Lossless Layout Compression for Maskless Lithography Systems

Vito Dai* and Avideh Zakhor

Video and Image Processing Lab
Department of Electrical Engineering and Computer Science
Univ. of California/Berkeley

ABSTRACT

Future lithography systems must produce more dense chips with smaller feature sizes, while maintaining throughput comparable to today's optical lithography systems. This places stringent data-handling requirements on the design of any maskless lithography system. Today's optical lithography systems transfer one layer of data from the mask to the entire wafer in about sixty seconds. To achieve a similar throughput for a direct-write maskless lithography system with a pixel size of 25 nm, data rates of about 10 Tb/s are required. In this paper, we propose an architecture for delivering such a data rate to a parallel array of writers. In arriving at this architecture, we conclude that pixel domain compression schemes are essential for delivering these high data rates. To achieve the desired compression ratios, we explore a number of binary lossless compression algorithms, and apply them to a variety of layers of typical circuits such as memory and control. The algorithms explored include the Joint Bi-Level Image Processing Group (JBIG), Ziv-Lempel (LZ77) as implemented by ZIP, as well as our own extension of Ziv-Lempel to two-dimensions. For all the layouts we tested, at least one of the above schemes achieves a compression ratio of 20 or larger, demonstrating the feasibility of the proposed system architecture.

Keywords: maskless, lithography, compression, JBIG, ZIP, LZ77, layout, pattern, direct-write

1. INTRODUCTION

Future lithography systems must produce more dense chips with smaller feature sizes, while maintaining throughput comparable to today's optical lithography systems. This places stringent data-handling requirements on the design of any direct-write maskless system. Optical projection systems use a mask to project the entire chip pattern in one flash. An entire wafer can then be written in a few hundred such flashes. In contrast, a direct-write maskless system must write each individual pixel of the chip pattern directly onto the wafer. To achieve writing speeds comparable to today's optical systems requires a direct-write system capable of transferring trillions of pixels per second onto the wafer. Our goal in this paper is to design a data processing system architecture, which is capable of meeting this enormous throughput requirement. In doing so, we will demonstrate that lossless binary compression plays an important role.

To arrive at this system, we begin, in section 2, with detailed device specifications and the resulting system specifications. Several designs are considered and discarded, based on memory, processing power, and throughput requirements. The final design we arrive at consists of storage disks, a processor board, and circuitry fabricated on the same chip as the hardware writers. To make this system design feasible, we estimate that a compression ratio of 25 is necessary to achieve the desired data rates. In section 3, we explore existing lossless compression schemes: the context-based arithmetic coding scheme as implemented by the Joint Bi-Level Image Processing Group (JBIG) [12], and the adaptive-dictionary based technique of Ziv and Lempel (LZ77) [13] as implemented by popular compression packages (ZIP) [7]. In addition, we devise and implement a two-dimensional variant of the LZ77 algorithm (2D-LZ) in section 3, and test its compression performance against that of JBIG and ZIP in section 4. Conclusions and directions for future research are included in section 5.

2. SYSTEM ARCHITECTURE

Maskless direct-write lithography is a next-generation lithographic technique, targeted for the sub-50 nm device generations. The left side of Table 1 presents relevant specifications for devices with a 50 nm minimum feature size. To meet these requirements, the corresponding specifications for a direct-write pixel-based lithography system are shown on the right side of Table 1. A minimum feature size of 50 nm requires the use of 25 nm pixels. Sub-nanometer edge placement can be

* Correspondence: Email: vdai@eecs.berkeley.edu; WWW: http://www-video.eecs.berkeley.edu; Telephone: 510 643 1587

achieved using 5-bit gray pixels. A 10 mm × 20 mm chip then represents $\frac{10\ mm}{25\ nm} \times \frac{20\ mm}{25\ nm} \times \frac{5\ bits}{pixel} \approx 1.6\ Tb$ of data per chip. A 300mm wafer containing 350 copies of the chip, results in 560 Tb of data per layer per wafer. Thus, to expose one layer of an entire wafer in one minute requires a throughput of $\frac{560\ Tb}{60\ s} \approx 9.4\ Tb/s$. These tera-pixel writing rates force the adoption of a massively parallel writing strategy and system architecture. Moreover, physical limitations of the system place a severe restriction on the processing power, memory size, and data bandwidth.

Device specifications		Direct-write specifications	
Minimum feature	50 nm	Pixel size	25 nm
Edge placement	< 1nm	Pixel depth	5 bits / 32 gray
Chip size	10 mm × 20 mm	Chip data (one layer)	1.6 Tb
Wafer size	300 mm	Wafer data (one layer)	560 Tb
Writing time (one layer)	60 seconds	Data rate	9.4 Tb/s

Table 1. Specifications for the devices with 50 nm minimum features

2.1 Writing strategy

As shown in Figure 1, one candidate for a maskless lithography system uses a bank of 80,000 writers operating in parallel at 24 MHz.[11] These writers, stacked vertically in a column, would be swept horizontally across the wafer, writing a strip 2 mm in height. Although this results in a 60 second throughput for one layer of a wafer, the problem of providing data to this enormous array of writers still remains. In the remainder of this paper, we address issues related to the design of the system that takes the chip layout stored on disks, and brings it to the massive array of writers.

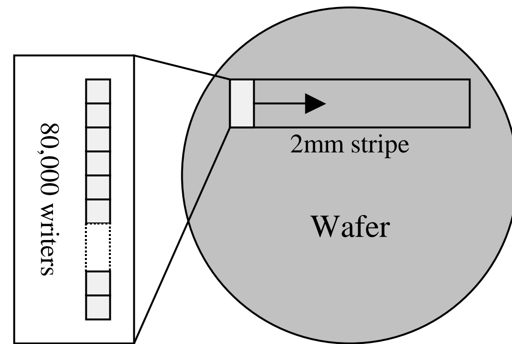

Figure 1. Hardware writing strategy

2.2 Data representation

An important issue intertwined with the overall system architecture is the appropriate choice of data representation at each stage of the system. The chip layout delivered to the 80,000 writers must be in the form of pixels. Hierarchical formats, such as those found in GDS-2 files, are compact as compared to the pixel representation. However, converting the hierarchal format to the pixels needed by the writers requires processing power to first flatten the hierarchy into polygons, and then to rasterize the polygons to pixels. An alternative is to use a less compact polygon representation, which would only require processing power to rasterize polygons to pixels. Flattening and rasterization are computationally expensive tasks requiring an enormous amount of processing and memory to perform. We will examine the use of all of these three representations in our proposed system: pixel, polygon, and hierarchical.

2.3 Architecture designs

The simplest design, as shown in Figure 2, is to connect the disks containing the layout directly to the writers. Here, we are forced to use a pixel representation because there is no processing available to rasterize polygons, or to flatten and rasterize hierarchical data. Based on the specifications, as presented in Table 1, the disks would need to output data at a rate of 9.4 Tb/s. Moreover, the bus that transfers this data to the on-chip hardware must

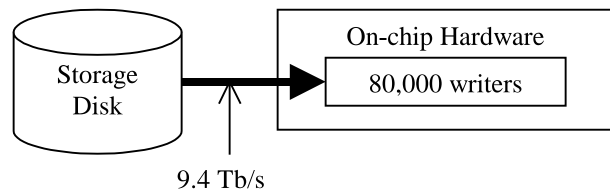

Figure 2. Direct connection from disk to writers

also carry 9.4 Tb/s of data. Clearly this design is infeasible because of the extremely high throughput requirements it places on storage disk technology.

The second design shown in Figure 3 attempts to solve the throughput problem by taking advantage of the fact that the chip layout is replicated many times over the wafer. Rather than sending the entire wafer image in one minute, the disks only output a single copy of the chip layout. This copy is stored in memory fabricated on the same substrate as the hardware

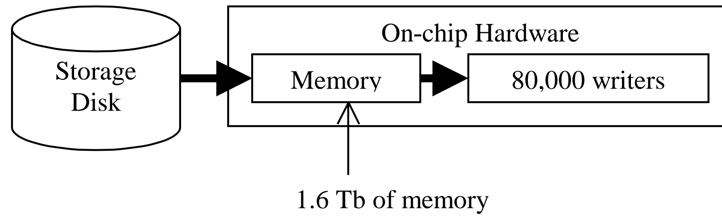

Figure 3. Storing a single layer of chip layout in on-chip memory

writers themselves, so as to provide data to the writers as they sweep across the wafer. Unfortunately, the entire chip image for one layer represents about 1.6 Tb of data, while the highest density DRAM chip available, we estimate will only be 16 Gb in size.[2] This design is therefore infeasible because of the extremely large amount of memory that must be present on the same die as the hardware writers.

It might appear to be possible to fit the chip layout in on-chip memory by either compressing the pixels, or using a compact representation such as the hierarchical or polygon representations mentioned in section 2.1. This requires additional processing circuitry to decompress the pixels, flatten the hierarchy or rasterize the polygons. In Figure 4, this processing

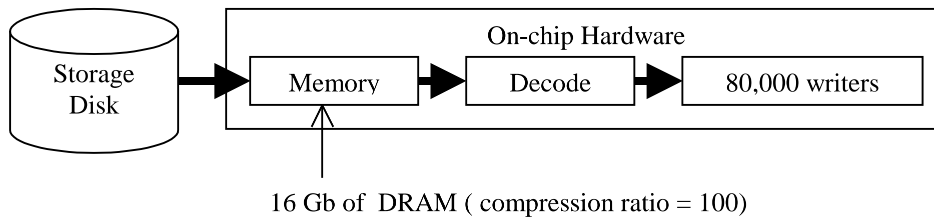

Figure 4. Storing a compressed layer of chip layout in on-chip memory

circuitry is called on-chip decode, and it shares die area with the on-chip memory and the writers. Even if all the on-chip area is devoted to memory, the maximum memory size that can be realistically built on the same substrate as the writers is about 16 Gb, resulting in a required compaction/compression ratio of about $\frac{1.6\ Tb}{16\ Gb} \approx 100$. However, this leaves no room for the added decode circuitry to be fabricated on the same die as the writers, thus making this approach infeasible. If we reduce the amount of memory to make room for the decode circuitry, we will need even higher compression ratios, resulting in more complex, larger decode circuitry.

To solve this memory-processing bottleneck it is possible to move the memory and decode off-chip onto a processor board, as shown in Figure 5. Now multiple memory chips can be available for storing chip layout data, and multiple processors can

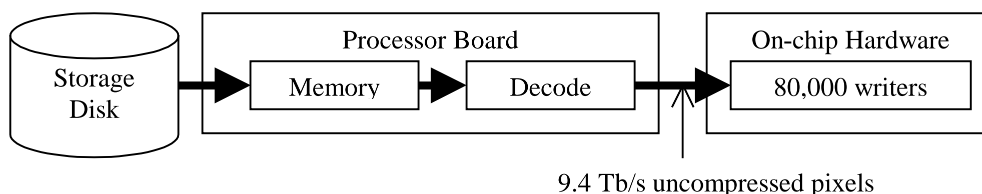

Figure 5. Moving memory and decode off-chip to a processor board

be available for performing decompression, rasterization, and even flattening. However, after decoding data into the bitmap pixel domain, we are again faced with a 9.4 Tb/s transfer of data from the processor board to the on-chip writers. We anticipate chips to have at most around 1000 pins, which can operate at about 400 MHz, limiting the throughput to the writers to at most 400 Gb/s. This represents about a factor of $\frac{9.4\ Tb/s}{400\ Gb/s} \approx 25$ difference, between the desired pixel data rate to the writers and the actual rates possible.

To overcome this problem, we propose to move the decode circuitry back on-chip as shown in Figure 6. Analyzing the system from the right to the left, it is possible to achieve the 9.4 Tb/s data transfer rate from the decoder to the writers

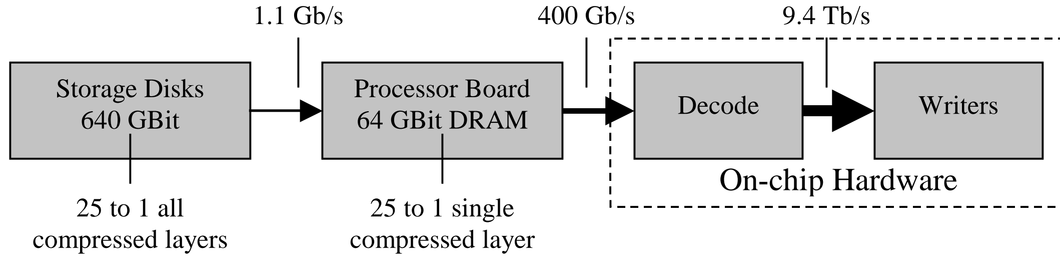

Figure 6. System archtecture

because they are connected with on-chip wiring, e.g. 80,000 wires operating at 25 MHz. The input to the decoder is limited to 400 Gb/s, limited by the amount of data that can go through the pins of a chip as mentioned previously. The data entering the on-chip decode at 400 Gb/s must, therefore, be compressed by at least 25 to 1, for the decoder to output 9.4 Tb/s. Because decoding circuitry is limited to the area of a single chip, it cannot perform complex operations such as flattening and rasterization. Thus, to the left of the on-chip decode, the system uses a 25 to 1 compressed pixel representation in the bitmap domain.

Given the 25 to 1 compressed pixel representation, the requirements for the rest of the system are relatively benign. As before, only one copy of the chip needs to be stored in the memory on the processor board, which is replicated hundreds of times as the wafer is written. In terms of uncompressed pixels, a chip layout represents about 1.6 Tb of information. Compressed by a factor of 25, the entire chip layout becomes $\frac{1.6\ Tb}{25} \approx 64\ Gb$ in size, and can be stored on multiple DRAM chips on the processor board. These DRAM chips must output 400 Gb/s, which could be accomplished, for example, with 125 DRAM chips each 32-bits wide operating at 100 MHz. Each DRAM chip would only be 512 Mb large, to satisfy total storage constraints. To supply data to the processor board, the storage disks need only output 64 Gb of compressed pixel data every minute, resulting in a transfer rate of $\frac{64\ Gb}{60\ s} \approx 1.1\ Gb/s$. Moreover, they need to store compressed pixel data for all layers of a chip, resulting in about 640 Gb of total storage for a 10-layer chip. These specifications are nearly within the capabilities of RAID systems today[5]. Our next goal, then, is to find a compression scheme that can compress layout in the pixel bitmap representation by a factor of 25.

3. DATA COMPRESSION

To find a compression scheme that can achieve a compression ratio of 25, we begin by apply existing lossless image compression techniques to rasterized chip layout. We have tested the performance of the context-based arithmetic coding scheme as implemented by the Joint Bi-Level Image Processing Group (JBIG) [12], and the adaptive-dictionary based technique of Ziv and Lempel (LZ77) [13] as implemented by popular compression packages (ZIP) [7]. In addition, we have devised and implemented a two-dimensional variant of the LZ77 algorithm (2D-LZ) and tested its compression performance against that of JBIG and ZIP.

3.1 JBIG and ZIP Compression

JBIG is a recent standard for lossless compression of bi-level images, developed jointly by the CCITT and ISO international standards bodies.[7] Optimized for compression of black and white images, JBIG can also be applied to gray images of about six bits per pixel, or sixty-four gray levels by encoding each bit plane separately, while maintaining compression efficiency. JBIG uses a ten-pixel context to estimate the probability of the next pixel being white or black. It then encodes the next pixel with an arithmetic coder based on that probability estimate. Assuming the probability estimate is reasonably accurate and heavily biased toward one color, as illustrated in Figure 7, the arithmetic coder can reduce the data rate to far below one bit per pixel. The more heavily biased toward one color, the more the rate can be reduced below one bit per pixel, and the greater the compression ratio.

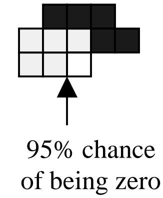

Figure 7. JBIG compression

ZIP is an implementation of the LZ77 compression method used in a variety of compression programs such as pkzip, zip, gzip, and winzip.[7] It is highly optimized in terms of both speed and compression efficiency. The ZIP algorithm treats the input as a stream of bytes, which in our case represents a consecutive string of eight pixels in raster scan order. To encode the next few bytes, it searches a window of up to 32 kilobytes of previously encoded characters to find the longest match to the next few bytes. If a long enough match is found, the match position and length is recorded; otherwise, a literal byte is encoded. For example, in Figure 8, on the first line, a match was found to “space,t,h,e” ten pixels back with a match length of four. On the second line, the only match available is the “s” which is too short. Therefore, a literal is generated instead. Literals and match lengths are encoded together using one Huffman code, and the match position is encoded using another Huffman code. Although the LZ77 algorithm was originally developed with text compression in mind, where recurring byte sequences represent recurring words, applied to image compression it can compress recurring sequences of pixels. In general, longer matches and frequent repetitions increase the compression ratio.

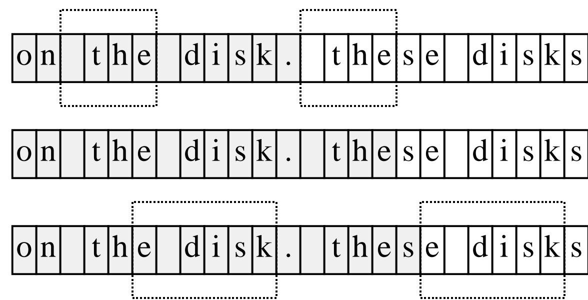

Figure 8. ZIP (LZ77) Compression

3.2 2D-LZ compression

We have extended the LZ77 algorithm to two dimensions, thereby taking advantage of the inherent two-dimensional nature of layout data, for the system architecture proposed in section 2. Pixels are still encoded using raster scan order. However, the linear search window, which appears in LZ77, is replaced with a rectangular search region of previously coded pixels. As illustrated in Figure 9, a match is now a rectangular *region*, specified with four coordinates: a pair of coordinates, *x* and *y*, specify the match position, and another pair of integers, *width* and *height*, specify the match. If a match of minimum size cannot be found, then a literal is outputted representing a vertical column of pixels. A sequence of control bits is also stored so the decoder can determine whether the output is a literal or a match. To further compress the output, five Huffman codes are used: one for each of the match coordinates *x*, *y*, *width*, *height*, and one for the literal. In order to find the largest match region, we exhaustively test each pixel in the search region (x,y). When a match at a particular (x,y) is found, we increase *width* as much as possible, while still ensuring a match; then we increase *height* as much as possible. This procedure

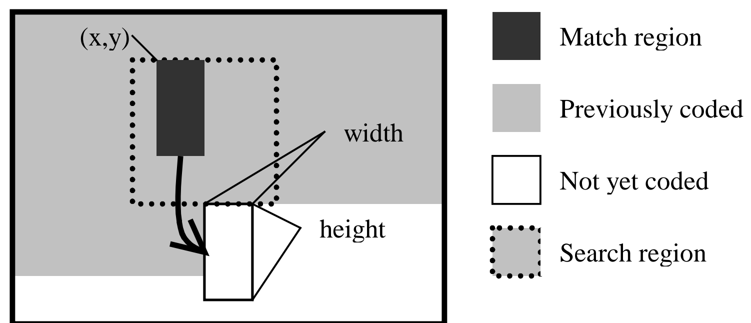

Figure 9. 2D-LZ Matching

guarantees the widest possible match size for a given match position. We then choose the match position that results in the largest match size and store this as the match region.

In our current implementation the search window is a 256×256 rectangular region centered above the pixel to be compressed. The match position is, therefore, specified in sixteen bits. The size of search region is chosen heuristically to achieve reasonable compression times and efficiency. A smaller search area yields less compression, because some potential matches may not be found. A larger search region dramatically increases the search time, which is proportional to the search region area. To improve compression efficiency, we allow the match region to extend beyond the search region, as shown in Figure 10; only the upper left hand corner of the match region actually needs to be inside the search region. By allowing these large matches, we can encode large repetitions in the layout data more effectively. Horizontally, the match *width* is limited to sixteen bits, or 65,536 pixels. Matches wider than this are uncommon, and can be encoded as two or more separate matches at a small cost to compression efficiency.

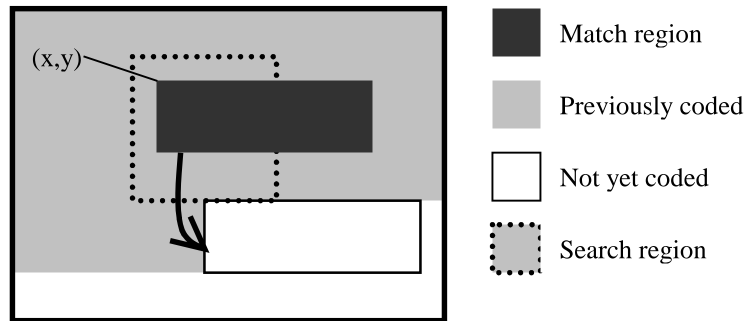

Figure 10. Extending the match region beyond the search region

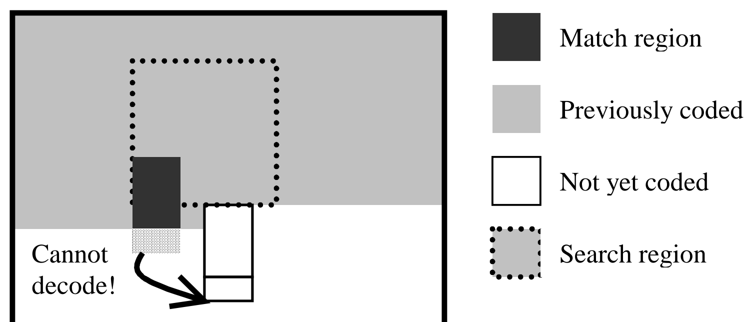

Figure 11. Cannot extend match region into not yet coded pixels

Vertically, the match height is often limited because match regions extending into not yet coded regions cannot be decoded, as shown in Figure 11. However, in the special case where the match position is centered directly above the pixel being coded, we can let the match region extend vertically into not yet coded areas, as illustrated in Figure 12. In Figure 12A, the gray area indicates the already encoded pixels, the white area indicates the pixels that are not coded yet, and the dotted rectangle denotes the search region where the upper left hand corner of the match region must reside. In Figure 12B, a large match, denoted by the rectangle, has been found centered directly above the pixel to be encoded. Note that the match region includes some pixels that are not yet coded, shown in light gray, which might be problematic when it comes time to decode. Figure 12C depicts the set of not yet coded pixels that the match region is matching to. Because the match region overlaps the region being matched to perfectly, it is possible to decode the light gray region completely first, as shown in Figure 12D, resulting in Figure 12E. Now that all the pixels in the match region have been decoded, the rest of the decoding can continue as normal as shown in Figure 12F. Tall matches such as these expose large vertical repetitions in the layout data. In these cases, the match *height* is also limited to sixteen bits, or 65,536 pixels. Our simulations with existing data indicate that matches taller than this are uncommon, and can be encoded as two or more separate matches at a small cost to compression efficiency.

Figure 12. Extending the match region into not yet coded pixels

Figure 13 illustrates another interesting facet of the 2D-LZ algorithm. It is entirely possible for the region to be encoded to cover previously coded pixels. Because these pixels have been encoded already, we do not check whether there is another match. These "don't care" pixels are always considered a match for the purpose of extending the match *width* and *height*, but they are subtracted from the match *size* when choosing the position that gives the largest match *size*. During decoding, the decoder also knows which pixels have been decoded previously, and will also ignore these "don't care" pixels.

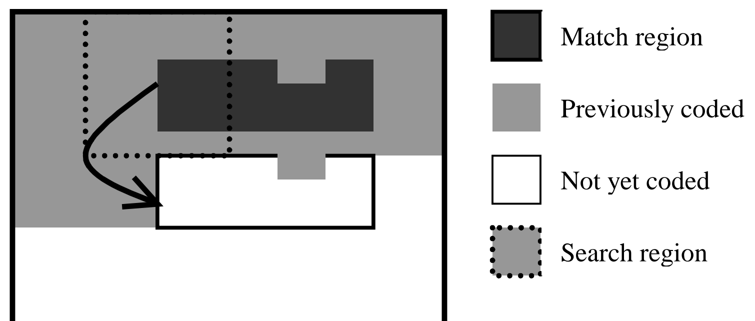

Figure 13. Holes in the rectangular match region

The decoding of 2D-LZ is simple. First the match region *x*, *y*, *width*, and *height*, and the literals are Huffman decoded. Similar to the encoder, the decoder also keeps a buffer of previously decoded pixels. The size of this

buffer must be large enough to contain the height of the search window and the width of the image for matching purposes. Each time a match is read, the decoder simply copies data from the corresponding match region among the previously decoded pixels and fills it in the not yet decoded area. “Don’t care” pixels, that is, pixels that have been previously decoded but appear in the match region, are discarded. If a literal is read, the decoder simply fills in a vertical column of pixels in the not yet coded area. The decoder does not need to perform any searches, and is therefore much simpler in design and implementation than the encoder.

4. COMPRESSION RESULTS

The results of our compression experiments for various layers of several layout types such as memory, control, and mixed logic are listed in Table 2. Memory cells tend to be dense and are composed of small, regularly repeated cells. Control logic is very irregular and somewhat less dense. Mixed logic comes from a section of a chip that contains both some memory cells and some glue logic intermingled with the cells. Compression ratios listed in bold in Table 2 are below the required compression ratio of 25, as suggested by the architecture presented in section 2.

Examining the third column of Table 2 reveals that JBIG performs well for compressing relatively sparse layout as in the control logic, mixed areas, and metal 2 layers. However, its performance suffers greatly in dense layout such as those found in memory cells. Even though the memory cells are very repetitive, JBIG’s limited ten-pixel context is not enough to model this repetition of cells. Theoretically we could increase the context size of the JBIG algorithm until it covers an entire cell. In practice, however, because the number of possible contexts increases exponentially with the number of context pixels, it is infeasible to use more than a few tens of pixels, whereas cells easily span hundreds of pixels.

Type	Layer	JBIG	ZIP	2D-LZ	2D-LZ ZIP
Memory Cells	Metal 2	58.7	88.0	94.6	171
	Metal 1	**9.77**	47.9	**22.9**	43.8
	Poly	**12.4**	50.7	35.8	64.9
Control Logic	Metal 2	47.0	**22.1**	25.1	**24.4**
	Metal 1	**20.0**	**10.9**	**11.9**	**11.2**
	Poly	41.6	**18.9**	**18.3**	**18.2**
Mixed	Metal 2	51.3	28.3	29.6	33.0
	Metal 1	**21.2**	**11.9**	**13.2**	**12.9**
	Poly	41.3	**22.9**	**23.5**	**24.9**
Large Area	Metal 1	35.5	26.3	43.8	44.5

Table 2. Compression ratios of JBIG, ZIP and 2D-LZ.

In contrast to JBIG, ZIP’s compression ratios in column four suggest that it is well suited to compressing dense repetitive layout data, exhibiting compression ratios of 50 or higher. Repetitive layout allows the ZIP algorithm to find plenty of long matches, which translates into large compression ratios. On the other hand, ZIP performs poorly on irregular layouts found in control and mixed logic. For these layouts, ZIP cannot find long matches, and frequently outputs literals, resulting in performance loss in these areas.

2D-LZ, as shown in the fifth column of Table 2, performs similarly to ZIP having been grounded in the same basic LZ77 scheme. As evidenced in the sixth column of Table 2, applying ZIP after 2D-LZ does increase the compression ratio by nearly a factor of two for memory layout, moreover, surpassing the performance of the basic ZIP algorithm. This increase suggests that the 2D-LZ algorithm can still be optimized further to improve its compression efficiency. The particular ZIP algorithm we use for our experiments is a commercial package on a desktop PC, and as such, has been finely tuned and highly optimized by many researchers and engineers over the years. Therefore, to improve the performance of 2D-LZ, it is necessary to replicate all of these optimizations from the ZIP implementation. While 2D-LZ algorithm described in section 3.2 already includes some of these improvements, clearly more work must be done to further optimize the 2D-LZ algorithm.

Examining the rows of Table 2, it is evident that while no single compression scheme has compression rations larger than 25 for all layouts, there exists at least one compression scheme with a ratio larger than 25 for most layouts. Thus, in most cases, we can achieve 25 to 1 compression by applying different compression schemes to different types of layout. Even for the rows where this fails, i.e. metal 1 control logic and metal 1 mixed layouts, JBIG still achieves a compression ratio of 20, which is very close to the desired compression ratio of 25. From an architectural point of view, the drawback of using different schemes for different layouts is that all the decoders, for all compression algorithms used, must be implemented in

hardware, making it more difficult to fit the decoding circuitry on the same substrate as the writers. Alternatively different layers can be written with different writers, each writer implementing the single best compression technique for that layer. Finally, to accommodate the large variation of compression ratios for different layouts, it is possible to write the layers at different speeds.

4.1 Dependency of compression ratio on region size

Compression ratios in the first nine rows of Table 2 are based on a 2048-pixel wide and 2048-pixel tall section of a chip. This height of the compressed section, 2048, is chosen to have approximately the same chip coverage experienced by the 80,000 writers. As described in section 2.1, 80,000 writers write a 2 mm stripe across the wafer, which, for a 10mm tall chip, is one-fifth of the chip's height. Since the layout data we test is from a chip only 12,000 pixels tall, the height of the section we compress, 2048, is approximately a fifth of our chip's height.

The width of the section compressed, 2048, is chosen more arbitrarily. All of the algorithms presented are adaptive, and need to process a small amount of setup data before reaching their peak compression efficiency. An effort was made so that there would be enough data to overcome initial adaptation overhead of the three compression algorithms. On the other hand, the system architecture presented at the end section 2 requires achieving a consistent compression ratio of 25 to 1 across different regions of a given layer of a chip. On a typical processor chip, for example, a horizontal strip across the chip may encounter several different types of circuits, including arrayed memory cells, control logic, wiring areas, and glue logic. The compression algorithm must maintain a 25 to 1 compression as the writers pass over each of these sections, or else on chip buffers are needed to smooth out variations in compression ratio from one region to another. Although not rigorously tested, we have found a section width of 2048 to be large enough to absorb the adaptation overhead, while small enough to achieve consistent compression performance for relatively small buffer sizes. Further investigation is needed to understand the precise relationship between section width, buffer size, compression ratio variations, and adaptation overhead.

One interesting result to note is the last row of Table 2. The large area metal 1 layout compressed here is an 8,192-pixel wide and 8,192-pixel tall section of a chip that is 10,000 pixels wide and 12,000 pixels tall. As such, it covers a large percentage of the chip including a large portion of memory cells, a small portion of control, and a portion of the pad area. The compression ratio here is representative of the average compression that can be achieved with each of the three schemes if there are no striping and buffer constraints as mentioned earlier. Here the performance of 2D-LZ stands out above that of JBIG and ZIP. 2D-LZ compresses memory cells much better than JBIG, and these cells occupy the majority of this chip layout. In addition, the two-dimensional nature of the 2D-LZ algorithm allows it to better exploit the two-dimensional correlations found in layout than ZIP can. Issues related to optimum buffer size and section size are wide open for future studies.

4.2 Decode complexity

A key consideration for the architecture proposed in section 2.3 is the decoding complexity of the three compression algorithms. JBIG decoders must maintain information about each of the 1024 contexts and update context probabilities in the same way as the encoder, and they must perform additions, bit-shifts, and comparisons to decode an the arithmetic code. Moreover, these operations must be performed for every bit. In contrast, both ZIP and 2D-LZ require mostly memory copying to fill in match information. To perform Huffman decoding, an adder, comparator, and single-bit shifter is necessary. However, these operations are only performed for every match block, rather than every bit. One drawback of the ZIP and 2D-LZ is that a large buffer of previously decoded pixels must be maintained for the purpose of decoding matches. While we have not performed extensive decode complexity tests and simulations, it might be worthwhile to report decoding times for JBIG, and ZIP. On a 600 MHz Intel Pentium III PC, running Windows NT 4.0, decoding a typical 2048×2048 region for JBIG requires about 3 seconds. On the same computer, the decoding of ZIP takes less than a second. 2D-LZ code has not yet been optimized for speed, and as such, we cannot currently report on its speed performance.

5. SUMMARY, CONCLUSION AND FUTURE WORK

We have proposed a data processing system architecture for next generation direct-write lithography, consisting of storage disks, a processor board, and decode circuitry fabricated on the same chip as the hardware writers, as shown in Figure 6. In our design, the pattern of an entire chip, compressed off-line, is stored on disk. These disks provide large permanent storage, but only low data throughput to the processor board. When the chip pattern needs to be written to wafer, only a single compressed layer is transferred to the processor board and stored there in DRAM memory. As the writers write a stripe

across the wafer, the processor board provides, in real-time, the necessary compressed data to on-chip hardware. In turn, the on-chip hardware decodes this data in real-time, and provides uncompressed pixel data to drive the writers. The critical bottleneck of this design lies in the transfer of data from the processor board to the on-chip hardware, which is limited in throughput to 400 Gb/s by the number of pins on the chip, e.g. 1,000 pins operating at 400 MHz. Another critical bottleneck is the real-time decode that must be done on-chip, which precludes such complex operations as rasterization. Considering that the writers require about ten terabits per second of data, and the processor board can deliver at most 400 Gb/s to the on-chip hardware, we estimate that a compression ratio of 25 is necessary to achieve the data rates desired.

To achieve this compression ratio, we have studied three compression algorithms and applied them to the problem of lossless layout compression for maskless lithography. JBIG, a compression standard developed for bi-level images, performs well for non-dense layout. However for dense, regularly arrayed memory cells, its performance is hampered by the limited ten-pixel context, which is not sufficient to model the repetition of large thousand pixel cells. On the other hand, ZIP, based on LZ77, takes full advantage of repetitions to compress memory cells, but performs poorly in non-regular layout. Our 2D-LZ improves on the basic LZ77 technique, by extending matching to two-dimensions. Several refinements of the basic 2D-LZ technique are implemented to improve compression performance, and there is reason to believe further refinements can further improve performance. For all the different layouts tested, at least one of the three compression schemes is able to achieve a compression ratio of at least 20. For most of the layouts, the compression ratio is greater than 25 for at least one of the schemes, demonstrating the feasibility of the proposed system architecture.

Nonetheless, the challenge remains to develop a single compression technique that can consistently achieve a compression ratio of 25 and higher, with as simple as possible decode complexity. We are currently investigating improvements to the 2D-LZ algorithm that can further improve its compression efficiency. In the future, we plan to investigate implementation issues related to decoder of each of the schemes presented. We also plan to explore representations that are more compact than the pixel bitmap representation, yet easily rasterizable so that the rasterization circuitry may fit in limited on-chip decode circuitry. Ultimately our goal is to understand the fundamental limits of compressing layout, and to analyze the tradeoff between compression efficiency and decode complexity.

ACKNOWLEDGEMENT

This work was conducted under the Research Network for Advanced Lithography, supported jointly by the Semiconductor Research Corporation and the Defense Advanced Research Project Agency. We would like to give special thanks to Uli Hofmann and Teri Stivers of Etec Systems, Inc. for helping us understand the complexity issues associated with bringing layout data to an array of mask writers.

REFERENCES

1. M. Gesley, "Mask patterning challenges for device fabrication below 100 nm", Microelectronic Engineering **41/42**, pp. 7-14, 1998.

2. *The National Technology Roadmap for Semiconductors, 1997 Edition*, Semiconductor Industry Association, San Jose, CA, 1997.

3. K. Keeton, R. Arpaci-Dusseau, D. A. Patterson, "IRAM and SmartSIMM: Overcoming the I/O Bus Bottleneck", *Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, International Symposium on Computer Architecture, 1997.

4. E. H. Laine, P. M. O'Leary, "IBM Chip Packaging Roadmap", *International Packaging Strategy Symposium*, SEMICON West, 1999.

5. "IBM fibre channel RAID storage server", IBM Corporation, 1999.

6. K. Sayood, *Introduction to Data Compression*, pp. 87-93, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996

7. A. Moffat, T. C. Bell, I. H. Witten, "Lossless compression for text and images", International Journal of High Speed Electronics and Systems **8** (1), pp. 179-231, 1997.

8. E. I. Ageenko, P. Franti, "Enhanced JBIG-based compression for satisfying objectives of engineering document management system", *Optical Engineering* **37** (5), pp. 1530-1538, SPIE, 1998.

9. R. Veltman, L. Ashida, "Geometrical library recognition for mask data compression", *Proceedings of the SPIE – The International Society of Optical Engineering* **2793**, pp.418-426, SPIE, 1996.

10. H. Yuanfu, W. Xunsen, "The methods of improving the compression ratio of LZ77 family data compression algorithms", *1996 3rd International Conference on Signal Processing Proceedings*, pp. 698-701, IEEE, New York, 1996.

11. 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.

12. CCITT, ITU-T Rec. T.82 & ISO/IEC 11544:1993, Information Technology – Coded Representation of Picture and Audio Information – Progress Bi-Level Image Compression, 1993.

13. J. Ziv, A. Lempel, "A universal algorithm for sequential data compression", *IEEE Trans. On Information Theory* **IT-30** (2), pp. 306-315, IEEE, 1984.