.

2.3 *Corner Transformation*

In the GDSII/OASIS representation of a structurally flattened single layer, the layout polygons are represented in terms of their corner points. While this representation is efficient for a system in which decoder memory is large, it is infeasible when the decoder memory is restricted because the decoder needs to access a memory block of size $(|x_1 - x_2| + 1) \times (|y_1 - y_2| + 1)$ for the encoder to connect an arbitrary pair of points (x_1, y_1) and (x_2, y_2) as in Fig. 9.

[†]This rasterization process itself is quite complex, but it is much simpler than that of the entire circuit because the size of the pattern is extremely small because of constraints on P_{SIZE}. For example, while it took several hours to generate a single layer image of a full example circuit, it only took a few seconds to generate the entire layer images of the frequent patterns.

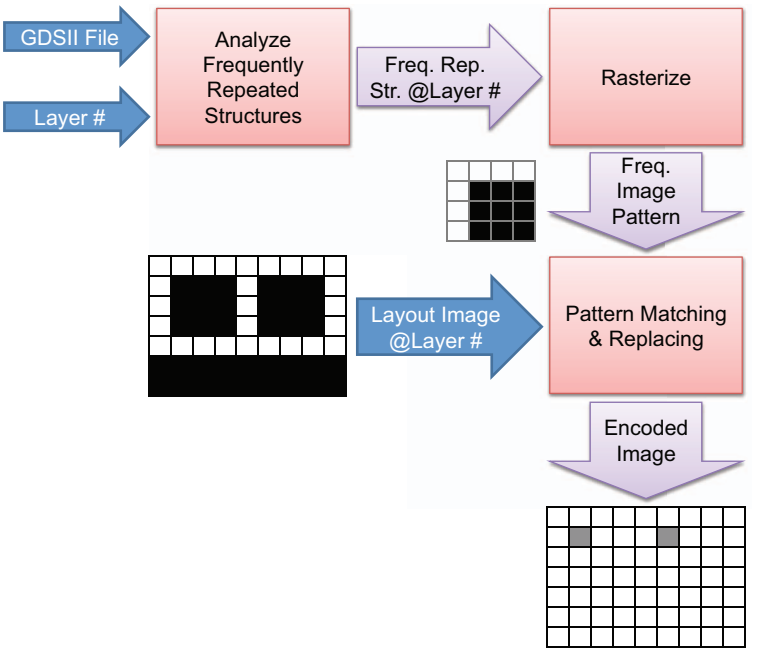

Fig. 8 Handling repeated structures—pattern matching.

However, if the angle of a contour line is constrained to a small set then there can be considerable simplification in the rasterizing process. In our previous research,[14] we restricted the contour lines to be either horizontal or vertical and decomposed an arbitrary polygon into a number of Manhattan polygons, i.e., polygons with right angle corners. This decomposition is well-suited to this application because most components of circuit layouts are produced using CAD tools which design the circuit in a rectilinear space, and even the non-Manhattan parts can be easily described by Manhattan components.

If the contour lines are constrained to be either horizontal or vertical and the decoder scans the image in raster order, i.e., each row in order from left to right, then when the decoder encounters a corner it only needs to decide whether it should reconstruct a horizontal and/or a vertical line. The raster order implies that a corner is either the beginning of a line going to the right and/or down or the end of a line. In our previous research,[14] we specified this decoding decision by representing each pixel with five possible symbols — 'not corner,' "right," "right and "down," "down," and "stop." However, as we will see, this five-symbol representation can be further simplified.

To motivate the simplified transformation, observe that a row (or a column) of the original layout image consists of alternating runs of 1's (fill) and runs of 0's (empty). We encode the pixels where there are transitions from 0 to 1 (or 1 to 0) using symbol "1" and encode the other places using symbol "0." Since most polygons are Manhattan, after applying this encoding in the horizontal direction we obtain alternating runs of 1's and 0's in the vertical direction as seen in Fig. 10(b). Therefore, we can re-apply this encoding in the other direction to produce the final corner image. We call this encoding scheme the binary corner transformation because the final encoded image is binary and the location of the 1-pixels indicate the corners of the polygons. In order to illustrate how the transform is applied, we will first discuss a two-step transformation process and then introduce a one-step transformation process which requires less memory during the encoding process and runs faster than the two-step transformation process.

The two-step transformation process consists of a horizontal encoding step and a vertical encoding step. In the horizontal encoding step, we process each row from left to right. For each row, the encoder initializes the previous pixel value to 0 (not filled). If the value of the current pixel differs from the previous one we encode it with a 1 and otherwise with a 0. After the horizontal encoding is completed, we use the intermediate encoded result as input to the vertical encoding process. This is identical to the horizontal encoding

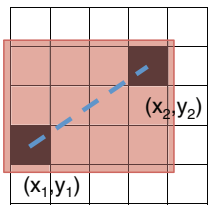

Fig. 9 Required decoder memory (shaded region) to reconstruct a line (dashed line) from (x_1, y_1) to (x_2, y_2).

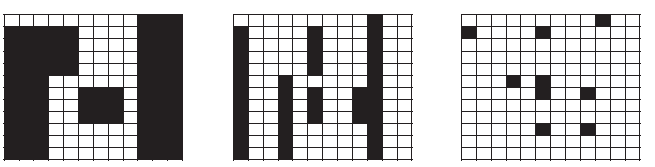

Fig. 10 Two-symbol corner transformation.

Algorithm 1 Transformation : Two-Step Algorithm

Input: Layer image IN $\in \{0,1\}^{C \cdot R}$

Output: Corner image OUT $\in \{0,1\}^{C \cdot R}$

Intermediate: Temporary image TEMP $\in \{0,1\}^{C \cdot R}$

{**Horizontal Encoding**}

1: Initialize TEMP$(x,y) = 0$, $\forall x, y$.
2: **for** $y = 1$ **to** R **do**
3: **for** $x = 1$ **to** C **do**
4: **if** IN$(x,y) \neq$ IN$(x-1,y)$ **then**
5: TEMP$(x,y) = 1$.
6: **end if**
7: **end for**
8: **end for**

{**Vertical Encoding**}

9: Initialize OUT$(x,y) = 0$, $\forall x, y$.
10: **for** $x = 1$ **to** C **do**
11: **for** $y = 1$ **to** R **do**
12: **if** TEMP$(x,y) \neq$ TEMP$(x,y-1)$ **then**
13: OUT$(x,y) = 1$.
14: **end if**
15: **end for**
16: **end for**

Algorithm 2 Transformation : One-Step Algorithm

Input: Layer image IN $\in \{0,1\}^{C \cdot R}$

Output: Corner image OUT $\in \{0,1\}^{C \cdot R}$

1: Initialize OUT$(x,y) = 0$, $\forall x, y$.
2: **for** $y = 1$ **to** R **do**
3: **for** $x = 1$ **to** C **do**
4: **if** IN$(x-1,y-1) =$ IN$(x,y-1)$ and IN$(x-1,y) \neq$ IN(x,y) **then**
5: OUT$(x,y) = 1$
6: **end if**
7: **if** IN$(x-1,y-1) \neq$ IN$(x,y-1)$ and IN$(x-1,y) =$ IN(x,y) **then**
8: OUT$(x,y) = 1$
9: **end if**
10: **end for**
11: **end for**

process except that instead of processing rows we process each column from top to bottom.

The algorithm is summarized in Algorithm 1. In the algorithm, x is the column index $[1, \cdots, C]$ of the image and y is the row index of the image $[1, \cdots, R]$.

As we can see from Line 13 of the algorithm, OUT$(x,y) = 1$ only if TEMP$(x,y) \neq$ TEMP$(x,y-1)$. That is, OUT$(x,y) = 1$ only if TEMP$(x,y) = 1$ and TEMP$(x,y-1) = 0$, or if TEMP$(x,y) = 0$ and TEMP$(x,y-1) = 1$. Since TEMP$(x,y) = 1$ only if IN$(x-1,y) \neq$ IN(x,y) as in Line 5, we can simplify the corner transform process as in Algorithm 2.

The preceding algorithm bypasses the need for intermediate memory. Here pixel (x,y) is processed as a function of the input pixels $(x-1,y)$, $(x,y-1)$ and $(x-1,y-1)$. This simplification results in a much faster running time. Finally, note that the transformation can handle layout images with width-1 lines as shown in Fig. 11.

2.4 *Frequent Pattern Replacement + Corner Transformation*

In Secs. 2.2 and 2.3, we separated the discussion of frequent pattern replacement and corner transformation. Since the output of frequent pattern replacement is a ternary image, a modification is needed to the corner transformation. Figure 12 illustrates the tweaking of the frequent pattern replacement and the corner transformation processes when $P = 1$. The frequent pattern replacement process now outputs two images, namely the matched pattern image and the residue image produced by removing the matched patterns from the original layout image. In this decomposition the pattern embeddings are compressed by the frequent pattern replacement and the residue image is compressed by the corner transformation.

Note that the filled pixels in our "corner" image are more closely related to transitions than to actual corners in the original layout image. Therefore, filled pixels in the corner image can appear to the right, down, or right-down of the corresponding actual corner in the layout image and can hence overwrite the frequent pattern stream if the frequent pattern does not contain an empty top row or an empty left column. Because we added an empty row and column to the frequent patterns, we can add the matched pattern image and the corner image to form the transformed image without any distortion.

Figure 12 illustrates the combination of the frequent pattern replacement and the corner transformation processes.

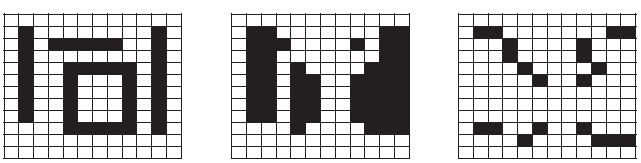

Fig. 11 Handling width-1 lines.

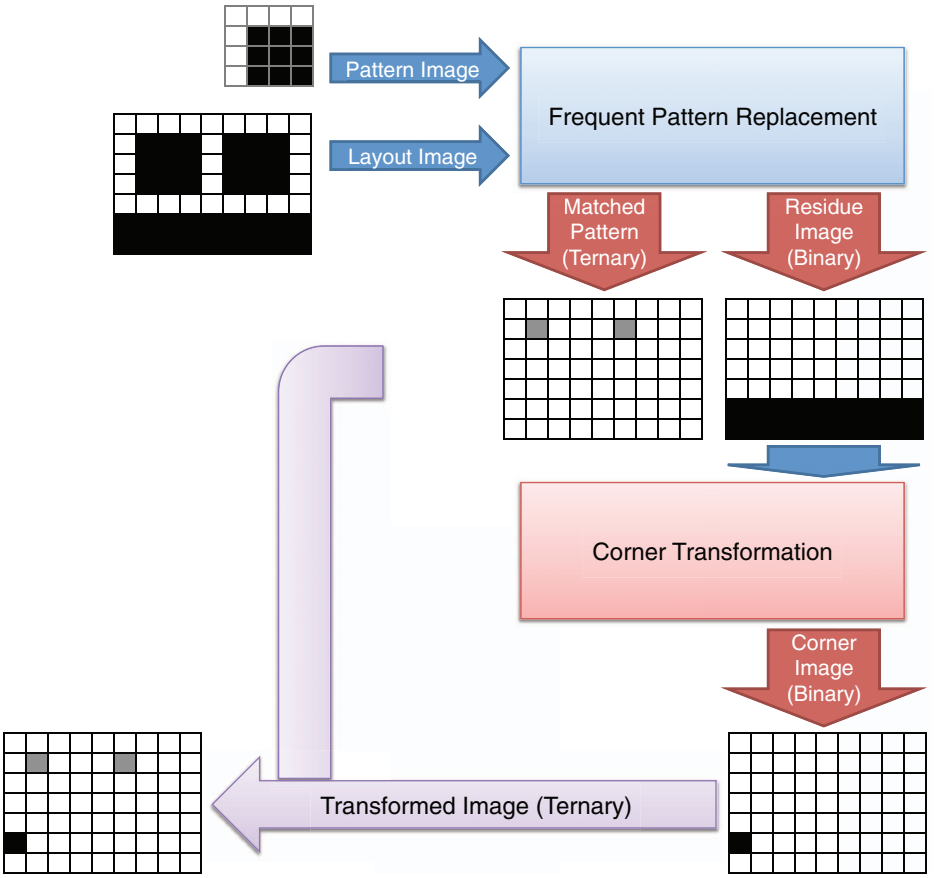

Fig. 12 Handling frequent pattern replacement and corner transformation in a unified system.

The frequent pattern replacement process now outputs two images, the matched pattern image and the residue image produced by removing the matched patterns from the original layout image. In this decomposition, the pattern embeddings are represented by the frequent pattern replacement process and the residue image is represented by the corner transformation. The outputs of the two processes are summed to obtain the transformed image. Observe that two nonzero symbols are never summed, and so the summation is well-defined and the transformed image is over the alphabet $\{0, 1, \$\}$.

We next describe an entropy coding scheme to compress the transformed image.

2.5 *Entropy Coding*

We expect the transformed image to contain long runs of zeroes, and it is therefore effective to use a type of run length encoding (RLE)[11] for compression. The nonzero pixels of the transformed image are written as they are, but each run of zeroes is represented by its run length which we encode with an M-ary representation. More specifically, we introduce new symbols "2," "3," $\cdots$, "M + 1" to represent the base-M symbols "0_M," "1_M," $\cdots$, "$(M-1)_M$." For example, if the transformed stream was "1 00000 00000 1 00000 0000 1 00000 00000 000" and $M = 3$, then the encoding of the stream is "1 323 1 322 1 333" because the run length are 10 ($= 101_3$), 9 ($= 100_3$), and 13 ($= 111_3$), and 2/3/4 is used to represent $0_3/1_3/2_3$.

These M symbols are to be encoded using arithmetic coding[12] for further compression. For arithmetic decoding, we need to allocate memory for each symbol, and hence, in our restricted decoder memory setting, we want to choose M as small as possible. However, small M is not desirable since there are very long runs of zeroes. These long runs of zeroes occur frequently if the circuit features are aligned in a grid manner.

Therefore, in order to obtain a high compression ratio while restricting the size of the decoder memory, we segment each line into k blocks of length L, and we introduce a new "EOB (end-of-block)" symbol "X." If a run of zeroes ends after a block, instead of representing the run length using an M-ary representation, we use the end-of-block symbol X. Hence, we encode a line of zeroes with k X's instead of roughly $\log_M(kL)$ symbols. Continuing the previous example, if $M = 2$, $k = 5$, and $L = 7$, then the transformed stream "1000000 0000100 0000000 1000000 0000000" is represented as "1X 3221X X 1X X," where $2/3$ ($= 0_2/1_2$) is used for the binary representations of runs of zeroes.

After applying EOB coding, there tends to be long runs of "X"s in the encoded stream. Therefore, we can apply run length encoding to this stream by reusing the M symbols for the initial runs of zeroes and introducing N new symbols for an N-ary representation of runs of "X"s . Persisting with the previous example, if $M = N = 2, k = 5$, and $L = 7$, then the next representation is the string "1 5 3221 45 1 45," where 2/3 (or 4/5) is used for the binary representation of runs of zeroes (or "X"s).

Our last encoding step compresses the preceding stream with the implementation of arithmetic coding provided by Witten et al.[15] The decoder requires four bytes per alphabet symbol, and since we used $M+N+3$ symbols, $4(M+N+3)$ bytes were used for arithmetic decoding. Note that M symbols are used for runs of zeroes, N symbols are used for runs of "X"s, 0/1 is used for the corner pixel, \$ is used to handle the frequent P patterns.

3 Decoder

The decoder consists of two parts: 1. an entropy decoder consisting of an arithmetic decoder, run length, and end-of-block decoder which outputs the transformed image, and 2. the transform decoder which reconstructs the layout image from the transformed image. The transform decoder also reconstructs the frequent patterns using the frequent pattern table which has been transmitted to the decoder. The ideas in the implementation of the first part are standard, and we omit them.

For simplicity we will demonstrate how the layer image can be reconstructed from the corner image and we will separately discuss the recovery of the frequent patterns. However, these two processes are conducted in a row-by-row fashion and are executed as a single process because the decoder has restricted memory.

Since each part of the decoding procedure (arithmetic decoding, run length decoding, and vertical/horizontal decoding) processes each symbol based on the previously processed symbols, the entire decoding process can be pipelined to improve the throughput. Observe also that the decoder can be implemented in hardware because the decoding process only requires simple branch and compare operations. For example, arithmetic coding and run length decoders are widely implemented in microcircuits.[16]

3.1 *Corner Transformation*

The corner transformation uses pixels from the previous row and column to decode the current pixel. Because the decoding process depends on the previous row, we designed the decoder to decode the corner image 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 a maskless writer. In our transform decoder we use a row buffer (BUFF). The 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 width bits of memory. "0" means "no transition" while "1" means "transition" which indicates the starting/ending point of a vertical line. Using the current row of the corner image and the buffer, we can reconstruct the layer image as in Algorithm 3. Note that the $\bigoplus$ operation is a binary XOR operation, and is only applied to binary summands.

Lines 4 to 8 process the buffer. If the buffer is filled, i.e., if there is a vertical fill, then the corresponding pixel is filled. Line 9 initializes the status of the horizontal fill. Lines 10 to 16 process each column of the corner image from left to right. If the pixel is 1, the decoder makes horizontal/vertical changes to the image. We first have to update the horizontal fill status (Line 12), fill the output pixel if necessary (Line 14), and update the buffer if necessary (Line 15). If the 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. For example, if the decoder was previously filling the horizontal line, it keeps filling the line (Line 14) and updates the buffer (Line 15) in order to process the next row.

Algorithm 3 Inverse Transformation

Input: Corner image $\mathtt{IN} \in \{0,1\}^{C \cdot R}$

Output: Layer image $\mathtt{OUT} \in \{0,1\}^{C \cdot R}$

Intermediate: Row Buffer $\mathtt{BUFF} \in \{0,1\}^{R}$

1: Initialize $\mathtt{BUFF}(x) = 0$, $\forall x$.
2: Initialize $\mathtt{OUT}(x,y) = 0$, $\forall x, y$.
3: **for** $y = 1$ **to** R **do**
4: **for** $x = 1$ **to** C **do**
5: **if** $\mathtt{BUFF}(x) = 1$ **then**
6: $\mathtt{OUT}(x,y) = 1$
7: **end if**
8: **end for**
9: $\mathtt{Fill} = 0$
10: **for** $x = 1$ **to** C **do**
11: **if** $\mathtt{IN}(x,y) = 1$ **then**
12: $\mathtt{Fill} = \mathtt{Fill} \bigoplus 1$
13: **end if**
14: $\mathtt{OUT}(x,y) = \mathtt{OUT}(x,y) \bigoplus \mathtt{Fill}$.
15: $\mathtt{BUFF}(x) = \mathtt{BUFF}(x) \bigoplus \mathtt{Fill}$.
16: **end for**
17: **end for**

3.2 *Pattern Reconstruction*

If the decoder finds string \$, it starts the pattern reconstruction process. Depending on the number P of patterns, the decoder reads $\lceil \log_2(P) \rceil$ pixels to determine which pattern p is used. If the pattern p is allowed to be rotated, then the decoder next reads another two pixels to determine its rotation. Similarly, if the pattern p is allowed to be flipped, then the decoder reads the next pixel to determine whether p is flipped or not. Hence, the [\$ PPP RR F] stream specifies the pattern to reconstruct.

In order to perform row-wise decoding of these patterns, we use a row buffer (BUFF). When the decoder finds the pattern string [\$ PPP RR F], it places that string into the corresponding buffer location to specify what pattern should be reconstructed. Furthermore, in order to specify which row of the pattern p the decoder should next process the decoder adds a $\lceil \log_2(h_p - q_{h_p}) \rceil$ bit binary representation of the next row number after the pattern

Table 1 Memory—Corner2 parameters.

	File size (bytes)			
Type	M	N	P_{SIZE} (bytes)	Decoder memory (kB)
Wire	128	128	0	4.5
Metal	64	64	930	4.9
Active	64	64	550	4.6
Poly	64	64	930	4.9
Via	64	64	800	4.8

Table 3 Memory—Compression ratio (x).

	Compression ratio (x)			
Type	Corner2	Corner	Block C4	JBIG
Wire	8,098	3,270	139	8,695
Metal	75	28	34	65
Active	914	286	133	555
Poly	109	47	38	104
Via	149	41	42	90
Net Average	192	67	57	149

string, where q_{h_p} is one more than the number of empty rows on the bottom of the pattern. Observe that if the bottom r rows of the pattern p are empty, then the decoder needs to reconstruct $h_p - (r+1)$ more rows. Recall that the shorter dimension of the pattern p, $\texttt{min}(w_p, h_p)$, may be no less than $2 + \lceil \log_2(P) \rceil + 2 \cdot \texttt{is(Rotation)} + \texttt{is(Flip)} + \lceil \texttt{max}\left[\log_2(w_p - q_{w_p}), \log_2(h_p - q_{h_p})\right] \rceil$.

For example, if we are decoding the compressed image in Fig. 8, then the 3×3 square instead of the larger 4×4 square (as in Fig. 8) is stored in the decoder memory. After reading the $ symbol in the second row, the decoder processes the first row of the 3×3 square pattern, and fills the corresponding three pixels of the second row. Then, it updates BUFF to [$0] starting from the leftmost corner of the pattern so that the decoder knows it should process the second row of the 3×3 square pattern. (Here we are assuming that $P = 1$. Since the only pattern is the same whether it is rotated or flipped, there is no need to specify the rotation and flip. The last bit 0 indicates that the decoder now has to reconstruct the second row of the 3×3 square pattern. Since $h_p = 3$ and $q_{h_p} = 1$, we only need 1 bit for the purpose.) When the decoder processes the third row, it reconstructs the second row of the 3×3 square pattern by filling the three pixels, and updates BUFF to [$1] so that the decoder knows it should process the last row of the 3×3 square pattern for the next row. A similar procedure applies to the third row, and after processing the fourth row, BUFF is updated to [00] which terminates the reconstruction of the 3×3 square pattern.

In order to operate this frequent pattern reconstruction along with the corner transformation, the decoder requires $\lceil \log_2(3) \times \text{width} \rceil$ bits for the row buffer and P_{SIZE} bits to store the entire pattern table.

4 Experimental Results

We tested the algorithm on two custom circuits — a memory circuit and a BFSK (binary frequency shift keying) transmitter circuit. 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. The data set used by Dai and Zakhor[2] was proprietary and unavailable to us. For the chips we studied we could run our algorithm Corner2 and Corner[14] on the entire layout image, but we experienced a memory shortage for the encoding process when we attempted to run Block C4[4] on the entire layout image. We therefore segmented the image into the largest components for which Block C4 could be applied. We also considered the standard binary image compression algorithm JBIG (Ref. 17) for comparison purposes. Note that because

Table 2 Memory—File size (bytes).

	File size (bytes)				
Type	Input	Corner2	Corner	Block C4	JBIG
Wire	150,314,718	18,561	45,969	1,083,996	17,288
Metal	75,272,468	997,951	2,644,137	2,238,276	1,160,802
Active	74,883,624	81,958	262,158	562,096	135,032
Poly	75,267,742	691,569	1,590,188	1,958,956	725,976
Via	112,943,616	757,741	2,779,929	2,675,916	1,251,210
Total	488,682,168	2,547,780	7,322,381	8,519,240	3,290,308

Table 4 Memory—Encoding time (s).

	Encoding time (s)			
Type	Corner2	Corner	Block C4	JBIG
Wire	4.23	14.39	1,799.98	7.15
Metal	79.45	12.29	1,849.90	7.72
Active	26.86	12.38	1,813.55	7.24
Poly	52.43	12.47	1,903.93	7.73
Via	45.22	9.56	1,819.31	7.56
Net Average	36.16	12.35	1,830.20	7.43

Table 5 Memory—Decoding time (s).

	Decoding time (s)			
Type	Corner2	Corner	Block C4	JBIG
Wire	2.51	2.18	52.93	2.96
Metal	3.35	2.88	54.24	3.49
Active	3.05	2.23	54.02	3.01
Poly	3.39	2.71	54.53	3.49
Via	3.10	2.62	55.30	3.36
Net Average	2.99	2.48	54.09	3.22

JBIG utilizes 2 to 3 line context-based prediction as well as a well-tuned arithmetic coding implementation, it requires to keep at least length $2 \cdot \text{width}$ bits of row buffer to apply row-by-row decoding. Furthermore, in order to update the prediction table, JBIG may require up to 2^{14} bytes of decoder memory which is larger than the requirements for the Block C4, Corner, and Corner2 decoding processes.

In our experiment, Corner2 was written in C/C++. Corner was implemented in C/C++, Block C4 was implemented in C#, and JBIG was implemented in C/C++ using JBIG-KIT (Ref. 18) and LibTIFF.[19] All of the experiments were performed on a laptop computer having a 2.53 GHz Intel Core 2 Duo CPU and 4 GB RAM.

4.1 *Memory*

For the memory circuit, Corner2 was set with parameters $k = 1$ and $L =$ width, and Corner was set with parameters $M = 128$, $k = 2$, and $L = 8192$. We found that given the nearly even distribution of memory cells among the circuit layers a choice of $k > 1$ results in poorer performance than the choice $k = 1$ because of the frequency of all-zero rows in the images. We chose different M, N, and P_{SIZE} depending on the layer so that the required decoder memory is similar to that of Corner and Block C4 while giving us the best compression performance. The parameters are reported in Table 1 and the decoder memory sizes were 4.9 kBytes for Corner2, Corner, and Block C4, and 20 kBytes for JBIG.

Table 2 shows the total file sizes in bytes of the algorithms. The input file sizes are measured for the raw images without any format (i.e., 1 pixel corresponds to a bit). The circuit layers are categorized into five types: wire, metal, active, poly, and via. Table 3 provides the compression ratios for Table 2. The compression ratios are defined as

$$\frac{\texttt{Input File Size}}{\texttt{Compressed File Size}}.$$

Note that the last row of Table 3 is not the average of preceding rows, but the "net average" which is defined as

$$\frac{\texttt{Total Input File Size}}{\texttt{Total Compressed File Size}}.$$

Because the memory circuit had highly regular wire layers (horizontal/vertical), we treated these layers (metal/poly) separately in the discussion. JBIG gave the highest compression results for the wire layers. However, the performance of JBIG was marginally better than Corner2 as we can see from Table 2. Observe that the wire layers consist of polygons over the entire circuit which are long in the horizontal or vertical direction. These are well-suited to the 3-line context-based prediction of JBIG and the corner representation of both Corner2 and Corner. The improvement of Corner2 over Corner is mainly due to the more efficient RLE methods introduced by Corner2. By contrast, Block C4 is less efficient because its search regions were block segmented for faster processing. For these wire layers our frequent pattern replacement algorithm did not extract any patterns, and therefore we were able to use larger M and N values for Corner2 while maintaining similar decoder memory.

For the other layers, Corner2 outperformed all other algorithms. The compression performance was the lowest for the metal layers (excluding the wire-grid layers) because the patterns were more complex and the truncation problem caused some pattern mismatch within the frequent pattern replacement process. However, both Corner2 and Corner attain higher compression ratios than Block C4 on average. The main reason that Corner2 outperforms Corner is because Corner2 applies additional RLE on EOBs and utilizes the frequent patterns — in this case, it was the pattern of

Table 6 BFSK—File size (bytes).

	File size (bytes)				
Type	Input	Corner2	Corner	Block C4	JBIG
Active	359,739,680	520,764	668,261	2,902,716	739,626
Poly	557,831,968	730,321	993,141	4,374,276	915,734
Metal	624,101,432	1,465,018	2,065,167	5,460,276	1,400,706
Via	465,249,560	970,053	1,376,307	5,027,234	1,074,292
Total	2,006,922,640	3,686,156	5,102,876	17,764,502	4,130,358

the repeated memory cell — while Corner did not. While JBIG had competing performance, Corner2 outperformed it about 30%, while requiring only a quarter of the decoder memory.

Tables 4 and 5 show the average run times of the algorithms for each layer type. As we can see from Table 4, the encoding time of Corner2 was 51 times faster than that of Block C4 which reduced the complexity of the earlier algorithm C4 (Ref. 2) by block segmentation. While both Block C4 and Corner2 utilize the frequent patterns, Corner2 runs much faster because it analyzes the input GDSII file instead of the image to choose the set of frequent patterns. This heuristic has no guarantee of optimality but it improves upon earlier work. However, Corner2 was about 2.9 times slower than Corner because the frequent pattern matching process is nontrivial for the memory circuit; the largest pattern size was 143×52 ($=930$ bytes).

Table 5 shows the average decoding runtimes for each layer. Because of the frequent pattern replacement, Corner2 had a slightly longer decoding time than Corner but was still 18 times faster than that of Block C4 and 8% faster than that of JBIG. Similar to Table 3, the last rows of Table 4 and 5 display "net average," which is defined as

$$\frac{\texttt{Total Encoding/Decoding Time}}{\texttt{Total Number of Layers}}.$$

4.2 *BFSK*

For the BFSK circuit, Corner2 extracts frequent patterns only for the via layer which was very small ($P_{\mathrm{SIZE}}=5$), and the other parameters were fixed to $M=256$, $N=256$, $k=4$, $L=8192$. The parameters of Corner were set to $M=128$, $k=4$, and $L=8192$. These choices allowed for similar decoder memory sizes for Corner2, Corner, and Block C4. For these parameter settings and this circuit the decoder memory sizes for Corner2, Corner, Block C4, and JBIG were, respectively, 8.4, 8.5, 8.4, and 24 kBytes.

Table 6 shows the total file sizes in bytes of the algorithms. The circuit layers are categorized into four types: active, metal, poly, and via. This time, the circuit did not contain the connection wire grid as in the memory array so the wire array was omitted. The compression ratios are computed in Table 7. As we can see from Table 7, Corner2 mostly outperforms other algorithms. The exception is for the metal layers with JBIG, but the difference was marginal (about 1%) (see Table 6).

We can also see that the transform-based techniques Corner2 and Corner both attain higher compression ratios than Block C4. While Block C4 relies on context prediction and finding repeating regions within an image, Corner2 and Corner use actual polygon corners, and these corners cannot be predicted correctly by Block C4's context predictor. Corner2 outperforms Corner mainly because of the improvement in the entropy encoder.

Tables 8 and 9 show the average run times of the algorithms for each layer type. As we can see from Table 8, the encoding time of Corner2 was 196 times faster than that of Block C4, but was about 7% slower than Corner. The improvement over Block C4 is due to the relatively high computational complexity of the context-based prediction and region copy components of the C4/Block C4 algorithm, and the encoding time of Corner2 is slightly longer than that of Corner because of more complex run-length encoding is used.

Table 9 shows that the decoding process for Corner2 is 5.7 times faster than Corner which is 5.2 times faster than Block C4. It was also 1.4 times faster than that of JBIG. The improvement in decoding time over Corner is due to the smaller output alphabet for the corner transformation process. Unlike Corner, which has to decide whether or not the corresponding pixel is a corner and to identify the type of each corner, Corner2 only needs to determine whether the pixel is a corner or not, and hence can be run faster than

Table 7 BFSK—Compression ratio (*x*).

	Compression ratio (x)			
Type	Corner2	Corner	Block C4	JBIG
Active	691	538	124	486
Poly	764	562	128	609
Metal	426	302	114	446
Via	480	338	93	433
Net Average	544	393	113	486

Table 8 BFSK—Encoding time (s).

	Encoding time (s)			
Type	Corner2	Corner	Block C4	JBIG
Active	11.53	13.53	2,636.80	14.58
Poly	17.40	21.05	3,951.47	23.10
Metal	32.70	24.47	4,450.12	28.72
Via	26.40	22.18	6,448.84	25.99
Net Average	17.97	16.78	3,137.88	18.90

Table 9 BFSK—Decoding time (s).

	Decoding time (s)			
Type	Corner2	Corner	Block C4	JBIG
Active	5.43	30.01	163.76	6.59
Poly	8.50	46.84	248.74	10.57
Metal	9.65	59.00	279.89	14.61
Via	8.97	50.10	278.68	12.88
Net Average	6.71	38.57	196.86	9.06

Corner during the transform process. Moreover, in contrast to the memory circuit, the frequent pattern replacement process was trivial for the BFSK circuit.

5 Conclusion

As we have shown in Sec. 4, the algorithm Corner2 requires the least decoder memory and simultaneously has the fastest decoding time and attains the highest compression ratios for both irregular (BFSK) and regular (memory core) circuits. The main improvement of Corner2 over C4 came from how we handle irregular and regular parts of the circuit layout image, and the efficiency of the final entropy coder. First, we use the corner location rather than prediction to handle the irregular parts. Because the context prediction used in C4 easily fails to predict the corners, it resulted in a longer bit sequence. Second, we use the frequent patterns extracted from the hierarchical circuit representation and apply the patterns to the entire circuit layout image. The main contribution of this is that using predefined frequent patterns are more efficient than the LZ-based copying of C4 when the copy region is limited to the previous line (for minimal decoder memory). Finally, our entropy encoder is more efficient for dealing with the long run of zeroes produced by the frequent pattern matching and corner transformation than the combinatorial coder used in C4.

The Corner2 decoder consists of arithmetic decoding, run length decoding, frequent pattern replacement, and corner transformation. Since arithmetic and RLE decoding are widely implemented in hardware, and the frequent pattern replacement and corner transformation processes can be implemented using simple branches, compares, and memory copies, the Corner2 decoding process can be implemented in hardware.

Because our work provides better compression performance, it can be used to solve the data delivery problem of maskless lithography systems with smaller features. Moreover, since our decoding speed is faster than C4 and Block C4, we can obtain higher throughput.

Finally, even though we only have been able to test the algorithm for 250/500 nm circuits, we can argue that it will work well for sub-90 nm circuits as well because as technology develops there are more geometrical constraints that are necessary to guarantee yield. Therefore, the layout polygon shapes that can be used in designing circuits tend to become simpler. However, the polygon density does not increase because as the technology develops, the targeting pixel size becomes smaller. Hence, when the circuit density doubles, the image size in terms of the number of pixels doubles, but the polygon density remains fixed. Since the performance of Corner2 depends on the corner/frequent pattern density within the layout image, the performance is expected to remain the same independent of the process technology.

For future research, we will investigate better frequent pattern extraction techniques. As we have mentioned in Sec. 2.2, extracting frequent patterns from the GDSII/OASIS format is simple and promising. However, this approach may not cover the entire image layer because of possible pattern mismatches coming from pattern truncation during the generation of binary layout images. We expect that a further study on pattern extraction would yield better compression results. Since the application is of the compress once, decompress multiple times variety, increasing the encoder complexity is not critical so long as the complexity of the decoding process remains fixed.

Acknowledgment

The authors thank Dr. Vito Dai for sharing his C4/Block C4 source code. S. A. Savari was supported in part by National Science Foundation Grant No. CCF-1017303.

1. N. Chokshi, D. S. Pickard, M. McCord, R. F. W. Pease, Y. Shroff, Y. Chen, W. G. Oldham, and D. Markle, "Maskless extreme ultraviolet lithography," *J. Vac. Sci. Technol. B*, **17**(6), 3047–3051 (1999).
2. V. Dai and A. Zakhor, "Lossless compression of VLSI layout image data," *IEEE Trans. Image Process.*, **15**(9), 2522–2530 (2006).
3. V. Dai, "Data compression for maskless lithography systems: Architecture, algorithms, and implementation," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley (2008), http://www-video.eecs.berkeley.edu/papers/vdai/phd-thesis.pdf.
4. H. Liu, V. Dai, A. Zakhor, and B. Nikolic, "Reduced complexity compression algorithms for direct-write maskless lithography systems," *J. Micro/Nanolith. MEMS MOEMS* **6**, 013007 (2007).
5. S. M. Rubin, *Computer Aids for VLSI Design*, 2nd Ed., Addison-Wesley, Boston, Massachusetts, App C (1987), http://www.rulabinsky.com/cavd/.
6. Y. Chen, A. B. Kahng, G. Robins, A. Zelikovsky, and Y. Zheng, "Evaluation of the new OASIS format for layout fill compression," in *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems*, Tel-Aviv, Israel, 13–15 Dec. 2004, pp. 377–382 (2004).
7. A. J. Reich, K. H. Nakagawa, and R. E. Boone, "OASIS vs. GDSII stream format efficiency," *Proc. SPIE* **5256**, 163–173 (2003).
8. C. Mack, *Fundamental Principles of Optical Lithography: The Science of Microfabrication*, Wiley, New York (2007).
9. F. Yesilkoy, K. Choi, M. Dagenais, and M. Peckerar, "Implementation of e-beam proximity effect correction using linear programming techniques for the fabrication of asymmetric bow-tie antennas," *Solid-State Electronics*, **54**(10), 1211–1215 (2010).
10. J. Yang and S. Savari, "A lossless circuit layout image compression algorithm for electron beam direct write lithography systems," *Proc. SPIE* **7970**, 79701U (2011).
11. S. Golomb, "Run length encodings," *IEEE Trans. Inf. Theory*, **12**(3), 399–401 (1966).
12. A. Moffat, R. M. Neal and I. H. Witten, "Arithmetic coding revisited," *ACM Trans. Inf. Syst.* **16**(3), 256–294 (1998).
13. A. Gu and A. Zakhor, "Lossless compression algorithms for hierarchical IC layout," *IEEE Trans. Semicond. Manufa.* **21**(2), 285–296 (2008).
14. J. Yang and S. A. Savari, "A lossless circuit layout image compression algorithm for maskless lithography systems," *Proceedings of the 2010 Data Compression Conference*, Snowbird, Utah, pp. 109–118 (2010).
15. I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communicat. ACM* **30**(6), 520–540 (1987).
16. M. Peon, R. R. Osorio, and J. D. Bruguera, "A VLSI implementation of an arithmetic coder for image compression," in *23rd EUROMICRO Conference '97 New Frontiers of Information Technology*, pp. 591–598 (1997).
17. JBIG, http://www.jpeg.org/jbig.
18. JBIG-KIT, http://www.cl.cam.ac.uk/∼mgk25/jbigkit/.
19. LibTIFF, http://www.libtiff.org.

Jeehong Yang received BS degree in electrical engineering from Yonsei University, Seoul, Korea, in 2003. He received an MS degree in electrical engineering and computer science from the University of Michigan, Ann Arbor, in 2005. He is currently a PhD candidate in electrical engineering and computer science at the University of Michigan, Ann Arbor. His research interests include data compression algorithms, information theory, and computer aided design.

Serap A. Savari is an associate professor in the Department of Electrical and Computer Engineering at Texas A&M University, College Station. She received SB and SM degrees in electrical engineering, the SM degree in operations research, and the PhD degree in electrical engineering and computer science, all from the Massachusetts Institute of Technology. She was a member of Technical Staff in the Computing Sciences Research Center at Bell Laboratories, Lucent Technologies from 1996 to 2003 and an associate professor in the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor from 2004 to 2007. Her research interests include information theory, data compression, network coding, and computer and communication systems. 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 been a member of the program committees for numerous conferences and workshops in information theory and in data compression.