Lossless Compression of VLSI Layout Image Data

Vito Dai, *Member, IEEE,* and Avideh Zakhor, *Fellow, IEEE*

Abstract— **We present a novel lossless compression algorithm called Context Copy Combinatorial Code (C4), which integrates the advantages of two very disparate compression techniques: context-based modeling and Lempel-Ziv (LZ) style copying. While the algorithm can be applied to many lossless compression applications, such as document image compression, our primary target application has been lossless compression of integrated circuit layout image data. These images contain 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. As part of C4, we have developed 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 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.**

Index Terms— **compression, lithography, maskless, C4.**

I. Introduction

FOR a next-generation 45-nm lithography system, using 25 nm, 5-bit grey pixels, a typical image of only one layer of a $2cm \times 1cm$ chip represents 1.6 terabits of data. A direct-write maskless lithography system with the same specifications requires data transfer rates of 10 terabits per second in order to meet the current industry production throughput of one wafer per layer per minute [3]. These enormous data sizes, and data transfer rates, motivate the application of lossless data compression to VLSI layout data.

VLSI designs produced by microchip designers consist of multiple layers of 2-D polygons stacked vertically, representing wires, transistors, etc. For pixel-based lithography writers, each layer is converted to a 2-D image. Pixels may be binary or grey depending on the design of the writer. A magnified sample of a binary image is shown in Fig. 1(a) and a grey image is shown in Fig. 1(b).

These lithography images differ from natural or even document images in several important ways. They are synthetically generated, highly structured, follow a rigid set of design rules, and contain highly repetitive regions cells of common structure.

In previous work [3], we have proposed a data-delivery system design capable of delivering these layout images to a massively parallel array of pixel-based lithography writers fabricated on a chip, e.g. an array of micromirrors described in [4]. This data-delivery system, shown in Figure. 2, meets the tera-pixel data rates through the use of compression. From top to bottom, first each layer of a microchip design is converted to a 2-D image, losslessly compressed by an estimated factor of 25, and stored to disk. Before the lithography writing process, a single compressed layer image is transferred from the disk to on-board DRAM memory. During the writing process, compressed data is streamed on-demand from the board to the chip containing the lithography writers. A parallel array of decoders, fabricated on the same substrate as the writers, must then decompress this data on-the-fly, and supply 10 Tb/s of data to the writers. The merits of this design, and the larger maskless lithography problem are beyond the scope of this paper. Here, we focus on the problem of finding a lossless compression algorithm for VLSI layout image capable of achieving such high lossless compression ratios. In addition, for this application, the decoder circuitry must be implemented in hardware as a parallel array of thousands of decoders, fabricated on the same integrated-circuit chip with a massively parallel array of lithography writers. As such, the decoder must have a very low implementation complexity. In contrast, the encoding is done off-line, and is under no such complexity constraint.

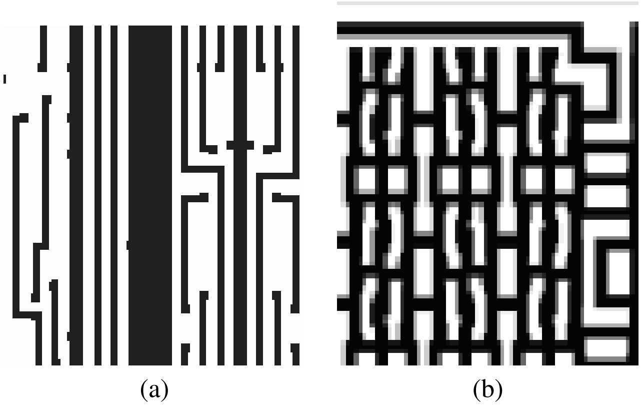

Fig. 1. A sample of layout image data (a) binary and (b) grey.

Our previous experiments [2], [3] have shown that Lempel-Ziv (LZ77) style copying [5], used in ZIP, results in high compression ratios on dense, repetitive circuits, such as arrays of memory cells. However, where these repetitions do not exist, such as control logic circuits, LZ-copying does not perform as well. In contrast, context-based prediction [6], used in JBIG [7], captures the local structure of lithography data, resulting in good compression ratios on non-repetitive circuits, but it fails to take advantage of repetitions, where they exist.

We have combined 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 hierarchical combinatorial coding (HCC) as a low-complexity alternative

Initial submission.

Video and Image Processing Lab, Department of Electrical Engineering and Computer Science, University of California/Berkeley. (e-mail: vdai@eecs.berkeley.edu; avz@eecs.berkeley.edu)

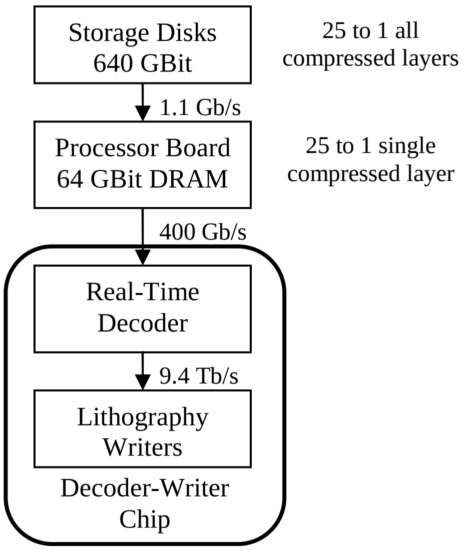

Fig. 2. System architecture of a data-delivery system for maskless lithography.

entropy coding technique to arithmetic coding [9] to be used within C4.

Section II describes the overall structure of C4. Section III describes the context-based prediction model used in C4. Section IV describes LZ-copying in two dimensions and how the C4 encoder segments the image into regions using LZ-copying and context-based prediction. Section V describes HCC used to code prediction errors. Section VI describes the extension of C4 to grey-pixel layout image data. Section VII includes the compression results of C4 in comparison to other existing compression techniques for integrated circuit layout data.

II. Overview of 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. However, neither predicted values nor copied values are 100% correct, so *error bits* are used to indicate the position of these prediction or copy errors. These error bits can be compressed using any binary entropy coder, but in C4, we apply our own hierarchical combinatorial coding (HCC) technique as a low-complexity alternative to arithmetic coding. Only the copy regions and compressed error bits are transmitted to the decoder.

In addition, as discussed in Section I, for our application to direct-write maskless lithography, the C4 decoding algorithm must 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]. As such, the C4 decoder must have a low implementation complexity. In contrast, the C4 encoder is under no such complexity constraint. This basic asymmetry in the complexity requirement between encoding and decoding is central to the design of the C4 algorithm.

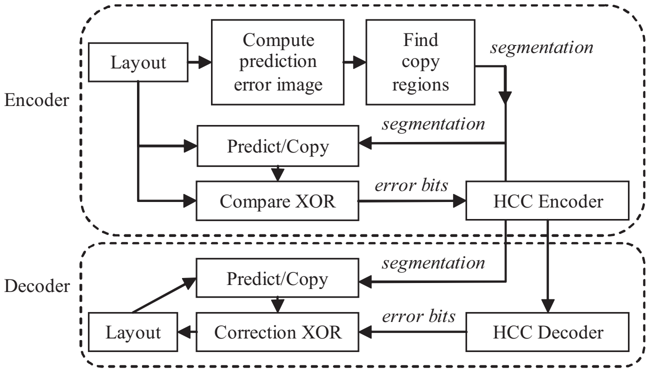

Fig. 3. Block diagram of C4 encoder and decoder for binary images.

Fig. 3 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 resulting error image is used to determine the segmentation map between *copy regions* and the *prediction region*, i.e. the set of pixels not contained in any copy region. As specified by the segmentation map, 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. Correctly predicted or copied pixels are indicated with a “0”, and incorrectly predicted or copied pixels 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.

The decoder mirrors the encoder, but skips the complex steps necessary to find the segmentation map, which are received from the encoder. Again as specified by the segmentation, the Predict/Copy block estimates each pixel value, either by copying or by prediction. The HCC decoder decompresses the error bits from the encoder. 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 must be inverted, equivalent to a Boolean XOR operation. Since there is no data modeling performed in the C4 decoder, it is considerably simpler to implement than the encoder, satisfying one of the requirements of our application domain.

III. Context-based Prediction Model

For our application domain, i.e. integrated-circuit layout compression, we use a simple 3-pixel binary context-based prediction model to use in C4, much simpler that the 10-pixel model used in JBIG [7]. Since the number of contexts scales exponentially with the number pixels used for prediction, this represents a significant complexity reduction of the C4 prediction mechanism at the decoder, as compared to JBIG. Nonetheless, this simple 3-pixel context captures the essential

TABLE I
THE 3-PIXEL CONTEXTS, PREDICTION, AND THE EMPIRICAL PREDICTION ERROR PROBABILITY FOR A SAMPLE LAYOUT

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

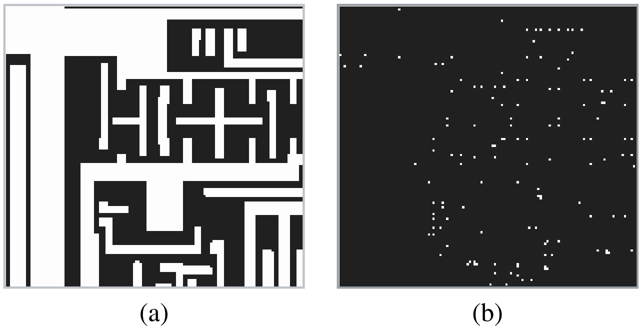

Fig. 4. Non-repetitive layout image data and its resulting prediction error image.

"Manhattan" structure of layout data, as well as some design rules, as seen in Table I.

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. Layout data is dominated by vertical edges, horizontal edges, and regions of constant intensity. The simple 3-pixel context predicts all these cases perfectly. Consequently, we do not expect much benefit to increasing the number of context pixels to 4 or higher, and in fact, this intuition matches our empirical observations. Visual inspection of the prediction error 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.

To generate the prediction error image, each correctly predicted pixel is marked with a "0", and each incorrectly predicted pixel is marked with a "1", creating a binary image which can be compressed with a standard binary entropy coder. The fewer the number of incorrect predictions, the higher the compression ratio achieved. Alternatively, we can group pixels by their context, forming 8 separate binary streams, each to be compressed separately by its own entropy coder. Empirically, for layout images, we have found that operating on the prediction error image is as efficient as grouping pixels by context.

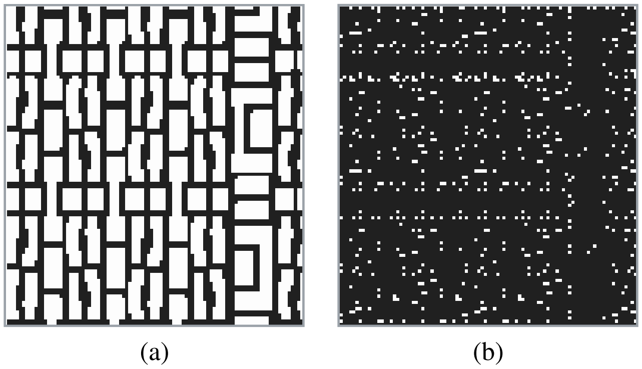

Fig. 5. Dense repetitive layout image data and its resulting prediction error image.

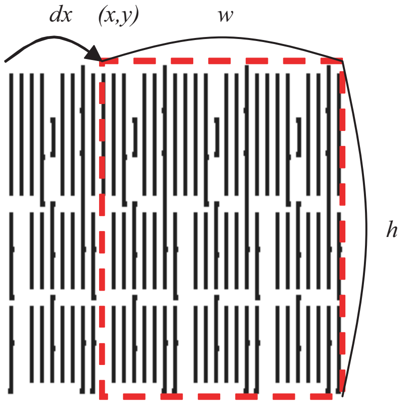

Fig. 6. Illustration of a copy left region.

An example of non-repetitive layout for which prediction works well is shown in Fig. 4(a), and its corresponding prediction error image is shown in Fig. 4(b).

In contrast to the non-repetitive layout shown in Fig. 4(a), some layout image data contains regions that are visually "dense" and repetitive. An example of such a region is shown in Fig. 5(a). This visual "denseness" results in a dense, large number of prediction errors as seen clearly in the prediction error image in Fig. 5(b).

The high density of prediction errors translates into low compression ratios using prediction alone. In C4, areas of dense repetitive layout are covered by copy regions to reduce the number of errors, as described in Section IV.

IV. Copy Regions and Segmentation

As seen in Fig. 5(a) of the previous section, some layout images are highly repetitive. We can take advantage of this repetitiveness to achieve compression by specifying *copy regions*, i.e. a rectangular region that is copied from another rectangular region preceding it in raster-scan order.

An example of a copy region is shown in the dashed rectangle in Fig. 6. As seen, a copy region is specified with six copy parameters: 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 Fig. 6, every pixel inside the region is copied from dx pixels to its left, and $dy = 0$. Although the entire region is copied, the copy itself need not be 100% correct. Similar to the prediction error map, there is a corresponding copy error map within the copy region. Each correctly copied pixel is indicated with a "0", and each incorrectly copied pixel each incorrectly predicted pixel is marked with a "1", creating a binary sub-image which can be compressed with a standard binary entropy coder.

This method of copying is related to LZ77 [5] copying, but extended to 2-D. Conceptually, this allows us to capture the 2D-repetitions found in layout. In addition, note that in Fig. 6 the single copy region spans three horizontal periodic repetitions, with period dx. In C4, an entire periodic array can be described with a single copy region. This simple feature is extremely important for extracting compression efficiency from layout data, which can contain large periodic 2D-arrays. This contrasts sharply with the method of copying used in JBIG2 [8] called "soft-pattern matching" [16], which constructs an explicit dictionary of pixel blocks that can be throughout the image. In JBIG2, each individual copy in a 2D periodic array must be separately referenced [8], leading to a lower compression efficiency in comparison to C4 for this type of data.

As described in Section II, the C4 encoder automatically segments the image into copy regions and the prediction region, i.e. all pixels not contained in any copy region. Each copy region has its own copy parameters and corresponding copy error map, and the background prediction region has a corresponding prediction error map. Together, the error maps merge to form a combined binary prediction/copy error map of the entire image, which is compressed using hierarchical combinatorial coding (HCC) as a binary entropy coder. The lower the number of the total sum of prediction and copy errors, the higher the compression ratio achieved. However, this improvement in compression by the introduction of copy regions, is offset by the *cost* in bits to specify the copy parameters (x, y, w, h, dx, dy) of each copy region. Moreover, copy regions that overlap with each other are undesirable: each pixel should only be coded once, to save as many bits as possible.

Ideally, we would like the C4 encoder to find the set of non-overlapping copy regions, which minimizes the sum of number of compressed prediction/copy error bits, plus the number of bits necessary to specify the parameters of each copy region. An exhaustive search over this space would involve going over all possible non-overlapping copy region sets, a combinatorial problem, generating the error bits for each set, and performing HCC compression on the error bits. This is clearly infeasible. To make the problem tractable, a number of simplifying assumptions and approximate metrics are adopted.

First we use entropy as a heuristic to estimate the number of bits generated by the HCC encoder to represent error pixels. If p denotes the percentage of prediction/copy error pixels over the entire image, then error pixels are assigned a per-pixel cost of $C = -log_2(p)$ bits, and correctly predicted or copied pixels are assigned a per-pixel cost of $-log_2(1-p) \approx 0$. Of course, given a segmentation map, p can be easily calculated by counting the number of prediction/copy error bits; at the same time, p affects how copy regions are generated in the first place, as discussed shortly. In C4, we solve this chicken and egg problem by first guessing a value of p, finding a segmentation map using this value, counting the percentage of prediction/copy error pixels, and using this percentage as a new value for p as input to the segmentation algorithm. This process can be iterated until the guess p matches the percentage of error pixels, but in practice we find that one iteration is sufficient if the starting guess is reasonable. Empirically, we have found a good starting guess to be the percentage of error pixels when no copy regions are used, then discounted by a constant factor of 4.

Next, for any given copy region, we compare the cost, in bits, of coding that region using copy, versus the cost of coding the region using prediction. If the cost of copying is lower, then the amount by which it is lower is the $benefit$ of using this region. The cost of copying is defined as the sum of the cost of describing the copy parameters, plus the cost of coding the copy error map. For our particular application domain, the description cost is 51 bits. 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, represented by 10 bits, denotes the distance left or above to copy from, and $left/above$, represented by 1 bit, denotes the direction left or above to copy from. This assumption is in line with the Manhattan structure of layout data. The cost of coding the copy error map is estimated as $C \times E_{copy}$, where C denotes the estimated per-pixel cost of an error pixel, as discussed previously, and E_{copy} denotes the number of copy error pixels in the region. Correctly copied pixels are assumed to have 0 cost, as discussed previously. So the total cost of copying is $51 + C \times E_{copy}$.

The cost of coding the region using prediction is the cost of coding the prediction error map of that region. It is estimated as $C \times E_{context}$, where $E_{context}$ denotes the number of prediction error pixels in the region. Finally, the $benefit$ of a region is the difference between these two costs, $C \times (E_{context} - E_{copy}) - 51$. Note that it is possible for a region to have negative benefit if $E_{context} - E_{copy} \leq (51/C)$. The threshold $T = (51/C)$ is used to quickly disqualify potential copy regions in the search algorithm presented below.

Using $benefit$ as a metric, the optimization goal is to find the set non-overlapping copy regions, which maximizes the sum of $benefit$ over all regions. This search space is combinatorial in size, so exhaustive search is prohibitively complex. Instead we adopt a greedy approach, similar to that used in the 2D-LZ algorithm described in [3]. The basic strategy used by the *find copy regions* algorithm in Fig. 3 is as follows: start with an empty list of copy regions; and in raster-scan order, add copy regions of maximum benefit, that do not overlap with regions previously added to the list. The completed list of copy regions is the *segmentation* of

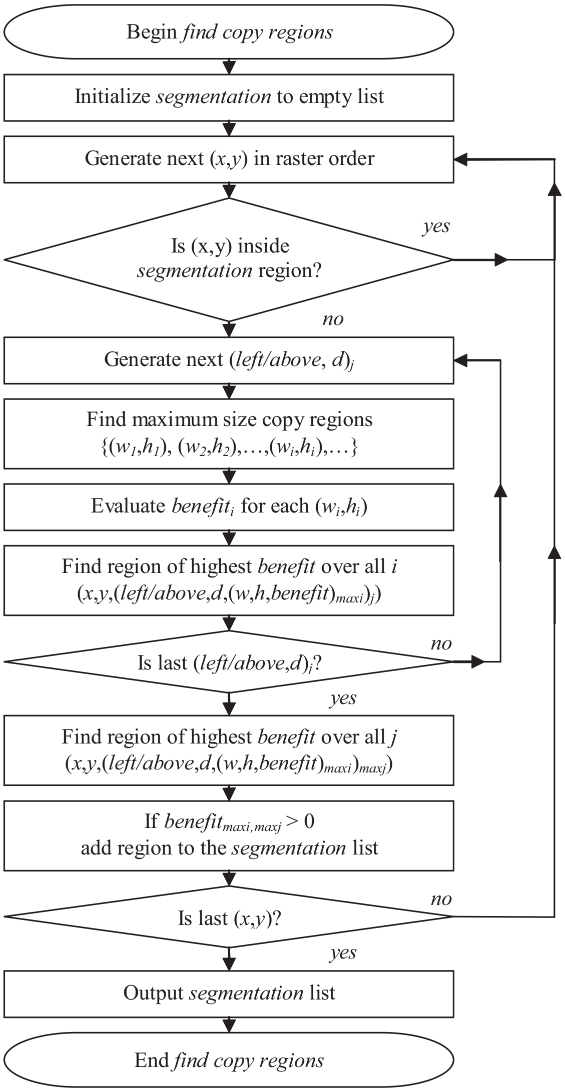
Begin *find copy regions*
Initialize *segmentation* to empty list
Generate next (*x*,*y*) in raster order
Is (x,y) inside *segmentation* region?
yes
no
Generate next (*left/above*, *d*)$_j$
Find maximum size copy regions {(w_1,h_1), (w_2,h_2),…,(w_i,h_i),…}
Evaluate *benefit*$_i$ for each (w_i,h_i)
Find region of highest *benefit* over all *i* (*x*,*y*,(*left/above*,*d*,(*w*,*h*,*benefit*)$_{maxi}$)$_j$)
Is last (*left/above*,*d*)$_j$?
no
yes
Find region of highest *benefit* over all *j* (*x*,*y*,(*left/above*,*d*,(*w*,*h*,*benefit*)$_{maxi}$)$_{maxj}$)
If *benefit*$_{maxi,maxj}$ > 0 add region to the *segmentation* list
Is last (*x*,*y*)?
no
yes
Output *segmentation* list
End *find copy regions*

Fig. 7. Flow diagram of the *find copy regions* algorithm.

the layout. A detailed flow diagram of the *find copy regions* algorithm is shown in Fig. 7, and described in the remainder of this section.

In raster-scan order, we iterate through all possible (x, y). If (x, y) is inside any region in the *segmentation* list, we move on to the next (x, y); otherwise, we iterate through all possible $(left/above, d)$. Next for a given $(x, y, left/above, d)$, we maximize the size of the copy region (w, h) with the constraint that a *stop pixel* is not encountered; we define a *stop pixel* to be any pixel inside a region in the *segmentation* list, or any pixel with a copy error. These conditions prevent overlap of copy regions, and prevent the occurrence of copy errors, respectively. Later, we describe how to relax this latter condition to allow for copy errors. The process of finding maximum size copy regions (w, h), is discussed in the next paragraph. Finally, we compute the benefit of all the maximum sized copy regions, and, if any region with positive benefit exists, we add the one with the highest positive benefit to the *segmentation* list.

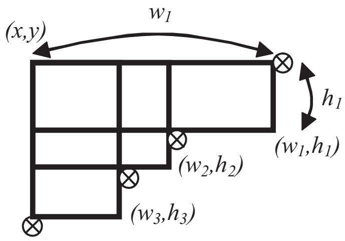

Fig. 8. Illustration of three maximum copy regions bordered by four stop pixels.

We now describe the process of finding the maximum size copy region (w, h). For any given $(x, y, left/above, d)$ there is actually a set of maximum size copy regions, bordered by stop pixels, because (w, h) is a two-dimensional quantity. This is illustrated in the example in Fig. 8. In the figure, the position of the stop pixels are marked with $\otimes$ and three overlapping maximum copy regions are shown (x, y, w_1, h_1) (x, y, w_2, h_2) and (x, y, w_3, h_3). The values w_1, h_1, w_2, h_2, w_3, and h_3 are found using the following procedure: initialize $w = 1$, $h = 1$. Increment w until a stop pixel is encountered; at this point $w = w_1$. Next increment h, and for each h increment w from 1 to w_1, until a stop pixel is encountered; at this point $h = h_1$, and $w = w_2$. Again increment h, and for each h increment w from 1 to w_2, until a stop pixel is encountered; at this point $h = h_2$, and $w = w_3$. Finally, increment h, and for each h increment w from 1 to w_3, until a stop pixel is encountered; at this point $h = h_3$, and $w = 1$. The maximum size algorithm is terminated when a stop pixel is encountered at $w = 1$.

As stated previously, any pixel inside a region in the *segmentation* list, and any pixel with a copy error, is a *stop pixel*. The latter condition prevents any copy errors inside a copy region. We relax this condition to merge smaller, error free, copy regions, into larger copy regions with a few number of copy errors. The basic premise is to tradeoff the 51-bits necessary to describe a new copy region against the introduction of bits needed to code copy errors, by excluding some copy error pixels from being *stop pixels*. For each copy error pixel, we look at a window of W pixels in a row, where the left most pixel is the copy error. If, in that window, the number of copy errors is less than the average number of errors expected, $E_{copy} < Wp$, and the number of copy errors is less than the number of prediction errors, $E_{copy} < E_{predict}$, then pixel with the copy error no longer considered to be a stop pixel. The size of the look-ahead window W is a user-defined input parameter to the C4 algorithm. Larger values of W correspond to fewer, larger copy regions, at the expense of increasing the number of copy errors.

V. Hierarchical Combinatorial Coding

We have proposed and developed combinatorial coding (CC) [9] as an alternative to arithmetic coding to encode the error bits in Fig. 3. The basis for CC is universal enumerative coding [10] which works as follows. For any binary sequence of known length N, let k denote the number of ones in that sequence. k ranges from 0 to N, and can be encoded using a minimal binary code [13], i.e. a simple Huffman code for uniform distributions, using $\lceil log_2(N + 1)\rceil$ bits. There are exactly $C(N,k) = N!/(N-k)!k!$ sequences of length N with k ones, which can be hypothetically listed. The index of our sequence in this list, known as the *ordinal* or *rank*, is an integer ranging from 1 to $C(N,k)$, which can again be encoded using a minimal binary code, using $\lceil log_2 C(N,k) \rceil$bits. Enumerative coding is theoretically shown to be optimal [10] if the bits to be compressed are independently and identically distributed (i.i.d.) as $Bernoulli(\theta)$ where θ denotes the unknown probability that "1" occurs, which in C4, corresponds to the percentage of error pixels in the prediction/copy error map. The drawback of computing an enumerative code directly is its complexity: the algorithm to find the rank corresponding to a particular binary sequence of length N, called *ranking* in the literature, is in time, is $O(N)$ in memory, and requires $O(N)$ bit precision arithmetic [10].

In CC, this problem is addressed 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 $(k, rank)$ pair for each block. Again, using the same assumption that input bits are i.i.d. as $Bernoulli(\theta)$, the number of ones k in a block of M bits are i.i.d. as $Binomial(M, \theta)$. Even though the parameter θ is unknown, as long as the Binomial distribution is not too skewed, e.g. $0.01 < \theta < 0.99$, a dynamic Huffman code efficiently compresses the k-values with little overhead, because the range of k is small. Given there are k ones in a block of M bits, the rank remains uniformly distributed, as in enumerative coding. Therefore, $rank$-values are efficiently coded using a minimum binary code.

The efficiency of CC, as described, is on par with arithmetic coding, except in cases of extremely skewed distributions, e.g. $\theta < 0.01$. In these cases, the probability that $k = 0$ approaches 1 for each block, causing the Huffman code to be inefficient. To address this issue, we have developed an extension to CC called hierarchical combinatorial coding (HCC). It works by binarizing sequence of k-values such that $k = 0$ is indicated with a "0" and $k = 1$ to 32 is indicated with a "1". CC is then applied to the binarized sequence of "0" and "1", and the value of k, ranging from 1 to 32 in the "1" case, is Huffman coded. Clearly, this procedure of CC encoding, binarizing the k-values, then CC encoding again can be recursively applied in a hierarchical fashion, to take care of any inefficiencies in the Huffman code for k-values, as θ approaches 0.

Fig. 9 is an example of HCC in action with 2-levels of hierarchy and block size $M = 4$. Only values in bold italics are coded and transmitted to the decoder. Looking at rows from bottom to top, the original data is in the lowest row labeled "bits – level 0". Applying CC with $M = 4$, the next two rows show the $rank$ and k value for each block in level 0. Note that when $k = 0$ no $rank$ value is needed as indicated by the hyphen. The high frequency of 0 in "k – level 0" makes it inefficient for coding directly using Huffman coding. Instead, we binarize "k – level 0", to form "bits – level 1", using the binarization procedure described in the previous paragraph. CC is recursively applied to "bits – level 1", to compute "$rank$ – level 1" and "k – level 1". Finally, to code the data, "k – level 1" is coded using a Huffman code, "$rank$ – level 1" is coded using a minimal binary code, *non-zero* values of "k – level 0" are coded using a Huffman code, and "$rank$ – level 0" is coded using a minimal binary code.

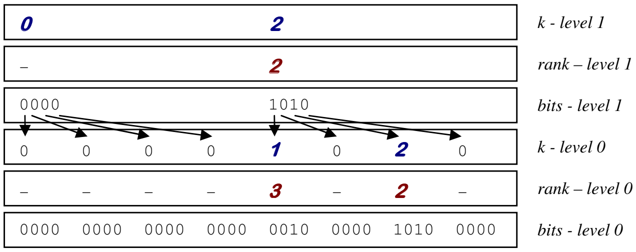

Fig. 9. 2-level HCC with a block size $M = 4$ for each level.

The rationale for choosing Huffman coding and minimal binary coding is the same as CC. If the input is assumed to be i.i.d. as $Bernoulli(\theta)$, all level $rank$-values are uniformly distributed, given the corresponding k-values in the same level. Furthermore, although the exact distribution of k-values is unknown, a dynamic Huffman code can adapt to the distribution with little overhead, because the dynamic range of k is small. Finally, for highly skewed distributions of k, which hurts the compression efficiency of Huffman coding, the binarization process reduces the skew by removing the most probable symbol $k = 0$.

Studying the example in Fig. 9, we can intuitively understand the efficiency of HCC: the single Huffman coded ***0*** in "k – level 1" decodes to M^2 zeroes in "bits – level 0". In general, for L-level HCC, a single Huffman coded ***0*** in level $L - 1$ corresponds to M^L zeroes in "bits - level 0". HCC's ability to effectively compress blocks of zeroes is critical to achieving high compression ratios, when the percentage of the error pixels is low.

In addition to achieving efficient compression, HCC also has several properties favorable to our application domain. First, the decoder is extremely simple to implement: the Huffman code tables are small because the range of k-values is small, unranking is accomplished with a simple table lookup, comparator, and adder, and minimal binary decoding is also accomplished by a simple table lookup and an adder. Second, the decoder is fast: blocks of $M^{(L+1)}$ zeroes can be decoded instantly when a zero is encountered at level L. Third, HCC is easily parallelizable: block sizes are fixed and block boundaries are independent of the data, so the compressed bitstream can be easily partitioned and distributed to multiple parallel HCC decoders. This is in contrast to run-length coding schemes such as Golomb codes [11], which also code for runs of zeroes, but have data-dependent block boundaries.

Independent of our development of HCC, a similar technique called Hierarchical Enumerative Coding (HEC) has been developed in [12]. The main difference between HEC and HCC is the method of coding k values at each level. HCC uses binarization and simple Huffman coding, whereas HEC uses hierarchical integer enumerative coding, which is more complex [12]. In addition, HEC requires more levels of hierarchy to achieve the same level of compression efficiency as HCC. Consequently, HCC is significantly less complex to

TABLE II

RESULT OF 3-PIXEL CONTEXT BASED BINARY IMAGE COMPRESSION ON A 242 KB LAYOUT IMAGE FOR A P3 800 MHZ PROCESSOR

Metric	Huf8	Arith.	Golomb	HEC	HCC
Comp. ratio	7.1	47	49	48	49
Enc. time(s)	0.99	7.46	0.52	2.43	0.54
Dec. time(s)	0.75	10.19	0.60	2.11	0.56

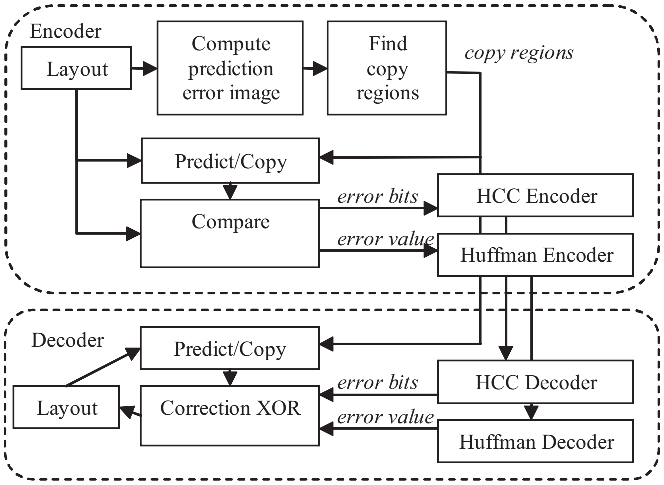

Fig. 10. Block diagram of C4 encoder and decoder for grey-pixel images.

compute than HEC.

To compare HCC with existing entropy coding techniques, we apply 3-pixel context based modeling as described in III to a 242 kb layout image and group pixels by context into 8 binary streams. We then apply Huffman coding to blocks of 8-bits, arithmetic coding, Golomb run-length coding, HEC, and HCC to each binary stream, and report the compression ratio obtained by each algorithm. In addition, we report the encoding and decoding times as a measure for complexity of these algorithms. The results are shown in Table II.

Among these techniques, HCC is one of the most efficient in terms of compression, and one of the fastest to encode and decode, justifying its use in C4. The only algorithm comparable in both efficiency and speed, among those tested, is Golomb run-length coding. However, as previously mentioned, HCC has fixed, data-independent block boundaries, which are advantageous for parallel hardware implementations; run-length coding does not. 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 in [13].

VI. 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, slight modifications need to be made to the prediction mechanism, and the representation of the error. Specifically, the local 3-pixel context based prediction described in Section 3, is replaced by 3-pixel linear prediction with saturation, to be described later; furthermore, in places of prediction or copy error, where the error bit is ”1”, an *error value* indicates the correct value of that pixel. A block diagram of the C4 encoder and decoder for grey-pixel images is shown in Fig. 10.

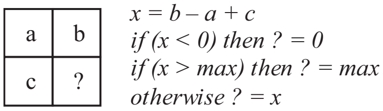

Fig. 11. 3-pixel linear prediction with saturation used in grey-pixel C4.

TABLE III

COMPRESSION RATIOS OF JBIG, JBIG2, ZIP, 2D-LZ, BZIP2 AND C4 FOR 2048 × 2048 BINARY LAYOUT IMAGE DATA.

Type	Layer	JBIG	JBIG2	ZIP	2D-LZ	BZIP2	C4
Mem. Cells	M2	59	68	88	233	260	332
	M1	10	12	48	79	56	90
	Poly	12	14	51	120	83	141
Ctrl. Logic	M2	47	52	22	26	32	50
	M1	20	23	11	11	11	22
	Poly	42	43	19	20	23	45
Encode Time (s)		6	11	2	640	4	720
Decode Time (s)		6	7	1	2	4	2

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 indicated with a “0”, and incorrectly predicted or copied pixels are indicated with a “1” with an error value generated indicating the true value of the pixel. The error bits are compressed with a HCC encoder, and the actual error values are compressed with a Huffman encoder.

As in 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 decodes the error bits, and the Huffman decoder decodes 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.

The linear prediction mechanism used in 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 Fig. 11. “?” 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. 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 described in Section 3. It is also similar to the median predictor used in JPEG-LS [15].

VII. 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 III. The original data are 2048×2048 binary images with 300 nm pixels sampled from an industry microprocessor layout, which corresponds to a 0.61 mm by 0.61 mm section, covering about 0.1% of the chip area. Each entry in the table corresponds to the compression ratio for 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, JBIG2, ZIP, 2D-LZ our 2D extension to the LZ77 copying [3], BZIP2 based on the Burrows-Wheeler Transform [14], and C4. The bold numbers indicate the highest compression for each row.

As seen, C4 outperforms all these algorithms for repetitive layouts, and is tied for first with JBIG2 for non-repetitive layouts. This is significant, because most layouts contain a heterogeneous mix of memory and control circuits. ZIP, 2D-LZ and BZIP2 take advantage of repetitions resulting in high compression ratios on memory cells. In contrast, where the layout becomes less regular, the context modeling of JBIG and JBIG2 has an advantage over ZIP, 2D-LZ, and BZIP2. It is worth noting that the compression efficiency of JBIG2 varies with the encoder implementation, and our tests are based on the JBIG2 encoder implementation used within Adobe Acrobat 6.0.

The last two rows report the encoder and decoder runtime of the various algorithms on a 1.8GHz Mobile Pentium 4 with 512MB of RAM running Windows XP. Each algorithm is asked to compress and decompress a suite of 10 binary layout files, and runtimes are measured to the nearest second by hand. Unfortunately, a more precise measurement has not been possible due to the varying input/output formats of the different software. The most striking result is the slow speed of the C4 encoder, in contrast to the fast performance of the C4 decoder. This is a direct consequence of the segmentation algorithm at the encoder, that is absent from the decoder implementation. Although it can be argued that encoder complexity is not a concern in our application domain, nonetheless, some algorithmic improvements and optimizations to improve the speed of the segmentation are needed.

TABLE IV

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

Layer	RLE	Huf	LZ77 256	LZ77 1024	ZIP	BZIP2	C4
M2	1.4	2.3	4.4	21	25	28	35
M1	1.0	1.7	2.9	5.0	7.8	11	15
Poly	1.1	1.6	3.3	4.6	6.6	10	14
Via	5.0	3.7	10	12	15	24	32
N	6.7	3.2	13	28	32	42	52
P	5.7	3.3	16	45	52	72	80
Enc (s)	1	1	6	10	4	8	1680
Dec (s)	1	1	2	2	2	8	3

TABLE V

Percent of each image covered by copy regions (Copy%), and its relation to compression ratios for Linear Prediction (LP), ZIP, and C4 for 5-bit grey layout image data.

Type	Layer	LP	ZIP	C4	Copy%
Repetitive	M1	3.3	7.8	18	94%
	Poly	2.1	6.6	18	99%
Non-Rep.	M1	14	12	16	18%
	Poly	7.3	9.6	14	42%
Mixed	M1	7.5	12	15	44%
	Poly	4.1	10	14	62%
	M2	15	26	35	33%
	N	18	32	52	21%
	P	29	52	80	33%
	Via	7.1	15	32	54%

Table IV is compression results for more modern layout image data with 65 nm pixels and 5-bit grey layout image data. For each layer, 5 blocks of 1024×1024 pixels are sampled from two different layouts, 3 from the first, and 2 from the second, and the *minimum* compression ratio achieved for each algorithm over all 5 samples is reported. 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. 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, and C4. Clearly, C4 still has the highest compression ratio among all these techniques. Some notable lossless grey-pixel image compression techniques have been excluded from this table including SPIHT and JPEG-LS. Our previous experiments in [2] have already shown that they do not perform well as simple ZIP compression on this class of data.

Again, the last two rows report the encoder and decoder runtime of the various algorithms on a 1.8GHz Mobile Pentium 4 with 512MB of RAM running Windows XP. Each algorithm is asked to compress and decompress a suite of 10 grey layout files, and runtimes are measured to the nearest second by hand. Again, the slow speed of the C4 encoder contrasts the fast performance of the C4 decoder.

In Table V, we show results for 10 sample images from the data set used to obtain Table IV, where each row is information on one sample image. In the first column "Type", we visually categorize each sample as repetitive, non-repetitive, or containing a mix of repetitive and non-repetitive regions. The second column is the chip layer from which the sample is drawn. The third column "LP" is the compression ratio achieved by linear prediction alone, equivalent to C4 compression with copy regions disabled. The fourth and fifth columns are the compression ratio achieved by ZIP and the full C4 compression respectively. The last column "Copy%" is the percent of the total sample image area covered by copy regions, when C4 compression is applied. Any pixel of the image not covered by copy regions is, by default, linearly predicted from its neighbors.

Clearly, the Copy% varies dramatically from image to image ranging from 18% to 99% across the 10 samples, testifying to C4's ability to adapt to different types of layouts. In general a high Copy% corresponds to repetitive layout, and low Copy% corresponds to non-repetitive layout. Also, the higher the Copy%, the more favorably ZIP compares to LP compression. This agrees with the intuition that LZ-style techniques work well for repetitive layout, and prediction techniques work well for non-repetitive layout. At one extreme, in the non-repetitive-M1 row, where 18% of the image is copied in C4, LP's compression ratio exceeds ZIP. At the other extreme, in the repetitive-Poly row, where 99% of the image is copied, ZIP's compression ratio is more than 3 times that of LP. This trend breaks down when the compression ratio becomes high for both LP and ZIP, e.g. the rows labeled Mixed-N and Mixed-P. These layouts contain large featureless areas, which are easily compressible by both copying and prediction. In these cases, C4 favors using prediction to avoid the overhead of specifying copy parameters.

VIII. Conclusion

A novel compression algorithm called C4 is presented, 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, C4 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 a wide variety of industry lithography image data.

Acknowledgment

This research is conducted under the Research Network for Advanced Lithography, supported jointly by SRC (01-MC-460) and DARPA (MDA972-01-1-0021).

References

[1] V. Dai and A. Zakhor, "Advanced Low-complexity Compression for Maskless Lithography Data", *Emerging Lithographic Technologies VIII*, Proc. of the SPIE Vol. 5374, pp. 610–618, 2004.

[2] V. Dai and A. Zakhor, "Lossless Compression Techniques for Maskless Lithography Data", *Emerging Lithographic Technologies VI*, Proc. of the SPIE Vol. 4688, pp. 583–594, 2002.

[3] V. Dai and A. Zakhor, "Lossless Layout Compression for Maskless Lithography Systems", *Emerging Lithographic Technologies IV*, Proc. of the SPIE Vol. 3997, pp. 467–477, 2000.

[4] N. Chokshi, Y. Shroff, W. G. Oldham, et al., "Maskless EUV Lithography", *Int. Conf. Electron, Ion, and Photon Beam Technology and Nanofabrication*, Macro Island, FL, June 1999.

[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] P. G. Howard, F. Kossentini, B. Martins, S. Forchammer, W. J. Rucklidge, "The Emerging JBIG2 Standard", *IEEE Trans. Circuits and Systems for Video Technology*, Vol. 8, No. 7, pp. 838-848, November 1998.

[9] V. Dai and A. Zakhor, "Binary Combinatorial Coding", *Proc. of the Data Compression Conference 2003*, p. 420, 2003.

[10] T. M. Cover, "Enumerative Source Coding", *IEEE Trans. on Information Theory*, IT-19 (1), pp. 73–77, 1973.

[11] S. W. Golomb, "Run-length Encodings", *IEEE Transactions on Information Theory*, IT-12 (3), pp. 399–401, 1966.

[12] L. Oktem and J. Astola, "Hierarchical enumerative coding of locally stationary binary data", *Electronics Letters*, 35 (17), pp. 1428–1429, 1999.

[13] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes, Second Edition*, Academic Press, 1999.

[14] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm", Technical report 124, Digital Equipment Corporation, Palo Alto CA, 1994.

[15] 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.

[16] P. G. Howard, "Text image compression using soft pattern matching", *Computer Journal*, vol.40, no.2-3, Oxford University Press for British Comput. Soc, UK, 1997, pp.146-56.

[17] P. Frnti and O. Nevalainen, "Compression of binary images by composite methods based on the block coding", Journal of Visual Communication and Image Representation, 6 (4), 366-377, December 1995.

[18] G. G. Langdon, Jr., J. Rissanen, "Compression of black-white images with arithmetic coding", *IEEE Transactions on Communications*, vol.COM-29, no.6, June 1981, pp.858-67. USA.

PLACE
PHOTO
HERE

Vito Dai (M'00) received the B.S. degree in electrical engineering at California Institute of Technology in 1998, and the M.S. degree in electrical engineering at University of California at Berkeley in 2000. He will complete his Ph.D. degree at U.C. Berkeley in Electrical Engineering in June 2006. He is currently a Sr. Engineer in the OPC group at Advanced Micro Devices.

PLACE
PHOTO
HERE

Avideh Zakhor received the B. S. degree from California Institute of Technology, Pasadena, and the S. M. and Ph. D. degrees from Massachusetts Institute of Technology, Cambridge, all in electrical engineering, in 1983, 1985, and 1987 respectively. In 1988, she joined the Faculty at U. C. Berkeley where she is currently 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 3D 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 5 U.S. patents, and is the co-author of the book, "Oversampled A/D Converters" with Soren Hein.

Prof. Zakhor was a General Motors scholar from 1982 to 1983, was a Hertz fellow from 1984 to 1988, received the Presidential Young Investigators (PYI) award, and Office of Naval Research (ONR) young investigator award in 1992. From 1998 to 2001, she was an elected member of IEEE Signal Processing Borad of Governers. In 2001, she was elected as IEEE fellow. She received the Okawa Prize in 2004.

She co-founded OPC technology in 1996, which was later acquired by Mentor Graphics (Nasdaq: MENT) in 1998, Truvideo in 2000, and Urban Scan in 2005.