Complexity Reduction for C4 Compression for Implementation in Maskless Lithography Datapath

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

ABSTRACT

Achieving the throughput of one wafer per minute per layer 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 previous work, we have shown that lossless binary compression plays a key role in the system architecture for such a maskless writing system. Recently, we developed a new compression technique *Context-Copy-Combinatorial-Code (C4)* specifically tailored to lithography data which exceeds the compression efficiency of all other existing techniques including BZIP2, 2D-LZ, and LZ77. The decoder for any chosen compression scheme must be replicated in hardware tens of thousands of times in any practical direct write lithography system utilizing compression. As such, decode implementation complexity has a significant impact on overall complexity. In this paper, we explore the tradeoff between the compression ratio, and decoder buffer size for C4. Specifically, we present a number of techniques to reduce the complexity for C4 compression. First, buffer compression is introduced as a method to reduce decoder buffer size by an order of magnitude without sacrificing compression efficiency. Second, linear prediction is used as a low-complexity alternative to both context-based prediction and binarization. Finally, we allow for copy errors, which improve the compression efficiency of C4 at small buffer sizes. With these techniques in place, for a fixed buffer size, C4 achieves a significantly higher compression ratio than those of existing compression algorithms. We also present a detailed functional block diagram of the C4 decoding algorithm as a first step towards a hardware realization.

Keywords: C4, maskless, complexity, implementation, decoder, prediction, buffer, memory

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 as achieved by today's optical lithography systems. Achieving 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[4], we have proposed a data processing system design capable of delivering tera-pixel data rates as shown in Figure 1. The success of this design is contingent on a data compression algorithm which achieves 25 to 1 compression 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.

In a previous paper[1], we proposed a lossless image compression algorithm called Context Copy Combinatorial Coding (C4), specifically tailored to compress layout images for application to maskless lithography. C4 combines the advantages of LZ-copying, which works well for compressing repetitive circuits such as memory cells, and JBIG context-based modeling, which works well for compressing non-repetitive circuits such as control logic. Consequently, C4 is a single compression technique which performs well for all types of layout: repetitive, non-repetitive, or a heterogeneous mix of both. Compression results show that 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, which is within range of feasibility of our original design.

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

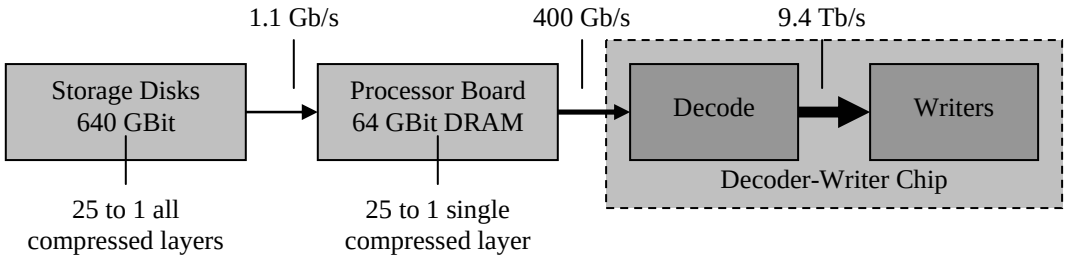


Figure 1. System architecture of a data-delivery system for maskless lithography

For our application to direct-write maskless lithography, the C4 decoding algorithm must also be implemented in hardware as a parallel array of thousands of C4 decoders fabricated on the same integrated-circuit chip as a massively parallel array of writers[3]. To meet this challenge, we present several improvements to C4 which lower its implementation complexity, reduce buffer usage, and improve the compression efficiency of C4 with limited buffering. In addition, we take the first steps towards an ASIC implementation of C4, by breaking down the C4 decoder algorithm into functional blocks, and refining each block, step-by-step.

In Section 2 we present an overview of the binary C4 algorithm. In Section 3, we propose a slight modification to the basic algorithm in order to accommodate copy errors. Although this is a small change, it has a large effect on compression efficiency. In Section 4, we discuss the complexity issues related to extending C4 to compress grey-level pixels through binarization. In Section 5, we introduce linear prediction as a more elegant alternative for extending C4 to grey-pixels, entirely avoiding the implementation issues of binarization. In Section 6, we introduce buffer compression, as a means of reducing decoder buffer usage. In Section 7, we present simulation results to update the compression/decoder buffer tradeoff results reflecting these changes to C4: copy error, linear prediction, and buffer compression. In Section 8, we describe the breakdown of C4 into functional blocks, identifying the most challenging blocks for implementation, and further refining those blocks.

2. OVERVIEW OF BINARY C4

The basic concept underlying C4 compression is to integrate the advantages of two disparate compression techniques: local context-based prediction, and LZ-style copying. This is accomplished by automatic *segmentation* of the image into *copy regions* and *prediction regions*. Each pixel inside a copy region is copied from a pixel preceding it in raster-scan order. Each pixel inside a prediction region, i.e. not contained in any copy region, is predicted from its local context. In the original C4[1], copied pixels are required to be correct, whereas predicted pixels may have *prediction errors*. The positions of these errors are specified by *error bits*, where "0" indicates a correct prediction, and "1" indicates an error. These error bits are then compressed using Hierarchical Combinatorial Coding (HCC), a low-complexity alternative to arithmetic coding, proposed by us in recent years[2]. If the frequency of error "1" is low, then HCC achieves a high compression ratio.

Therefore, the goal of the C4 encoder is to minimize the frequency of error; we have two strategies for accomplishing this, depending on the properties of the layout. First, non-repetitive layout data tend to be sparse in features, so pixels can be accurately predicted[4]. Consequently, the frequency of prediction errors is low, and the compression ratio is high. Second, dense, repetitive layout data can be easily copied, so that most of the layout is covered by copy regions[4]. Since copy regions are error-free, this also lowers the frequency of error, leading to higher compression ratio. Therefore, central to C4 is the automatic segmentation of the layout image into copy and prediction regions. In the past, we have developed a greedy search algorithm[1], to maximize the coverage of error-free copy regions in areas where the number of prediction errors is high.

Figure 2 shows a high-level block diagram of the C4 encoder and decoder for binary layout images. First, a prediction error image is generated from the layout, using a simple 3-pixel context-based prediction model. Next, the "Find copy regions" block uses the error image to do automatic segmentation as described above, generating a *segmentation* map between *copy regions* and the *prediction region*. 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", equivalent to a Boolean XOR operation. These error bits are compressed without loss by the HCC encoder, which are transmitted to the decoder, along with the segmentation map.

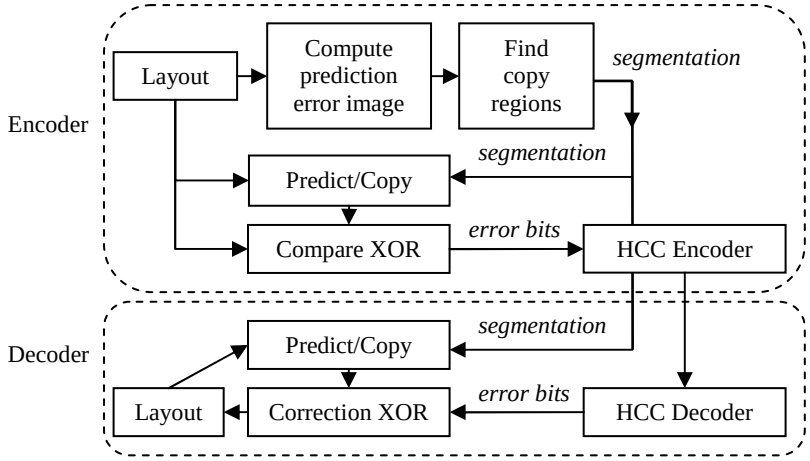


Figure 2. Block diagram of C4 encoder and decoder for binary images

The decoder mirrors the encoder, but skips the complex process necessary to find the segmentation map, which are received from the encoder. The HCC decoder decompresses the error 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 bit is "0", the value is correct, and if the error bit is "1", the value is incorrect and must be inverted, equivalent to a Boolean XOR operation. There is no data modeling performed in the C4 decoder, so it is considerably simpler to implement than the encoder, satisfying one of the requirements of our application domain.

3. EXTENDING C4 TO COPY ERRORS

As previously mentioned, in the original C4[1], copied pixels are required to be correct, whereas predicted pixels may contain errors. In this paper, we modify C4 to allow some copied pixels to be incorrect, resulting in *copy errors*. This dramatically changes the automatic segmentation algorithm described previously, and implemented in the "Find copy regions" block in Figure 2. Specifically, this allows the algorithm to find copy regions which would otherwise have been broken into smaller regions or overlooked. This is particularly beneficial when the search area for copy regions is constrained by limited buffering, resulting in increased compression efficiency. We quantify this effect in our experimental results in Section 7.

Implementing copy errors requires no modification to the high-level block diagram of C4 in Figure 2. Other than the "Find copy region" block, the remaining blocks are largely unchanged. In particular, it has no effect on the structure and complexity of the decoder. Copy errors are handled in exactly the same manner as prediction errors. At the encoder, 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, *whether copied or predicted*, are indicated with a "1", equivalent to a Boolean XOR operation. The resulting error bit stream includes information on both copy and prediction errors, and are compressed by HCC. Similarly, at the decoder, the Predict/Copy block estimates each pixel value, either by copying or by prediction. If the

error bit is “0”, the value is correct, and if the error bit is “1”, the value, *whether copied or predicted*, is incorrect and must be inverted, equivalent to a Boolean XOR operation.

4. BINARIZATION – COMPLEXITY CONCERNS

The original C4 algorithm is a binary image compression technique extended to grey pixels by through binarization[1], as shown in Figure 3. While functional from a compression point of view, binarization introduces the added complexity of a “binarize” block at the encoder, and, more importantly, an added “undo-binarize” block at the decoder. Since implementation complexity of the C4 decoder is critical to our datapath design in Figure 1, we need to understand the impact of the “undo-binarize” block on decoder implementation complexity. To this end, we begin by describing the binarization process.

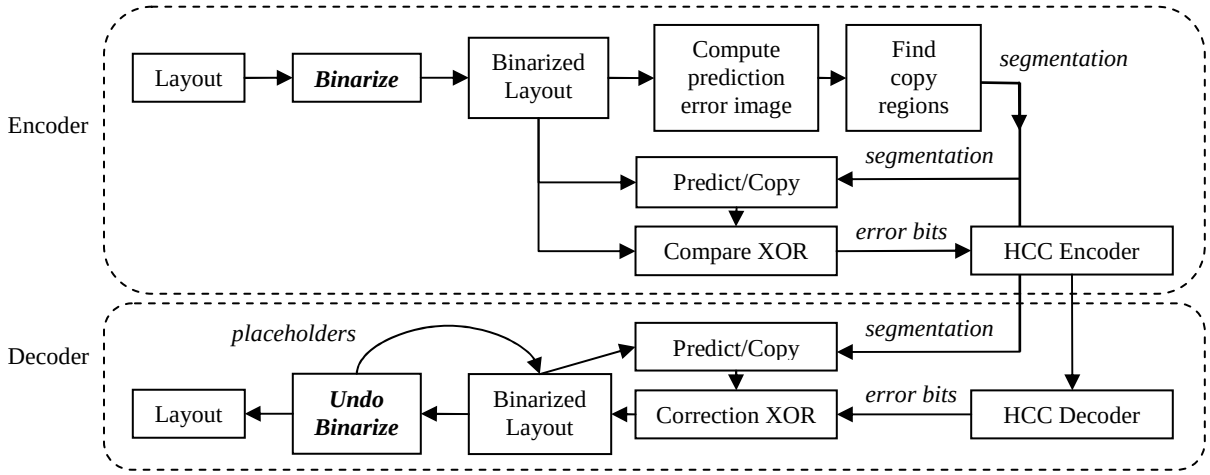


Figure 3. Block diagram of C4 encoder and decoder extended to grey-pixel images through binarization

Figure 4 illustrates the binarization process for an image with 4 grey levels. On the left, the original layout image has 4 grey-levels, ranging from 0 representing pure black, to 3 representing pure white. Each “binarize *L*” step creates a binary image where “0” represents grey-level *L*, and “1” represents all other grey-levels. It also leaves behind a residual image with pixels of the remaining grey-levels. For example, “binarize $L = 0$” creates a binary image where “0” represents level *0*, and a “1” represents *non-0*. The residual leaves only non-0 pixels in the image; all level 0 pixels are removed leaving an empty “.” marker in its place. We continue this process by applying “binarize $L = 3$” and finally “binarize $L = 1$”. As the binarization process continues, the residual image resulting from on “binarize *L*” step is fed into the next “binarize *L´*” step. In each “binarize *L´*” step the empty “.” pixels from the previous step are untouched, and the resulting output is yet another binarized image *L´* and residual image, as shown in Figure 4. As binarization progresses, the resulting binarized and residual images evolves with more empty “.” pixels, until the residual image is entirely made of empty “.” pixels. At this time, the binarization process at the encoder is complete.

The order of binarization is determined by the frequency of occurrence of each grey level in the layout image, beginning with the most frequent, and ending with the least frequent. In most layouts with *N* grey levels, the frequency of black level *0* is the highest, and therefore, binarized first. This is followed by white level *N-1*, and the remaining grey levels in random order, depending on the statistics of the layout.

Each of the binarized images, encircled by dashed lines in Figure 4, is separately compressed by a binary C4 encoder, and decompressed by the binary C4 decoder. Empty “.” pixels act as placeholders, from a conceptual point of view, in order to properly structure the binarized images during binarization and de-binarization. For instance, “binarized image L = 3” is represented as “1000”. Without placeholders, the C4 decoder does not know to place these bits in the lower

right hand corner of the image. The placeholders, however, are not transmitted from the encoder to the decoder. This is because the first "binarized image $L = 0$" is guaranteed to contain no placeholders, and from that image, subsequent placeholders can be regenerated by the "Undo Binarize" block. Thus, the de-binarization order of the grey levels is the same as the binarization order.

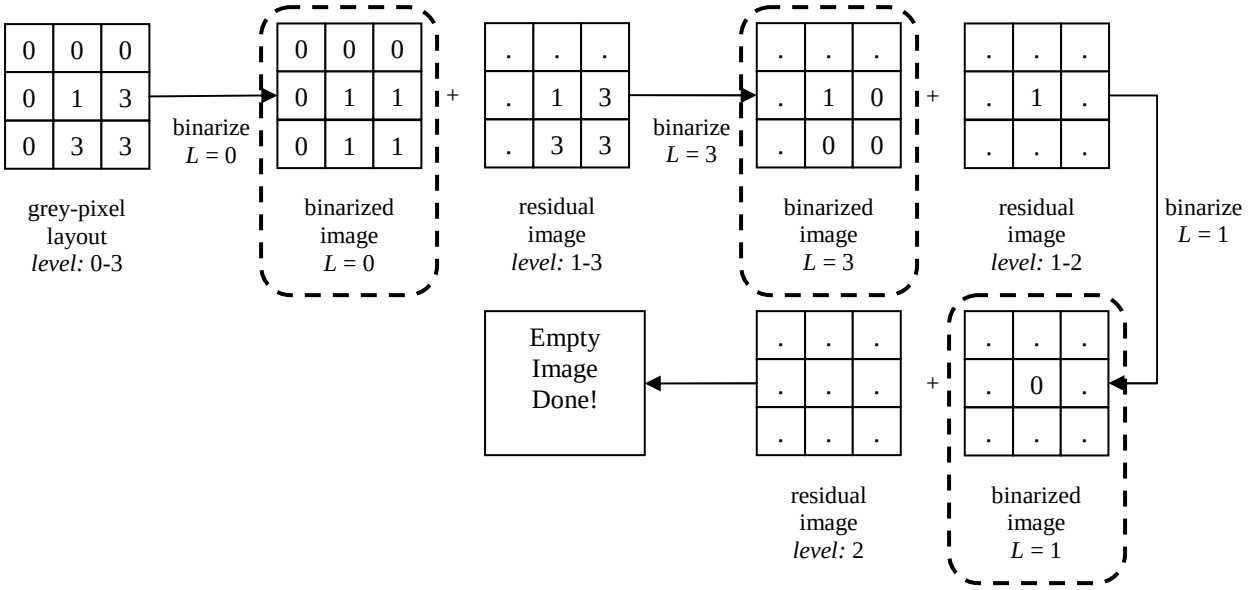


Figure 4. Binarization of a grey-pixel image

To undo the binarization at the C4 decoder, we begin to reconstruct the original grey-pixel layout one grey level at a time from the sequence of binarized images. The process is shown in Figure 5. Each "binarized image *L*" partially fills the grey-pixel layout with pixels of level *L*. Then, the placeholder map necessary to reconstruct the next binarized image is created by inverting the partially-filled grey-pixel layout; that is, filled pixels with values become an empty "." placeholder, and empty pixels are given a dummy value "0". For example, first, the binary C4 decoder decompresses "Binarized image $L = 0$"; no placeholders are needed for this step. Next, "0" in the binarized image is reconstructed as level 0, and "1" is left empty ".", forming "grey-pixel layout level 0". Next, we invert "grey-pixel layout level 0" to produce placeholders. Next, using these placeholders, the binary C4 decoder decompresses "binarized image $L = 3$". Next, each "0" in the binarized image is replaced by a level 3 in the grey-pixel layout, forming "grey-pixel layout *level*: 0,3". Again, we invert the grey-pixel layout to produce placeholders. Using these placeholders, the binary C4 decoder decompresses "binarized image $L = 1$". Finally, we fill in the last remaining pixel in the grey-pixel layout by replacing the "0" in the binarized image, with a level 1 grey pixel. Thus, the same order in which binarized images are generated at the encoder is used in the decoder to undo binarization.

The main complexity of "Undo-binarize" has to do with maintaining a partially filled, grey-pixel layout, which is reconstructed using multiple passes, in an unpredictable order depending upon the structure of the layout. Performing "Undo-binarize" in this fashion requires the use of a memory buffer capable of storing the entire block of layout being de-binarized, which for our test layout images, is 1024 × 1024 × 5-bits or 5 Mbits of buffer. In fact, the size of this de-binarization buffer is larger than the C4 decoder buffer used for copying, as analyzed in[1]. While a buffer of this size is insignificant for a desktop computer, it becomes prohibitive for a large array of decoders to be fabricated on the same substrate as the writers.

It is conceivable to possible subdivide the entire image into smaller sub-blocks in order to reduce buffer size in "undo-binarize". The binary C4 algorithm, however, would still need to be applied to the entire image to preserve compression efficiency[1]. Consequently, after each "binarized image" sub-block is decompressed, the binary C4 decoder is left in a

partially decoded state which must be preserved in memory, one state for each binarization level. In addition, state-swapping creates latency in the decoder, leading to slower decoding speeds, as well as the additional control circuitry needed to implement it. We believe that linear prediction, presented in the next section, is a more elegant alternative to binarization for extending binary C4 to grey-pixels; it also results in lower implementation complexity.

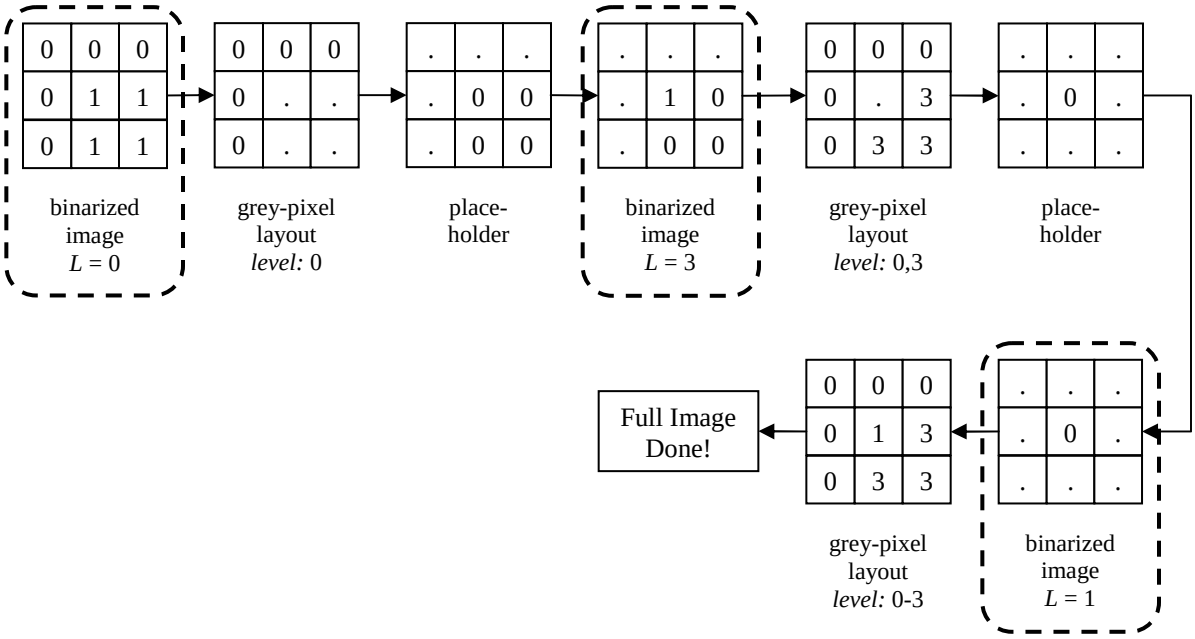


Figure 5. "Undo-binarize" – reconstructing the grey-pixel layout from binarized images

5. LINEAR PREDICTION

To extend C4 to encode 5-bit grey-pixel layout image through linear prediction, we modify the prediction mechanism, and the representation of the error, as compared to original binary C4 encoding and decoding[1]. Specifically, the local 3-pixel binary context based prediction mentioned in Section 3, is replaced by 3-pixel grey linear prediction with saturation, to be described later; furthermore, in places of prediction or copy error, where "1" is used to denote an error bit, we use a grey-level *error value* to indicate the correct value of that pixel. The resulting algorithm is called C4 with Linear Prediction, denoted by C4+LP, and the block diagram of the C4+LP encoder and decoder is shown in Figure 6.

First, a prediction error image is generated from the layout, using a simple 3-pixel linear prediction model. The error image is a binary image, where "0" denotes a correctly predicted grey-pixel value and "1" denotes a prediction error. The copy regions are found as before in binary C4, with no change in the algorithm. As specified by the copy regions, the Predict/Copy generates pixel values either using copying or linear prediction. The result is compared to the actual value in the layout image. Correctly predicted or copied pixels are represented by a "0", and incorrectly predicted or copied pixels are represented by a "1", together with a grey-level error value indicating the true value of the pixel. We call the stream of "0" and "1", indicating the location of incorrectly predicted or copied pixels, *error bits*. This binary stream is compressed by the HCC encoder. The grey-level error values are compressed by a Huffman encoder.

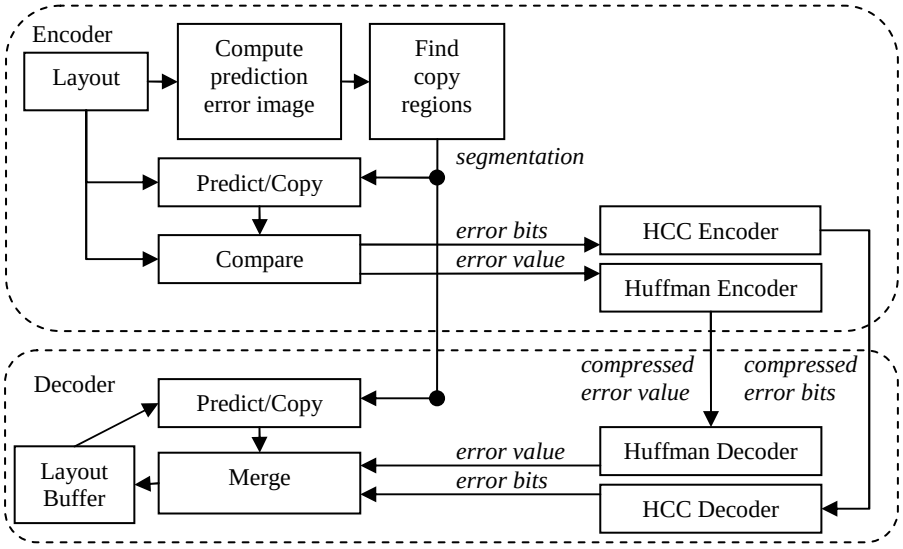


Figure 6. Block diagram of C4+LP encoder and decoder for grey-pixel images

As with binary C4, the grey-pixel C4 decoder mirrors the encoder, but skips the complex steps necessary to find the copy regions. The Predict/Copy block generates pixel values either using copying or linear prediction according to the copy regions. The HCC decoder decompresses the error bits, and the Huffman decoder decompresses the error values. If the error bit is "0" the prediction or copy is correct, and if the error bit is "1" the prediction or copy is incorrect and the actual pixel value is the error value.

Our proposed linear prediction mechanism for grey-pixel C4 is analogous to the context-based prediction used in binary C4. Each pixel is predicted from its 3-pixel neighborhood as shown in Figure 7. Pixel *z* is predicted as a linear combination of its local 3-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 $z = c$, continuing 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, there should be an equivalent step up from *c* to *z*, such that $c + d = z$, continuing that vertical edge. Likewise, if there is a step up from *a* to *c*, such that $a + d = c$, as in a horizontal edge, there should be an equivalent step up from *b* to *z*, such that $b + d = z$, continuing that horizontal edge. Thus, these equations predict horizontal and vertical edges, as well as 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. It is also similar to the median predictor used in JPEG-LS[13].

a	*b*
c	z

x = b – a + c
if (x < 0) then z = 0
if (x > max) then z = max
otherwise z = x

Figure 7. 3-pixel linear prediction with saturation used in grey-pixel C4

From the algorithm in Figure 7, it is also clear that the implementation of linear prediction is simple, involving 5-bit addition, subtraction, and overflow detection for 5-bit grey-level pixels. Linear prediction enables us to entirely avoid the added complexity of "Undo-binarize" and its associated buffering or state-swapping requirements, presented in the previous section. The compression ratios achieved by binarization and linear prediction are nearly identical, and in fact,

does not affect the compression results we report in Section 7. Consequently, linear prediction is our method of choice for grey-pixel C4 compression.

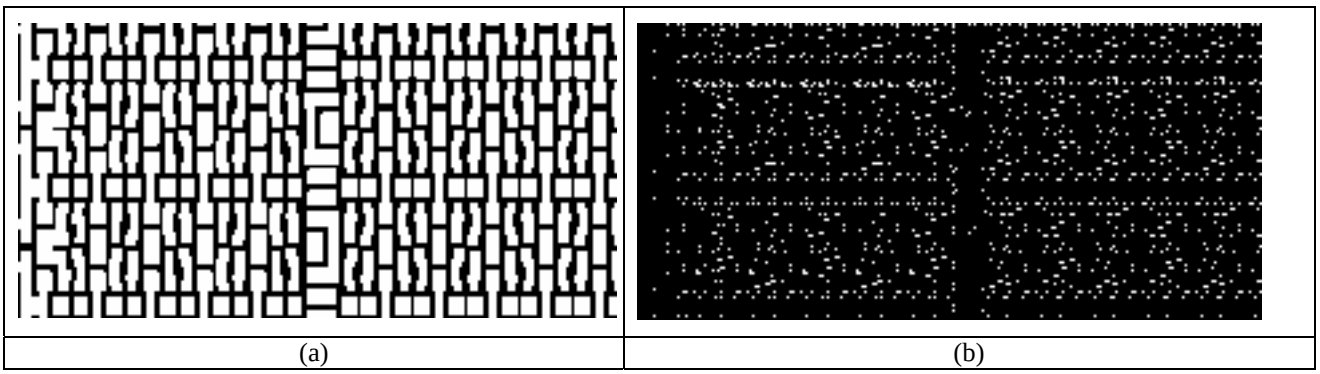
6. BUFFER COMPRESSION

Among the various blocks of the C4+LP decoder in Figure 6, the Layout Buffer, which supplies data for Predict/Copy, remains the largest single block of memory in the decoder, and, most likely, the largest single block in physical size as well. The size of this buffer ranges from 10kbits to 5Mbits, depending on the number of stored rows of the layout used to support copy operations[1]. Buffer compression addresses this issue by storing the incoming, already compressed, data from the encoder in a buffer, so that more rows may be stored using less memory. We can then reuse this data by copying from the compressed domain as necessary. The resulting algorithm, C4 with Buffer Compression and Linear Prediction (C4+BC+LP), represents a significant departure from C4+LP algorithm described in Section 5, although the basic premise of combining the benefit of copying and context-based prediction remains the same. We describe C4+BC+LP in detail in the remainder of this section.

As in C4+LP, C4+BC+LP begins by computing the prediction error image of the layout. Recall from Section 5, each pixel of the layout image is linearly predicted from its neighbors, and results in a prediction (*error bit*, *error value*) pair, where (0, *no value*) indicates a correct prediction, and (1, *error value*) indicates a prediction error. Here, we name this (*error bit*, *error value*) pair, an *error pixel*, and together all the error pixels form a prediction error image.

In C4+BC+LP, we find copy regions not on the original layout image, but instead, on this prediction error image. We interpret these copy regions as follows: each error pixel inside a copy region is copied from an error pixel preceding it in raster order; each error pixel outside a copy region is compressed and *transmitted* from encoder to decoder, where the error bits are compressed using HCC, and error values are compressed using Huffman coding. Therefore, the result is not a *segmentation* on the layout image, between prediction and copy regions; rather, all pixels in the layout image are predicted. Specifically, it is a *segmentation* on the prediction error image, between *transmit* and copy regions, where the transmit region is composed of all error pixels not contained in a copy region.

This method of copying in the prediction error domain, as described in BC, performs because there is a close relationship between the layout image, and the prediction error image. In particular, dense repetitive layout images, as seen in Figure 8a, also have dense repetitive prediction error images, as seen in Figure 8b. In fact, the correspondence is so close, that copy regions found on the prediction error image are very similar to copy regions found on the original image, with the exception that copy errors, as described in Section 3, are not allowed in C4+BC+LP. We have chosen to omit copy errors within BC by design, as it requires an additional compressed copy error stream from encoder to decoder. This additional stream adds complexity to the encoder and decoder, unlike copy errors in Section 3 which requires no change to the decoder structure. This is in contradiction with our stated goal for buffer compression, which is to reduce implementation complexity of the decoder. Nonetheless, from a conceptual point of view, integrating copy errors into the C4+BC+LP framework remains a possibility for future research.



(a) (b)

Figure 8. Dense repetitive layout image data and its resulting prediction error image

Figure 9 is a block diagram of the C4+BC+LP encoder and decoder. At the encoder, first the prediction error image is computed from the layout using linear prediction. Next Find Copy Regions block automatically segments the prediction error image as described above, generating a segmentation map between copy regions and transmit regions. In the Predict/Compare block, for each pixel inside the transmit region, we compute a prediction error pixel, i.e. an (error bit, error value) pair. For each pixel inside a copy region, we skip this step: these error pixels are generated via copying at the decoder. Error bits and error values from error pixels inside transmit regions are compressed by the HCC encoder and Huffman encoder, respectively. The compressed error bits and error values are transmitted to the decoder, along with the segmentation information.

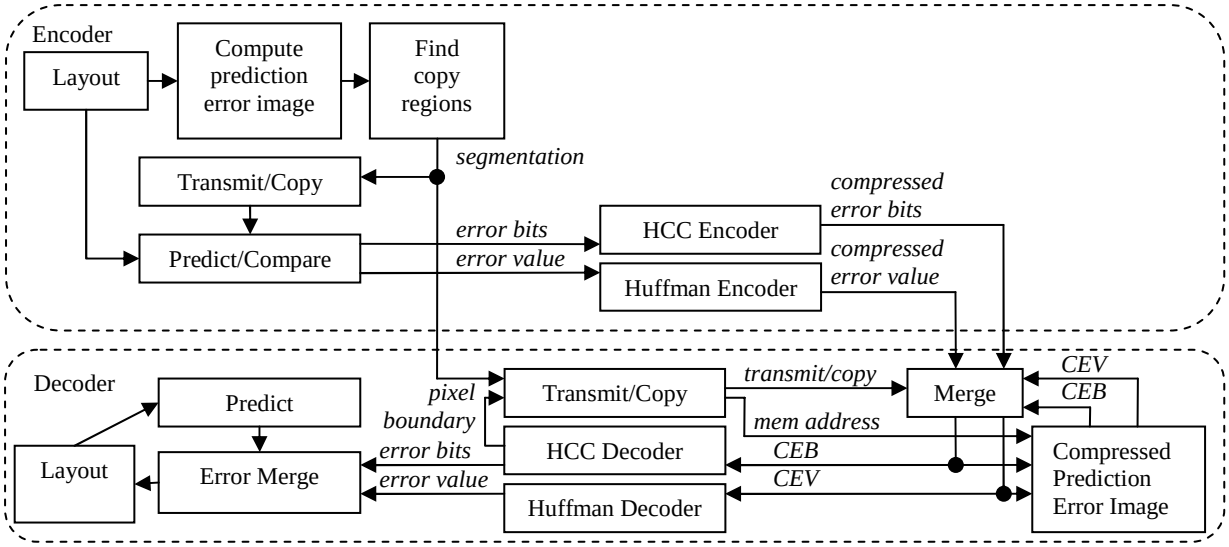


Figure 9. Block diagram of C4+BC+LP encoder and decoder. CEB and CEV stand for compressed error bits and compressed error values, respectively.

At the decoder, the Transmit/Copy decodes the segmentation into control signals to determine whether the current pixel is inside a transmit region or a copy region. If the current pixel is inside a transmit region then Merge blocks accepts compressed error bits (CEB) and compressed error values (CEB) from the encoder, but if the current pixel is inside a copy region, the Merge block accepts *CEB* and *CEV* from the buffer which stores a compressed version of the prediction error image. The Transmit/Copy block is responsible for generating the correct memory address to read from the compressed prediction error image buffer based on the segmentation. At the output of the Merge block, CEB are decoded by the HCC decoder, and CEV are decoded by the Huffman decoder. In addition, CEB and CEV are also stored in the compressed prediction error image buffer, for reuse in future copy regions. The HCC decoder must also supply pixel boundary information to the Transmit/Copy block. Due to compression, the number of bits needed to represent each error pixel varies with the compression ratio. Consequently, only the HCC decoder can determine where pixel boundaries are in the compressed data stream.

Once the error bits and error values are decoded, the Predict block estimates each pixel value based on its context using linear prediction. If the error bit is “0”, the predicted value is correct and is the layout pixel value. If the error bit is “1”, then the predicted value is incorrect, and the error value is the layout pixel value. Note that in contrast to C4+LP, there is no copying after HCC and Huffman coding.

5.1. Latency, compression efficiency, and buffer size reduction of C4+BC+LP

At the heart of the C4+BC+LP decoder is the Transmit/Copy block which must decode segmentation of the prediction error image into transmit/copy signals for merging CEB and CEV. In addition it also manages the memory used to store the compressed prediction error image for copying. This task is made even more complex by compression, because each

error pixel is represented by a variable number of bits. If the data is uncompressed, to find a pixel *d* rows above to copy from, the offset in memory is *d* × *row width* × *bits per pixel* bits. However, because *bits per pixel* varies in the compressed domain, finding that exact pixel boundary, requires searching sequentially through the compressed prediction image memory via the HCC decoder. To aid in this search, the Transmit/Copy maintains a list of memory pointers corresponding to the beginning of each compressed prediction image row. Consequently, the maximum latency cost to find the right memory position to copy from is, in the worst case, the width of an image row. This search must be performed once for each row of a copy region, for all copy regions. So the maximum worst-case total latency is *image width* × *number of copy regions* × *average height of copy regions.*

This latency is significant, though worst-case analysis is likely to be overly pessimistic. For example, it is possible to reduce the latency by enabling partial HCC decoding. Using partial HCC decoding, it would then be possible to fast-forward through the compressed prediction image, block-by-block, rather than pixel-by-pixel. This would speed up the search algorithm by a constant factor relative to the HCC block size. Another possibility is to limit the number of copy regions. Doing so reduces compression efficiency, but it may be a worthwhile tradeoff to reduce the latency of C4+BC+LP.

In addition to latency, BC also incurs some compression inefficiency when compared to C4 compression without BC. As previously mentioned, C4+BC+LP does not allow for copy errors and, as such, copy regions defined on the prediction error image are not exactly the same as the copy regions defined on the layout image. These two effects combined reduce the compression benefit of copying, leading to reduced compression efficiency for C4+BC+LP, as compared to C4 with copy error and linear prediction (C4+CE+LP). Nonetheless, the reduced compression efficiency of C4+BC+LP is a worthwhile tradeoff, when compared to the reduction in complexity offered by the smaller decoder buffer. This tradeoff is quantified in the results section.

Finally, although C4+BC+LP benefits from a smaller buffer for copying from the compressed prediction error image, it incurs an overhead of an extra Predict buffer used to store past layout pixel values for linear prediction. This buffer does not exist in previous versions of C4, because linear prediction shares the same layout image buffer as that used by copy. The size of this Predict buffer is 10kbits, corresponding to 2 rows of a 5-bit grey pixel image, with 1024-pixels per row. This overhead becomes significant when the total buffer size of C4+BC+LP is restricted to less than 80kbits or 10kBytes. This effect is also quantified in the next section.

7. SIMULATION RESULTS

Throughout this paper, we have made various changes to the original C4 algorithm[1] proposed in 2004. In Section 3, we allowed for copy errors (CE) to increase the size copy regions and improve compression efficiency. In Sections 4 and 5, we motivated the change from binarization, as a means of extending C4 to grey-pixels, to the more elegant linear prediction (LP), which though significantly less complex, does not significantly alter the compression efficiency of C4. In Section 6, we discussed buffer compression as a method of significantly reducing buffer usage by copying in the compressed prediction error domain. In this section we update the tradeoff curves from our earlier paper[1], between decoder buffer size and compression ratio for various algorithms, including run-length (RLE), Huffman, LZ77, ZIP, BZIP2, and the original version of C4 developed in 2004, which we refer to as C4 SPIE 2004[1]. In addition, we include 2 new curves, C4 with copy error and linear prediction, denoted by C4+CE+LP, and C4 with buffer compression and linear prediction, denoted by C4+BC+LP. Results are shown in Figure 10.

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 on modern layout image data with 65 nm pixels and 5-bit grey levels. 5 blocks of 1024 × 1024 pixels are sampled by hand from 2 different layouts, 3 from the first layout, and 2 from the second, and the *minimum* compression ratio achieved for each algorithm on the Poly layer is plotted. The reason for using minimum rather than the average is that in a practical system, the minimum compression ration over a layout determines the speed at which the writers operate. Otherwise, we would need to add yet another large buffer to absorb compression ratio variability, so as to smooth out the data rate. Nonetheless, we have empirically determined that our main conclusion remains the same, if we

were to use average compression ratio, or if we were to compare these compression schemes for each of the 5 blocks individually.

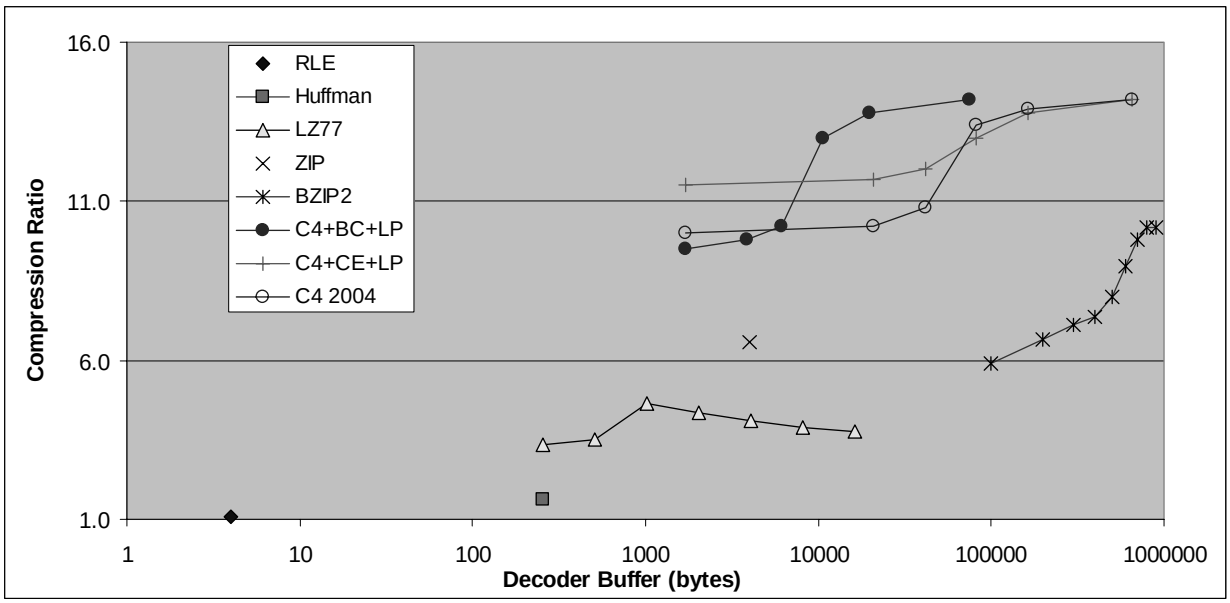


Figure 10.Tradeoff between decoder buffer size and compression ratio for various algorithms on Poly layer

Each individual curve represents one compression algorithm. C4 curves are generated by varying the number of stored rows available for copying from above. We have 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 of uncompressed buffer. Naturally, these numbers are smaller when buffer compression is enabled in the C4+BC+LP curve. The buffer size for C4+BC+LP includes the smaller buffer necessary to support linear prediction.

Comparing the three C4 curves, the highest compression ratio, corresponding to 1024 stored rows, is nearly identical. However C4+LP+BC achieves this ratio using 10 times less buffering than C4 without BC. In contrast, comparing the three C4 curves at the smallest buffer size, corresponding to 2 stored rows, all three C4 versions use about 2kB of buffer, but C4+CE+LP has the highest compression at 11.5, followed by C4 SPIE 2004 at 10.0, and C4+BC+LP at 9.5. In addition, the C4+CE+LP is also much simpler to implement as it replaces binarization with linear prediction. This reduction in complexity is not reflected by the tradeoff curves show in Figure 10, which only consider decoder buffering as a measure of complexity.

Copy errors result in the most benefit when the copy region search area is limited. In this case, copy errors allow for copy regions to be found, which would otherwise be broken up into several smaller regions, or simply go undetected. In contrast, at smaller buffer sizes corresponding to 2, 32, or 64 stored rows, C4+LP+BC has slightly lower compression ratio than C4 SPIE 2004, yet provides no reduction in buffer size, because the buffering needed for LP dominates. Based on these results, the optimal strategy is to use C4+LP+CE with 2 stored rows if buffering is limited to a few kilobytes.

8. BLOCK DIAGRAM OF THE C4 DECODER

To use C4 compression in a maskless lithography datapath shown in Figure 1, the C4 decoder must ultimately be implemented as ASIC circuitry built on the same substrate as the array of writers[4]. The first step in this endeavor is to break down the C4 decoder algorithm into functional blocks. Subsequently, each block needs to be refined, step-by-

step, down to the gate level, and finally the transistor level. In this section, we consider the first steps of breaking C4 down to functional blocks, and refining the blocks, in particular, the region decoder.

We begin with the high level block diagram of the C4 decoder with copy error and linear prediction (C4+CE+LP), as shown in Figure 6. The reason for choosing C4+CE+LP is two-fold: first, it has the highest compression efficiency on test data when the buffer size is small, and second, unlike C4+BC+LP, it has no latency issue to be resolved.

In Figure 11, we have slightly rearranged Figure 6, to emphasize the inputs and outputs. The C4 decoder has 3 main inputs, a segmentation, a compressed error location map, and a stream of compressed error values. The segmentation is a list of copy regions, which indicate whether each pixel is copied or predicted. The compressed error location map is a compressed stream of bits, decompressed by the HCC Decoder block. Each bit in this stream controls whether the Merge block accepts the Predict/Copy value, or the error value input. The error value stream is generated by Huffman decoding the compressed error value input. If the error location bit is *0*, the Predict/Copy value is correct and is sent as output to the writers. If the error location bit is 1, the Predict/Copy value is incorrect, and the *error value* is sent as output to the writers. In addition, the output to the writers is stored in a Layout Buffer, to be used in the future for prediction or copying.

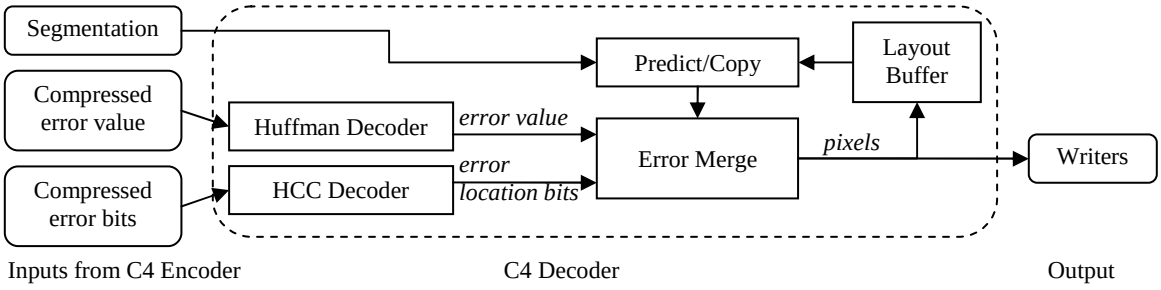


Figure 11. Block diagram of the C4 decoder for grey-pixel images using linear prediction

Of the 5 blocks in the C4, only one is discussed in detail in this section, namely the Predict/Copy block to be covered in subsection 5.1. The implementation of the remaining 3 blocks are either fairly straightforward, or may be extracted directly from existing work in the literature[14]. The function of the Merge block is essentially that of a multiplexer. The Layout Buffer may be implemented using random access on-chip memory, such as multi-ported SRAM, with 5-bit I/O. Other implementation possibilities for this memory, such as a systolic array[14], are beyond the scope of this paper. We have discussed the Huffman decoder implementation in detail, in a previous paper[14], even though we note that the symbol table for this 5-bit Huffman decoder for 5-bit pixel values, is about 8 times smaller than that of the 8-bit Huffman decoder presented earlier[14]. This corresponds to a considerable complexity reduction. The HCC decoder likewise is composed of a 5-bit Huffman decoder, a uniform decoder, and a combinatorial decoder, all three of which are straightforward to implement from algorithms described previously using a simple adders and memory.

8.1 Predict/Copy Block

The Predict/Copy block receives the segmentation information from the C4 encoder and generates either a copied pixel value, or predicted pixel value based on this segmentation. Data needed to perform the copy or prediction is drawn from the Layout Buffer. In Figure 12, we refine the Predict/Copy block into 4 smaller blocks: Region Decoder, Predict, Copy, and Merge.

The Region Decoder decodes the *segmentation* input into control signals for the other blocks. Specifically, it provides the copy parameters (*dir*, *d*) to Copy block, and *predict/copy* signal to the Merge block. The parameter *dir*, short for direction, indicates whether to copy from the *left* or from *above*, and the parameter *d* indicates the distance in pixels to copy from. Together, (*dir*, *d*) are used by the Copy Block to compute the memory address from which to read Layout Buffer, using a shifter and adder. The Predict block performs linear prediction as described previously in Section 5,

using a pair of adders with overflow detection. The *predict/copy* signal selects which input is accepted by the Merge block, a basic multiplexer, Predict or Copy, and generates an output accordingly. The output leaving Predict/Copy goes through another Merge block, which chooses between the Predict/Copy output and the error value, based on the error bit. Copy, Predict, and Merge are simple circuits commonly used in digital design.

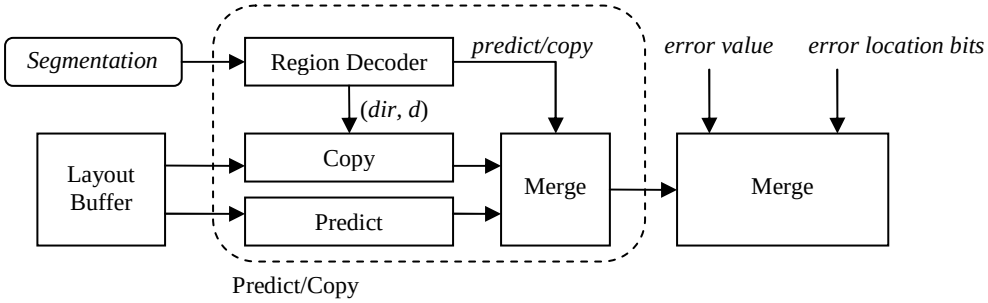


Figure 12. Refinement of the Predict/Copy block into 4 sub-blocks: Region Decoder, Predict, Copy, and Merge.

In contrast, the Region Decoder is considerably more complex than the other 3 blocks. The *segmentation* input is a list of copy regions, as described our previous paper[1]. From an implementation point of view, each copy region can be thought of as a rectangle in an image at location (*x*,*y*), height of *h*, width of *w*, and "color" of (*dir*, *d*). All areas outside the list of rectangles have a "color" *predict*. Figure 13 illustrates this visual interpretation of copy regions as colored rectangles. Using this interpretation, for each pixel, the output of the region decoder can be thought of as the "color" of that pixel in the set {*predict*, (dir_1, d_1), (dir_2, d_2), … }. The problem of region decoding can, therefore, be interpreted as a problem of rasterizing non-overlapping colored rectangles.

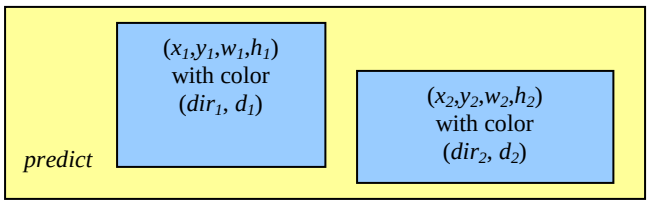


Figure 13. Illustration of copy regions as colored rectangles

8.2. Rasterizing Rectangles

An efficient algorithm for rasterizing Mahattan-style rectangles is well-known in the computational geometry literature as part of a family of plane-sweep algorithms[15]. We discuss the algorithm here only to explain the data structures needed to implement Region Decoder in hardware. Plane-sweep works by using a horizontal sweepline that moves from the top to the bottom of the image, as illustrated in Figure 14. We call rectangles intersecting the sweepline *active regions*. Information on active regions is maintained in an *active region list*. The list is ordered by *x*-position of each active region from left to right. For example, in $sweepline_1$, there is one rectangle in the active region list.

Traversing from left to right within a sweepline, there are a series of segments of constant color, alternating between *predict* and (*dir*, *d*). Therefore, decoding a *sweepline* into pixels becomes a simple matter of outputting a series of (*color*, *width*) pairs to a run-length decoder where color alternates between *predict* and (*dir*, *d*). This can be done by simply traversing the active region list, in order from left to right and outputting the (*x*, *w*, *dir*, *d*) for each active region. For example, in $sweepline_1$, there is one active region. From its (*x*, *w*, *dir*, *d*) we can compute the following: the first segment is (*predict*, $x - 0$), the next segment is ((*dir*, *d*), *w*), and the last segment is (*predict*, $1024 - (x + w)$). In general, the predict segment between two active regions *i* and *i*+1 is (*predict*, $x_{i+1} - (x_i + w_i)$), and the segment corresponding to active region *i* is ((dir_i, d_i), w_i). Next, we consider how to advance the sweepline row-by-row.

As the sweepline advances from *sweepline*$_1$ to *sweepline*$_2$, there is no change in the active region list, but the *residual height* of the active region, defined as the distance from the sweepline to the bottom of the active region, decreases. When the sweepline advances to *sweepline*$_3$, a new region becomes active which must be inserted into the list of active regions in the correct order. When the sweepline advance to *sweepline*$_4$, the residual height of the left active region is reduced to zero, so that region now becomes inactive, and must be removed from active region list.

Therefore each row the sweepline advances, 4 steps must be taken. First the residual height of all active regions is reduced by one. If any residual height is reduced to zero, that active region is deleted from the region list. Second, we must check for new regions becoming active. This is done by comparing the *y*-position of a new region streaming in from the *segmentation* list, against the *y*-position of the sweepline. If it matches, then the new region must be inserted in the proper position in the active region list based on its *x*-position. Third, we traverse through the (*x*, *w*, *dir*, *d*) of each active region to compute the sequence of colored segments, as described above. Fourth, the sequence of colored segments are converted to a "color" value per image pixel, *predict* or (*dir*, *d*). This "color" is the output of the Region Decoder.

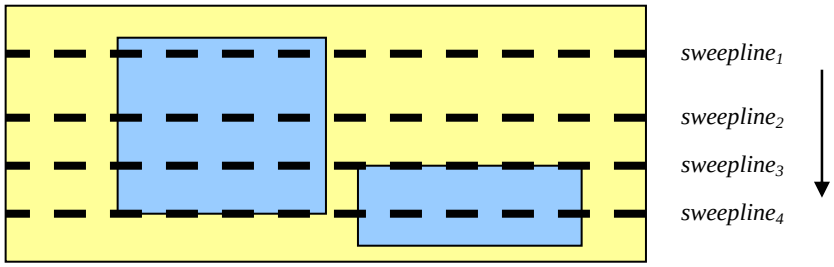


Figure 14. Illustration of the plane-sweep algorithm for active regions

8.3 Data Structures

The data structure necessary to implement the active region list on a computer is a sorted linked-list[16]. Each record in the list represents one active region: (*x*, *w*, residual height *h*, copy parameters (*dir*, *d*)). These records are not sequentially ordered in memory. Instead, each record contains a *link* pointer which points to the memory position of the next record. Stepping through the list of active regions involves following the chain of *link*s. Inserting a record into the list involves, stepping through the list to the correct position, allocating space for the record in unused or *free* memory, and then fixing up the *link* pointers to add the record to the chain. Deleting a record involves fixing up the *link*s to remove the record from the chain, and then marking the memory position of the deleted record as *free*. On a computer, unused memory is called a heap, which is managed by the operating system (OS). Of course, there is no room for an OS our decoder circuit, so the memory management must be mimicked in hardware.

8.4 Subblocks in the Region Decoder

Figure 15 is a further refinement of the Region Decoder block into sub-blocks for implementing the sweepline algorithm. From top to bottom, the Control block on top controls all the other blocks, and executes the 3 steps necessary to advance the sweepline: *Trim()* decrements residual height of each active region, and deletes regions with a residual height of 0; *Add()* inserts new region incoming from the *Segmentation* stream into the active region list when appropriate; and *Output()* traverses the active region list, outputting (*x*, *w*, *dir*, *d*) for each region, which is then translated into (*color*, *w*) segments, and rasterized. The Input New Region block helps execute Add() by maintaining the *y*-position of the sweepline, and comparing it to the *y*-position of the incoming copy region in the *Segmentation* stream. If it matches, then it further checks that the *x*-position of the incoming copy region is correct for insertion to the active region list. The Active Region Memory stores the (*x*, *w*, *h*, *dir*, *d*) of each active region in the list. The Raster Segment block helps execute Output() by converting an input sequence of (*x*, *w*, *dir*, *d*) into a sequence of (*color*, *width*) segments as described previously. It also includes a run-length decoder to rasterize these segments to generate the output (*predict/copy*, *dir*, *d*) for the region decoder. The Linked List block controls all operations relating to the linked

list, including stepping through the list Step(), inserting to the list Insert(), and deleting from the list Delete(). It directly controls the *link* Memory which stores the *link* information for each record. It also keeps track of the *current* pointer, which is the address in the Active Region Memory that is being written to, or read from. It also corresponds to the memory address where a record is being inserted or deleted. Finally, the Free Stack block mimics the behavior of the heap by keeping track of which memory addresses are unused or *free* using a stack[16]. When a new active region is being inserted, *Alloc()* provides the pointer *new*, to an unused memory position in Active Region Memory. When an active region is being deleted, *Free()* marks the *current* pointer as free. The FS Memory block provides the memory to support Free Stack operations.

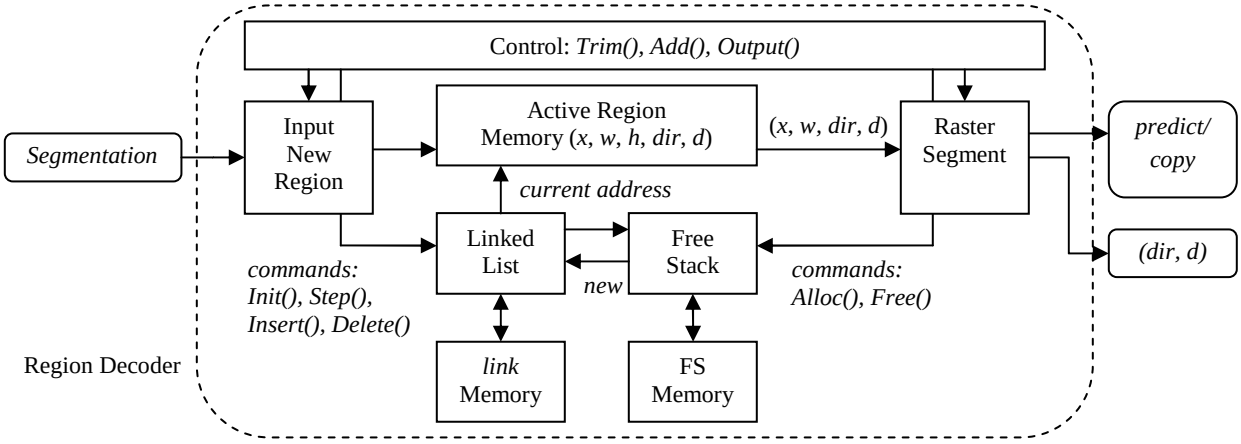


Figure 15. Refinement of the Region Decoder into sub-blocks.

Overall the Region Decoder involves simple hardware structures, such as registers, memory, adders and comparators, controlled by a cascade of small state-machines, each executing short control commands. Total memory usage is on the order of a few kilobits. The next step in the future towards a physical implementation of the Region Decoder is to implement the state machine controllers for each of these blocks, along with the hardware structures on a FPGA. Although the overall behavior is fairly complex, the physical size of the circuit implementation of the Region Decoder should be fairly modest.

9. SUMMARY AND FUTURE WORK

We have presented several improvements to our earlier developed compression techinque *Context-Copy-Combinatorial-Code (C4)*[1]. Specifically, we introduced the idea of allowing for copy errors, which has the effect of discovering copy regions which would otherwise be broken into several smaller regions or ignored altogether. CE improves the compression efficiency of C4 when available buffer size in hardware is limited, resulting in a restricted copy region search area. We introduced the idea of linear prediction to replace binarization. Not only is LP more elegant and less complex to implement than binarization, but it also results in no loss in compression efficiency. We introduced the idea of buffer compression, which involves buffering data for copying at the compressed input of the C4 decoder, and copying in the compressed prediction error domain. BC reduces the buffering requirements of C4 by a large factor, nearly 10 for the test Poly data, but only when the number of stored rows is large. BE and CE are mutually exclusive techniques, but they compliment each other in that they benefit different regions of the compression ratio vs. decoder buffer size tradeoff curve for C4.

The question remains how applicable these compression ratios are to general layout data beyond the test set. To answer this question conclusively, a significantly larger study is needed, over a wide variety of test layout, involving, ideally, all blocks in the layout. This is certainly a worthwhile subject of future research.

In addition, we have presented a breakdown of the C4 decoder into functional blocks, and further refined the most challenging block to implement, the Region Decoder sub-block of the Predict/Copy block. The Region Decoder is essentially a fast color rectangle rasterizer. Although the algorithm remains fairly complex, each sub-block of the region decoder consists of either simple digital circuits, such as adders, registers and memory, or simple state machines executing short commands composed together. Future work involves refining the C4 decoder to the gate level, for implementation on a FPGA, and finally the transistor level, for implementation in ASIC. We also continue to search for methods to improve the efficiency of C4, while reducing its implementation complexity.

ACKNOWLEDGEMENTS

This research is conducted under the Research Network for Advanced Lithography, supported jointly by the funding of the Semiconductor Research Corporation (2005-OC-460) and the Defense Advanced Research Project Agency (W911NF-04-1-0304). We would also like acknowledge Hsin-I Cindy Liu and Borivoje Nikolić of U.C. Berkeley for their invaluable insight into hardware design. They motivated much of the work presented in Section 8 of this paper.

REFERENCES

1. V. Dai and A. Zakhor, "Advanced Low-complexity Compression for Maskless Lithography Data", Emerging Lithographic Technologies VIII, Proc. of the SPIE, 5374, pp. 610-618, 2004.
2. V. Dai and A. Zakhor, “Binary Combinatorial Coding”, Proc. of the Data Compression Conference 2003, p. 420, 2003.
3. V. Dai and A. Zakhor, "Lossless Compression Techniques for Maskless Lithography Data", Emerging Lithographic Technologies VI, Proc. of the SPIE, 4688, pp. 583-594, 2002.
4. V. Dai and A. Zakhor, "Lossless Layout Compression for Maskless Lithography Systems", Emerging Lithographic Technologies IV, Elizabeth A. Dobisz, Editor, 3997, pp. 467-477, SPIE, 2000.
5. J. Ziv, and A. Lempel, “A universal algorithm for sequential data compression”, IEEE Trans. on Information Theory, IT-23 (3), pp.337-43, 1977.
6. J. Rissanen and G. G. Langdon, “Universal Modeling and Coding”, IEEE Trans. on Information Theory, IT-27 (1), pp. 12-23, 1981.
7. 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.
8. T. M. Cover, “Enumerative Source Coding”, IEEE Trans. on Information Theory, IT-19 (1), pp. 73-77, 1973.
9. S. W. Golomb, “Run-length Encodings”, IEEE Transactions on Information Theory, IT-12 (3), pp. 399-401, 1966.
10. L. Oktem and J. Astola, “Hierarchical enumerative coding of locally stationary binary data”, Electronics Letters, 35 (17), pp. 1428-1429, 1999.
11. I. H. Witten, A. Moffat, and T. C. Bell, Managing Gigabytes, Second Edition, Academic Press, 1999.
12. M. Burrows, and D. J. Wheeler, “A block-sorting lossless data compression algorithm”, Technical report 124, Digital Equipment Corporation, Palo Alto CA, 1994.
13. M. J. Weinberger, G. Seroussi, and G. Sapiro, “The LOCO-I lossless image compression algorithm: principles and standardization into JPEG-LS”, IEEE Transactions on Image Processing, 9 (8), pp.1309-1324, 2000.
14. B. Nikolić, B. Wild, V. Dai, Y. Shroff, B. Warlick, A. Zakhor, and W. G. Oldham, “Layout Decompression Chip for Maskless Lithography” in *Emerging Lithographic Technologies VIII*, Proceedings of the SPIE, San Jose, California, Vol. 5374, No. 1, February 2004, pp. 1092-1099.
15. M. J. Laszlo, *Computational Geometry and Computer Graphics in C++*, Prentice-Hall Inc., Upper Saddle River, NJ, 1996, pp. 173-202.
16. T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990, pp. 200-209.