Advanced low-complexity compression for maskless lithography data

Vito Dai* and Avideh Zakhor
Video and Image Processing Lab
Department of Electrical Engineering and Computer Science
Univ. of California/Berkeley

ABSTRACT

A direct-write maskless lithography system using 25nm for 50nm feature sizes requires data rates of about 10 Tb/s to maintain a throughput of one wafer per minute per layer achieved by today's optical lithography systems. In a previous paper, we presented an architecture that achieves this data rate contingent on 25 to 1 compression of lithography data, and on implementation of a real-time decompressor fabricated on the same chip as a massively parallel array of lithography writers for 50 nm feature sizes. A number of compression techniques, including JBIG, ZIP, the novel 2D-LZ, and BZIP2 were demonstrated to achieve sufficiently high compression ratios on lithography data to make the architecture feasible, although no single technique could achieve this for all test layouts. In this paper we present a novel lossless compression algorithm called Context Copy Combinatorial Code (C4) specifically tailored for lithography data. It successfully combines the advantages of context-based modeling in JBIG and copying in ZIP to achieve higher compression ratios across all test layouts. As part of C4, we have developed a low-complexity binary entropy coding technique called combinatorial coding which is simultaneously as efficient as arithmetic coding and as fast as Huffman coding. Compression results show C4 outperforms JBIG, ZIP, BZIP2, and 2D-LZ, and achieves lossless compression ratios greater than 22 for binary layout image data, and greater than 14 for grey-pixel image data. The tradeoff between decoder buffer size, which directly affects implementation complexity and compression ratio is examined. For the same buffer size, C4 achieves higher compression than LZ77, ZIP, and BZIP2.

Keywords: compression lithography maskless datapath data-rate layout LZ77 combinatorial BZIP2 algorithm

1. INTRODUCTION

Future lithography systems must produce more dense chips with smaller feature sizes, while maintaining the throughput of one wafer per sixty seconds per layer achieved by today's optical lithography systems. To achieve this throughput with a direct-write maskless lithography system, using 25 nm pixels for 50 nm feature sizes, requires data rates of about 10 Tb/s. In our previous work[1], we have proposed a basic system design of a data processing system capable of delivering tera-pixel data rates as shown in Figure 1.

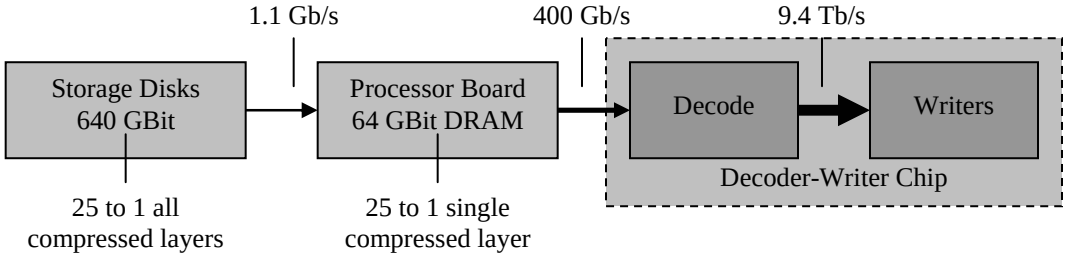


Figure 1. System architecture of a data-delivery system for maskless lithography

This design consists of storage disks, a processor board with DRAM memory, and a decoder-writer chip with data-decoding circuitry fabricated together with a massive array of pixel writers. Layout data for all layers of a single chip is compressed off-line and stored on the disks. Before the writing process begins, only a single compressed layer is transferred from disks to the processor board DRAM memory and stored there. As the writers write a stripe across the wafer, compressed data is streamed from the processor board to the decoder-writer chip in real-time as needed. The on-chip decoding circuitry, in real-time, expands the compressed data stream into the data signals necessary to control the writers. The success of this design is contingent on a data compression algorithm which achieves 25 to 1 compression

*vdai@eecs.berkeley.edu; 510-643-1587; http://www-video.eecs.berkeley.edu/~vdai

ratio on all types of lithography data, and the implementation of a real-time decompression circuitry on the same chip as a massively parallel array of writers.

Applying existing compression techniques to these images, our previous experiments have shown that Lempel-Ziv (LZ) style copying[2], used in ZIP, results in high compression ratios on dense, repetitive circuits, such as arrays of memory cells[1]. However, where these repetitions do not exist, such as control logic circuits, LZ-copying does not perform as well. In contrast, we have shown that context-based prediction[3], used in JBIG[4], captures the local structure of lithography data, and results in good compression ratios on non-repetitive circuits, but it fails to take advantage of repetitions, where they exist[1].

In this paper, we combine the advantages of LZ-copying and JBIG context-modeling into a new lossless image compression technique called Context Copy Combinatorial Coding (C4). C4 is a single compression technique which performs well for all types of layout: repetitive, non-repetitive, or a heterogeneous mix of both. In addition, we have developed combinatorial coding (CC) as a low-complexity alternative entropy coding technique to arithmetic coding[5] to be used within C4.

Section 2 describes the overall structure of C4. Section 3 describes the context-based prediction model used in C4. Section 4 describes LZ-copying in two dimensions and how the C4 encoder segments the image into regions using LZ-copying, and regions using context-based prediction. Section 5 describes CC used to code prediction errors. Section 6 describes the binarization process that extends C4, a primarily binary image compression algorithm, to grey-pixel layout image data. Section 7 includes the compression results of C4 in comparison to other existing compression techniques. Section 8 examines the trade off between decoder buffering, which directly affects decoder implementation complexity, and compression ratio for C4 and other techniques.

2. OVERVIEW OF C4

Figure 2 shows a high-level block diagram of the C4 encoder and decoder. To encode a grey pixel image, it is first converted to a set of binary images. Next, a simple 3-pixel context-based prediction model is computed for the binary image. The resulting prediction error statistics, and prediction error image are used by the Copy block to decide on *copy regions*, i.e. regions of the image encoded with LZ-copying. Pixels outside these copy-regions are encoded using the context-based prediction. The Predict/Copy block generates prediction error bits according to whether the prediction is correct “0” or incorrect “1”. These bits are compressed by the CC encoder. The output of the CC encoder and the copy regions are transmitted to the decoder.

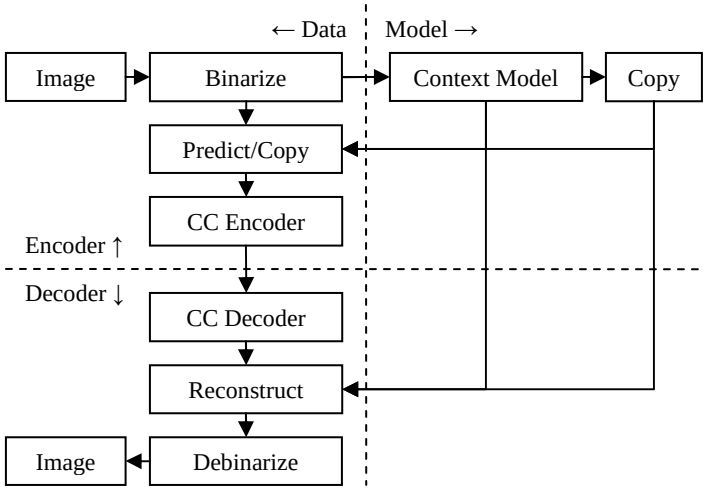


Figure 2. Block diagram of C4 encoder and decoder

At the decoder, the process is reversed. Prediction error bits are decoded using CC, regions outside the copy regions are reconstructed using prediction error bits, and regions inside the copy regions are reconstructed using copy operations. Finally, multiple binary images are merged to restore the original grey-pixel image. No data modeling is performed in the C4 decoder, making it considerably simpler to implement than the encoder, as it is one of the requirements of our application domain.

3. CONTEXT MODEL OF C4

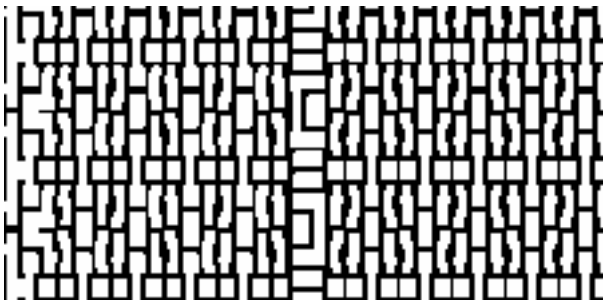
C4 uses a simple 3-pixel binary context-based prediction model, much simpler that the 10-pixel model used in JBIG. Nonetheless, it captures the essential "Manhattan" structure of layout data, as well as some design rules, as seen in Table 1. The pixels used to predict the current coded pixel are the ones above, left, and above-left of the current pixel. The first column shows the 8 possible 3-pixel contexts, the second column shows the prediction, the third column shows what a prediction error represents, and the fourth column shows the empirical prediction error probability for an example layout. From these results, it is clear that the prediction mechanism works extremely well; visual inspection of the prediction error image reveals that prediction errors primarily occur at the corners in the layout. The two exceptional 0% error cases in rows 5 and 6 represent design rule violations. When encoding, the image is scanned in raster order, and pixels are grouped by context. Each correctly predicted pixel is marked with a "0" and each incorrectly predicted pixel is marked with a "1", creating a binary string for each of the 8 contexts. Each of these strings can then be compressed with a standard binary entropy coder. However, in C4, some of these bits are first eliminated by the copying process described in Section 4.

Context	Prediction	Error	Error probability
			0.0055
			0.071
			0.039
			0
			0
			0.022
			0.037
			0.0031

Table 1. The 3-pixel contexts, prediction, and the empirical prediction error probability for a sample layout

4. COPY REGIONS

Layout image data contains regions that are visually "dense" and repetitive. An example of such a region is shown in Figure 3(a). This visual "denseness" is directly correlated to number of prediction errors resulting from context modeling, as seen clearly in the prediction error image in Figure 3(b). The high density of prediction errors translates into low compression ratios for context-based encoding.



(a)

(b)

Figure 3. Dense repetitive layout image data and its resulting prediction error image

Alternatively, since this layout is repetitive, it is possible to specify *copy regions*, e.g. a rectangular region that is a copy of another previously encoded region. An example of such a region is the dashed rectangle shown in Figure 4. A copy

region is specified with six variables: position of the upper left corner *x*,*y*, width *w*, height *h*, distance to the left to copy from *dx*, and distance above to copy from *dy*. For the copy region in Figure 4, every pixel inside the region can be copied from *dx* pixels to its left, and *dy* = 0.

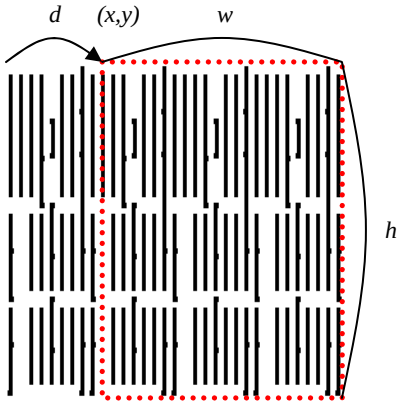


Figure 4. Illustration of a copy left region

Not all valid copy regions are worthwhile as far as compression efficiency is concerned. It *costs* bits to specify the (*x*,*y*,*w*,*h*,*dx*,*dy*) of a copy region. This is weighed against the *benefit*, bits saved because pixels inside the copy region do not need to be coded using context-prediction. Moreover, copy regions overlap with each other, which is undesirable: each pixel should only be coded once, to save as many bits as possible. The goal of the C4 encoder is to find the set of non-overlapping copy regions which maximizes *benefit*, while minimizing cost. These regions are found using exhaustive search, with a number of simplifying assumptions and approximations adopted to make the problem tractable.

Benefit is estimated on a per-pixel basis using entropy as a heuristic: pixels which are incorrectly predicted are assigned a benefit of $-\log_2 p$ bits, where *p* denotes the context dependent prediction error probability. Pixels which are correctly predicted are assigned a benefit of zero, based on a further assumption that $-\log_2(1-p) \approx 0$. The benefit of a copy region is the sum of pixel-benefits inside the region.

Cost is fixed at 51-bits per copy region. Here we have restricted *x*, *y*, *w*, *h* to 10-bits each which is reasonable for our 1024 × 1024 test images. In addition, we assume that copies are either from above, or to the left, so (*dx*, *dy*) is replaced by (*left/above*, *d*) and represented with 11 bits, where *d* denotes the distance left or above to copy from. This assumption is in line with the Manhattan structure of layout data.

The exhaustive search algorithm works as follows: make a list called "*valid*" of all valid copy regions (*x*, *y*, *w*, *h*, *left/above*, *d*), excluding regions whose cost exceed benefit. Next generate a new list called "*worthwhile*" by going through the regions in the *valid* list in order of decreasing benefit. A region is added to the *worthwhile* list if it does not overlap with existing regions already in the *worthwhile* list at the time it is being tested. Finally, reorder this *worthwhile* list in raster order. This is the list provided to the Predict/Copy block of the encoder, and the Reconstruct block of the decoder in Figure 1.

To improve the speed and reduce the memory usage of the exhaustive search, we make the following observations; (a) if x_0, y_0, *left/above*$_0$, d_0, w_0, h_0 is invalid then any x_0, y_0, *left/above*$_0$, d_0, w_1 , h_1 where $w_1 \geq w_0$ and $h_1 \geq h_0$ is also invalid. This reduces the search for valid copy regions; (b) If x_0, y_0, *left/above*$_0$, d_0, w_0, h_0 is valid then any x_0, y_0, *left/above*$_0$, d_0, w_1 , h_1 where $w_1 \leq w_0$ and $h_1 \leq h_0$ is also valid and has a smaller *benefit*. This implies the existence of *largest valid regions* which cannot be contained by other valid regions. These *largest valid regions* can be used in place of an entire family of smaller valid regions contained within, with guaranteed smaller *benefits*. This has the effect of reducing the memory requirements of storing the *valid* list.

5. COMBINATORIAL CODING

We have proposed and developed combinatorial coding (CC) as an alternative to arithmetic coding to encode the prediction bits[5]. The basis for CC is universal enumerative coding[6] which works as follows. For any binary sequence of known length *N*, count the number of ones in that sequence *k* which ranges from 0 to *N*. Represent *k* in binary format using $\lceil \log_2(N+1) \rceil$ bits. Now hypothetically make a list of all possible sequences of length *N* with *k* ones. There are

exactly $C(N,k)$ sequences in this list, one of which corresponds to the one we are need to code. The index of the sequence under consideration is called the *ordinal* or *rank*, and can be represented in binary format using $\lceil \log_2 C(N,k) \rceil$ bits. Enumerative coding is optimal in that as N approaches infinity, the average number of bits used approaches entropy for independently and identically distributed (i.i.d.) Bernoulli(p) binary data with unknown source probability p. However, computing the enumerative code takes $O(N)$ in time, $O(N^3)$ in memory for storing combination values, and requires $O(N)$ bit precision arithmetic[6]. The latter two requirements make direct implementation of enumerative coding using the ranking algorithm impractical.

CC addresses this problem by first dividing the bit sequence into blocks of fixed size M. For today's 32-bit architecture computers, $M = 32$ is a convenient and efficient choice. Enumerative coding is then applied separately to each block, generating a sequence of k, *rank* values, one for each block. The k values which range from 0 to M can be efficiently coded with a dynamic Huffman code with little overhead for a small Huffman table. Then, given M and k, *rank* values have a uniform distribution which can be efficiently coded using a uniform code, or minimum binary code, similar to that used in Golomb coding[7].

The efficiency of CC, as described, is on par with arithmetic coding, except in cases of extremely skewed distributions, e.g. $p < 0.01$. In these cases, the distribution for k which is i.i.d. $Binomial(M, p)$ becomes extremely skewed, and the Huffman code is no longer efficient because $k = 0$ happens too frequently. This is remedied by applying frequency based binarization, described in appendix A, to the stream of k-values resulting in a binary stream and a non-binary stream with $k = 0$ removed. The binary stream is coded with a separate combinatorial code, and the non-binary stream is coded with Huffman coding.

Independent of our development of CC, a similar technique called called Hierarchical Enumerative Coding (HEC) has been developed[8]. The main difference between the two techniques is the method of coding k values after enumerative coding of each block of bits. HEC constructs a sum tree from the k values and applies integer enumerative coding to encode elements of this tree. Based on our own experiments with HEC and the results presented by Oktem and Astola[8], HEC performs well for highly skewed distributions, and when the data exhibits some globally non-stationary but locally stationary behavior. When applied to synthetic i.i.d. data sets where $0.05 < p < 0.5$, HEC does not perform as well as both arithmetic coding and CC. There is a theoretical explanation for this: integer enumerative coding assigns equal code lengths to all integer vectors with the same sum, whereas in the stationary i.i.d. case, the integer vectors in the sum tree have a hypergeometric distribution.

To compare CC with existing entropy coding techniques, we apply 3-pixel context based prediction as described in Section 3 to a layout image data and generated 8 binary streams. We then apply Huffman coding to blocks of 8-bits, arithmetic coding, Golomb run-length coding, HEC, and CC to each binary stream. The results are shown in Table 2. Among these techniques, CC is one of the most efficient in terms of compression, and one of the fastest to encode and decode, justifying its use in C4. Run-times are reported for 100 iterations on an 800 MHz Pentium III workstation. All algorithms are written in C# and optimized with the assistance of VTune to eliminate bottlenecks. The arithmetic coding algorithm is based on that described by Witten, Moffat, and Bell[9].

	Orig.	Huf8	Arith.	Golomb	HEC	CC
Comp. Ratio	1 (242 kb)	7.1	47	49	48	49
Enc. Time (s)	N/A	0.99	7.46	0.52	2.43	0.54
Dec. Time (s)	N/A	0.75	10.19	0.60	2.11	0.56

Table 2. Result of 3-pixel context based binary image compression

It is also worth noting that CC enjoys an implementation advantage over Golomb run-length coding which comes nearest to it for both compression ratio and speed. Specifically, CC operates on small, fixed-size blocks, in contrast to Golomb run-length coding, which uses variable-sized blocks of arbitrary length. This simplifies the implementation of CC decoder in hardware, and allows the data stream to be easily subdivided for parallel decoding.

6. EXTENSION TO GREY PIXELS

So far, C4 as described is a binary image compression technique. To extend C4 to encode 5-bit grey-pixel layout image data typically used in maskless lithography applications, with pixel values ranging from 0 to 31, we apply frequency-based binarization twice as described in Appendix A. The first time, a binary image is formed from '0' and not '0'. The

remaining non-binary image is passed again through frequency based binarization. This time, the resulting binary image is '31' and not '31'. Finally the remaining non-binary pixels are coded using standard Huffman coding.

7. COMPRESSION RESULTS

We apply a suite of existing and general lossless compression techniques as well as C4 to binary layout image data. Compression results are listed in Table 3. The original data are 2048 × 2048 binary images with 300 nm pixels sampled from an industry microprocessor layout. Each row of the table represents one such image. The first column "Type" indicates where the sample comes from, memory, control, or a mixture of the two. Memory circuits are typically extremely dense but highly repetitive. In contrast, control circuits are highly irregular, but typically much less dense. The second column "Layer" indicates which layer of the chip the image comes from. Poly and Metal1 layers are typically the densest, and mostly correspond to wire routing and formation of transistors. The remaining columns from left to right are compression ratios achieved by: JBIG, ZIP, 2D-LZ a 2D extension to the LZ77 copying[1], BZIP2 based on the Burrows-Wheeler Transform[10], and C4. The bold numbers indicate the highest compression for each row.

Type	Layer	JBIG	ZIP	2D-LZ	BZIP2	C4
Mem. Cells	M2	59	88	233	260	**332**
	M1	10	48	79	56	**90**
	Poly	12	51	120	83	**141**
Ctrl. Logic	M2	47	22	26	32	**50**
	M1	20	11	11	11	**22**
	Poly	42	19	20	23	**45**
Mixed	M2	51	28	34	39	**52**
	M1	21	12	13	12	**25**
	Poly	41	23	27	28	**47**

Table 3. Compression ratios of JBIG, ZIP, 2D-LZ, BZIP2 and C4 for binary layout image data

As seen, C4 successfully combines both context modeling and copying to outperform all these algorithms for all types of layouts. This is significant, because most layouts contain a heterogeneous mix of these components. It is unusual that ZIP and BZIP2, which are typically thought of as text-based compressors, achieve such high compression ratios on this image data, in particular for memory cells. The success of these algorithms no doubt hinges on the repetitive nature of layout data, which is exploited by copying. In contrast, where the layout becomes less regular, the context modeling of JBIG has an advantage over ZIP, 2D-LZ, and BZIP2.

Layer	Run length	Huffman	LZ77 (256)	LZ77 (1024)	ZIP	BZIP2	C4	C4 *maxdy* = 1
Metal 2	1.4	2.3	4.4	21	25	28	**35**	27
Metal 1	1.0	1.7	2.9	5.0	7.8	11	**15**	10
Poly	1.1	1.6	3.3	4.6	6.6	10	**14**	10
Via	5.0	3.7	10	12	15	24	**32**	27
N	6.7	3.2	13	28	32	42	**52**	45
P	5.7	3.3	16	45	52	72	**80**	76
Dec. Buffer	4 Bytes	256 Bytes	256 Bytes	1.0 kB	4.0 kB	900 kB	656 kB	1.7 kB

Table 4. Compression ratio of run length, Huffman, LZ77, ZIP, BZIP2, and C4 for 5-bit grey layout image data

Table 4 shows compression results for more modern layout image data with 65 nm pixels and 5-bit grey layout image data. Blocks of 1024 × 1024 pixels are sampled across the layout and the *minimum* compression ratio achieved for each algorithm and each layer is reported in rows. The reason for using minimum rather than the average has to do with limited buffering in the actual hardware implementation of maskless lithography writers. Specifically, the compression ratio must be consistent across all portions of the layout as much as possible, so as to achieve a tractable hardware implementation. The last row is the amount of buffer memory used by the decoder of each of the algorithms. The size of the decoder buffer has a direct impact on the decoder's implementation complexity. From left to right, compression ratios are reported in columns for a simple run-length encoder, Huffman encoder, LZ77 with a history buffer length of 256, LZ77 with a history buffer length of 1024, ZIP, BZIP2, C4, and C4 with limited vertical copying. The bold numbers indicate the highest compression for each row. Clearly, C4 is the strongest performer in terms of compression efficiency. Some notable lossless grey-pixel image compression techniques have been excluded from this table including

SPIHT and JPEG-LS. Previous experiments[11] have already shown that they do not perform well as simple ZIP compression on this class of data.

The rightmost column of Table 4 shows compression results for C4, where we have limited the vertical copy operation, described in Section 4, to just one row above the current row. As seen by comparing the last two columns, this restriction significantly reduces decoder buffering to a small 1.7kB as compared to the 656 kB of the full C4 decoder, with some cost to compression efficiency. Nonetheless, this restricted form of C4 achieves compression ratios on par with BZIP2, the highest among non-C4 algorithms tested, using a buffer size comparable to LZ77 and ZIP. The tradeoff between decoder buffer size and compression efficiency for each of these algorithms is explored more fully in Section 8.

8. DECODER BUFFER SIZE

Compression algorithms, in general, require the use of a memory buffer at the decoder to store information about the compressed data. In general, larger buffers allow more information to be stored, which may lead to higher compression ratios. On the other hand, the size of this buffer, which must be implemented in decoder hardware, has a direct impact on implementation complexity. For copy-based compression algorithms, such as C4, LZ77, and ZIP, the buffer stores previously decoded data to copy from. For BZIP2, the buffer stores the block of data over which block-sorting is performed. For simpler schemes such as Huffman coding, the buffer stores symbol statistics and the code table. For the very simple run-length code, the buffer stores the current symbol being repeated, and the number of repetitions remaining.

In C4, most of the decoder buffer is used to support the copy operations described in Section 4. Copying from above requires storing multiple rows of image data at the decoder to enable the copy operation. To allow copies from 1 to *maxdy* pixels above, the decoder needs buffering to store the current row, and *maxdy* rows above, for a total of *maxdy*+1 rows. Clearly, increasing *maxdy* potentially allows more copy regions to be discovered. However each stored row requires *image width* × *bits per pixel* = *bits per row* bits of buffering. Total decoder buffering is *bits per row* × (*maxdy*+1) + *constant* used as overhead to support other aspects of C4. For the 1024 × 1024 image with 5-bit grey pixels used in our experiments, *bits per row* is 5120 bits or 640 bytes, and the constant is 3410 bits or 427 bytes.

We tested C4 with 2, 32, 64, 128, 256, 1024 stored rows, corresponding to total decoder buffer of 1.7 kB, 21 kB, 41 kB, 82 kB, 164 kB, 656 kB, respectively. Results of the tradeoff between decoder buffer and compression ratio for C4, as compared to run-length, Huffman, LZ77, ZIP, and BZIP2, are shown in Figure 5. The x-axis is the decoder buffer size plotted on a logarithmic scale, and the y-axis is the compression ratio achieved on the Poly layer as in Table 4. Each individual curve represents one compression algorithm.

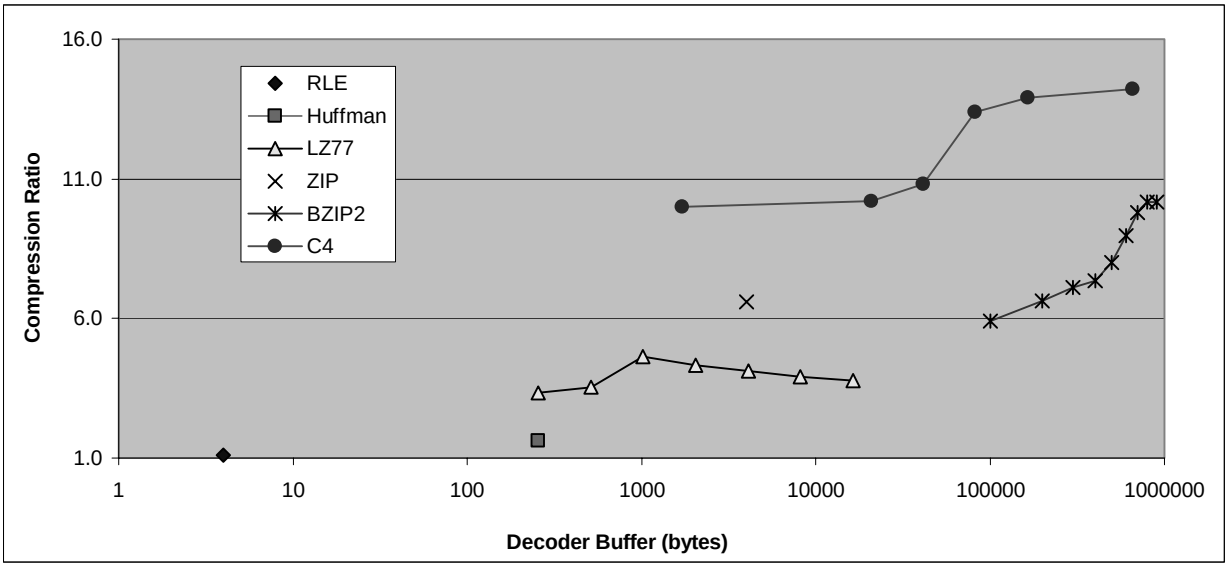


Figure 5. Tradeoff between decoder buffer size and compression ratio for various algorithms on Poly layer

The C4 curve lies above the curves for BZIP2, ZIP, and LZ77, indicating that C4 offers a higher compression ratio for the same decoder buffer size as BZIP2 and LZ77. Note that individual curves in Figure 5 have an S-shape, indicating that there is a specific "sweet-spot" for each algorithm with regards to buffer size and compression efficiency. This point corresponds to a buffer size of 1kB for LZ77, 82kB for C4, and 800kB for BZIP2. Increasing the buffer size beyond this point does not improve the compression efficiency by much. Decreasing the buffer size below this point dramatically reduces compression efficiency. Overall, there is an upward trend to the right, implying that larger decoder buffers result in higher compression efficiency, at a cost of increased decoder implementation complexity.

9. SUMMARY AND FUTURE WORK

In this paper we present a novel compression algorithm called C4, which successfully integrates the advantages of two very disparate compression techniques: context-based modeling and LZ-style copying. This is particularly important in the context of layout image data compression which contains a heterogeneous mix of data: dense repetitive data better suited to LZ-style coding, and less dense structured data, better suited to context based encoding. In addition, it utilizes a novel binary entropy coding technique called combinatorial coding which is simultaneously as efficient as arithmetic coding and as fast as Huffman coding. Compression results show that C4 achieves superior compression results over JBIG, ZIP, BZIP2 and 2D-LZ for lithography image data. Also, decoder buffer size requirements, which impacts decoder implementation complexity, is studied for run-length, Huffman, LZ77, ZIP, BZIP2 and C4. For the same buffer size as LZ77, ZIP and BZIP2, C4 offers better compression. Future work involves keeping data in the C4 decoder buffer in compressed form to further reduce buffer size, mapping the C4 algorithm to hardware, and applying C4 to compress non-lithography images.

APPENDIX A. FREQUENCY BASED BINARIZATION

Frequency based binarization is a method to remove the most frequently occurring symbol in a data stream and replace it binary bitmap of its position. The result is 2 separate data streams, one binary stream representing the bitmap, and one non-binary stream of the remaining symbols. The remaining symbols can be recursively binarized again until the entire stream has been binarized, but typically this is not useful. For example consider this data stream: AAABA ACAAD AAAFA. The most frequent symbol in this case is clearly 'A'. We create two data streams from this sequence as follows: 00010 01001 00010 is a binary stream where '0' denotes the position of 'A' and '1' denotes the position of 'not A'. The residual non-binary stream BCDF describes what the '1' represents. Alternatively, we can preserve the structure of the original stream if desired by denoting instead the residual non-binary stream as XXXBX XCXXD XXXFX, where X denotes 'don't care'. X symbols can be used in modeling, but ignored in coding.

ACKNOWLEDGEMENT

This research is conducted under the Research Network for Advanced Lithography, supported jointly by the funding of the Semiconductor Research Corporation (01-MC-460) and the Defense Advanced Research Project Agency (MDA972-01-1-0021).

REFERENCES

1. V. Dai and A. Zakhor, "Lossless Layout Compression for Maskless Lithography", *Emerging Lithographic Technologies IV*, Elizabeth A. Dobisz, Editor, Proceedings of the SPIE, **3997**, pp. 467-477, 2000.
2. J. Ziv, and A. Lempel, "A universal algorithm for sequential data compression", *IEEE Transactions on Information Theory*, **IT-23** (3), pp. 337-43, 1977.
3. J. Rissanen and G. G. Langdon, "Universal Modeling and Coding", *IEEE Transactions on Information Theory*, **IT-27** (1), pp. 12-23, 1981.
4. CCITT, ITU-T Rec. T.82 & ISO/IEC 11544:1993, Information Technology – Coded Representation of Picture and Audio Information – Progressive Bi-Level Image Comp., 1993.
5. V. Dai and A. Zakhor, "Binary Combinatorial Coding", *Proceedings of the Data Compression Conference 2003*, pp. 420, 2003.
6. T. M. Cover, "Enumerative Source Coding", *IEEE Transactions on Information Theory*, **IT-19** (1), pp. 73-77, 1973.
7. S. W. Golomb, "Run-length Encodings", *IEEE Transactions on Information Theory*, **IT-12** (3), pp. 399-401, 1966.

8. L. Oktem and J. Astola, “Hierarchical enumerative coding of locally stationary binary data”, *Electronics Letters*, **35** (17), pp. 1428-1429, 1999.
9. I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes*, Second Edition, Academic Press, 1999.
10. M. Burrows, and D. J. Wheeler, “A block-sorting lossless data compression algorithm”, Technical report 124, Digital Equipment Corporation, Palo Alto CA, 1994.
11. V. Dai and A. Zakhor, “Lossless Compression Techniques for Maskless Lithography Data”, *Emerging Lithographic Technologies VI*, Elizabeth A. Dobisz, Editor, Proceedings of the SPIE, **4688**, pp. 583-594, 2002.