Reduced complexity compression algorithms for direct-write maskless lithography systems

Hsin-I Liu
Vito Dai
Avideh Zakhor
Borivoje Nikolić
University of California, Berkeley
Department of Electrical Engineering
and Computer Science
Berkeley, California
E-mail: hsil@eecs.berkeley.edu

Abstract. Achieving the throughput of one wafer layer per minute with a direct-write maskless lithography system, using 22-nm pixels for 45-nm feature sizes, requires data rates of about 12 Tb/s. In our previous work, we developed a novel lossless compression technique specifically tailored to flattened, rasterized, layout data called *context copy combinatorial code (C4)*, which exceeds the compression efficiency of all other existing techniques including BZIP2, 2D-LZ, and LZ77, especially under a limited decoder buffer size, as required for hardware implementation. In this work, we present two variations of the C4 algorithm. The first variation, block C4, lowers the encoding time of C4 by several orders of magnitude, concurrently with lowering the decoder complexity. The second variation, which involves replacing the hierarchical combinatorial coding part of C4 with Golomb run-length coding, significantly reduces the decoder power and area as compared to block C4. We refer to this algorithm as *block Golomb context copy code (block GC3)*. We present the detailed functional block diagrams of block C4 and block GC3 decoders, along with their hardware performance estimates as the first step of implementing the writer chip for maskless lithography. *© 2007 Society of Photo-Optical Instrumentation Engineers.* [DOI: 10.1117/1.2435202]

Subject terms: maskless lithography; complexity; implementation; decoder; prediction; buffer; memory; segmentation.

Paper 06021RR received Apr. 6, 2006; revised manuscript received Oct. 2, 2006; accepted for publication Oct. 9, 2006; published online Feb. 2, 2007. This paper is a revision of a paper presented at the SPIE conference on Emerging Lithographic Technologies X, San Jose, California, Feb. 2006. The paper presented there appears (unrefereed) in SPIE Proceedings Vol. 6151.

1 Introduction

Future lithography systems must produce chips with smaller feature sizes, while maintaining throughput comparable to today's optical lithography systems. This places stringent data handling requirements on the design of any direct-write maskless system. Optical projection systems use a mask to project the entire chip pattern in one flash. An entire wafer can then be written in a few hundreds of such flashes. To be competitive with today's optical lithography systems, direct-write maskless lithography needs to achieve a throughput of one wafer layer per minute. In addition, for 45-nm technology to achieve the 1-nm edge placement required to comply with the minimum grid size specification as well as the 22-nm pixel size as the design rule scale, a 5-bit per pixel data representation is needed to refine the edge placement precision of pixels to less than 1 nm. Combining these together, the data rate requirement for a maskless lithography system is[1,2]

$$\frac{(300\ \text{nm}^2)}{(22\ \text{nm}^2)} \times \frac{\pi}{4} \times \frac{5\ \text{bits}}{60\ \text{s}} = 12\ \text{Tb/s}.$$

To achieve such a data rate, we have recently proposed[1] a data path architecture, shown in Fig. 1. In this architecture, rasterized, flattened layouts of an integrated circuit (IC) are compressed and stored in a mass storage system. Assuming a 10:1 compression ratio for all layers, the layout of a 22×22-mm chip with 40 layers occupies 20 Tb, as illustrated in Fig. 1. The compressed layouts are then transferred to the processor board with enough memory to store one layer at a time. This board will transfer the compressed layout to the writer chip, composed of a large number of decoders and actual writing elements. The outputs of the decoders correspond to uncompressed layout data and are fed into D/A converters driving the writing elements, such as a micromirror array or electron-beam writers.

In the proposed data delivery path, compression is needed to minimize the transfer rate between the processor board and the writer chip, and also to minimize the required disk space to store the layout. In Fig. 1, the total writer data rate of 12 Tb/s is reduced to 1.2 Tb/s, assuming a compression ratio of 10:1, and then is further reduced to 10 Gb/s by using on-board memory with high performance I/O interfaces.[3] Since there are a large number of decoders operating in parallel on the writer chip, an important requirement for any compression algorithm is to have a very

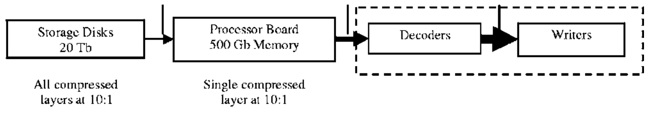


Fig. 1 The data delivery path of maskless lithography.

1537-1646/2007/$25.00 © 2007 SPIE

low decoder complexity. To this end, we have proposed a lossless layout compression algorithm for flattened, rasterized data called context-copy-combinatorial-code (C4), which has been shown to outperform all existing techniques such as BZIP2, 2D-LZ, and LZ77 in terms of compression efficiency, especially under limited decoder buffer size, as required for hardware implementation. However, a major drawback of the C4 algorithm, as presented in our previous work, is its encode complexity.[4] For example, compressing one rasterized, 15-layer layout of a 10×10-mm chip would take 2000 processor months. This can be attributed to the exhaustive search nature of the segmentation portion of the C4 algorithm.

In this work, we present two variations of the basic C4 algorithm. In Sec. 2, we present an overall structure of the C4 algorithm. In Sec. 3, we introduce block C4 to improve the encode complexity of C4 by a few orders of magnitude. As it turns out, block C4 also results in lower decoder complexity. In Sec. 4, we replace the hierarchical combinatorial coding (HCC) part of C4 with Golomb run-length coding to obtain a lower decode complexity algorithm called block Golomb context copy coding (block GC3). In Sec. 5, we discuss hardware implementation aspects of the decoder for block C4 and block GC3. In Sec. 6, we discuss area, power, and speed estimates of block C4 and block GC3 decoders for direct-write maskless lithography systems.

2 Overview of C4

The basic concept underlying C4 compression is to integrate the advantages of two disparate compression techniques: local context-based prediction[5] and Lempel-Ziv (LZ) style copying.[6] The premise is simple: assuming pixels are transmitted in raster order, if a pixel value can be copied from a pixel preceding it, then it does not need to be transmitted again. The encoder simply needs to specify to the decoder how to perform the copy. Likewise, if a pixel value can be predicted from its neighbors, then it does not need to be transmitted. The decoder simply applies the same prediction mechanism to calculate the pixel value. By avoiding redundant transmissions of copied or predicted pixels, C4 achieves compression. For this to work, however, the encoder needs to send additional information to the decoder, the segmentation map, and the error location map.

First, the encoder needs to decide whether copying or prediction is more advantageous; if copying is used, some information needs to be provided as to where to copy from, e.g., from 5 pixels to the left or from the row above. This information is represented as a *segmentation map*, and represents *additional information* that must be transmitted from the encoder to the decoder. Clearly, generating segmentation on a per-pixel basis would increase the amount of information in the segmentation map to such an extent that it may exceed the size of the uncompressed original image. To achieve a high compression rate, the C4 encoder must carefully group individual pixels into *copy regions* and *prediction regions*. In the case of C4, *copy regions* are defined as nonoverlapping rectangles. Each copy region is represented as four coordinates (x, y, w, h), where x and y represent the position of the rectangle, and w and h represent the width and height, respectively. In addition, two parameters are used to specify how to perform the copy (dir, d), where dir represents the direction to copy from, left or above, and d represents the distance in pixels. Thus, in total, six parameters are associated with each copy region. The prediction region is, by definition, any pixel not included in a copy region.

The automatic segmentation of a layout into copy and prediction regions by the encoder is an extremely compute-intensive task, and is vital to the compression efficiency of C4. Later, in Sec. 3, we examine in detail how and why this is the case, and present a new method called block C4, which applies a lower complexity segmentation algorithm. Fortunately, for application to the maskless lithography data path in Fig. 1, the segmentation complexity does not affect the decoder, as it simply receives the segmentation map from the encoder.

In addition to segmentation, the encoder needs to specify to the decoder the location and value of *error pixels*. To increase the compression efficiency of C4, we deliberately allow some pixels in the copy or prediction regions to be erroneous, with the intention of correcting them at a later stage. For example, if a large region is nearly an identical copy of a previously encoded region with the exception of a few pixels, it is still advantageous to classify it as a copy region, provided the pixels in error are taken care of. These incorrectly copied or predicted pixels are called *error pixels*, and their location and values must also be transmitted from encoder to decoder. The locations of these error pixels are represented by a binary *error location map*, where a 0 indicates a correctly predicted/copied pixel, and a 1 indicates an error pixel. This error location map can be compressed with any standard binary compression technique, such as arithmetic coding. In C4, we use hierarchical combinatorial coding (HCC), a low-complexity alternative to arithmetic coding proposed in our previous work.[7] Later, in Sec. 4, we examine an alternative to HCC for compressing the error location map, Golomb coding, and evaluate its effect on decoder complexity and compression efficiency. In addition, the value of error pixels, i.e., error values, must also be transmitted from encoder to decoder. Error values are compressed using any entropy coding technique, such as Huffman coding.

The number of error pixels has a direct effect on the compression efficiency of C4. Fewer error pixels translate to fewer *error values* to be encoded. It also translates to fewer 1's in the error location map, resulting in higher compression efficiency for HCC. Consequently, the goal of the C4 encoder is to minimize the frequency of error pixels, which is determined entirely by two factors: the properties of the layout itself, and the efficacy of the segmentation algorithm in taking advantage of these properties.

2.1 *Factors Affecting the Compression Efficiency of C4*

At the beginning of this section, we stated that "the basic concept underlying C4 compression is to integrate the advantages of two disparate compression techniques: local context-based prediction and LZ-style copying." Why should this strategy be advantageous for compressing layout? First, layout generated by layout designers are mostly Manhattan structured. This means that the majority of pixels may be classified as part of a vertical edge, horizontal

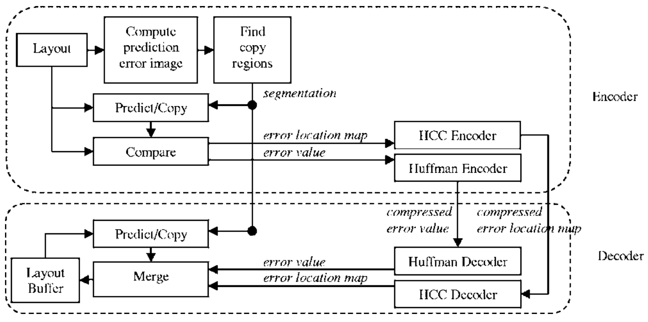


Fig. 2 Block diagram of C4+LP encoder and decoder for gray-pixel images.

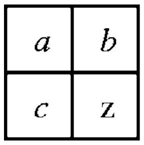
edge, or a region of constant intensity. In context-based prediction, C4 examines neighboring pixels of a given pixel to determine whether it is part of an edge, or a region of constant intensity, and makes a prediction accordingly. Given the raster scan ordering, horizontal edges continue right, vertical edges continue down, and constant intensity regions remain the same color. This prediction mechanism typically fails only at corners of polygons, so the number of prediction error pixels is proportional to the number of polygon vertices. Therefore, for sparse features, it is advantageous to apply predictions, as empirically verified.[1]

For nonsparse layouts, such as dense memory layouts, C4 takes advantage of the fact that a single cell is replicated many times in the layout. After one cell pattern is encoded with prediction, the remainder can be generated with copying. Consequently, for dense repetitive features, it is advantageous to apply copy regions, also verified empirically.[1]

In general, a layout contains a heterogeneous mix of sparse irregular features and dense repetitive features. It is the task of the C4 encoder to appropriately place as few copy regions as possible to capture the repetitive features, so as to minimize the total number of error pixels. This is done with a greedy heuristic search algorithm that attempts to find few copy regions with maximal coverage of the layout, resulting in a minimum number of error pixels.[4] A high-level overview of the algorithm is presented in Sec. 3.

2.2 *Block Diagram of the C4 Encoder and Decoder*

Figure 2 shows a high-level block diagram of the C4 encoder and decoder for flattened, rasterized gray-level layout images. First, a prediction error image is generated from the layout, using a simple three-pixel prediction model to be described shortly. Next, the “find copy regions” block uses the error image to do automatic segmentation as described before, generating a segmentation map between copy regions and the prediction regions. As specified by the segmentation, the predict/copy block estimates each pixel value, either by copying or by prediction. The result is compared to the actual value in the layout image. Correct pixel values are indicated with a 0 and incorrect values are indicated with a 1. The pixel error location is compressed without loss by the HCC encoder, and the corresponding pixel error value is compressed by the Huffman encoder. These compressed bit streams are transmitted to the decoder, along with the segmentation map.



$$
\begin{aligned}
&x = b - a + c\\
&\mathit{if}\,(x < 0)\ \mathit{then}\ z = 0\\
&\mathit{if}\,(x > max)\ \mathit{then}\ z = max\\
&\mathit{otherwise}\ z = x
\end{aligned}
$$

Fig. 3 Three-pixel linear prediction with saturation used in gray-pixel C4.

The decoder mirrors the encoder, but skips the complex process necessary to find the segmentation map, which is received from the encoder. The HCC decoder decompresses the error location bits from the encoder. As specified by the segmentation, the predict/copy block estimates each pixel value, either by copying or by prediction. If the error location bit is 0, the pixel value is correct, and if the error location bit is 1, the pixel value is incorrect and must be replaced by the actual pixel value decoded from the Huffman decoder. There is no segmentation performed in the C4 decoder, so it is considerably simpler to implement than the encoder, satisfying one of the requirements of the data path architecture in Fig. 1.

The prediction algorithm used is linear prediction, where each pixel is predicted from its three-pixel neighborhood, as shown in Fig. 3. Pixel z is predicted as a linear combination of its local three-pixel neighborhood a, b, and c. If the prediction value is negative or exceeds the maximum allowed pixel value max, the result is clipped to 0 or max, respectively. The intuition behind this predictor is simple: pixel b is related to pixel a, the same way pixel z relates to pixel c. For example, if $b=a$, as in a region of constant intensity, then predicting $z=c$ continues that region of constant intensity. Also, if there is a step up from a to b, such that $a+d=b$, as in a vertical edge, then predicting an equivalent step up from c to z, such that $c+d=z$, continues that vertical edge. Likewise, if there is a step up from a to c, such that $a+d=c$, as in a horizontal edge, then predicting an equivalent step up from b to z, such that $b+d=z$, continues that horizontal edge. Thus, these equations predict a continuation of horizontal edges, vertical edges, and regions of constant intensity. Interestingly, this linear predictor can also be applied to a binary image by setting max $=1$, resulting in the same predicted values as binary context-based prediction proposed in our previous work.[4] It is also similar to the median predictor used in JPEG-LS.[5] The linear prediction is used in both the encoder and decoder, as shown in Fig. 2.

3 Block C4

Block C4 is an improvement over the C4 compression algorithm.[4] Similar to C4, it is designed to compress flattened, rasterized layout data, and provides similar compression efficiency but at a tiny fraction of the encoding time. In Table 1, we compare the compression efficiency and encoding time for two 1024×1024 five-bit grayscale layout images, generated from two different sections of the polylayer of a layout. The flavor of C4 used here and throughout this work is C4+LP, the variant of C4 with the lowest decoder implementation complexity. LP stands for linear prediction of a pixel based on its surrounding pixels, as described in Sec. 2. Encoding times are generated on an

Table 1 Comparison of compression ratio and encode times of C4 versus block C4.

Layout	C4 compression ratio	C4 encode time	Block C4 compression ratio	Block C4 encode time
Polymemory	7.60	1608 s (26.8 min)	7.63	14.0 s (115× speed-up)
Polycontrol	9.18	12113 s (3.4 h)	9.18	13.9 s (865× speed-up)

Advance Micro Devices (AMD) Athlon64™ 3200+ Windows XP desktop with 1 GB of memory. As seen, block C4 is 115 times faster for the polymemory layout, and 865 times faster on the polycontrol layout with no noticeable loss in compression efficiency. There are some layouts for which block C4 has significantly less compression efficiency, but the speed advantage is universal. A more complete table of results appears in Table 2 of Sec. 3.4.

The significance of a speed-up of this magnitude in encoding time cannot be understated. Indeed, if we extrapolate from an average encoding time of 30 min per 1024×1024 layout image, a 20×10-mm chip die drawn on a 22-nm five-bit grayscale grid would take over 22 CPU years to encode. Block C4 reduces this to number to 60 CPU days, still a large number, but manageable by today's multicore, multi-CPU computer systems.

Another benefit of block C4, apparent in Table 1, is an approximately constant computation time of about 14 s per 1024×1024 layout image, independent of the layout data, as compared to widely varying computation times of C4, from 27 min to 3.4 h. A predictable and consistent computation time is important to project planning, for example, to maximize tool usage.

The remainder of this section describes how block C4 achieves this encoding speed-up with no loss in compression efficiency. In Sec. 3.1, we introduce the segmentation algorithm of C4 and contrast it with block C4. In Sec. 3.2, we examine the problem of choosing a block size for block C4. In Sec. 3.3, we describe how the block C4 segmentation is encoded for compression efficiency.

3.1 *Segmentation in C4 Versus Block C4*

The basic concept underlying both C4 and block C4 compression is exactly the same. Layout data are characterized by a heterogeneous mix of repetitive and nonrepetitive structures, examples of which are shown in Figs. 4(a) and 4(b), respectively. Repetitive structures are compressed efficiently using LZ-style copying, whereas nonrepetitive structures are better compressed using localized context-

Table 2 Comparison of compression ratio and encode times of C4 versus block C4.

Layout	C4 compression ratio	C4 encode time	Block C4 compression ratio	Block C4 encode time
Polymemory	7.60	1608 s (26.8 min)	7.63	14.0 s (115× speed-up)
Polycontrol	9.18	12113 s (3.4 h)	9.18	13.9 s (865× speed-up)
Polymixed	10.6	1523 s (25.4 min)	11.35	13.9 s (110× speed-up)
M1-memory	13.1	3841 s (1.1 h)	9.50	13.9 s (276× speed-up)
M1-control	18.7	13045 s (3.6 h)	17.3	13.9 s (938× speed-up)
M1-mixed	15.5	13902 s (3.9 h)	14.7	13.9 s (1000× speed-up)
Via-dense	10.2	3350 s (55.8 min)	15.5	14.1 s (237× speed-up)
Via-sparse	16.0	7478 s (2.1 h)	21.6	13.7 s (546× speed-up)

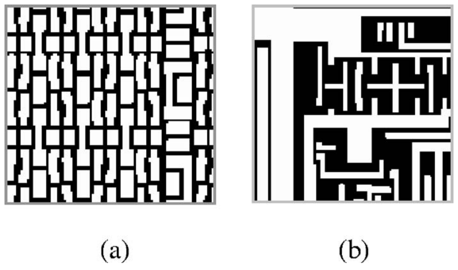


Fig. 4 (a) Repetitive and (b) nonrepetitive layouts.

prediction techniques.[1] The task of both the C4 and block C4 encoder is to automatically partition the image into repetitive copy regions and nonrepetitive prediction regions in a process called *segmentation*. The result is a *segmentation map*, which indicates whether copy or prediction should be used to compress each pixel of the image. Once the segmentation into prediction versus copy is complete, it is straightforward to encode each pixel according to this segmentation map. The segmentation map must also be encoded and included as part of the compressed data, so that the decoder knows which algorithm to apply to each pixel for decoding.

The task of computing the segmentation map accounts for nearly all the computation time of the C4 encoder. Of the encode times reported in Table 1, the encode time excluding segmentation is a constant 1.2 s, for both C4 and block C4. In other words, more than 99.9% of the encode time of C4 and 91% of the encode complexity of block C4 is attributable to segmentation.

In C4, the segmentation is described as a list of rectangular copy regions. An example of a copy region is shown in Fig. 5. Each copy region is a rectangle, enclosing a repetitive section of a layout, described by six attributes: the rectangle position (x, y), its width and height (w, h), the orthogonal direction of the copy ($dir = left$ or $above$), and the distance to copy from (d), i.e., the period of the repetition.

What makes automated C4 segmentation such a complex task is that the "best" segmentation, or even a "good" segmentation, is hardly obvious. Even in such a simple ex-

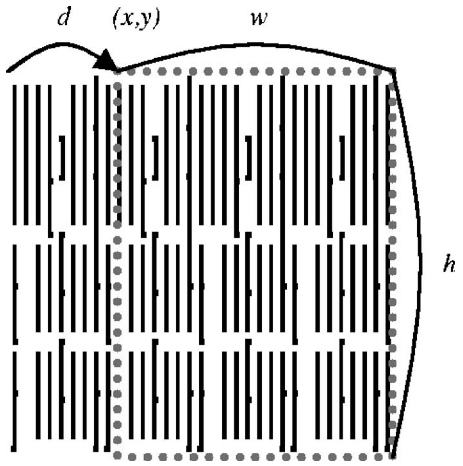


Fig. 5 Illustration of a copy region.

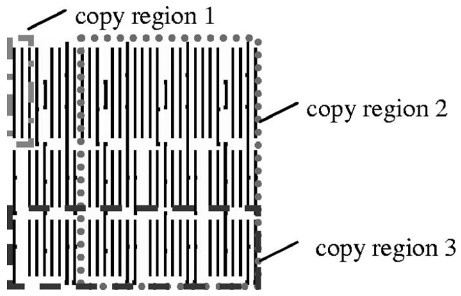


Fig. 6 Illustration of a few potential copy regions that may be defined on the same layout.

ample shown in Fig. 5, there are many potential copy regions, a few of which are illustrated in Fig. 6 as dotted and dashed rectangles. The number of all possible copy regions is of the order of $O(N^5)$ for $N \times N$ pixel layout, and choosing the best set of copy regions for a given layout is a combinatorial problem. Exhaustive search in this space is prohibitively complex, and C4 already adopts a number of greedy heuristics to make the problem tractable.[4] Clearly, further complexity reduction of the segmentation algorithm is desirable.

Block C4 adopts a far more restrictive segmentation algorithm than C4, and as such is much faster to compute. Specifically, block C4 restricts both the position and sizes to fixed $M \times M$ blocks on a grid, whereas C4 allows for copy regions to be placed in arbitrary (x, y) positions with arbitrary (w, h) sizes. Figure 7 illustrates the difference between block C4 and C4 segmentation. In Fig. 7(a), the segmentation for C4 is composed of three rectangular copy regions, with six attributes (x, y, w, h, dir, d) describing each copy region. In Fig. 7(b), the segmentation for block C4 is composed of 20 $M \times M$ tiles, with each tile marked as either prediction (P) or the copy with direction and distance (dir, d). This simple change reduces the number of possible copy regions to

$$O\left(\frac{N^3}{M^2}\right) \sim \frac{N^2}{M^2} \times O(N),$$

a substantial N^2M^2 reduction in search space compared to C4. For the experiment in Table 1, we have $N = 1024$ and $M = 8$, so the copy region search space has been reduced by a factor of 64 million. However, this complexity reduction

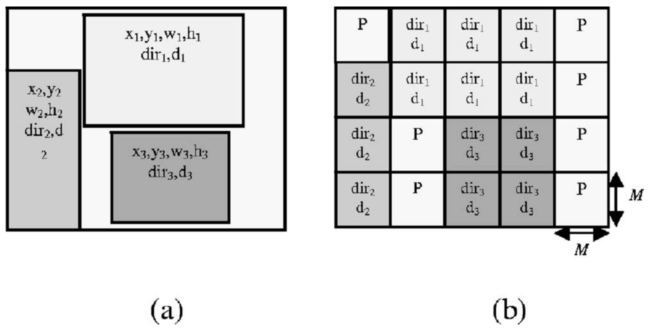


Fig. 7 Segmentation map of (a) C4 versus (b) block C4.

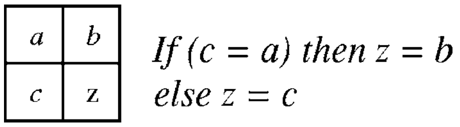


Fig. 8 Three-block prediction for encoding segmentation in block C4.

could potentially come at the expense of compression efficiency.

3.2 *Choosing a Block Size for Block C4*

The three large copy regions in the C4 segmentation map in Fig. 7(a) have been divided into 13 small square blocks in block C4 in Fig. 7(b) in this example. In general, a large repetitive $w \times h$ region is broken up into wh/M^2 tiles in block C4. Each copy region tile in block C4 is represented with only two attributes (dir, d) rather than the six per copy region (x, y, w, h, dir, d) in C4. If a sufficiently large tile is broken up, there may be a net increase in the amount of data needed to represent the segmentation information, which adversely affects the compression ratio of block C4. Smaller values of M accentuate this effect, motivating the use of larger M.

However, large values of M could also be disadvantageous. Comparing the segmentation map of C4 in Fig. 7(a) to that of block C4 in Fig. 7(b), the rectangles are forced to snap to the coarse grid in block C4. In C4, the rectangle boundaries are optimized to delineate repetitive regions from nonrepetitive regions. In block C4, the coarse grid causes this delineation to be suboptimal. Consequently, at the boundary of the copy regions, repetitive regions are predicted and nonrepetitive regions are copied. This suboptimal segmentation could potentially lower the compression efficiency. Of course, the smaller and finer the grid, the lower the occurrence of grid snapping, hence motivating the use of a smaller M.

These arguments suggest that there is an optimal M value that trades off between grid snapping and the break-up of large copy regions. We have empirically found $M=8$ to exhibit the best compression efficiency for nearly all test cases, as compared to $M=4$ or $M=16$. In the remainder of this work, we use $M=8$ in all of our block C4 experimental results, unless otherwise stated.

3.3 *Context-Based Block Prediction for Encoding Block C4 Segmentation*

To further improve the compression efficiency of block C4, we note that the segmentation shown in Fig. 7(b) is highly structured. Indeed, the segmentation can be used to represent boundaries in a layout separating repetitive regions from nonrepetitive regions, and that these repetitions are caused by design cell hierarchies, which are placed on an orthogonal grid. Consequently, block C4 segmentation has an orthogonal structure, and C4 already employs a reasonably efficient method for compressing orthogonal structures placed on a grid, namely context-based prediction.[4]

To encode the segmentation, blocks are treated as pixels and the attributes (P, dir, d) as colors of each block. Each block is predicted from its three-block neighborhood, as shown in Fig. 8. For vertical edges corresponding to $c=a$, it is likely for z to be equal to b. Similarly for horizontal edges corresponding to $a=b$, it is likely for z to be equal to c. Consequently, the prediction shown in Fig. 8 only fails around corner blocks, which are assumed to occur less frequently than horizontal or vertical edges. Applying context-based block prediction to the segmentation in Fig. 9(a), we obtain Fig. 9(b), where $\surd$ marks indicate correct predictions. The pattern of $\surd$ marks could be compressed using HCC[7] or any other binary coding techniques, and the remaining values of (P, dir, d) could be Huffman coded, exactly analogous to the method of coding copy/prediction error bits and values used in C4[2]. For block C4, we choose to use a Golomb run-length coder to compress segmentation error locations. This is because the segmentation error location amounts to a very small percentage of the output bit stream, and as such, applying a complex scheme such as HCC is hard to justify.

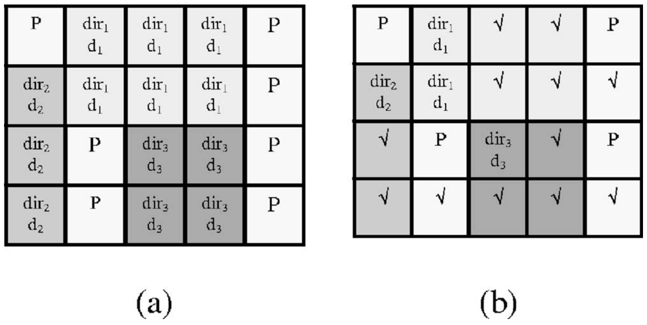


Fig. 9 (a) Block C4 segmentation map (b) with context-based prediction.

3.4 *Compression Results for Block C4*

As we have seen in the previous sections, block C4 speeds up C4 by introducing a coarse fixed grid of 8×8 pixel blocks for the segmentation. This change dramatically reduces the size of the search space for copy regions, resulting in a large speed-up of the encoding time. Even though the coarse grid results in lowered compression efficiency, we have mitigated this by an appropriate choice of the block size, and the application of the context-based block prediction. The full table of results, comparing block C4 to C4, is shown in Table 2. In it, we compare the compression efficiency and encoding time of various 1024×1024 five-bit grayscale images, generated from different sections and layers of an industry microchip. In columns, from left to right, are the layer image name, C4 compression ratio, C4 encode time, block C4 compression ratio, and block C4 encode time. Both C4 and block C4 use the smallest 1.7-kB buffer, corresponding to only two stored rows of data. Encoding times are generated on an AMD Athlon64™ 3200+ Windows XP desktop with 1 GB of memory.

A quick glance at this table makes clear that the speed advantage of block C4 over C4 is universal, i.e., more than 100 times faster than C4, and consistent, i.e., 13.7 to 14.1 s for all layers and layout types tested. In general, the compression efficiency of block C4 matches that of C4. One exception is row 5 of Table 2, where C4 exceeds the compression efficiency of block C4 on the highly regular M1-memory layout.

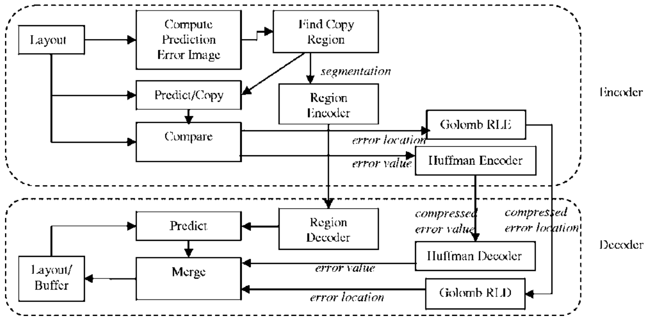


Fig. 10 The encoding/decoding architecture of block GC3.

For this layout, C4′s compression ratio is 13.1, while block C4′s compression ratio is 9.5. In this particular case, the layout is extremely repetitive, and C4 covers 99% of the entire 1024×1024 image with only 132 copy regions. Moreover, many of these copy regions are long narrow strips, less than eight pixels wide, which block C4 cannot possibly duplicate. Consequently, block C4 exhibits a loss of compression efficiency as compared to C4 in this particular case.

On the other hand, in the last two rows of Table 2, the compression ratio of block C4 exceeds the compression ratio of C4 for the dense and sparse "Via" layout. The Via layer consists of a large number of small squares scattered like flakes across the image. It is best compressed with a large number of small copy regions each covering a few squares. Block C4 has two advantages in this case: it uses fewer bits to represent each copy region than C4, and it takes advantage of correlations between copy regions using context-based block prediction. For example, in the Via-sparse layer image, C4 applies 945 copy regions to cover ∼50% of the layout. In contrast, block C4 covers ∼96% of the same image with copy regions, thereby achieving significantly higher compression ratio.

4 Block GC3—Alternate Way to Compress the Error Location

In both C4 and block C4, the error location bits are compressed using HCC. While HCC is useful for encoding the highly skewed binary data in a lossless fashion,[7] when it comes down to hardware implementation the hierarchical structure of HCC implies repetitive hardware blocks and inevitable decoding latency from the top level to the final output. Moreover, as we show in Sec. 6, the HCC block becomes the bottleneck of the entire system due to its long delay. To overcome this problem, we propose to replace HCC in block C4 by a Golomb run-length coder,[8] resulting in a new compression algorithm called block GC3. As such, the Golomb run-length coder in block GC3 is now used to encode error locations of both the pixels in the layout and the segmentation blocks in the segmentation map. Figure 10 shows the block diagram for block GC3, which is more or less identical to that of C4 shown in Fig. 2, with the exception of the pixel error location encoding scheme and segmentation map compression, as discussed in the previous section.

Coding the pixel error location of layouts with the Golomb run-length code could potentially lower the compression efficiency. Figure 11 shows a binary stream coded with both a HCC and Golomb run-length coder. In the upper path, the stream is coded with a Golomb run-length coder. In this case, the input stream is either coded as (0), denoting a stream of B zeroes, where B denotes a predefined bucket size, or coded as $(1,n)$, indicating that a 1 occurs after n zeroes. In general, the algorithm of the Golomb code requires integer multiplication and division. To simplify it to bit-shifting operation, we restrict B to be power of 2. These parameters are further converted into a bit stream, where parameter (0) is translated into a 1-bit codeword and $(1,n)$ takes $1+\log_2 B$ bits to encode. Therefore, a stream with successive 1's can potentially be encoded into a longer code than a stream with 1's that are far apart from each other. On the other hand, in the lower path of Fig. 11, HCC counts the number of 1's within a fixed block size and codes it using enumerative code.[4] In Fig. 11, the block size is 8 and attributes *(2, 11)* denote the 11th greatest 8-bit sequence with two 1's, i.e., 01000010. The attributes *(2, 11)* are further translated to codewords 010 and 01011, which are the binary representations of 2 and 11, respectively. As long as the number of 1's inside the block is fixed, HCC results in a fixed length bit stream regardless of the input distribution.

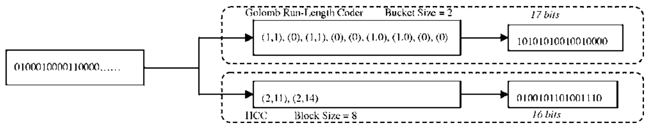


Fig. 11 Golomb run-length encoding process.

Based on this, block GC3 can result in potential compression efficiency loss for certain classes of images. Specifically, Fig. 12 shows a typical layout with successive prediction errors occurring at the corner of Manhattan shapes due to the linear prediction property. Since error locations are not distributed in an independently and identically-distributed (i.i.d.) fashion, there is potential compression efficiency loss due to the Golomb run-length coder as compared to HCC. To alleviate this problem, we adapt the bucket size for the Golomb run-length coder from layer to layer.

As shown in Table 3, block GC3 results in about 10 to 15 % lower compression efficiency than block C4 over different process layers of layouts, assuming decoder buffer size of 1.7 kB. The test images in Table 3 are 1024×1024 five-bit grayscale rasterized, flattened layouts, examples of which are shown in Figs. 4(a) and 4(b). Similarly, Fig. 13 compares the minimum compression efficiency of block C4, block GC3, and a few other existing

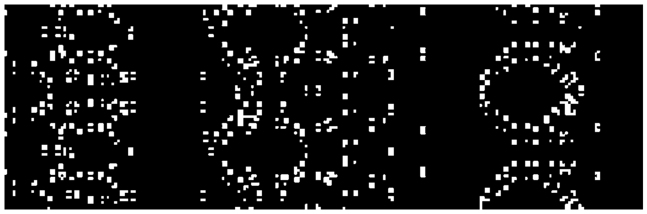


Fig. 12 Visualization of pixel error location for a layout image.

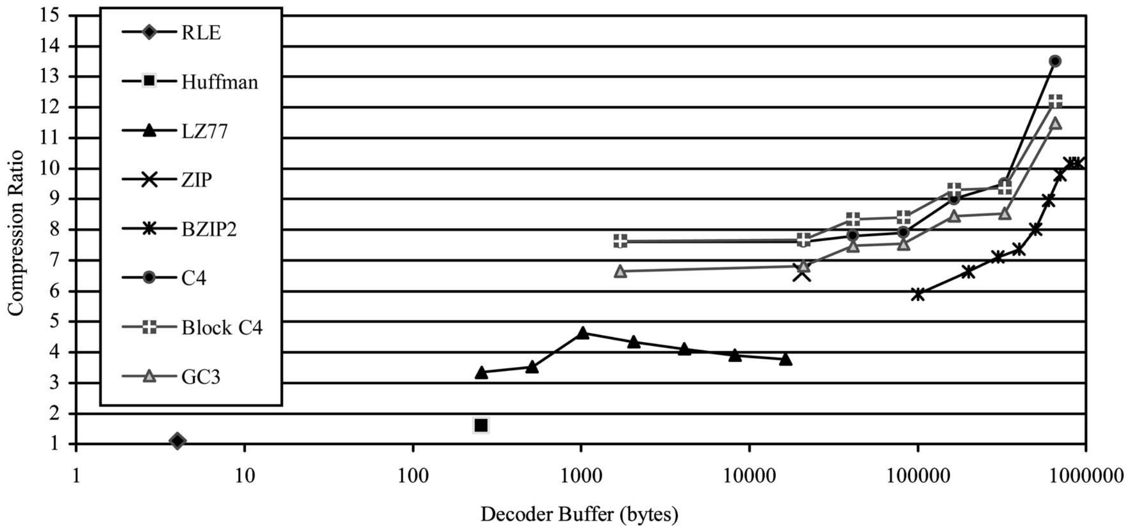


Fig. 13 Compression efficiency and buffer size tradeoff for block C4 and block GC3.

lossless compression schemes as a function of decoder buffer size.[9] The minimum is computed over ten 1024 $\times$ 1024 images manually selected among five layers of two IC layouts. In practice, we focus on 1.7-kB buffer size for hardware implementation purposes. While block GC3 results in slightly lower compression efficiency than block C4 for nearly all decoder buffer sizes, it outperforms all other existing lossless compression schemes such as LZ77, ZIP, BZIP2, Huffman, and RLE.

5 Decoder Architecture

For the decoder to be used in a maskless lithography data path, it must be implemented as a custom digital circuit and included on the same chip with the writer array. In addition, to achieve a system with a high level of parallelism, the decoder must have the data-flow architecture and high throughput. By analyzing the functional blocks of the block C4 and block GC3 algorithms, we devise the data-flow architecture for the decoder.[9]

Table 3 Compression ratio comparison between block C4 and block GC3 for different layers of layout.

Layers	Compression ratio (block C4)	Compression ratio (block GC3)	Bucket size for block GC3
Metal 1 mixed	14.21	12.67	16
Metal 2 mixed	33.81	28.83	64
N active mixed	43.10	36.51	64
P active mixed	66.17	59.24	128
Polymixed	11.00	9.633	16

The block diagram of block C4 decoder is shown in Fig. 14. There are three main inputs: the segmentation, the compressed error location, and the compressed error value. The segmentation is fed into the region decoder, generating a segmentation map as needed by the decoding process. Using this map, the decoded predict/copy property of each pixel can be used to select between the predicted value from "linear prediction" and the copied value from "history buffer" in the control/merge stage by a multiplexer (MUX), as shown in Fig. 15. The compressed pixel error location is decoded by HCC, resulting in an error location map, which indicates the locations of invalid predict/copy pixels. In the decoder, this map contributes to another control signal in the control/merge stage to select the final output pixel value from either the predict/copy value or the decompressed error value generated by the Huffman decoder. The output data are written back to "history buffer" for future usage, either for linear prediction or for copying, where the appropriate access position in the buffer is generated by the address generator. All the decoding operations are combinations of basic logic and arithmetic operations, such as selection, addition, and subtraction. By applying the tradeoffs described in Sec. 4, the total amount of needed memory inside a single block C4 decoder is about 2 KB, which can be implemented using on-chip SRAM.

The block diagram of block GC3 is almost identical to

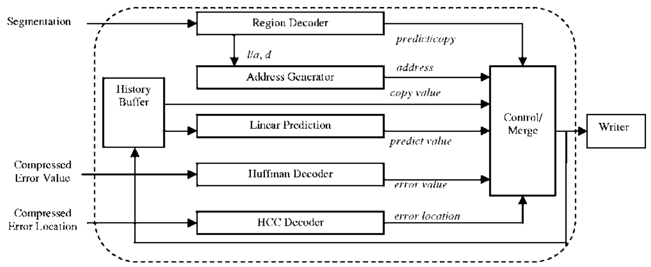


Fig. 14 Functional block diagram of block C4 decoder.

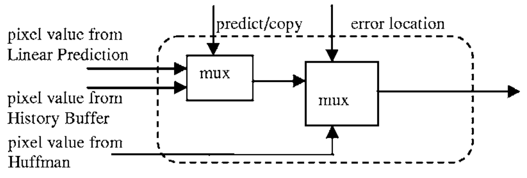

pixel value from Linear Prediction
pixel value from History Buffer
pixel value from Huffman
predict/copy
error location
mux
mux

Fig. 15 Block diagram of merge/control block.

that of block C4 shown in Fig. 14, since it only replaces the HCC block of block C4 by a Golomb run-length decoder.

In the remainder of this section, we discuss the architecture for the block C4 and block GC3 decoders. Even though there are seven major blocks shown in Fig. 14, we focus on the two most challenging blocks of the design, namely the region decoder and HCC. For block GC3, we only discuss the design of the Golomb run-length decoder as its main distinctive feature.

5.1 *Region Decoder*

In the description of the block C4 algorithm in Sec. 3, a segmentation map is introduced to represent the predict/copy segmentation of the layout. Similar to an actual IC layout, the segmentation map is also Manhattan shaped, and can be compressed by prediction algorithms. However, since the segmentation map is an artificially generated image, there is no correlation between the values of adjacent segments. As a result, the segmentation predictor shown in Fig. 8 is used in the region decoder rather than the linear predictor used for pixel predictions in a layout.

Figure 16 shows the block diagram of the region decoder for block C4. The segmentation input has been separated into two streams, the compressed error location and the error value. The core of the region decoder is the segmentation predictor. As shown in Fig. 16, the output of the region decoder is selected to be either the error value or the output of the segmentation predictor, depending on the segmentation error location provided by the Golomb run-length decoder. Similar to the linear predictor for the layout image, the segmentation predictor of block C4 applies the three-block-based prediction: the segmentation of the current microblock is determined by the three adjacent blocks from upper, left, and upper left microblocks under the conditions shown in Fig. 8.

The design of a Golomb run-length decoder is adapted from techniques in the existing literature,[10] as shown in Fig. 17. It is the combination of a barrel shifter and a conventional run-length decoder. The barrel shifter is used as a data buffer to compact the coded error location into an 8-bit data stream, and the decoded result is used as the input to a run-length decoder, resulting in a binary output stream. In

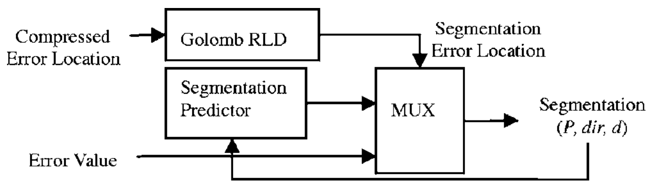


Fig. 16 Block diagram of block C4 region decoder.

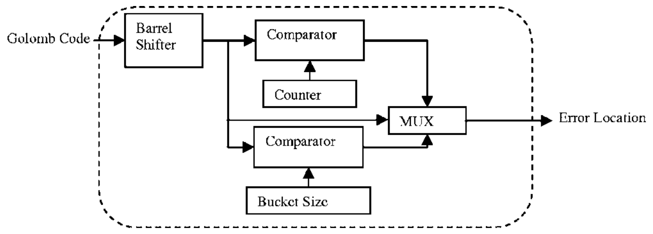


Fig. 17 Block diagram of the Golomb run-length decoder.

contrast to the approach in the literature, we only use one barrel shifter in our design to reduce the hardware overhead.

The proposed region decoder has several architectural advantages over the original C4 region decoder.[9] First, it is implemented as a regular data path, in contrast to a linked-list structure, thus eliminating feedback and latency issues. Second, the output of the block C4 region decoder is the control signal over an 8×8 microblock, which lowers the output rate of the region decoder by 64, and reduces the power consumption. Finally, the length of the segmentation parameter is reduced from 51 bits ($[x,y,w,h,dir,dist]$) to 19 bits, i.e., 8-bit error location and 11-bit error value, resulting in fewer I/O pins in the decoder.

5.2 *Hierarchical Combinatorial Coding Block Design*

Combinatorial coding (CC) is an algorithm for compressing a binary sequence of 0's and 1's. For block C4, it represents a binary pixel error location map. A 0 represents a correctly predicted or copied pixel, and a 1 represents a prediction/copy error. CC encodes these data by dividing the bit sequence into blocks of fixed size H, e.g., $H=4$, and computing k_{block} the number of 1's in each block. If $k_i=0$, this means block i has no 1's so it is encoded as 0000, with a single value k_i. If $k_i>0$, e.g., $k_i=1$, then it needs to be disambiguated between the list of possible 4-bit sequences with one 1: $\{1000,0100,0010,0001\}$. This can be done with an integer representing an index into that list denoted $rank_{block}$. For example, *(2, 3)* represents the third greatest 4-bit sequence with two 1's among all possibilities: $\{1100,1010,1001,0110,0101,0011\}$, i.e., 1001. In this manner, any block i of H bits can be encoded as a pair of integers (k_i, $rank_i$). The theoretical details of how this achieves compression can be found in our previous work,[7] but intuitively it can be expressed as follows: if the data contains contiguous sequences of 0's, and if the length of these all 0 sequences matches the block size H, each block of H 0's can be concisely encoded as ($k_i=0$) with no rank value, effectively compressing the data.

Computational complexity of CC grows as the factorial of the block size H. Hierarchical combinatorial coding (HCC) avoids this issue by limiting H to a small value, and recursively applying CC to create a hierarchy, as shown in Fig. 18.

In Fig. 18, the original binary sequence is the lowest row of the hierarchy bits–level 0. It has been encoded using CC as k–level 0 and “rank–level 0” with a block size $H=4$. We now recursively apply CC on top of CC by first converting the integers in k–level 0 to binary bits–level 1 as follows: 0 is represented as 0, and nonzero integers are represented as

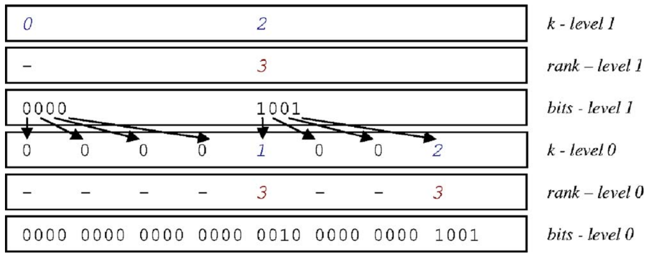


Fig. 18 Two-level HCC with a block size $H=4$ for each level.

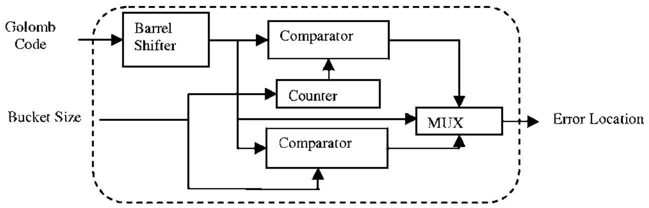


Fig. 20 Block diagram of the Golomb run-length pixel error location decoder in block GC3.

1. Applying CC to bits – level 1 results in k–level 1 and rank–level 1. The advantage of the hierarchical representation is that a single 0 in k–level 1 now represents 16 zeros in bits–level 0. In general, a single 0 in k-level L corresponds to H^{L+1} zeroes in bits–level 0, compressing large blocks of 0's more efficiently. The disadvantage of decoding the HCC is that it requires multiple CC decoding steps as we traverse the hierarchy from top to bottom.

The task of traversing the hierarchy of HCC decoding turns out to be the main throughput bottleneck of the HCC decoder, which in turn is the throughput bottleneck of the entire block C4 decoder. The block diagram of a sequential HCC decoder is shown in Fig. 19(a). Block C4 uses a three-level $H=8$ HCC decoder. The dashed lines separate the HCC levels from top to bottom, and the data that moves between levels are the bits–level L. Three CC blocks represent (k, *rank*) decoders for levels 2, 1, and 0, from top to bottom, respectively. CC–level 2 decodes to bits–level 2. If bits–level 2 is a 0 bit, the MUX selects the run-length decoder (RLD) block, which generates eight zeros for bits–level 1. Otherwise, the MUX selects the CC–level 1 block to decode a (k, *rank*) pair. Likewise, bits level–1 controls the MUX in level 0. A 0 causes the RLD block to generate eight zeros, and a 1 causes CC–level 0 to decode a (k, *rank*) pair. In this sequential design, the output of a lower level must wait for the output of a higher level to be available before it can continue. Consequently, the control signal corresponding to when the output of the lowest level bits level–0 is ready resembles Fig. 19(c). While levels 2 and 1 are decoding, as indicated by the shaded boxes, the output

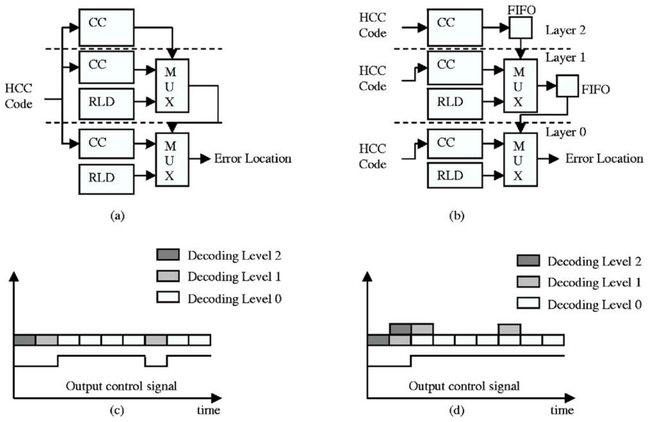


Fig. 19 The decoding process of HCC in (a) top-to-bottom fashion and (b) parallel scheme. The timing analysis of (c) top-to-bottom fashion and (d) parallel scheme.

of layer 0 must stall, reducing the effective overall throughput of the HCC block.

To overcome the problem of HCC decoding, we can parallelize the operation by introducing a first-in first-out (FIFO) buffer between HCC levels, as indicated by the additional squares in Fig. 19(b), and by dividing the input (k, *rank*) values for each HCC level into multiple substreams. The idea is that after an initial delay to fill the buffers of levels 2 and 1, level 0 can decode continually as long as the buffers are not empty. This is guaranteed because one level-2 output bit corresponds to eight level-1 output bits, and 64 level-0 output bits. Level 2 and level 1 can continue to decode into these buffers while level 0 is working. Consequently, the output control signal of the parallel design resembles Fig. 19(d), where only the initial delay is noticeable. The control mechanism of the parallel design is also considerably simpler than the sequential design, because each HCC level can now be controlled independently of the other HCC levels, halting only when its output buffer is full, or its input buffer is empty. Only a 2-byte FIFO is introduced between each level.

5.3 *Golomb Run-Length Decoder for Block GC3*

Since the pixel error location in block GC3 is encoded with a Golomb run-length coder, the pixel error location decoder of block GC3 resembles the Golomb run-length decoder for the segmentation map in the region decoder of block C4. However, for pixel error locations, it is advantageous to use a variable bucket size in the Golomb run-length coder for different process layers to improve compression efficiency, as discussed in Sec. 3. The block diagram of a Golomb run-length decoder for error location is shown in Fig. 20. The only difference between Figs. 17 and 20 is that the variable bucket size is introduced as an input signal to the decoder. The main advantage of this implementation over HCC is that it is "on the fly," and does not require any stall cycles during the decoding process due to its regular data path structure.

6 Decoder Performance

By applying the block diagram discussed in the last section, we have implemented block C4 and block GC3 decoders with logic synthesis tools in a general-purpose 90-nm bulk complementary metal-oxide semiconductor (CMOS) technology. To accurately estimate the speed, power, and area, each of the building blocks has been translated from the hardware description language to the gate level. Table 4 shows the estimate of area, speed, throughput, and power for block C4 and block GC3. Comparing the LZ77 decoder implemented in our previous work,[11] the block C4 decoder

Table 4 Estimated hardware performance comparison of different data path for direct-write maskless lithography systems.

Block		Area (μm^2)	Tp (ns)	Throughput (output/cycle)	Power (mW)
Golomb		4845	1.63	1	1.8
HCC		43,135	1.43	0.71	7.4
Huffman		2745	0.83	1/codeword+2	1.6
Linear prediction		2674	1.06	1	1.1
Address generator		1897	0.6	0.94	0.9
Region decoder		5965	1.39	1	1.7
Control/merge		4166	0.69	1	1.6
Memory		40,080	1.03	1	1.0
Block C4	Single decoder	100,665	1.43	0.71	24.6
	Total (200)	20,254,592	1.43	0.71	6.375 (W)
Block GC3	Single decoder	62,375	1.63	0.94	19.1
	Total (186)	11,550,960	1.63	0.94	5.192 (W)
Direct write		–	–	–	10.8 (W)

uses half of the area and results in twice as much compression efficiency. Furthermore, the block GC3 decoder is half the size of the block C4 decoder.

Table 4 also shows that the critical path for block C4 is in the HCC block, whereas for block GC3, the critical path is in the Golomb run-length decoder. This timing analysis indicates that the single block C4 decoder can operate at 700-MHz clock rate, while block GC3 can only run at 600 MHz.

The throughput of the decoder is summarized in Table 4. The throughput is estimated by multiplying the latency for each block by its usage probability. The usage probability is determined by analyzing the encoder statistics of the test layouts. Although the latency of HCC in block C4 has been smoothed out by applying parallel decoding, there are still some inevitable stall periods in the HCC block, resulting in a throughput of 71% of the peak. In contrast, block GC3 only has a few stalls in the predict/copy segment transition caused by the address generator. The throughput of block GC3 is around 94% of the peak. Combining the clock rate, throughput, and the five-bit output pixel value, we compute the output rate of block C4 and block GC3 decoders to be 2.48 and 2.7 Gb/s, respectively, for a 1024×1024 image. As a result, 200 block C4 decoders or 186 block GC3 decoders are needed to achieve a 500-Gb/s output rate, which corresponds to three wafer layers per hour. To be competitive with today's optical mask lithography systems, which generate one wafer layer per minute, about 5000 decoders are needed to run in parallel.

Table 4 also shows the power consumption of different maskless lithography data path approaches using gate-level simulations of block C4 and block GC3 decoders. As for the direct-write technique, it is possible to apply 80 high-speed I/O pins operating at 6.4 Gb/s to achieve a 500 Gb/s data rate without data compression.[3] However, such a method would result in the total power of the writer to be 10.8 W. By applying block C4 and block GC3 to the data, the power consumption of the writer chip is substantially reduced. The decrease in the input data rate proportionally decreases the I/O power, thus with the average compression rate higher than 10, the I/O power is reduced to less than 1 W. As shown in Table 4, block C4 and block GC3 achieve 41 and 51% power reduction, respectively, as compared to the direct-write technique at 500-Gb/s output rate. As compared to the block C4 decoder, the block GC3 decoder achieves 42% area reduction and 18% power reduction, making block GC3 an attractive option to be implemented in practical maskless lithography systems.

Summary and Future Work

We discuss two variations of C4 to reduce its complexity overhead in both encoding and decoding processes. Block C4 can solve the encoding latency issue by changing the segmentation algorithm into a prediction-based scheme, reducing the encoding time by 2 orders of magnitude. Block GC3, which replaces HCC by a Golomb run-length code for pixel error location coding, can further reduce the hardware decoder overhead of block C4 by 42% in area and 18% in power.

With scaling of minimum features, the resolution enhancement techniques, e.g., proximity correction, are expected to significantly affect the mask layouts. These techniques add more complex features to the layout, affecting its Manhattan structure. For example, more complex corners will result in more prediction errors in the C4 algorithms, which will affect the compression efficiency. However, the impact will be less severe for dense layouts, where the corrected patterns are repetitive as the original shapes and can be compressed by applying the copying technique. The exact compression efficiency has to be derived by testing the actual enhanced layouts.

In addition, the C4 algorithm can be used to compress rasterized mask layouts for pixel-based mask writers. The compressed files can be stored in the mass storage, and decompressed "on the fly" in software as needed by the mask writer. This flow would require the C4 algorithm to be implemented as a plug-in software and integrated into commercial CAD tools.

We plan to fabricate the block GC3 decoders in the near future. Subsequently, the integration of decoder and writer chip poses new challenges. One of them is the input balancing problem, where the input data of different decoders are transmitted from common input pins at a constant rate from the storage device to the processor board. In such a case, the size of the on-board memory and the access mechanism has to be carefully designed. Also, another important topic to investigate is the mixed signal processing problem from the digital memory to the D/A converters.

Acknowledgments

This research was conducted under the Research Network for Advanced Lithography, supported jointly by the Semiconductor Research Corporation (2005-OC-460) and the Defense Advanced Research Project Agency (W911NF-04-1-0304). We would also like to acknowledge Brian Richards of the University of California, Berkeley for his invaluable input on hardware design implementation.

References

1. V. Dai and A. Zakhor, "Lossless layout compression for maskless lithography," *Proc. SPIE* **3997**, 467–477 (2000).
2. ITRS, *International Technology Roadmap for Semiconductors 2005 Edition – Lithography*, see http://www.itrs.net
3. K. Chang, S. Pamarti, K. Kaviani, E. Alon, X. Shi, T. J. Chin, J. Shen, G. Yip, C. Madden, R. Schmitt, C. Yuan, F. Assaderaghi, and M. Horowitz, "Clocking and circuit design for a parallel I/O on a first-generation CELL processor," *IEEE Intl. Solid-State Circuit Conf., Digest of Technical Papers*, vol. 1, pp 526–615 (2005).
4. V. Dai and A. Zakhor, "Advanced low-complexity compression for maskless lithography data," *Proc. SPIE* **5374**(1), 610–618 (2004).
5. M. J. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS," *IEEE Trans. Image Process.* **9**(8), 1309–1324 (2000).
6. J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory* **IT-23**(3), 337–343 (1977).
7. V. Dai and A. Zakhor, "Binary combinatorial coding," *Proc. Data Compression Conf.*, pp. 420, IEEE, Piscataway, NJ (2003).
8. S. W. Golomb, "Run-length encodings," *IEEE Trans. Inf. Theory* **IT-12**(3), 399–401 (1966).
9. V. Dai and A. Zakhor, "Complexity reduction for C4 compression for implementation in maskless lithography datapath," *Proc. SPIE* **5751**(1), 385–400 (2005).
10. M. T. Sun, "VLSI architecture and implementation of a high-speed entropy decoder," *IEEE Intl. Symp. Circuits and Syst.*, pp. 200–203 (1991).
11. B. Nikolić, B. Wild, V. Dai, Y. Shroff, B. Warlick, A. Zakhor, and W. G. Oldham, "Layout decompression chip for maskless lithography," *Proc. SPIE* **5374**(1), 1092–1099 (2004).

Hsin-I Liu received her BS degree in electrical engineering from National Taiwan University in 2003. Since 2003, she has been with the Department of Electrical Engineering and Computer Science at the University of California at Berkeley, where she is pursuing her PhD degree. She is currently a member of the Video and Image Processing Laboratory. Her research interests include image compression, image processing, system architecture design, and digital integrated circuit design.

Vito Dai received the BS degree in electrical engineering at the California Institute of Technology in 1998, and the MS degree in electrical engineering at the University of California (UC) at Berkeley in 2000. He will complete his PhD degree at UC Berkeley in electrical engineering in 2006. He is currently a senior engineer in the Optical Proximity Correction group at Advanced Micro Devices.

Avideh Zakhor received a BS degree from the California Institute of Technology, Pasadena, and MS and PhD degrees from the Massachusetts Institute of Technology, Cambridge, all in electrical engineering, in 1983, 1985, and 1987, respectively. In 1988, she joined the Faculty at UC Berkeley, where she is currently a professor in the Department of Electrical Engineering and Computer Sciences. Her research interests are in the general area of image and video processing, multimedia communication, and 3-D modeling. Together with her students, she has won a number of best paper awards, including the IEEE Signal Processing Society in 1997, IEEE Circuits and Systems Society in 1997 and 1999, international conference on image processing in 1999, and Packet Video Workshop in 2002. She holds five U.S. patents, and is the coauthor of the book, *Oversampled A/D Converters* with Soren Hein (Kluwer Academic Publishers, 1993).

Borivoje Nikolić received the DiplIng and MSc degrees in electrical engineering from the University of Belgrade, Yugoslavia, in 1992 and 1994, respectively, and the PhD degree from the University of California at Davis in 1999. He was on the faculty of the University of Belgrade from 1992 to 1996. He spent two years with Silicon Systems, Incorporated, Texas Instruments Storage Products Group, San Jose, California, working on disk-drive signal processing electronics. In 1999, he joined the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, where he is now an associate professor. His research activities include high-speed and low-power digital integrated circuits and VLSI implementation of communications and signal processing algorithms. He is coauthor of the book *Digital Integrated Circuits: A Design Perspective*, Second Edition (Prentice-Hall, 2003).