Lossless Circuit Layout Image Compression Algorithms for Multiple Electron Beam Direct Write Lithography Systems

by

Jeehong Yang

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Electrical Engineering: Systems)
in The University of Michigan
2012

Doctoral Committee:

Associate Professor Achilleas Anastasopoulos, Co-Chair
Associate Professor Serap A. Savari, Texas A&M University, Co-Chair
Professor L. Jay Guo
Professor Andrew E. Yagle
Associate Professor Elizaveta Levina

© Jeehong Yang 2012
All Rights Reserved

To my parents.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Professor Serap A. Savari for her extraordinary support and guidance during my graduate studies. She guided me with patience and full-hearted support. She carefully reviewed all my works and write-ups as well as connected me with people who helped me. None of my achievements would have been possible without her.

I am also grateful to my committee members: Professor Achilleas Anastasopoulos, Professor Andrew E. Yagle, Professor L. Jay Guo, and Professor Elizaveta Levina for their valuable suggestions and comments. I am incredibly lucky to have them on my committee.

I am thankful to Professor Sunil P. Khatri and Professor H. Rusty Harris at Texas A&M University for their advice and help which initialized this research. I would also like to thank Vito Dai and Alan Gu for their valuable comments towards the research as well as sharing their source code with me.

My special thanks go to my family – my parents, my wife, and my daughter. I am indebted to my family for their unconditional love and support throughout my life. It would not have been possible to stand where I am now without them. Finally, I am grateful to all my friends who prayed for me during all these years and thank God for His grace and faithfulness.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF ABBREVIATIONS

ASIC Application Specific Integrated Circuits

BFSK Binary Frequency Shift Keying

BIP Binary Integer Programming

C4 Context-Copy-Combinatorial Coding

CPU Central Processing Unit

CAD Computer Aided Design

DPG Digital Pattern Generator

DW Direct-Write

EBL Electron Beam Lithography

EOB End-Of-Block

FPGA Field Programmable Gate Array

FPR Frequent Pattern Replacement

GPU Graphics Processing Unit

GP-GPU General Purpose Graphics Processing Unit (GPU)

HCC Hierarchical Combinatorial Coding

HVM High Volume Manufacturing

JBIG Joint Binary Image Group

LER Line-Edge Roughness

LUT Look-Up Tables

LZ Lempel-Ziv

MEB Multiple Electron Beam

MEMS Micro-Electro-Mechanical Structures

ML2 Maskless Lithography

NGL Next Generation Lithography

OPC Optical Proximity Correction

REBL Reflective Electron Beam Lithography

RLE Run-Length Encoding

VLSI Very Large Scale Integrated Circuits

ABSTRACT

Lossless Circuit Layout Image Compression Algorithms for Multiple Electron Beam Direct Write Lithography Systems

by

Jeehong Yang

Co-Chairs: Achilleas Anastasopoulos and Serap A. Savari

As technology develops, electronic devices are becoming faster, more power efficient, and smaller. All of these technological advances were possible because improvements in photolithography processes enabled the fabrication of smaller microelectronic circuits.

In order to continue these technological advances, many engineers have been introducing alternative lithographical methods. Among them, Multiple Electron Beam (MEB) is considered a strong candidate because of its high resolution as well as cost efficiency. However, there are more problems that we have to solve before MEB can replace conventional lithography systems, and one of these is the data delivery issue. For MEB systems to maintain sufficient throughput, many bits must be transmitted simultaneously to the electron beam writer array. This raises the question of how to provide the massive layout image data (several hundred terabits) to the MEB systems. Because of a bandwidth shortage between the storage where the layer images are deposited and the MEB system, obtaining competitive throughput using

a MEB system is not possible with conventional data delivery methods.

In this thesis, we introduce a data delivery system using lossless image compression to solve the data delivery issue. By transmitting a compressed layout image and quickly decompressing it on-the-fly at the e-beam writer array of an MEB system, we can transmit the huge layout image through a bandwidth limited channel.

Our compression algorithm is inspired by the compactness of the GDSII/OASIS format and is designed to take advantage of ideas like corner representation and the copying of repeated structures. However, we avoid the complex flattening and rasterizing processes and offer a simple decoding process. In order to take advantage of the repeated structures, we propose an algorithm that discovers the frequent structures from the layout description (GDSII) as well as the layout image and replace the discovered structures with a simpler representation. In order to make an efficient corner representation while maintaining a simple decoding process, we introduce a transformation which represents the corner points efficiently combined with an entropy encoder. The proposed compression algorithm provides a high compression performance while having a simple decoder architecture which enables the decoding process to be handled as an add-on hardware.

CHAPTER I

Introduction

As technology develops, we have been able to manufacture integrated circuits having smaller features. That development means that we have been able to integrate more transistors in the same area, run more complex computations at a time, and enable the circuit to operate with less power. Because of these technology innovations, we are now living in a world where mobile computing is everywhere. All of these achievements would have been impossible without improvements in lithography technologies. Conventional lithography technology is reaching its limit and for further development alternative lithography technologies have been considered [1].

To overcome this issue and advance lithography technology, many scientists have been investigating alternative lithographical methods. Among them, Electron Beam Lithography (Electron Beam Lithography (EBL)) is considered as a useful candidate [2]. EBL consists of three parts [3]: 1) a digital layout image which is stored at the storage system, 2) a data delivery path through which the layout image is transmitted, and 3) an electron beam writer which writes the transmitted layout image on the photoresist using one or more electron beams. Unlike other lithographic methods where the layout image has to be produced in a physical form such as physical masks, EBL systems do not require physical masks. Instead of masking the layout image patterns from the light source, an EBL system writes the layout image digitally

pixel-by-pixel using electron beams.

EBL is a strong candidate for the Next Generation Lithography (NGL) because EBL systems have a number of advantages over conventional photolithography systems [4]: 1) EBL systems are well-known to obtain *very high resolutions*. By using the electron beam as its light source, EBL offers far better resolution than what conventional photolithography systems can offer. 2) EBL systems *do not require physical masks*. Conventional photolithography requires high quality physical masks which are very expensive [5] to fabricate and maintain [6]. However, EBL systems do not require masks, and the software controlled e-beam writer instead writes the mask pattern directly to the photoresist layer. Moreover, because of the maskless feature, the circuit layouts EBL systems write can easily be modified resulting in simpler prototyping [7].

However, EBL systems have a drawback over physical mask lithography systems: they are *very slow* [1, 7]. Because EBL systems write layout images pixel by pixel and writing each pixel requires some time for the photoresist to react, the throughput of EBL is extremely low and is hence not suitable for the mass production of circuits.

Over the decades, scientists have been trying to solve this problem and are recently attacking the problem by applying *multiple* electron beam writers to the system [8, 9, 10]. By writing multiple pixels at a time, it is possible to decrease the writing time and increase the throughput. Furthermore, by carefully selecting the number of electron beam writers of the EBL systems, it is possible to match the throughput of conventional photolithography systems. Many innovative concepts regarding multiple electron beam lithography have been conceived and are being developed for various applications, such as mask writing, prototyping, writing critical layers in High Volume Manufacturing (HVM), and writing all layers in HVM. Recently, it was shown that Multiple Electron Beam (MEB) systems targeting the Direct-Write (DW) application, i.e., writing all layers of a circuit using MEB, is the most economical option for the

next generation lithography technology, especially for 450 mm wafer technology [2].

There are still a few more problems that we have to solve before MEB can replace conventional photolithography systems, and one of these is the data delivery issue [3]. For MEB systems to maintain sufficient throughput, many bits must be simultaneously transmitted to the array of electron beam writers. This raises a question on how to provide the massive layout image data (which is typically several hundred terabits per wafer) to the MEB systems. Because of a bandwidth shortage between the storage where the layer images are deposited and the MEB system, obtaining competitive throughput using a MEB system is not possible with conventional data delivery methods.

In this thesis, we will approach the MEB data delivery problem using data compression. Data compression is a classical way to deal with data transfer on bandwidth limited channels and is a natural approach to this type of problem. Throughout the thesis, we describe data compression algorithms that are tailored to compress layout images that are to be used for MEB systems. In order to propose a data compression algorithm, we start by analyzing the characteristics of layout images and design a compression algorithm that could take advantage of such characteristics while enabling the decoder to decompress it with the restrictions it has.

This thesis consists of seven chapters. In the remaining part of Chapter I, we will describe how EBL / MEB systems work, why data compression is necessary for the data delivery system, and how layout images are generated. In Chapter II, we will summarize the related work which motivated this research. We will discuss the properties of layout images and how others dealt with those properties. In Chapter III, we introduce the `Corner2` algorithm which was designed to compress binary layout images using the characteristics we derived from Chapter II. In Chapter IV, we illustrate the proof-of-concept Field Programmable Gate Array (FPGA) implementation of the `Corner2` decoder verifying that `Corner2` is suitable for the data delivery sys-

tem. In Chapter V, we introduce `Corner2-BIP` which improves the frequent pattern discovery algorithm of `Corner2` using binary integer programming. In Chapter VI, we illustrate `Corner2-MEB` by modifying `Corner2-BIP` so that it is suitable for the MAPPER [8] system. Finally, we conclude with future research directions in Chapter VII.

1.1 Electron Beam Lithography

Before describing EBL, we will discuss conventional photolithography processes and compare EBL processes to them; i.e., we explore the strengths and weaknesses of EBL compared to conventional photolithography.

1.1.1 Conventional Photolithography

The conventional photolithography process is depicted in Figure 1.1. First the wafer is coated uniformly using a chemical called "photo-resist." Second, after the photo-resist is covered with the mask of the circuit layout image that we wish to fabricate, it is exposed to light. The photo-resist then undergoes a chemical change making it soluble to a photo-resist developer. Third, this photo-resist developer is applied to the photo-resist so that the photo-resist only covers the parts of the wafer which were covered by the mask. Fourth, the uppermost layer of the substrate in the areas that are not covered by photo-resist is removed (level change is shown with light gray). Finally, the photo-resist is removed and the result of this process is a wafer etched by the mask pattern. Since the photolithography process adds one layer at a time to the wafer, the process is repeated for each circuit layer.

In short, conventional photolithography is much like letterpress printing where the physical mask serve as the letterpress, the light source acts like ink, and exposure is analogous to press. As letterpress printing requires separate letterpress for each page of the book, a physical mask is required for each circuit layer in photolithography.

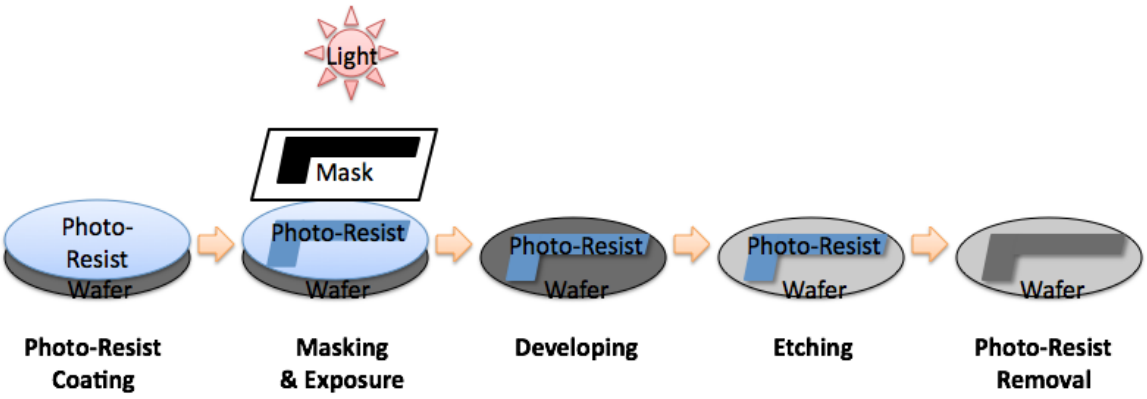

Figure 1.1: Conventional Photolithography Process

Similarly to letterpress printing, conventional photolithography is suitable for HVM because the physical mask take cares of the complex pattern generation making the entire printing process simple and fast.

However, conventional photolithography also has disadvantages: Producing physical masks for smaller circuits is becoming harder and more expensive. The current mask set cost is over 5 million US dollars [5] because the physical mask itself has to shrink and the mask pattern becomes more complex in order to handle the optical proximity effect. Moreover, as we target smaller circuits, maintaining - cleaning and storing - the physical masks becomes nontrivial [6].

Finally, conventional photolithography does not have limitless resolution as there is a limit on the smallest feature size that we can print due to the optical proximity effect [11]. Because of this limit, many researchers have been searching for alternative lithography technologies. Among them, EBL is considered as a strong candidate.

1.1.2 Electron Beam Lithography

EBL consists of three parts as shown in Figure 1.2 [3]: 1) a digital layout image which is stored at the storage system, 2) a data delivery path through which the layout image is transmitted, and 3) an electron beam writer which writes the transmitted layout image on the photoresist using an electron beam. Unlike conventional

photolithography where the layout image has to be made in a physical form such as physical mask, EBL systems do not require a physical mask. Because of this reason, EBL is sometimes called Maskless Lithography (ML2). Instead of masking the layout image patterns from the light source, an EBL system writes the layout image digitally pixel-by-pixel using the electron beam writer.

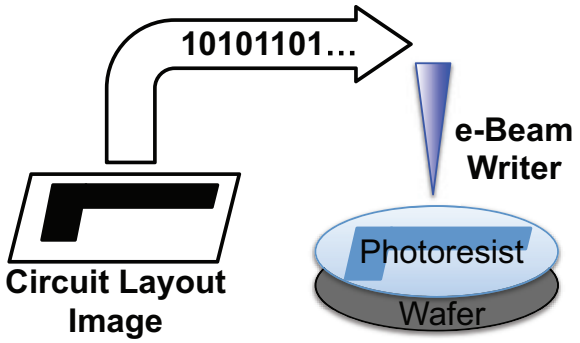

Figure 1.2: Electron Beam Lithography Systems

In short, EBL is analogous to inkjet printing where the electron beam writer serves as the nozzle and the electrons as the ink. As an inkjet printer controls the nozzle whether to release ink to the paper or not, an electron beam writer controls whether the electron beam should expose the corresponding photo-resist area or not. Just as inkjet printers can print the digital input, EBL can print any layout image transmitted to the electron beam writer. However, unlike inkjet printers, the EBL tools are very expensive (around 50 million US dollars) [5].

EBL systems have a number of advantages over conventional photolithography systems:

- EBL systems are well known to obtain *very high resolutions*. Unlike other lithography technologies, the achievable minimum feature size of EBL is not limited by light diffraction. Because of the property, it is well known that we can obtain very high resolutions (or very small features) using EBL. For example, images in resist of 10 *nm* have been demonstrated [12] and with alternative techniques

resolutions down to 3 *nm* have been demonstrated [13]. By using the electron as its light source, EBL offers far better resolution than what conventional photolithography systems can offer. In fact, it gives the finest resolution among all particle-based (e.g., photon, proton, ion, or plasma) lithography tools.

- EBL systems *do not require physical masks*. As we explained earlier, conventional photolithography requires high quality physical masks which have high costs to both fabricate and maintain. However, EBL writes the mask pattern directly to the photo-resist using the software controlled e-beam writer. That means as far as one sets up an EBL system, it can fabricate any circuit (targeting the same developing technology), just by changing the circuit mask image in the input (or control) system. Moreover, because of the maskless feature, EBL is suitable for rapid prototyping [7].

However, there is a significant drawback in EBL and that has so far prohibited EBL to become the major lithography technology. That is its *low throughput* [1, 7]. For this reason, most current EBL applications concentrate on prototyping sophisticated devices as in Figure 1.3 [7]. The figure plots an EBL application as a function of the minimum feature size and writing speed. Depending on these parameters the applications can be categorized into four types:

1. high resolution research applications, including nanometer scale structures for basic physics research, single electron devices, quantum dots, and quantum wires,

2. the manufacturing of high frequency devices for communications, integrated optics, and Micro-Electro-Mechanical Structures (MEMS),

3. Very Large Scale Integrated Circuits (VLSI) prototyping, manufacturing of sparse critical levels, and Application Specific Integrated Circuits (ASIC), and

4. mainstream VLSI manufacturing.

Among these applications, all except the last application are widely practiced using EBL [7].

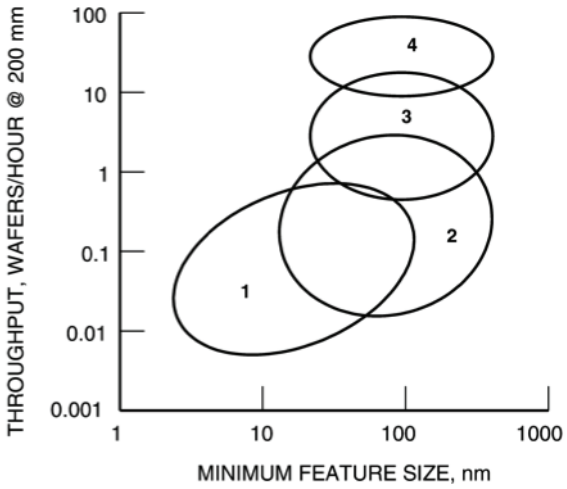

Figure 1.3: Applications of Electron Beam Lithography

Traditionally, EBL draws the mask image to the resist pixel-by-pixel and this is extremely slow when compared to conventional photolithography. Moreover, just as the printing speed decreases when we increase the resolution (or DPI) of inkjet printing, the EBL throughput likewise decreases when it targets smaller circuits because as we increase the resolution by making the pixel size smaller, the number of pixels EBL has to write in the same area increases. Low throughput (or slow writing speed) is because the electron dose must be high enough to affect the photo-resist layer. For example, if the dose to resist is 2 $\mu C/cm^2$, it means a 30 nm x 30 nm pixel will become resist only if 180 nC or more charge hits the pixel area. Therefore, the easiest way to increase the throughput of EBL is to make the photo-resist more sensible and/or increase the writer current so that more charges hit the area at the same time. However, neither of them is practical for real applications because a more sensible photo-resist results in noisier image (just as high ISO pictures occur with more noise in digital cameras) and increasing the writer current results in a blurry

beam by the physical phenomenon called the *Coulomb interaction* [14].

In order to produce a highly concentrated electron beam, EBL operates in high voltage (5∼100kV). Since it is impossible to switch on and off a high voltage device at the speed that is suitable for EBL applications, controlling the electron beam exposure by turning it on and off is not desirable. Instead an electronically controlled *blanker* [15] is used to pass or block the electron beam. This blanker can be considered as an optical switch as shown in Figure 1.4 [15]. In Figure 1.4, the electron beam enters the blanker from the top and exits from the bottom. If no electric field was induced in the blanker, the electron beam simply reaches the bottom. But if an electric field is induced, the electron beam makes a curve to reach the aperture which blocks it from escaping the blanker.

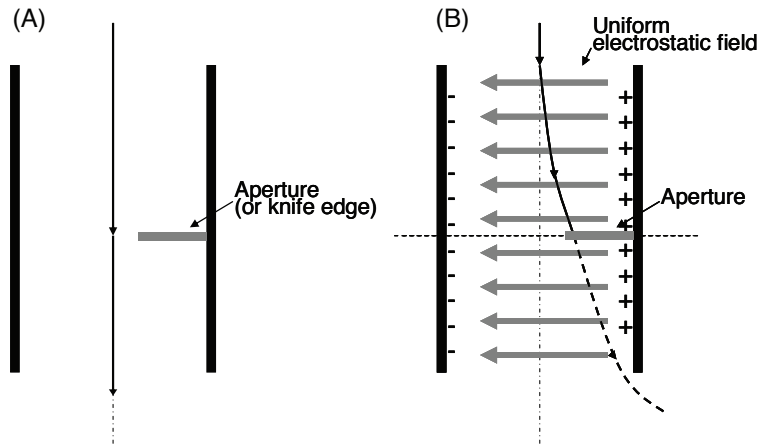

Figure 1.4: Basic design of blanker

Because the blanker could operate at a high speed, it is possible to use it to control the electron beam and obtain high throughput. However, at high speeds the electron beam exposure time becomes too short for the photo-resist to react. Therefore a high speed blanker combined with high sensitivity photo-resist and high current electron beams is a direction of future research for EBL systems. Technically speaking, because the blanker is sufficiently fast, if there exists a data delivery system that can support the data rate the blanker requires, then the EBL throughput is limited by the photo-

resist sensitivity and the electron beam current. Because of that, these two parameter contributes to the wafer writing time which is as follows [15]:

$$t = NAD/j \tag{1.1}$$

where N is the number of chips per wafer, A is the area of a chip, D is the dose to resist[1], and j is the beam current density.

1.2 Multiple Electron Beam Lithography

As we have mentioned in the previous section, neither making the photo-resist more sensitive nor increasing the current density is suitable for practical applications. Therefore, the best way to reduce the EBL writing time is by adding more writers and processing them in *parallel*. This is similar to Central Processing Unit (CPU) development in that parallel architecture became critical when the CPU clock reached its limit. By writing multiple pixels at a time, it is possible to decrease the writing time and increase the throughput. Furthermore, by carefully selecting the number of electron beam writers of the EBL systems, Multiple Electron Beam (MEB) lithography systems are anticipated to match the throughput of conventional photolithography systems.

Because MEB inherits EBL, it has high resolution and is maskless, while it also has high throughput making it a strong candidate for next generation lithography. In fact, Lin [2] claims that for writing all layers of a circuit for HVM, MEB is the most economical option for the next generation lithography technology especially when a 450 mm wafer is considered.

However, in order to do that, all the writers should work simultaneously in parallel: we have to generate e-beams, position the beams, and apply digital masking

[1]The charge dose that is required to change the photo-resist status. Usually in $\mu C/cm^2$.

simultaneously. Generating an array of e-beams has been studied throughly [16] but will not be discussed in detail in this thesis because it is outside of the main scope of our work. In short, a single light source sprays the light and using a special lens, the sprayed beams are focused to make an array of electron beams. Then each beam passes through a digital masking circuit which decides whether the corresponding pixel should be exposed or not.

There are two ways to control the writers and those controllers are called *blanker* and *Digital Pattern Generator (DPG)*. Blanker is used for MAPPER [8], and DPG is used for Reflective Electron Beam Lithography (REBL) [10]. We concentrate here these MEB systems because they represent the two main MEB system architectures that are actively under development at present. Details of these systems are out of scope of this thesis and only how the data is handled in the MEB systems will be outlined in the following subsections.

1.2.1 Reflective Electron Beam Lithography

An overview of the Reflective Electron Beam Lithography (REBL) [10] system is shown in Figure 1.5. REBL has an electron beam source called an electron gun. The array of electron beams generated by the electron gun are masked by a Digital Pattern Generator (DPG), and the masked electron beams hit the resist. The DPG serves as an electron mirror where it can absorb the corresponding electron beam or reflect it back to where it came from. Using the illumination optics, the electron beam induced by the electron gun is turned to hit the DPG, and the electron beam that was reflected by DPG goes straight down to hit the wafer. The DPG consists of 1 million pixels (4096×248) which control 1 million electron beams to simultaneously write a block (corresponding to 4096×248 pixels) of each wafer. Note that REBL does not write a block and stop, move the stage (or shift the wafer), and write a block again. Instead, REBL writes the block continuously while the stage is moving.

Because of this reason, the wafer is covered multiple time (in an integral form) by the REBL writer. In fact, the DPG was asymmetric in order to shorten the integral effect on the writing direction.

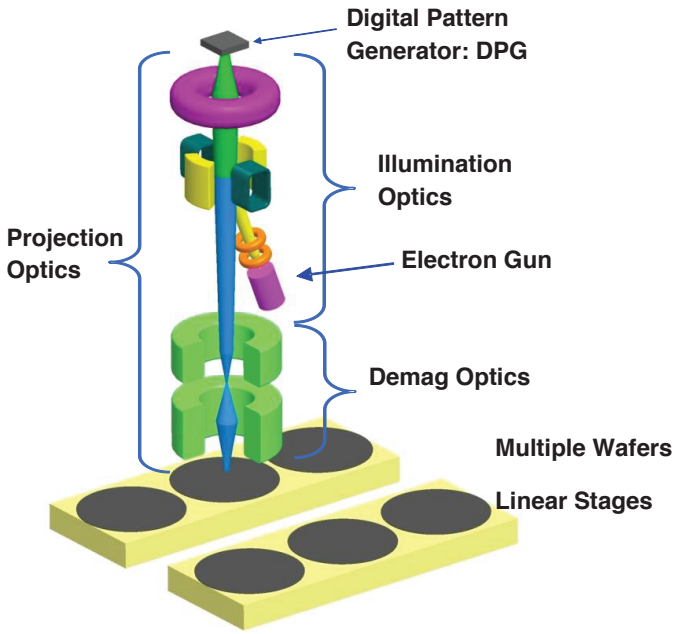

Figure 1.5: REBL System Overview

For HVM, a REBL system applies linear stage as in Figure 1.6. That is, there is more than one REBL system with each writing more than one wafer. For the example in Figure 1.6, there are six REBL platforms with each platform containing six REBL writers covering some number of wafers. Each REBL platform is designed to write multiple wafers because in order to cover the entire wafer using a REBL platform, the stage has to move back-and-forth (or in this case, left-and-right). However, changing direction requires the stage to reduce speed making the throughput lower. Therefore, in order to prevent that throughput decrement, we write as many wafers as possible before the stage changes its direction.

Currently the REBL system is targeting 45 nm nodes [10] at 5–7 wafer layers per hour, and is extending the technology to 16 nm half-pitch nodes and beyond using the HVM setting as in Figure 1.6. It is reported that the DPG would have to handle

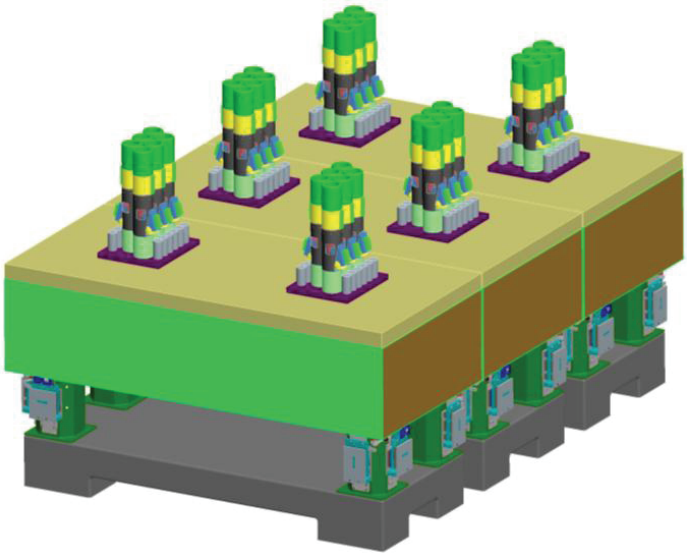

Figure 1.6: REBL HVM Setting using Linear Stage

up to 20 Tbps of data [10] and with the help of a data compression algorithm [17] they were able to reduce the required data rate to 1 Tbps.

1.2.2 MAPPER

An overview of the MAPPER system [8] is shown in Figure 1.7. An electron source generates the array of electron beams and the beams are individually focused by multiple lenses. Finally the e-beam array passes through the blanker array where it is passed to hit the wafer or it is blocked depending on the control signal. Each blanker of the blanker array is controlled by an optical sensor and the red lasers reaching the blanker array diagonally in the Figure 1.7 are the control signals.

Unlike REBL systems where a group of (1 million) electron beams forms a large writing region, MAPPER allows each electron beam to cover a region independently. The blankers in the blanker array are aligned as in Figure 1.8. The blankers are separated from each other in a staggered grid. This spacing is necessary to avoid the Coulomb interaction and the staggering is used so that it has to consider fewer stitching issues. Each blanker writes a stripe of 2 μm height from left to right in

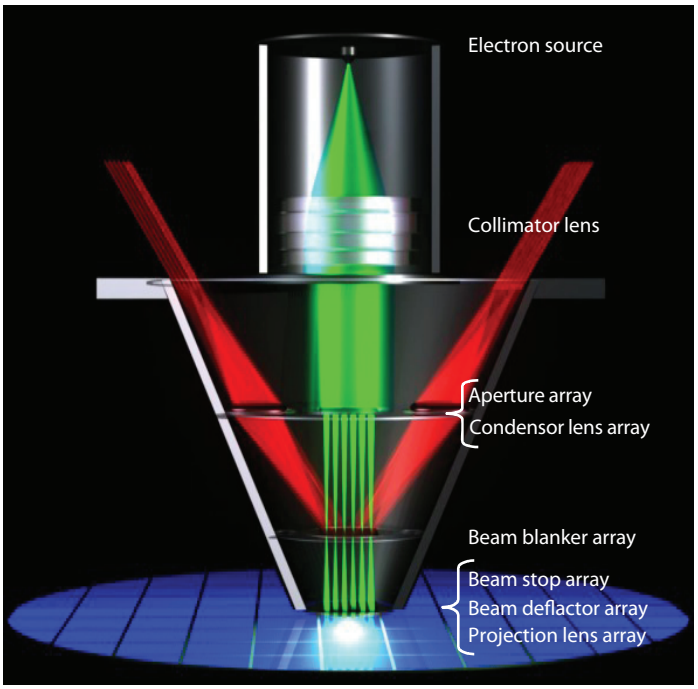

Figure 1.7: MAPPER System Overview

zig-zag order (going up and down) because the blanker writing regions are staggered by 2 μm as shown in the right part of Figure 1.8, the stripe continues until it reaches the end of the wafer with the help of the moving stage. Within the height-2 μm stripe, the electron beam is controlled by a 2.25 nm grid so that the lines it produce are not rough.

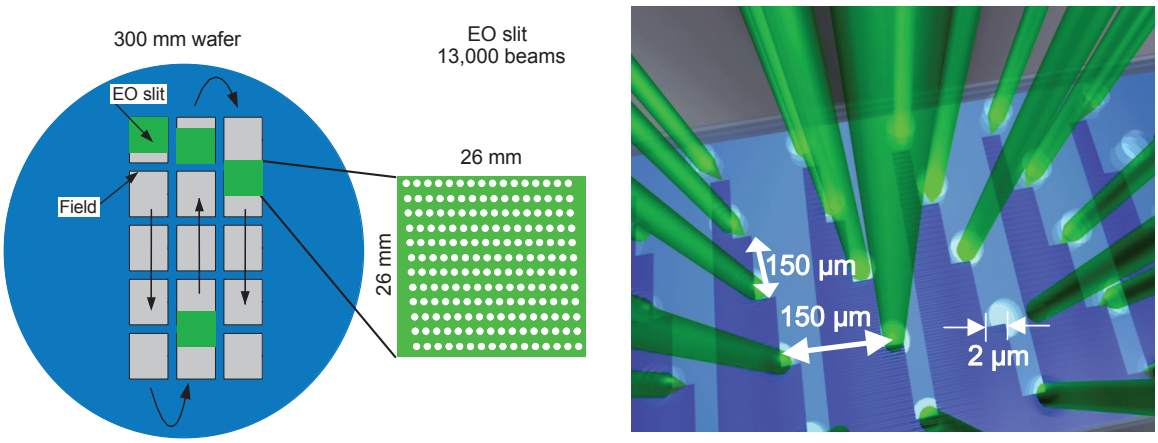

Figure 1.8: MAPPER Writing Strategy

Currently the MAPPER system is targeting 45 nm nodes [8] at 10–20 wafer layers

per hour, and is extending the technology to 16 nm half-pitch nodes and beyond using multiple MAPPER clusters. It is reported that the blanker array requires 7.5 Gbps for each channel, meaning a data rate of 97.5 Tbps has to be handled at the MAPPER system.

1.3 Data Delivery System Architectures for Multiple Electron Beam Lithography Systems

As we have mentioned earlier, an MEB system has to handle a huge amount of data. For example, in order to write a $10 \times 20 mm^2$ circuit on a 300 mm wafer targeting a 45 nm node, we need 2.1 Tbits to represent a circuit layer and 735 Tbits for the entire wafer layer [3]. Considering the requirement, Dai [3] argued that in order for the MEB systems to match the throughput of conventional photolithography (which is one wafer layer per minute), they would have to handle 12 Tbps of data for 45 nm nodes.

This requirement will increase as the targeting nodes gets smaller. In the following subsections, we will review Dai's argument on what kind of data delivery systems could be proposed and why data compression is necessary. Throughout the following subsections, we will discuss the data delivery system transferring 12 Tbps of data for 45 nm nodes.

1.3.1 Direct-Connection Architecture

The simplest design is to connect the storage disks containing the circuit layer directly to the writers as shown in Figure 1.9.(a). In order to match the throughput requirement, the storage disks need to output data at a rate of 12 Tbps. Moreover, the bus that transfers this data to the electron beam controller must also carry 12 Tbps of data. Clearly this design is infeasible because there exists no storage disk with

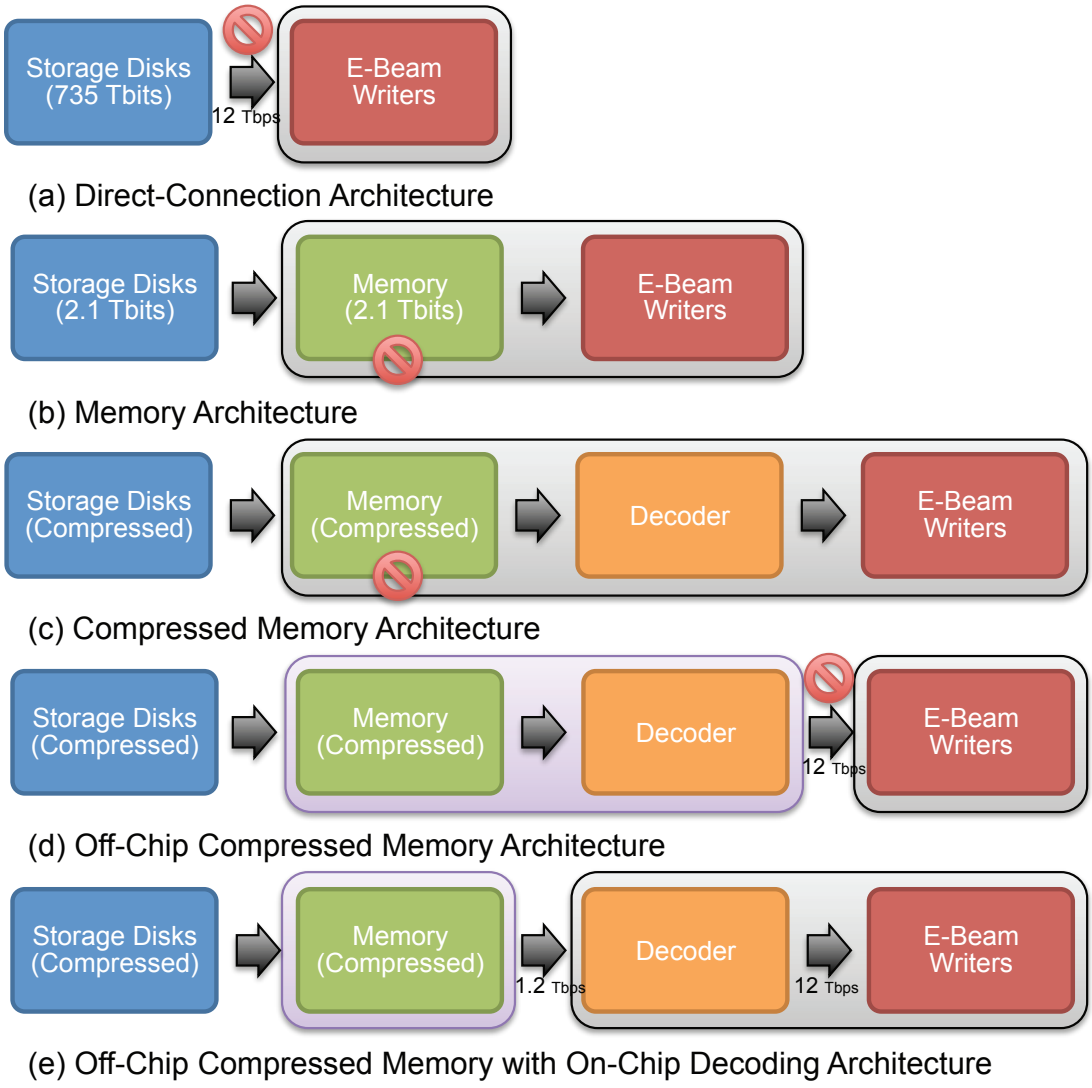

Figure 1.9: Possible Data Delivery System Architectures for MEB

this extremely high throughput. One can argue that using direct-connection on each electron beam writer reduces the throughput requirement for each direct-connection, but considering the number of electron beams in a MEB system (tens of thousands $\sim$ million), it is also infeasible.

1.3.2 Memory Architecture

Instead of taking the entire wafer layer, we can take advantage of the fact that the circuit layer is replicated many times [2] over the wafer. So, in the second design, rather than sending the entire wafer image in one minute, the storage disks only transmit the circuit layer information. Then, this information is stored in the *on-chip memory* fabricated on the same substrate as the writers as in Figure 1.9.(b). Because the memory is placed on the same silicon substrate as the writers, the 12 Tbps data transfer rate should be achievable between the memory and the writers. However, now we need memory for one circuit layer, 2.1 Tbits of data, and considering the huge memory requirement, this option is likely to be infeasible because of the extremely large amount of memory that must be present on the same die as the writers.

1.3.3 Compressed Memory Architecture

We can apply compression to the circuit layer image that is to be stored in the on-chip memory. Because the data is compressed, this data cannot be directly used by the writers without further data processing. Therefore, we need to have an additional *on-chip decoder* where the compressed circuit layer image is decompressed into its original form to control the writers as in Figure 1.9.(c). However, this architecture is still challenging because we still have to store the entire compressed circuit layer image on the memory. Considering we have to implement the decoder and the memory on the same die as the writer, this is infeasible.

1.3.4 Off-Chip Compressed Memory Architecture

To resolve the competition for circuit area between the memory and the decoder, it is possible to move the memory and decoder off the writer chip onto a processor board as in Figure 1.9.(d). Now multiple memory chips are available for storing

[2] For the example, the $10 \times 20mm^2$ circuit is repeated 350 times on a 300 mm wafer.

chip layer data, and multiple processors are available for performing decompression. However, after the data is decoded, we still require 12 Tbps transfer rate from the processor board to the writer circuits. Considering the anticipated state-of-the-art board-to-chip communications is expected to be 1.2 Tbps, which can be achieved using 128 pins operating at 6.4 Gbps [18], this architecture is still infeasible.

1.3.5 Off-Chip Compressed Memory with On-Chip Decoding Architecture

The previous architecture was infeasible because of the bandwidth limit on transferring decompressed data from the processing board to the writers. By moving the decoder back on-chip and leaving the memory off-chip as in Figure 1.9.(e), this board-to-chip transfer is compressed, improving the effective throughput. It is possible to achieve the 12 Tbps data transfer rate from the decoder to the writers because they are fabricated on the same substrate. As we have mentioned earlier, the input to the decoder is limited to 1.2 Tbps which is the communication bandwidth limit from board to chip. Therefore, if the data entering the on-chip decoder is compressed 10 times, the decoder will be able to produce 12 Tbps given that the decoder is implementable with high throughput.

The decoder has two limits:

1. It has to be *small* in size. Because the decoder and the writer has to be fabricated on the same substrate, we expect the decoder circuit area to be limited by the area of a single chip.

2. It must have *extremely high throughput*. We want the decoder to output 12 Tbps in total. So, the decoding circuit should not contain complex operation and instead should have a simple architecture.

In order for this architecture to work, we need to have high throughput board-

to-chip communication (e.g., 1.2 Tbps), a compression algorithm always compacting the circuit layer image at least by the factor of

$$\frac{\text{Transfer rate of Decoder to Writer}}{\text{Transfer rate of Memory to Decoder}}$$

and a decoder structure that is small and fast enough for the application.

Since the application is compress-once-and-decode-multiple-times, the encoding cost is not that important. Therefore, the encoder does not have any limit as long as the it can be compressed in a reasonable time on a reasonable computer system. However, the compression algorithm has to have a decoding process in which the decoder structure does not become too complex or slow.

1.4 Layer Image Generation

In this section, we will discuss how the circuit layer images are generated and use these images throughout the following chapters.

Circuit layouts are typically stored in GDSII [19] or OASIS [20] formats. GDSII and OASIS represent circuit features such as polygons and lines and describe them by their corner points [19, 21]. GDSII and OASIS formatted data are far more compact than the uncompressed image of a circuit layer. Therefore it may initially appear that the GDSII and OASIS formats are good candidates for this particular application. However, this is not the case because maskless writers operate directly on pixel bit streams and GDSII and OASIS layout representations must be converted into layout images before the lithography process begins. In general this conversion requires (1) eliminating hierarchical structures by replacing all of the copied parts with actual features, (2) arranging the circuit features such as polygons and lines into the corresponding layers of the circuit, and (3) rasterizing (see Figure 1.10). As the conversion usually takes hours using a complex computer system with large memory

it cannot be performed within the decoder chip.

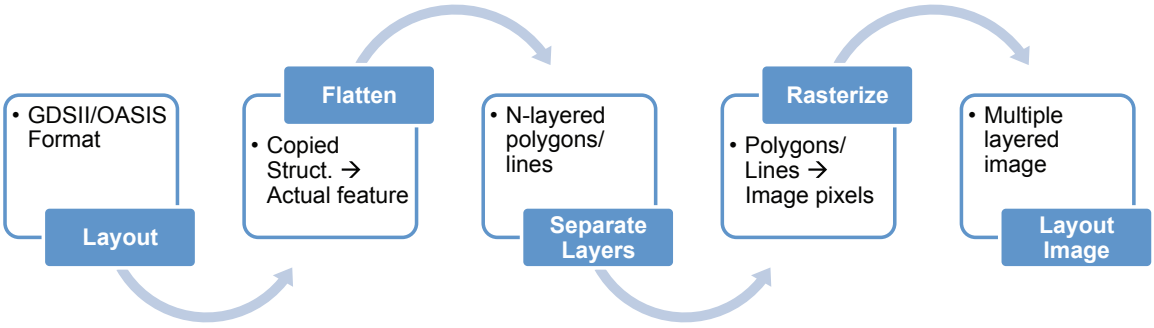

Figure 1.10: Generating Layer Images: From GDSII to Bitmap Image

Optical Proximity Correction (OPC) [11] is widely used for conventional lithography systems to adjust the shapes of mask features, but it is not in general necessary for EBL when the direct write application is considered because OPC is conventionally used to compensate for the image errors due to the diffraction effect, but this is not an issue for EBL using electron beams. The only exception is when EBL is used to make precise masks for the fabrication of circuits via photolithography. Since mask making is not a high volume manufacturing application, we will focus on the more interesting direct write applications only. Throughout the entire thesis we need not worry about OPC.

We instead consider electron beam proximity correction for maskless lithography systems to obtain good quality Line-Edge Roughness (LER). This is achieved by applying a multi-level electron beam dosage to each pixel [22]. As shown in Figure 1.11.(a) and Figure 1.11.(b), uniform electron beam doses result in blurry boundary edges because of the electron beam proximity effect. To compensate for that phenomenon, a higher electron beam dosage was applied to the boundary pixels as in Figure 1.11.(c) and the proximity effect has been corrected as in Figure 1.11.(d).

It is possible to represent the proximity corrected layout image using gray images [22]. However, this data eventually has to be reinterpreted as a binary image because the lithography writer does not produce a multi-level electron beam dose during a

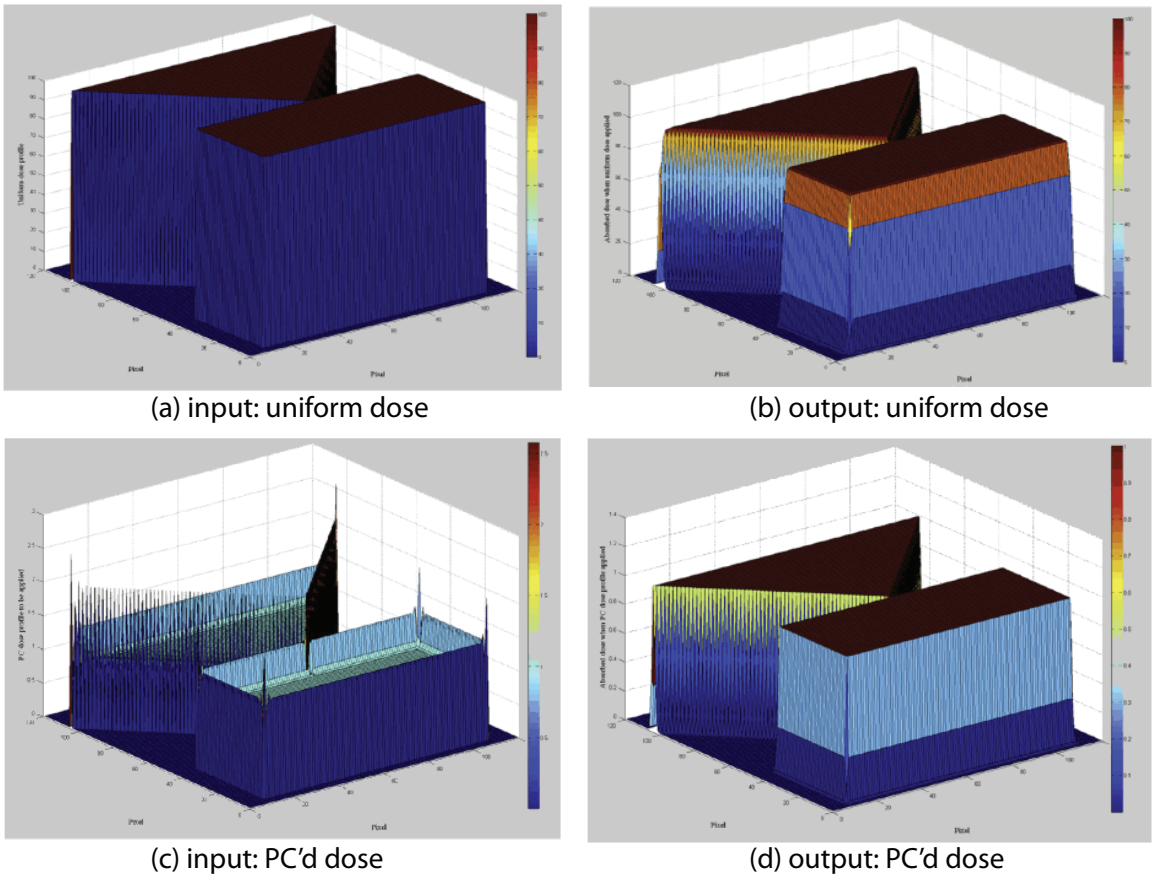

Figure 1.11: Proximity Correction using Gray Tone Exposure

single write time. Instead the lithography writer utilizes an electron beam writer to write the corresponding pixel multiple times so that the pixel is exposed with the targeted electron beam dosage; i.e., each electron beam writer uses a proximity-corrected layout image to write a portion of a gray image pixel which corresponds to a block of binary pixels. This thesis considers an *idealized pixel printing model* as in [3], however, this model can be applied to general proximity correction methods by reinterpreting the proximity corrected gray image as a binary image.

Under the idealized pixel printing model, we generated the gray images by the following process as in [3]. First, we start with the GDSII or OASIS layout. Second, as illustrated in Figure 1.10, we rasterize the layout image in a 1 nm grid and output a large binary image. Third, this binary image is segmented in blocks and quantized

with the appropriate gray level. For example, if we are targeting a 45 nm process technology, the electron beam pixel size chosen would be 22 nm (half the minimum feature size) and a block of 22×22 binary pixels would make up a single gray pixel. To obtain a 1nm edge placement Dai [3] suggested counting the number of fills in every 22×22 pixel block and quantizing that number to one among 32 levels.

The generated gray image is then reinterpreted as a binary image so that it directly maps to the lithography writer control signal, by changing the grid size. For the previous example, we want a 22 nm pixel to have a 1 nm edge replacement. That means, we need at least 22 dose levels [3]. Therefore, instead of 32 levels we can choose a quantization of 25 ($= 5 \times 5$) levels. Since every electron beam dose will increase a single level, each 22 nm pixel is written 25 times or, equivalently, that each control signal covers a 4.4 nm ($= 22 \ / \ 5$) pixel size. For simplicity, we can recompute the numbers so that each control signal covers a 4 nm ($= 22 \ / \ 5.5$) pixel size and is one among 30 levels ($\approx 5.5 \times 5.5$) for each 22 nm pixel. Finally, this new binary representation can be obtained by rasterizing the layout GDSII or OASIS file to the targeted grid (4 nm for the example).

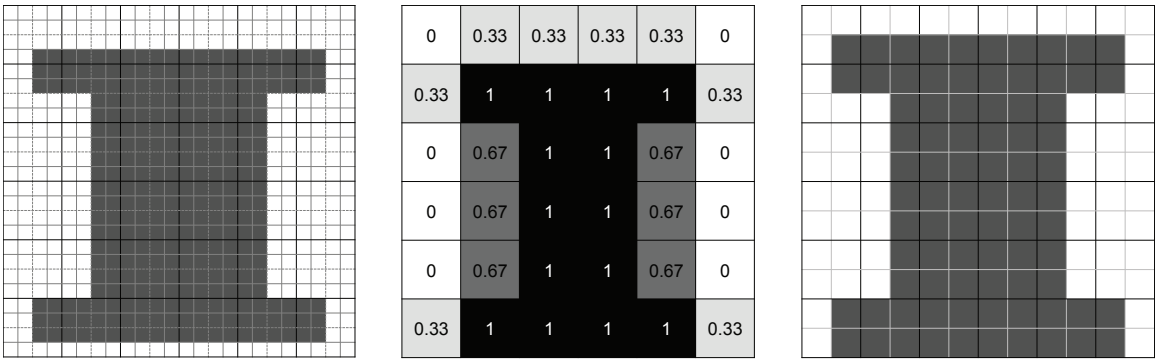

Figure 1.12: Binary Image vs. Gray Image

A simple illustration of this binary image to gray image and gray image to binary

[3]Note that 32-level was chosen for this example so that each pixel can be represented with five bits, but 22-level is the requirement.

conversion is shown in Figure 1.12. Here the grid size of the gray image is set to 4 nm and the grid size of the binary image is set to 2 nm. Figure 1.12.(a) shows the binary rasterized image at a grid size of 1 nm. By grouping 4×4 blocks of this image as a pixel and quantizing the number of fills to 4 levels (2 bits), we obtain Figure 1.12.(b) which corresponds to the gray image at a grid size of 4 nm. Because each gray image pixel has 4 levels, we could interpret this as a 4 nm pixel written 4 times. Here a 4 nm gray pixel corresponds to a 2×2 block of binary pixels from a 2 nm binary grid as in Figure 1.12.(c). Observe that Figure 1.12.(c) is not generated from Figure 1.12.(b) but could be generated from Figure 1.12.(a) by forming the appropriate 2×2 blocks.

We assume that all of the circuit layer images are binary images throughout the thesis unless specifically mentioned.

CHAPTER II

Prior Work on Lossless Data Compression Algorithms for Maskless Lithography Systems

In the proposed data-delivery path of Chapter I, compression is needed to minimize the transfer rate between the processor board and the writer chip, and also to minimize the required disk space to store the layout data. Since multiple decoders should operate in parallel on the writer to achieve the projected output data rate, it is crucial for any compression algorithm to have an extremely low decompression complexity for the application. In this chapter, previous works on lossless layout image compression will be reviewed.

2.1 Basic Properties of Layout Images

Even though circuit designers use computers to design complex circuits, they are *human designed*, and hence, are not randomly designed. In fact, they are well-structured and well-organized because the circuit designers build their circuits by their building blocks which they call *cells*. Since there are more than a billion elements in a modern microelectronic chip, this structured designed is inevitable. Instead of designing new components every time, circuit designers design a cell and try to reuse it whenever the same functionality is required in the circuit. Moreover, when they

embed the cells, the designers tend to align them so that the cells could be easily managed and are efficiently positioned in the circuit area. Because of these reasons, circuit layouts tend to be repetitive in some regions while irregular in other regions.

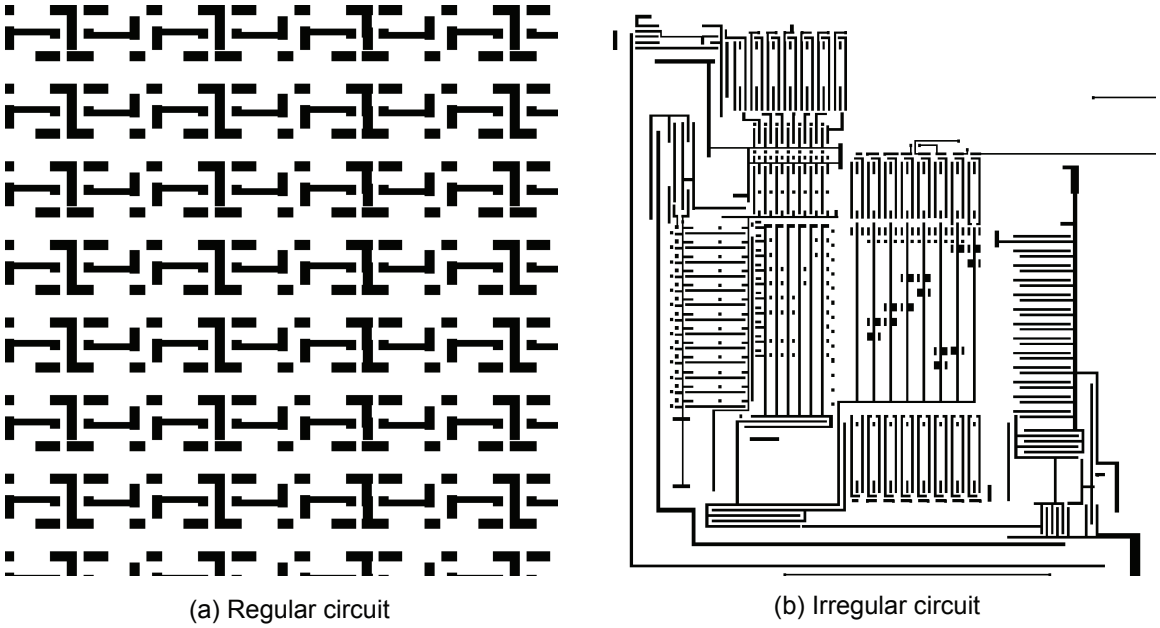

Figure 2.1: Circuit Layout Image Examples

Figure 2.1 shows examples of circuit layout images. Figure 2.1.(a) was extracted from a memory circuit while Figure 2.1.(b) was extracted from a Binary Frequency Shift Keying (BFSK) transmitter. From the figure, we can notice three characteristics of the layout image: First, most polygons in the layout image are Manhattan, i.e., all of the polygon corners have right angles because of the circuit design space. Because it is computationally infeasible to optimize a circuit element with an arbitrary shape, it has been customary to restrict the design space to a rectilinear space making the circuit elements Manhattan. In some cases, in order to relax this restriction while having manageable optimization complexity, the design space is formed by two rectilinear spaces - one regular and one tilted in 45 degrees - enabling diagonal lines in the layout image, but these patterns can still be decomposed into Manhattan polygons.

Second, it is easy to see there are patterns which are highly repetitive in Figure 2.1 because of the block-based design. Circuits are highly repetitive in a memory region where an array of memory cells are perfectly aligned. Moreover, as parallel processing develops, the number of processing blocks such as GPU blocks tend to grow making the circuits more repetitive.

Third, layout images typically are black-and-white images. This can be interpreted as the mask where the light source is blocked or passed as well as the electron beam control signal where the electron beam is blanked or passed.

2.2 Overview of `C4`

Because of the circuit layout image properties derived in Section 2.1, Dai [3] proposed the `Context-Copy-Combinatorial Coding (C4)` compression algorithms. In this section, we will overview the `C4` algorithm which is the base of the entire `C4` compression algorithm family.

While investigating layout images Dai noticed their three main characteristics. By applying a number of well known compression algorithms such as `JBIG`, `ZIP`, and `BZIP2` on example layout images, Dai found that the context prediction used in `JBIG` enabled efficient compression for irregular Manhattan polygons while the Lempel-Ziv (LZ)-style copying utilized in `ZIP` and `BZIP2` resulted in a high compression ratio for repetitive layout images. Based on this observation, Dai designed a lossless compression that takes the advantage of context prediction and LZ-style copying.

First, Dai expanded the LZ77 algorithm [23] so that it is suitable for 2D images. In the proposed 2D-LZ algorithm [24], pixels are encoded in raster order, i.e., each row in order from left to right, and the linear search window which appears in LZ77 is replaced with a rectangular search region of previously coded pixels. This search window is illustrated in Figure 2.2 [3].

As demonstrated in Figure 2.2, a match is now a rectangular region, specified

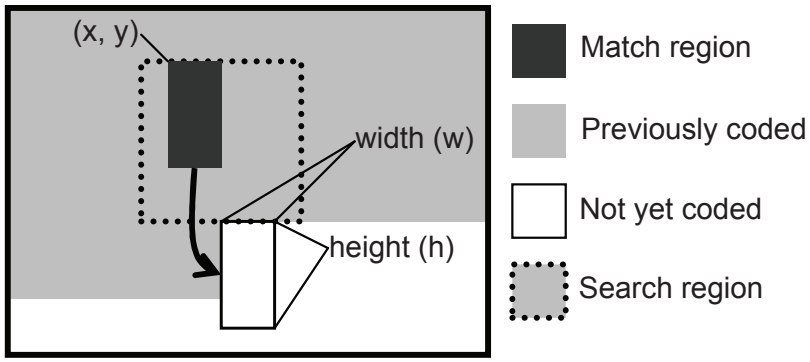

Figure 2.2: 2D-LZ Search Region

with four coordinates: a pair of coordinates (x, y) specifying the match position and another pair of integers (w, h) specifying the match dimension (width and height). In order to find the largest match region, each pixel is exhaustively tested in the search region (x, y). When a match is found at a particular (x, y), we increase the width w as much as possible, while still ensuring a match; then we increase the height h as much as possible. This procedure 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. If a match of minimum size cannot be found, then a symbol is output to represent a vertical column of pixels. A sequence of control bits is also stored so the decoder can determine whether the output is a symbol or a match. Finally, to further compress the output, each representation is encoded with separate Huffman codes [25] which means five [1] Huffman codes are used for 2D-LZ.

In order to decode 2D-LZ, the match region (x, y) and (w, h), or the mismatched symbols 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 found, the decoder simply copies data from the corresponding

[1] One Huffman code is used for each of the match coordinates x, y, w, and h, and another Huffman code is used for the symbols

match region (x, y) and (w, h) among the previously decoded pixels and fills it in the current decoding area. If a mismatched symbol is read, the decoder simply fills in a vertical column of pixels in the current decoding area. The decoder does not need to perform any searches, and is therefore much simpler in design and implementation than the encoder.

Second, in order to handle irregular patterns while making the context prediction simpler, Dai simplified the 10-pixel context-based prediction model of `JBIG` to a 3-pixel context-based prediction model (up-left, up, and left). In Figure 2.3, 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 layer image. From the results, it is clear that this 3-pixel context prediction works extremely well. Since the layer image is dominated by vertical edges, horizontal edges, and regions of constant intensity, this simple 3-pixel context predicts all of these cases perfectly. Furthermore, it is shown in [3] that using more pixels for the context does not improve the prediction performance.

This type of prediction typically fails at the polygon corners, so the number of prediction error pixels is proportional to the number of polygon corners. Therefore, for sparse features, it is advantageous to apply prediction. On the other hand, for dense or repetitive layouts, if the encoder can automatically find the repetition within the image and code it appropriately, the copy error pixels, i.e., the pixels which can not be copied, will be dramatically reduced. The compression efficiency is directly related to the numbers of image error pixels, which is the sum of prediction error pixels and copy error pixels.

By integrating the advantages of the two separate compression techniques – 2D-LZ and context-based prediction, Dai proposed the `C4` compression algorithm. In order to use two different compression techniques for a layout image, the encoder

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

Figure 2.3: `C4` Context Prediction Example

first divides the layout images into "predict" and "copy" regions, which are non-overlapping rectangles. In a copy region, each pixel is copied from a pixel preceding it in raster-scan order while each pixel inside a prediction region, i.e. not contained in any copy region, is predicted from its neighboring context. However, since neither predicted values nor copied values are 100% correct, error bits are used to indicate the position of these prediction or copy errors. These error bits are compressed using a technique called Hierarchical Combinatorial Coding (HCC) [26] which is a low-complexity alternative to arithmetic coding. Finally, the copy regions and compressed error bits are transmitted to the decoder so that the decoder can reconstruct the image. By avoiding the redundant transmission of copied or predicted pixels, `C4` achieves a high compression efficiency.

Finally, in order to reconstruct the original image, the decoder needs to store the 2D-LZ dictionary, keep track of the context model for prediction, and have Huffman tables to decode five Huffman code streams. Dai reported that in order to search $maxdy$ previous rows for 2D-LZ, the decoder requires $width \times (maxdy + 1) + 3410$ bits of memory. For the smallest decoding memory requirement, we set $maxdy = 1$

throughout the thesis and set the parameters of the proposed algorithms so that the decoder memory requirements are similar.

It was shown that the `C4` compression algorithm can efficiently compress both regular and irregular circuits in [3]. However, the encoding complexity was tremendous ($\sim$ 18 CPU years), prohibiting it to handle the layout images of a full chip. This encoding complexity is due to the complex segmentation algorithm partitioning copy regions and prediction regions, and was mitigated by the improved `Block C4` compression algorithm [27].

2.3 Overview of `Block C4`

The basic concept underlying both `C4` and `Block C4` compression is exactly the same. Repetitive structures are better compressed using LZ-based copying, whereas non-repetitive structures are better compressed using localized context-based prediction techniques, described in Section 2.2. The task of both the `C4` and `Block C4` encoder is to automatically segment the image between repetitive copy regions and non-repetitive prediction regions. The resulting segmentation map indicates which algorithm should be used to compress each pixel of the image, i.e. either copy or prediction. Once the segmentation is complete, it becomes a simple matter to encode each pixel according to this segmentation map. Dai analyzed that the `C4` encoding process and found that over 99.9% of the encoding time is devoted to the segmentation. In order to reduce the encoding complexity, the segmentation algorithm was revised for the `Block C4` encoder. By doing so, it was possible to speed up the encoding process by 100–900 times [3].

In order to understand why the segmentation algorithm for `C4` is too complex, it is important to understand how the copy region is detected from the layout image because the segmentation is described as a list of rectangular copy regions. An example of a copy region is shown in Figure 2.4.(a). Recall that each copy region is a rectangle,

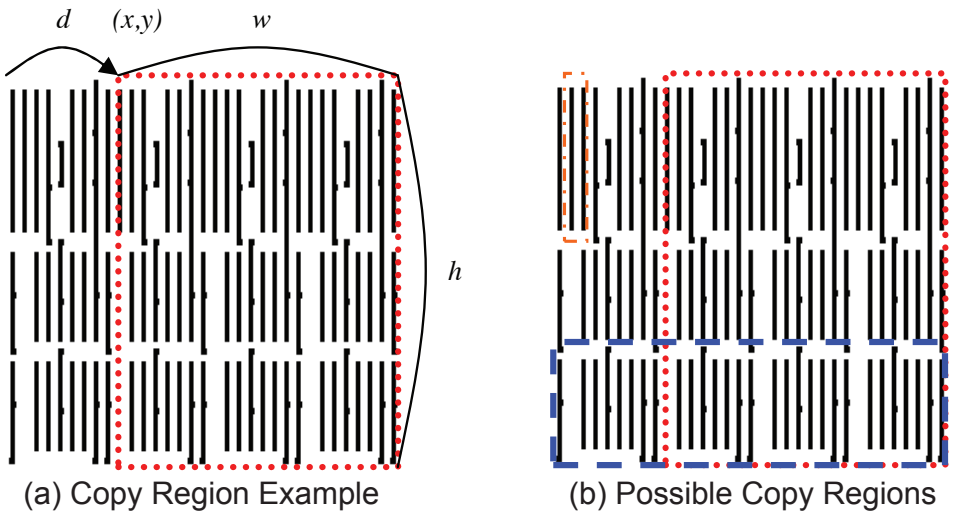

Figure 2.4: `C4` Copy Region

enclosing a repetitive section of a layer, described by 6 values, the rectangle position (x, y), (w, h), the direction of the copy (dir) which can be either $left$ or $above$, and the distance from the original copy (d). Finding the "best" segmentation is hardly obvious. This is because even in such a simple example shown in Figure 2.4.(a), there are many potential copy regions; a few of them are illustrated in Figure 2.4.(b) as colored rectangles, e.g the dashed blue rectangle and a dot-dashed orange rectangle. If we assume that the layout image is a $N \times N$ image, the number of all possible copy regions is $O(N^5)$ where each copy region parameter (x, y, w, h, d) contributes as $O(N)$. Therefore, exhaustive search on this space is prohibitively complex and clearly further complexity reduction of the segmentation algorithm is desirable.

`Block C4` adopts an entirely different segmentation algorithm from `C4` which is far more restrictive, and hence, much faster to compute. While `C4` allows for copy regions to be placed in arbitrary (x, y) positions with arbitrary (w, h) sizes, `Block C4` restricts both the position and sizes to fixed $M \times M$ blocks on a grid. Figure 2.5 illustrates the difference between `Block C4` and `C4` segmentation. Figure 2.5.(a) shows a segmentation example for `C4` which consists of three rectangular copy regions with 6 values (x, y, w, h, dir, d) describing each copy region. The same image is segmented

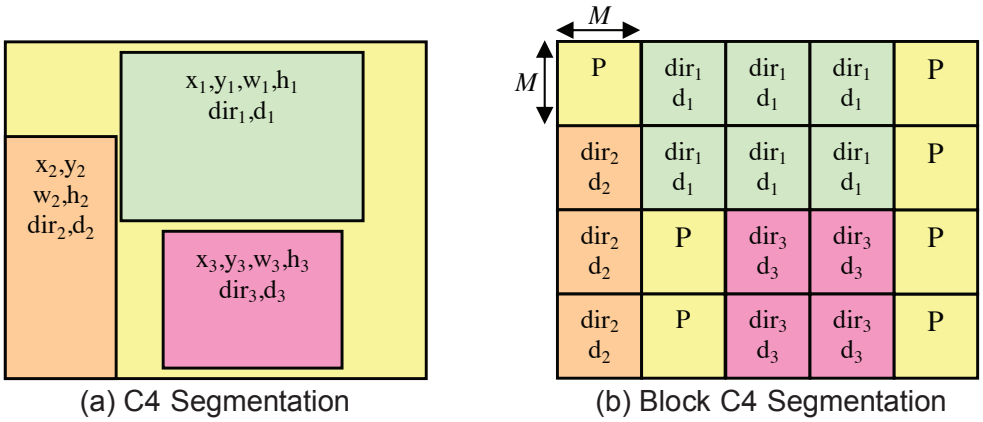

Figure 2.5: `C4` vs. `Block C4` Segmentation

for `Block C4` as shown Figure 2.5 which consists of twenty $M \times M$ tiles with each tile marked as either prediction (P) or the copy direction and distance (dir, d). This simple change reduces the number of possible copy regions to $O(N^3/M^2)$ which is a substantial $O(N^2M^2)$ reduction in search space compared to `C4`.

Dai reported that this simplified segmentation algorithm did not harm the compression performance significantly and sometimes even improved it by covering more small copy regions that the `C4` algorithm missed.

2.4 Experimental Results

Since the data set used by Dai [3] was proprietary and unavailable to us, we tested the `Block C4` algorithm on two custom circuits that were available to us – a memory circuit and a BFSK transmitter circuit. These circuits are the ones for which we provide experimental results throughout the thesis. `Block C4` was implemented in C# and was provided by Vito Dai. Since we experienced a memory shortage for the encoding process when we attempted to run `Block C4` on the entire layout image, we segmented the image into the largest components for which `Block C4` could be applied. We tested the compression algorithm on a laptop computer having a 2.53 GHz Intel Core 2 Duo CPU and 4 GB RAM.

Throughout the thesis, we define the compression ratio as

$$\frac{\texttt{Input File Size}}{\texttt{Compressed File Size}}.$$

2.4.1 Memory

An overview of the memory circuit is shown in Figure 2.6. Since the memory circuit is dense, we only show a small portion of the layout image with all of the layers in different colors in Figure 2.6. The memory core was targeting 500 nm lithography technology containing 13 layers with the memory cell structure in Figure 2.7 repeated 32,768 times throughout the layout image.

The compression ratio, encoding time, and decoding time of `Block C4` are shown in Table 2.1. Note that the last row of compression ratio is not the sum of preceding rows, but the "net average" which is defined as

$$\frac{\texttt{Total Input File Size}}{\texttt{Total Compressed File Size}}.$$

We can see that `Block C4` is compressing the layer images by a factor of 24–148 with an average compression ratio of 57. We also see that each layer requires about half an hour to be compressed while about a minute to be decompressed. Finally, the `Block C4` decompression algorithm required 4.9 kBytes of memory.

2.4.2 BFSK

An overview of the custom designed BFSK transmitter is shown in Figure 2.8. The BFSK transmitter was targeting 250 nm lithography technology containing 19 layers of mostly irregular features. The circuit consists of I/O pins (surrounding), large antenna area (center square), active circuits (top), and BFSK transmitter circuitry (bottom right).

Figure 2.6: Memory Circuit

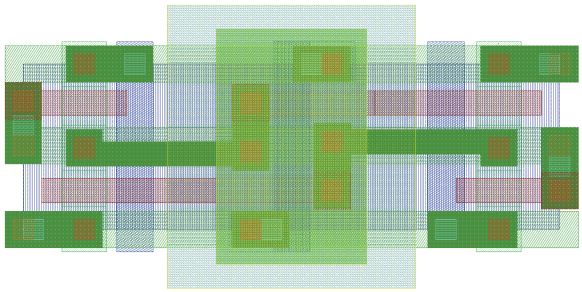

Figure 2.7: Memory Cell

The experimental results of `Block C4` are shown in Table 2.2. As we can see from the table, `Block C4` is compressing the layer images by a factor of 59–153 with an

Layer	Compression Ratio (x)	Encoding Time (s)	Decoding Time (s)
1	147.78	1,874.79	55.44
2	79.80	1,890.78	54.99
3	147.78	1,787.97	52.90
4	54.12	1,814.79	53.70
5	133.22	1,830.77	53.55
6	133.22	1,796.32	54.48
7	31.58	1,846.09	54.63
8	24.40	1,885.01	54.79
9	38.17	1,817.26	54.83
10	25.31	1,917.07	54.07
11	75.70	1,794.59	56.43
12	121.69	1,775.43	51.50
13	141.02	1,761.72	51.88
Total	57.36	23,792.61	703.18

Table 2.1: `Block C4` Compression Ratio - Memory

average compression ratio of 113. Since the size of the BFSK circuit was larger than that of the memory circuit, it required much more time for encoding and decoding as well as memory (8.4 kBytes) for the decoder.

2.5 Other Related Works

There are a number of works that are related to `Block C4`. Liu et al. [28] improved `Block C4` by applying a Golomb run-length code [29] to encode the error locations instead of HCC for improved compression performance as well as simpler decoder structure. Cramer et al. [17] tailored the compression algorithm for REBL systems with rotary stages [10].

We initially introduced a compression algorithm named `Corner` [30] which represents the layout polygons using their corners. In order to restrict the decoding complexity, we made sure that the corners do not produce arbitrary angle lines, but only horizontal or vertical lines and described the direction of the lines the corresponding corner point will reconstruct such as 'right', 'right and down', 'down', and

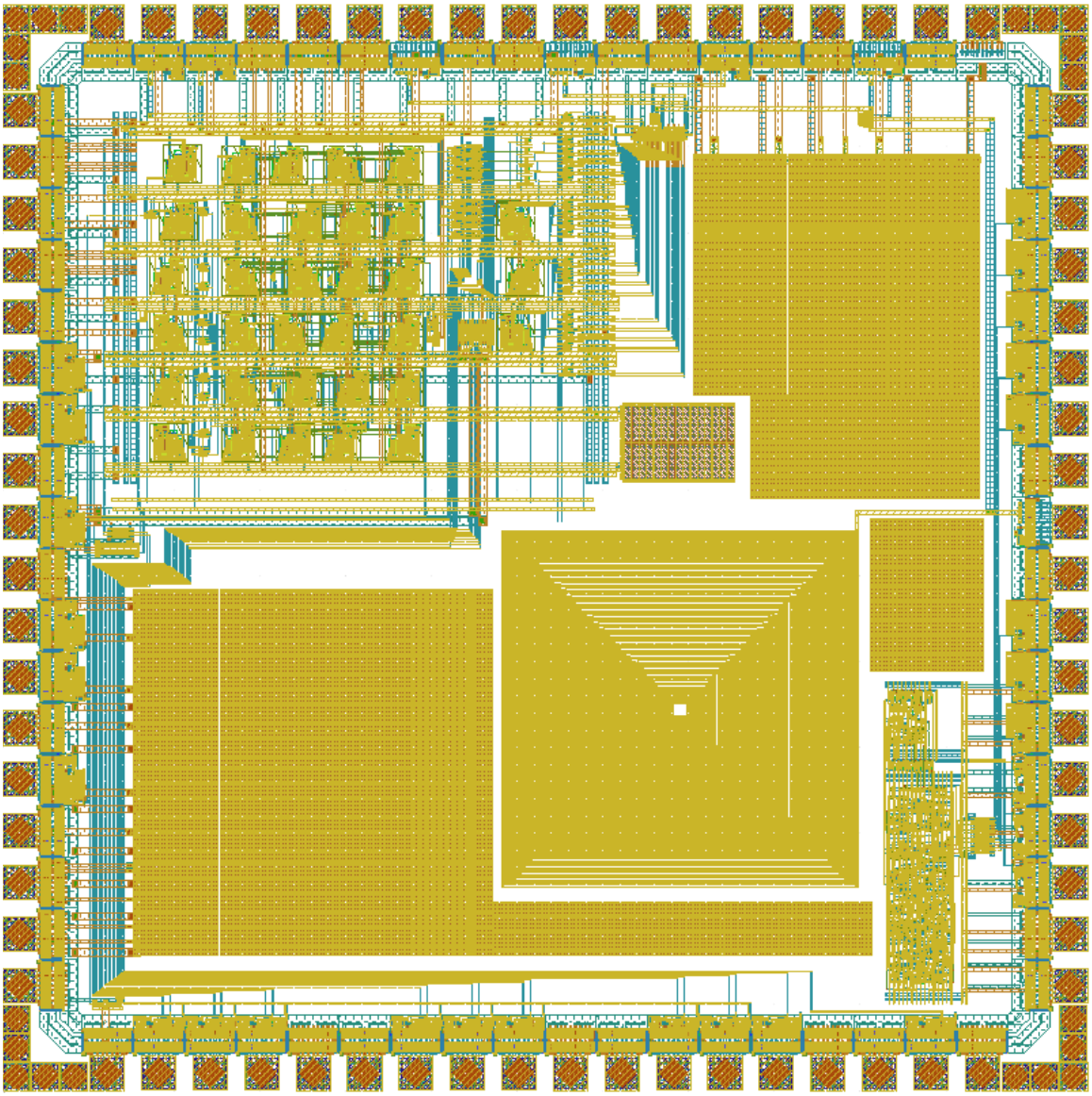

Figure 2.8: BFSK Circuit

'stop.' We later found a more efficient approach which we introduce in Section 3.1.3.

We also expanded the `Corner2` compression algorithm which is illustrated in Chapter III to gray level images to compare its performance against `Block C4` in gray level images. The `Corner-Gray` [31] algorithm utilizes the fact that for the layout images we generated, the pixel intensity only changes at the boundaries (or edges of the layout polygons). It encodes the boundaries using the *corner transform* which will be discussed in Section 3.1.3 and encodes the boundary pixel values using an entropy

Layer	Compression Ratio (x)	Encoding Time (s)	Decoding Time (s)
1	150.84	450.57	29.50
2	149.90	3,790.74	242.07
3	136.75	3,800.22	237.42
4	146.58	3,759.72	237.35
5	59.02	12,467.89	280.67
6	127.80	4,485.38	274.15
7	110.89	4,476.01	282.03
8	81.40	4,588.22	289.53
9	121.91	4,392.86	275.56
10	152.88	4,865.70	293.04
11	86.02	4,414.41	281.57
12	138.35	4,427.67	278.34
13	88.80	4,650.64	290.37
14	139.43	4,412.76	276.35
15	91.61	4,364.18	277.88
16	140.95	4,380.78	275.98
17	150.45	3,756.04	236.50
18	151.58	3,049.21	190.71
19	150.79	454.33	27.68
Total	112.97	80,987.33	4,576.71

Table 2.2: `Block C4` Compression Ratio - BFSK

encoder. The experimental results shows that `Corner-Gray` had a marginal improvement over `Block C4` on average, but was not always outperforming `Block C4`. Since `Corner-Gray` did not have a way to handle circuit regularities, there is room for improvement. We did not continue pursuing this direction because the low level electron beam writer control signals are not gray level, but binary.

Krecinic et al. [32] introduced a vertex-based circuit layout image representation format. Their research can be viewed as a variant of the *corner transformation* which will be introduced in Section 3.1.3 along with a version of Run-Length Encoding (RLE) to compress circuit layout images. However, they did not account for the circuit regularity which will be illustrated in Section 3.1.2 and Section 5.2 or use more advanced entropy encoding techniques to further compress the representation as in Section 3.1.5.

CHAPTER III

Corner2 Lossless Compression Algorithm

As we have seen in Chapter II, when compressing circuit layouts we need a good strategy to handle their regularities and irregularities. `Block C4` [3, 27] used an LZ-based region copy to handle the regularities and context prediction to handle the irregular parts of the circuit layout images. However, there are some drawbacks with these approaches. First, because the regularities are handled by LZ-based parsing while the LZ-copy region was limited to the previous row only for a compact decoder architecture, the approach can handle limited pattern repetitions.

Second, the context prediction mostly fails for the polygon corners. The 3-pixel context prediction worked well for most cases with low error probability (~ 0.07) as shown in Figure 2.3, but it failed at the boundaries – i.e., polygon corners. Since polygons are efficiently represented by their corners, having a good context prediction inside/outside the polygons is not really impressive. Rather, it would be more interesting if one can come up with more efficient ways to represent the polygon corners and reconstruct them without the complex rasterization process shown in Figure 1.10.

Our compression algorithm is inspired by the compactness of the GDSII/OASIS format and is designed to take advantage of ideas like corner representation and the copying of repeated structures. However, we avoid the complex flattening and

rasterizing processes and offer a simple decoding process. In the following subsections, we will show how our compression / decompression algorithm works and why it is superior to `Block C4`.

3.1 The Compression Algorithm

3.1.1 Overview

An overview of the compression algorithm, `Corner2`, is shown in Figure 3.1. We begin by seeking frequently occurring patterns in order to handle circuit regularities. The Frequent Pattern Replacement (FPR) process is based on dictionary-based compression which replaces the pattern embeddings that are in the dictionary with a simple representation. By doing so, the encoder replaces roughly all embeddings of frequently repeated cells over the layout image and is hence better suited than the LZ-based region copy used in `Block C4`.

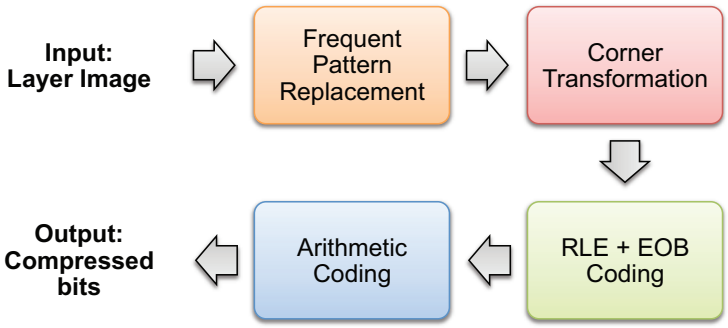

Figure 3.1: `Corner2` Encoder Overview

In order to handle circuit irregularities on the non-matched image regions, we next apply a transform to extract polygon corners. This transformation generates a *transitional corner* image [1] of the non-matched image regions. This transformed image is designed so that it will be easy to rasterize without a complex decoding

[1]The transformed image is similar to the polygon corner image, but they are not exactly the same.

system. Combined with the following entropy coding schemes, this transformation can represent circuit irregularity more efficiently than `Block C4`.

Finally, we apply a series of entropy encoders such as RLE [29], End-Of-Block (EOB) coding, and arithmetic coding [33] to produce a compressed bit stream of the transformed image.

In Section 3.1.2 and Section 3.1.3, we will first describe how the FPR and corner transform processes work as separate processes. In Section 3.1.4, we will illustrate how we tweak them to work in a unified system, and in Section 3.1.5 we will describe the entropy coding processes.

3.1.2 Frequent Pattern Replacement

GDSII/OASIS formats are designed to take advantage of the hierarchical structure present within circuit layouts. In particular, if a substructure is repeatedly used in a circuit layout, GDSII/OASIS identifies it and then refers to it whenever the substructure occurs. For example, the GDSII/OASIS representation for an 8-bit adder, which is implemented using two 4-bit adders, will consist of a definition of a 4-bit adder and the description of the 8-bit adder in terms of two 4-bit adders as in Figure 3.2.

By searching the GDSII/OASIS file and counting the number of references for each definition, we obtain a list of frequent patterns in the mask image. We could alternatively run a complex pattern matching algorithm to detect the frequent patterns, but the outcome would not be significantly different due to the block-wise design of typical circuits. Our approach is not efficient for compression purposes if the input GDSII/OASIS file is unstructured, but this issue can be handled by a preprocessing algorithm proposed by [34] which efficiently restructures input GDSII/OASIS files.

Figure 3.3 offers an overview of frequent pattern extraction from GDSII files. The inputs to the procedure are the GDSII file and the layer number of the layout

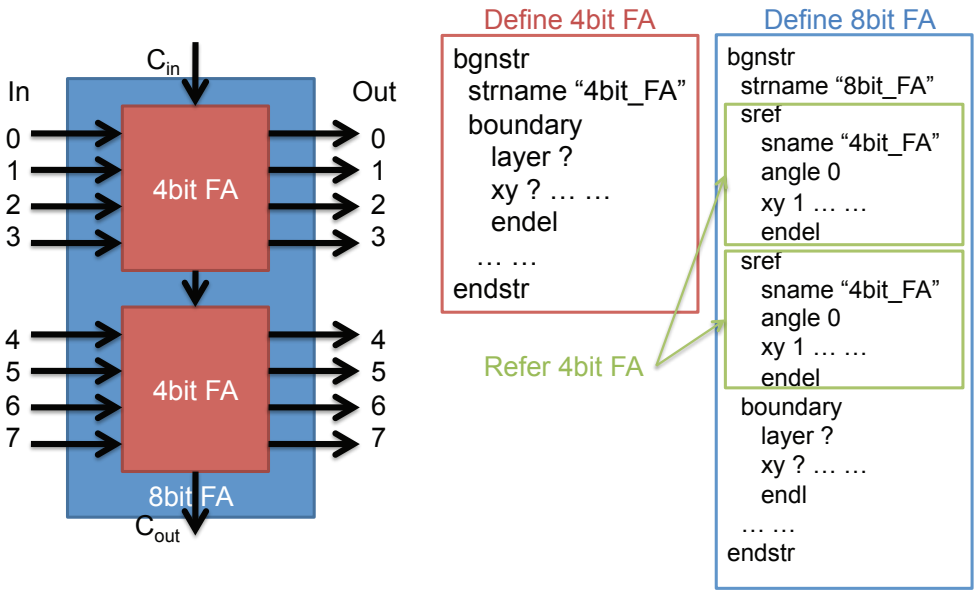

Figure 3.2: 8-bit adder using two 4-bit adders

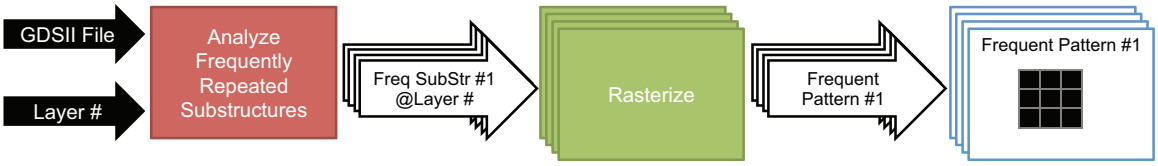

Figure 3.3: Extracting Frequent Patterns from GDSII

image, and the outputs of this process are the rasterized images of the discovered frequent patterns. The first step of this frequent pattern extraction is to analyze frequently repeated substructures and extract the substructures from the layout description. To do that, the encoder detects all of the substructures that are defined in the GDSII/OASIS representation. Next, the encoder counts the number of references of each substructure extracted in the previous step. The encoder proceeds to order the substructures by the number of their occurrences and select the most frequent P of them, where P is chosen so that the representation of the P substructures requires less than P_{size} bytes of memory. Note that we are not fixing the number P, but P_{size}, the memory that is required to store the P patterns. Also, note that because of the decoder memory constraint, P_{size} is very small, and hence, we can only pick a small number of patterns. Each of the P substructures undergoes the rasterizing process

to generate the corresponding pattern image [2].

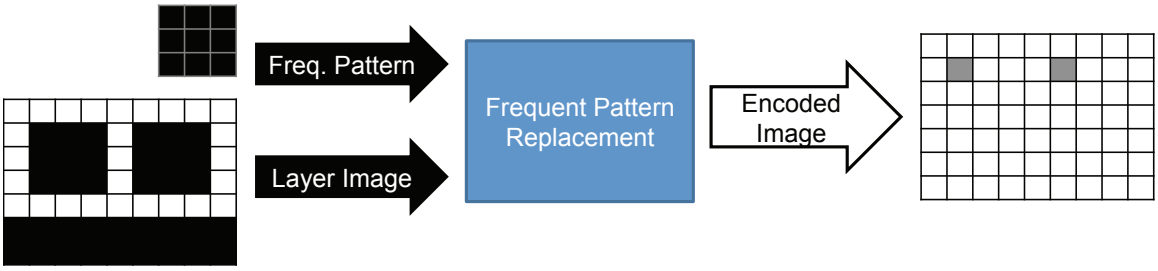

Figure 3.4: Frequent Pattern Replacement

Once the frequent patterns are generated, we replace the pattern embeddings from the layout image. Figure 3.4 offers an example of the FPR process with one frequent pattern. Given the binary layout image and the frequent pattern images, the FPR encoder seeks the P patterns within the image. When the FPR encoder seeks a pattern, it actually wraps the pattern around with empty rows (in the top and bottom) and columns (in the left and right) so that the pattern embedding is isolated (i.e., not connected to other polygons). We apply this to avoid interference with corner transformation, which will be further discussed in Section 3.1.4. Whenever one of the patterns is matched within the search region, the encoder will replace the top left part of the corresponding part of the image with a string described below and will replace the rest of filled pixels that have been matched with "0"s (or empty).

In order to restrict the decoder memory, the pattern is encoded using the following symbol string format:

$$[\$ \texttt{Pattern\#}]$$

The symbol \$ is a flag for a pattern. The choice of this flag will become clearer in the next subsection. `Pattern#` is a length $\lceil \log_2(P) \rceil$ binary string representing the pattern number. If $P = 1$, then `Pattern#` can be omitted.

[2]This rasterization process itself is quite complex, but it is much simpler than that of the entire circuit because the size of the pattern is extremely small due to constraints on P_{size}. For example, while it took several hours to generate a single layer image of a full example circuit, it only took several seconds to generate the frequent patterns for the entire circuit layers.

Suppose the pattern p dimension is $w_p \times h_p$. Then in order to make this encoded stream fit in the top row of the image, we assume the original image width w_p is no shorter than

$$1 + \lceil \log_2(P) \rceil + \lceil \log_2(h_p - q_{h_p}) \rceil,$$

where q_{h_p} is one more than the number of empty bottom rows of the pattern. The final term $\lceil \log_2(h_p - q_{h_p}) \rceil$ is added because of the decoder memory restriction which will be explained in Section 3.2.2.

In Figure 3.4, we illustrated an example where $P = 1$ and the rasterized layer image of the frequent substructure is a 3×3 square. Since $P = 1$, `Pattern#` can be omitted and the encoder outputs \$ (depicted by one 'gray' pixel) for the pixel corresponding to the top-left corner of each 3×3 square pattern in the input layout image and 'white' pixels for the remaining 8 pixels. For this pattern, $w_p = 3$ and is greater than $1 + \lceil \log_2 1 \rceil + \lceil \log_2(3-1) \rceil = 2$.

3.1.3 Corner Transformation

In the GDSII/OASIS representation of a structurally flattened single layer, the layout polygons are represented in terms of their corner points. While this representation is efficient for a system in which decoder memory is large, it is infeasible when the decoder memory is restricted because the decoder needs to access a memory block of size $(|x_1 - x_2| + 1) \times (|y_1 - y_2| + 1)$ for the encoder to connect an arbitrary pair of points (x_1, y_1) and (x_2, y_2) as in Figure 3.5.

However, if the angle of a contour line is constrained to a small set then there can be considerable simplification in the rasterizing process. In our previous research [30], we restricted the contour lines to be either horizontal or vertical and decomposed an arbitrary polygon into a number of Manhattan polygons, i.e., polygons with right angle corners. This decomposition is well-suited to this application because most components of circuit layouts are produced using Computer Aided Design (CAD)

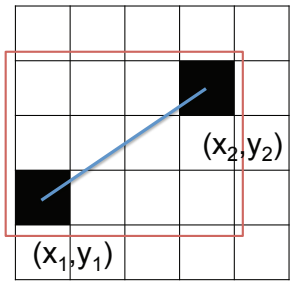

Figure 3.5: Required decoder memory to reconstruct a line from (x_1, y_1) to (x_2, y_2).

tools which design the circuit in a rectilinear space, and even the non-Manhattan parts can be easily decomposed to Manhattan components.

If the contour lines are constrained to be either horizontal or vertical and the decoder scans the image in raster order, i.e., each row in order from left to right, then when the decoder encounters a corner it only needs to decide whether it should reconstruct a horizontal and/or a vertical line. The raster order implies that a corner is either the beginning of a line going to the right and/or down or the end of a line. In our previous research [30], we specified this decoding decision by representing each pixel with five possible symbols – 'not corner,' 'right,' 'right and down,' 'down,' and 'stop.' However, as we will see this five-symbol representation can be further simplified.

To motivate the simplified transformation observe that a row (or a column) of the original layout image consists of alternating runs of 1s (fill) and runs of 0s (empty). We encode the pixels where there are transitions from 0 to 1 (or 1 to 0) using symbol "1" and encode the other places using symbol "0." Since most polygons are Manhattan, after applying this encoding in the horizontal direction we obtain alternating runs of 1s and 0s in the vertical direction as seen in Figure 3.6 (b). Therefore we can re-apply this encoding in the other direction to produce the final corner image. We call this encoding scheme the *binary corner transformation* because the final encoded image is binary and the location of the "1"-pixels indicate the corners of the polygons. In order to illustrate how the transform is applied, we will first discuss a two-step

transformation process and then introduce a one-step transformation process which requires less memory during the encoding process and runs faster than the two-step transformation process.

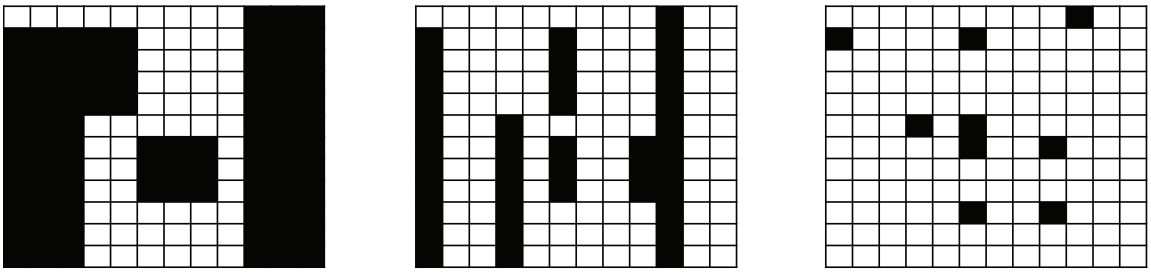

Figure 3.6: Two-Symbol Corner Transformation

The two-step transformation process consists of a horizontal encoding step and a vertical encoding step. In the horizontal encoding step, we process each row from left to right. For each row, the encoder initializes the previous pixel value to 0 (not filled). If the value of the current pixel differs from the previous one we encode it with a "1" and otherwise with a "0." After the horizontal encoding is completed we use the intermediate encoded result as input to the vertical encoding process. This is identical to the horizontal encoding process except that instead of processing rows we process each column from top to bottom.

The algorithm is summarized in Algorithm 1. In the algorithm, x is the column index $[1, \cdots, C]$ of the image and y is the row index of the image $[1, \cdots, R]$.

As we can see from Line 13 of the algorithm, $\texttt{OUT}(x, y) = 1$ only if $\texttt{TEMP}(x, y) \neq \texttt{TEMP}(x, y-1)$. That is, $\texttt{OUT}(x, y) = 1$ only if $\texttt{TEMP}(x, y) = 1$ and $\texttt{TEMP}(x, y-1) = 0$, or if $\texttt{TEMP}(x, y) = 0$ and $\texttt{TEMP}(x, y-1) = 1$. Since $\texttt{TEMP}(x, y) = 1$ only if $\texttt{IN}(x-1, y) \neq \texttt{IN}(x, y)$ as in Line 5, we can simplify the corner transform process as in Algorithm 2.

The preceding algorithm bypasses the need for intermediate memory. Here pixel (x, y) is processed as a function of the input pixels $(x-1, y)$, $(x, y-1)$, and $(x-1, y-1)$. This simplification results in a much faster running time. Finally, note that the

Algorithm 1 Transformation : Two-Step Algorithm

Input: Layer image $\texttt{IN} \in \{0,1\}^{C \cdot R}$
Output: Corner image $\texttt{OUT} \in \{0,1\}^{C \cdot R}$
Intermediate: Temporary image $\texttt{TEMP} \in \{0,1\}^{C \cdot R}$

{**Horizontal Encoding**}
Initialize $\texttt{TEMP}(x,y) = 0$, $\forall x, y$.
for $y = 1$ **to** R **do**
  for $x = 1$ **to** C **do**
    if $\texttt{IN}(x,y) \neq \texttt{IN}(x-1,y)$ **then**
      $\texttt{TEMP}(x,y) = 1$.
    end if
  end for
end for

{**Vertical Encoding**}
Initialize $\texttt{OUT}(x,y) = 0$, $\forall x, y$.
for $x = 1$ **to** C **do**
  for $y = 1$ **to** R **do**
    if $\texttt{TEMP}(x,y) \neq \texttt{TEMP}(x,y-1)$ **then**
      $\texttt{OUT}(x,y) = 1$.
    end if
  end for
end for

Algorithm 2 Transformation : One-Step Algorithm

Input: Layer image $\texttt{IN} \in \{0,1\}^{C \cdot R}$
Output: Corner image $\texttt{OUT} \in \{0,1\}^{C \cdot R}$

Initialize $\texttt{OUT}(x,y) = 0$, $\forall x, y$.
for $y = 1$ **to** R **do**
  for $x = 1$ **to** C **do**
    if $\texttt{IN}(x-1,y-1) = \texttt{IN}(x,y-1)$ and $\texttt{IN}(x-1,y) \neq \texttt{IN}(x,y)$ **then**
      $\texttt{OUT}(x,y) = 1$
    end if
    if $\texttt{IN}(x-1,y-1) \neq \texttt{IN}(x,y-1)$ and $\texttt{IN}(x-1,y) = \texttt{IN}(x,y)$ **then**
      $\texttt{OUT}(x,y) = 1$
    end if
  end for
end for

transformation can handle layout images with width-1 lines as shown in Figure 3.7.

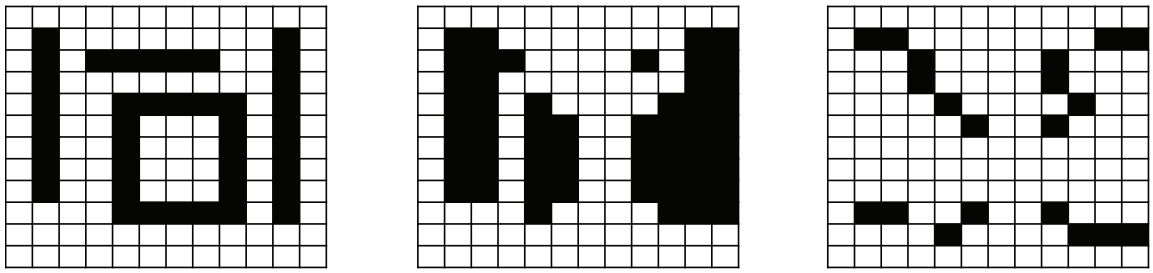

Figure 3.7: Handling width-1 lines

3.1.4 Frequent Pattern Replacement + Corner Transformation

In the previous two subsections we separated the discussion of FPR and corner transformation. Since the output of FPR is a ternary image (\$, 0, 1), a modification is needed to the corner transformation. Figure 3.8 illustrates the tweaking of the FPR and the corner transformation processes when $P = 1$. The FPR process now outputs two images, namely the *matched pattern image* and the *residue image* produced by removing the matched patterns from the original layout image. In this decomposition the pattern embeddings are compressed by the FPR and the residue image is compressed by the corner transformation.

Note that the filled pixels in our "corner" image are more closely related to transitions than to actual corners in the original layout image. Therefore filled pixels in the corner image can appear to the right, down, or right-down of the corresponding actual corner in the layout image and can hence overwrite the frequent pattern stream if the frequent pattern does not contain an empty top row or an empty left column. Because we surrounded frequent patterns with empty rows and columns when pattern matching was considered, we can add the matched pattern image and the corner image to form the compressed image without any distortion.

Figure 3.8 illustrates the combination of the FPR and the corner transformation processes. The FPR process now outputs two images, the matched pattern image and the residue image produced by removing the matched patterns from the original

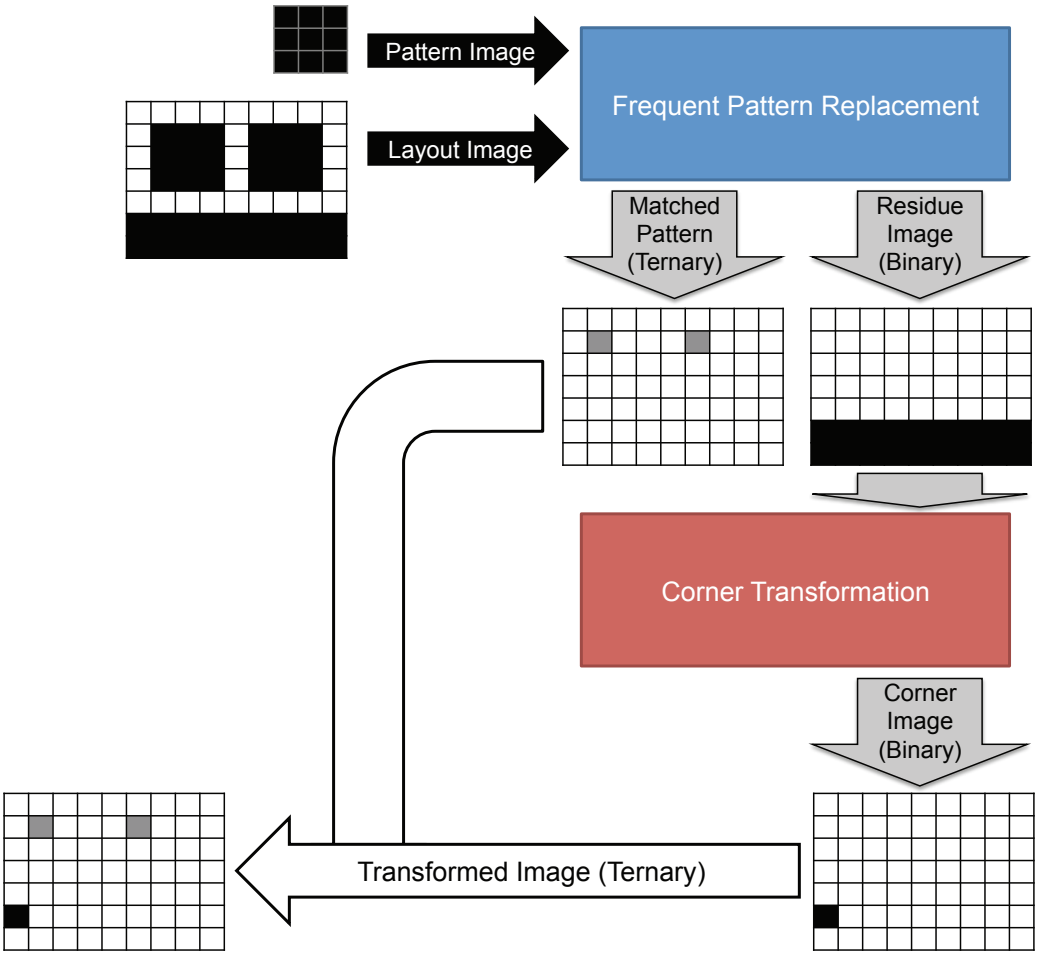

Figure 3.8: Handling FPR + Corner Transformation in a Unified System

layout image. In this decomposition, the pattern embeddings are represented by the FPR process and the residue image is represented by the corner transformation. The outputs of the two processes are summed to obtain the transformed image. Observe that two non-zero symbols are never summed, and so the summation is well-defined and the transformed image is over the alphabet $\{0, 1, \$\}$.

3.1.5 Entropy Coding

We next describe an entropy coding scheme to compress the transformed image.

We expect the transformed image to contain long runs of zeroes, and it is therefore effective to use a type of RLE [29] for compression. The nonzero pixels of the

transformed image are written as they are, but each run of zeroes is represented by its run length which we encode with an M-ary representation. More specifically, we introduce new symbols "2", "3", $\cdots$, "M+1" to represent the base-M symbols "0_M", "1_M", $\cdots$, "$(M-1)_M$". For example, if the transformed stream was "1 00000 00000 1 00000 0000 1 00000 00000 000" and $M = 3$, then the encoding of the stream is "1 323 1 322 1 333" because the run lengths are 10 (=101_3), 9 (=100_3), and 13 (=111_3), and 2/3/4 is used to represent $0_3/1_3/2_3$.

These M symbols are to be encoded using arithmetic coding [33] for further compression. For arithmetic decoding, we need to allocate memory for each symbol, and hence, in our restricted decoder memory setting, we want to choose M as small as possible. However, small M is not desirable since there are very long runs of zeroes. These long runs of zeroes occur frequently if the circuit features are aligned in a grid manner.

Therefore, in order to obtain a high compression ratio while restricting the size of the decoder memory, we segment each line into k blocks of length L, and we introduce a new "EOB" symbol "X". If a run of zeroes ends after a block, instead of representing the run length using an M-ary representation, we use the end-of-block symbol X. Hence, we encode a line of zeroes with k X's instead of roughly $\log_M (kL)$ symbols. Continuing the previous example, if $M = 2$, $k = 5$, and $L = 7$, then the transformed stream "1000000 0000100 0000000 1000000 0000000" is represented as "1X 3221X X 1X X," where 2/3 (=$0_2/1_2$) is used for the binary representations of runs of zeroes.

After applying EOB coding, there tend to be long runs of "X"s in the encoded stream. Therefore, we can apply run length encoding to this stream by reusing the M symbols for the initial runs of zeroes and introducing N new symbols for an N-ary representation of runs of "X"s . Persisting with the previous example, if $M = N = 2$, $k = 5$, and $L = 7$, then the next representation is the string "1 5 3221 54 1 54," where

2/3 (or 4/5) is used for the binary representation of runs of zeroes (or "X"s).

Our last encoding step compresses the preceding stream with the implementation of arithmetic coding provided by [35]. The decoder requires four bytes per alphabet symbol, and since we used $M + N + 2$ symbols, $4(M + N + 2)$ bytes were used for arithmetic decoding. Note that M symbols are used for runs of zeroes, N symbols are used for runs of "X"s, 0/1 is used for the corner pixel, \$ is used to handle the frequent P patterns.

3.2 Decoder

The decoder consists of two parts: (1) an entropy decoder consisting of an arithmetic, run length, and end-of-block decoder which outputs the transformed image, (2) the transform decoder which reconstructs the layout image from the transformed image. The transform decoder reconstructs the layout image using both inverse corner transformation and frequent pattern reconstruction with the help of the frequent pattern dictionary which has been transmitted to the decoder. The ideas in the implementation of the first part are standard, and we omit them.

For simplicity we will demonstrate how the layer image can be reconstructed from the corner image and we will separately discuss the recovery of the frequent patterns. However, these two processes are conducted in a row-by-row fashion because the decoder has restricted memory and is executed as a single process because the corner image and the matched pattern image do not overlap as explained in Section 3.1.4.

Since each part of the decoding procedure (arithmetic decoding, run length decoding, and vertical/horizontal decoding) processes each symbol based on the previously processed symbols, the entire decoding process can be pipelined to improve the throughput. Observe also that the decoder can be implemented in hardware because the decoding process only requires simple branch and compare operations. For example, arithmetic coding is widely implemented in microcircuits [36].

3.2.1 Inverse Corner Transformation

The corner transformation uses pixels from the previous row and column to decode the current pixel. Because the decoding process depends on the previous row, we designed the decoder to decode the corner image in a row-by-row manner instead of in its entirety in order for this process to be compatible with the restricted memory available to a maskless writer. In our transform decoder we use a row buffer (`BUFF`). The buffer is used to store the status of the previous (decoded) row. It uses two symbols, 0 and 1, to represent its status, and hence, the buffer requires *width* bits of memory. "0" means 'no transition' while "1" means 'transition' which indicates the starting/ending point of a vertical line. Using the current row of the corner image and the buffer, we can reconstruct the layer image as in Algorithm 3. Note that the $\bigoplus$ operation is a binary `XOR` operation, and is only applied to binary summands.

Algorithm 3 Inverse Transformation

Input: Corner image $\texttt{IN} \in \{0,1\}^{C \cdot R}$
Output: Layer image $\texttt{OUT} \in \{0,1\}^{C \cdot R}$
Intermediate: Row Buffer $\texttt{BUFF} \in \{0,1\}^{R}$

1: Initialize $\texttt{BUFF}(x) = 0$, $\forall x$.
2: Initialize $\texttt{OUT}(x, y) = 0$, $\forall x, y$.
3: **for** $y = 1$ **to** R **do**
4: $\quad \texttt{Fill} = 0$
5: $\quad$ **for** $x = 1$ **to** C **do**
6: $\quad\quad$ **if** $\texttt{BUFF}(x) = 1$ **then**
7: $\quad\quad\quad \texttt{OUT}(x, y) = 1$
8: $\quad\quad$ **end if**
9: $\quad\quad$ **if** $\texttt{IN}(x, y) = 1$ **then**
10: $\quad\quad\quad \texttt{Fill} = \texttt{Fill} \bigoplus 1$
11: $\quad\quad$ **end if**
12: $\quad\quad \texttt{OUT}(x, y) = \texttt{OUT}(x, y) \bigoplus \texttt{Fill}$.
13: $\quad\quad \texttt{BUFF}(x) = \texttt{BUFF}(x) \bigoplus \texttt{Fill}$.
14: $\quad$ **end for**
15: **end for**

Line 4 initializes the status of the horizontal fill. Lines 6–8 process the buffer. If the buffer is filled, i.e., if there is a vertical fill, then the corresponding pixel is

filled. Lines 10–16 process each column of the corner image from left to right. If the pixel is a "1", then the decoder makes horizontal/vertical changes to the image. We first have to update the horizontal fill status (Line 10), fill the output pixel if necessary (Line 12), and update the buffer if necessary (Line 13). If the pixel is "0", the decoder makes no horizontal/vertical changes to the image, but fills the output pixels and updates the buffer according to the fill status. For example, if the decoder was previously filling the horizontal line, it keeps filling the line (Line 12) and updates the buffer (Line 13) in order to process the next row.

3.2.2 Frequent Pattern Reconstruction

If the decoder finds the string "$", it starts the pattern reconstruction process. Depending on the number P of patterns, the decoder reads $\lceil \log_2(P) \rceil$ pixels to determine which pattern p is used. Hence, the [$ `Pattern#`] stream specifies the pattern to reconstruct.

In order to perform row-wise decoding of these patterns, we use a row buffer (`BUFF`). When the decoder finds the pattern string [$ `Pattern#`], it places that string into the corresponding frequent pattern dictionary location to specify what pattern should be reconstructed. Furthermore, in order to specify which row of the pattern p the decoder should next process the decoder adds `Row#` which is a $\lceil \log_2(h_p - q_{h_p}) \rceil$ bit binary representation of the next row number after the pattern string, where q_{h_p} is one more than the number of empty rows on the bottom of the pattern. Observe that if the bottom r rows of the pattern p are empty, then the decoder needs to reconstruct $h_p - (r+1)$ more rows. Recall that the width of the pattern p, w_p, may be no less than $1 + \lceil \log_2(P) \rceil + \lceil \log_2(h_p - q_{h_p}) \rceil$.

For example, if we are decoding the compressed image in Figure 3.3, then the 3×3 square is stored in the decoder memory. After reading the $ symbol in the second row, the decoder processes the first row of the 3×3 square pattern, and fills the

corresponding three pixels of the second row. Then, it updates `BUFF` to [$0] starting from the leftmost corner of the pattern so that the decoder knows it should process the second row of the 3×3 square pattern. (Here we are assuming that $P = 1$ and so we can omit `Pattern#`. The last bit "0" indicates that the decoder now has to reconstruct the second row of the 3×3 square pattern. Since $h_p = 3$ and $q_{h_p} = 1$, we only need 1 bit for the purpose.) When the decoder processes the third row, it reconstructs the second row of the 3×3 square pattern by filling the three pixels, and updates `BUFF` to [$1] so that the decoder knows it should process the last row of the 3×3 square pattern for the next row. A similar procedure applies to the third row, and after processing the fourth row, `BUFF` is updated to [00] which terminates the reconstruction of the 3×3 square pattern.

In order to operate this frequent pattern reconstruction along with the corner transformation, the decoder requires $\lceil \log_2(3) \times width \rceil$ bits for the row buffer and P_{size} bits to store the entire pattern table.

3.3 Experimental Results

We tested the algorithm on two benchmark circuits introduced in Section 2.4. For the chips we studied we could run our algorithm `Corner2` on the entire layout image. We also considered the standard binary image compression algorithm `Joint Binary Image Group (JBIG)` [37] as well as `Block C4` [27] for comparison purposes. Note that because `JBIG` utilizes 2–3 lines of context-based prediction as well as a well-tuned arithmetic coding implementation, it requires keeping at least length $2 \cdot width$ bits of row buffer to apply row-by-row decoding. Furthermore, in order to update the prediction table, `JBIG` may require up to 2^{14} bytes of decoder memory which is larger than the requirements for the `Block C4` and `Corner2` decoding processes.

In our experiment, `Corner2` was written in C/C++ and `JBIG` was implemented in C/C++ using JBIG-KIT [38] and LibTIFF [39]. All of the experiments ran on a

laptop computer having a 2.53 GHz Intel Core 2 Duo CPU and 4 GB RAM.

3.3.1 Memory

For the memory circuit, `Corner2` used parameters $M = 64$, $N = 64$, $k = 1$, and $L = width$. We found that given the nearly even distribution of memory cells among the circuit layers a choice of $k > 1$ results in poorer performance than the choice $k = 1$ because of the frequency of all-zero rows in the images. We chose different M, N, and P_{size} depending on the layer so that the required decoder memory is similar to that of `Block C4` while giving us the best compression performance. The decoder memory sizes were 4.9 kBytes for `Corner2` and `Block C4`, and 20 kBytes for `JBIG`.

Compression Ratio (x)			
Layer	`Corner2`	`Block C4`	`JBIG`
1	18,658	148	28,169
2	1,216	80	169
3	18,658	148	28,169
4	86	54	83
5	920	133	555
6	920	133	555
7	84	32	53
8	67	24	53
9	170	38	100
10	57	25	75
11	442	76	228
12	3,169	122	3,576
13	9,616	141	9,188
Average	192	57	149

Table 3.1: `Corner2` Compression Ratio - Memory

Table 3.1 provides the compression ratios of the memory circuit layers for the algorithms we tested. The compression ratios are defined as

$$\frac{\texttt{Input File Size}}{\texttt{Compressed File Size}}.$$

Note that the last row of Table 3.1 is not the average of preceding rows, but the "net average" which is defined as

$$\frac{\texttt{Total Input File Size}}{\texttt{Total Compressed File Size}}.$$

`Corner2`, which utilizes a corner transform, attains compression ratios which are 1.2–3.3 times higher than those for Block C4. Furthermore, `Corner2` outperforms `JBIG` by 29.3%.

Encoding Time (s)			
Layer	`Corner2`	`Block C4`	`JBIG`
1	4.29	1,875	7.15
2	26.36	1,891	7.56
3	4.29	1,788	7.16
4	48.93	1,815	7.51
5	26.85	1,831	7.24
6	26.86	1,796	7.24
7	25.27	1,846	7.90
8	109.97	1,885	7.93
9	30.18	1,817	7.47
10	78.49	1,917	7.90
11	80.22	1,795	7.30
12	4.16	1,775	7.24
13	4.16	1,762	7.04
Total	470.05	23,793	96.64

Table 3.2: `Corner2` Encoding Time - Memory

Table 3.2 and Table 3.3 shows the encoding and decoding times of the algorithms on the memory circuit layers. The last row of the tables offer the total run times of the encoding and decoding processes. As we can see from Table 3.2, the encoding time of `Corner2` was 51 times faster than that of `Block C4`. While both `Block C4` and `Corner2` utilize the frequent patterns, `Corner2` runs much faster because it doesn't use the image to decide which patterns are frequent. Rather, it analyzes the input GDSII file. The decoding time of `Corner2` was 18 times faster than that of `Block`

Decoding Time (s)			
Layer	`Corner2`	`Block C4`	`JBIG`
1	2.53	55.44	2.94
2	3.34	54.99	3.32
3	2.53	52.90	2.94
4	3.20	53.70	3.31
5	3.05	53.55	3.01
6	3.05	54.48	3.00
7	3.23	54.63	3.68
8	3.51	54.79	3.67
9	3.05	54.83	3.28
10	3.44	54.07	3.66
11	3.04	56.43	3.12
12	2.49	51.50	3.07
13	2.48	51.88	2.89
Total	38.93	703.18	41.90

Table 3.3: `Corner2` Decoding Time - Memory

`C4` and 8% faster than that of `JBIG` as shown in Table 3.3.

Observe that the memory circuit is covered entirely with a single memory cell pattern, and so $P = 0$ (which happens when the memory cell did not contain any polygon in the corresponding layer) or 1.

3.3.2 BFSK

For the BFSK circuit, `Corner2` was run with parameters $M = 64$, $N = 64$, $k = 4$, $L = 8192$, and $P_{size} = 10$ bytes. These choices allowed for similar decoder memory sizes for `Corner2` and `Block C4`. For these parameter settings and this circuit the decoder memory sizes for `Corner2`, `Block C4`, and `JBIG` were respectively 7.9 kBytes, 8.4 kBytes, and 16 kBytes.

Table 3.4 shows the compression ratios of the algorithms. `Corner2` attain a 3.5–4.6 times higher compression ratio than `Block C4`. `Block C4` relies on context prediction and finding repeating regions within an image. By contrast, `Corner2` use actual polygon corners, and these corners cannot be predicted correctly by `Block C4`'s context

Compression Ratio (x)			
Layer	`Corner2`	`Block C4`	`JBIG`
1	9,226	151	11,103
2	6,760	150	10,855
3	1,186	137	1,985
4	3,270	147	5,503
5	1,033	59	1,560
6	580	128	902
7	390	111	600
8	167	81	128
9	452	122	1,044
10	195,365	153	109,570
11	200	86	151
12	1,006	138	2,025
13	203	89	151
14	1,093	139	2,228
15	230	92	169
16	1,296	141	2,605
17	6,656	150	10,923
18	19,579	152	22,939
19	9,173	151	11,027
Average	515	113	486

Table 3.4: `Corner2` Compression Ratio - BFSK

predictor. Furthermore, `Corner2` outperforms `JBIG` by 6%.

Table 3.5 and Table 3.6 show the run times of the algorithms on the BFSK circuit layers. As we can see from Table 3.5, the encoding time of `Corner2` was 196 times faster than that of `Block C4`, which reduced the complexity of the earlier algorithm `C4` [40] by segmentation. The improvement over `Block C4` is due to the relative computational complexity of the context-based prediction and region copy components of the `C4` / `Block C4` algorithms.

The decoding process for `Corner2` is 29.6 times faster than that of `Block C4`. It was also 1.4 times faster than than of `JBIG` as shown in Table 3.6.

Encoding Time (s)			
Layer	`Corner2`	`Block C4`	`JBIG`
1	2.41	451	2.41
2	17.88	3,791	20.92
3	18.31	3,800	22.74
4	17.21	3,760	21.54
5	28.07	12,468	23.05
6	26.43	4,485	24.09
7	37.09	4,476	29.68
8	17.91	4,588	31.17
9	35.56	4,393	30.12
10	20.73	4,866	22.97
11	17.90	4,414	29.65
12	33.20	4,428	27.16
13	17.88	4,651	29.56
14	34.38	4,413	27.02
15	17.80	4,364	29.03
16	38.54	4,381	25.61
17	15.74	3,756	20.72
18	14.26	3,049	16.09
19	2.42	454	2.41
Total	413.75	80,987	435.95

Table 3.5: `Corner2` Encoding Time - BFSK

Decoding Time (s)			
Layer	`Corner2`	`Block C4`	`JBIG`
1	0.84	29.50	0.98
2	8.22	242.07	8.78
3	8.24	237.42	10.53
4	8.22	237.35	9.41
5	8.35	280.67	10.83
6	8.33	274.15	11.87
7	9.74	282.03	15.53
8	9.68	289.53	17.04
9	9.67	275.56	15.92
10	9.41	293.04	9.32
11	9.64	281.57	15.57
12	9.57	278.34	13.24
13	9.62	290.37	15.44
14	9.57	276.35	12.90
15	9.60	277.88	15.01
16	9.59	275.98	11.64
17	8.21	236.50	8.70
18	6.47	190.71	6.63
19	0.84	27.68	0.98
Total	153.81	4,576.71	210.33

Table 3.6: `Corner2` Decoding Time - BFSK

CHAPTER IV

FPGA Implementation of Corner2 Decoder

To enable hardware implementation the `Corner2` decoding process is designed to operate in a row-by-row fashion. That is, instead of decoding the entire layout image at once using large memory, the decoder is able to decode a row of the layout image using limited memory. Because of these properties, we claimed that the `Corner2` decoder can be implemented in hardware with limited decoder memory (or cache).

In this chapter, we present an FPGA implementation of the `Corner2` decoder. The rest of this chapter consists of two parts; we explain the `Corner2` decoder architecture in Section 4.1 and report the FPGA synthesis results in Section 4.2.

4.1 `Corner2` Decoder Architecture

The `Corner2` decompression process is shown in Figure 4.1. First, the compressed bit stream is decoded at the arithmetic decoder. Second, the decoded symbol stream goes through the run length and EOB decoding process reconstructing the (ternary) transformed image. Third, the ternary image is handled by the inverse transformation block to reconstruct the original image. The inverse transformation process handles both frequent pattern reconstruction for the regular circuit parts and inverse corner transformation for the irregular circuit parts. The entire inverse transformation process is applied in a row-by-row fashion requiring limited decoder memory. This was

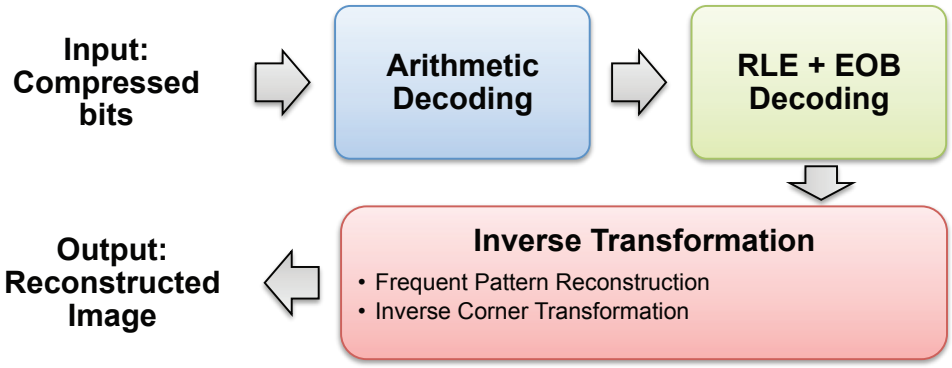

Figure 4.1: Overview of the `Corner2` Decompression Process

possible because the FPR and corner transformation were designed specifically with row-by-row decoding in mind.

As we have discussed in Chapter III, the decoder requires $\lceil \log_2(3) \times width \rceil$ bits to keep intermediate information for row-by-row decoding, P_{size} bits of memory for the frequent pattern dictionary, and $32(M + N + 2)$ bits for arithmetic decoding, where $width$ is the width of the layout image and M and N are RLE/EOB parameters.

Since the entropy decoding part is straightforward we are more interested in the "inverse transformation" block of the `Corner2` decoder. The inverse transformation is applied to a row of the ternary transformed image to produce a row of the decompressed layout image. The inverse transformation block has the following architecture shown in Figure 4.2 [41]. It consists of two main processes - the inverse corner transformation and the frequent pattern reconstruction. Both processes utilize the row buffer for row-by-row decoding, and the frequent pattern reconstruction process requires an additional frequent pattern dictionary which stores the frequent pattern information such as each pattern's width, height, and image.

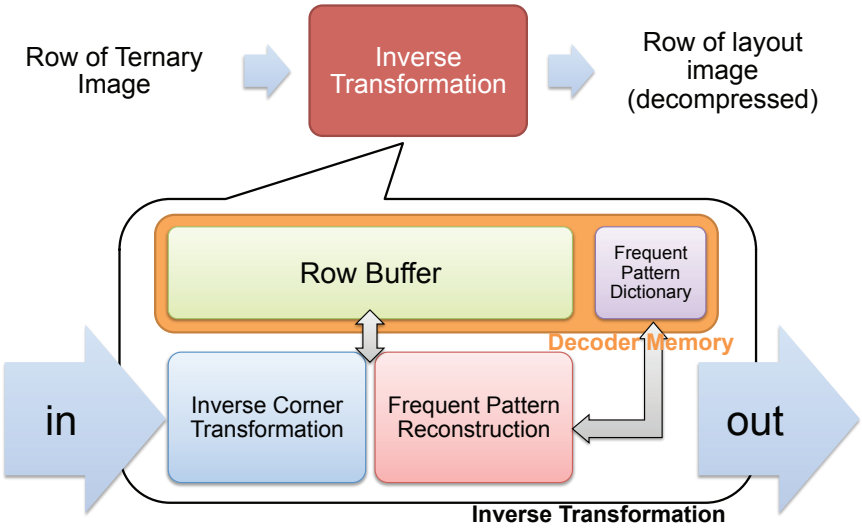

Figure 4.2: Architecture of the `Corner2` inverse transformation block

4.2 FPGA Synthesis Results

Since the `Corner2` decoding algorithm [42] was implemented to simulate a decoder circuit implemented in hardware, we were able to design the decoder circuit without major modifications. We used `Impulse CoDeveloper` [43] to generate hardware description language (HDL) files from the `Corner2` decoder source code and used `Xilinx` CAD tools to design the FPGA circuit from the HDL files.

The designed `Corner2` decoder consists of the RLE and EOB decoders and the inverse transformation process of Figure 4.1. The arithmetic decoder was omitted because we used a well-known implementation of arithmetic coding [35] and there are various arithmetic decoders implemented in hardware [36, 44].

Table 4.1 shows our custom `Corner2` decoder with 5 kbytes of decoder memory. It was implemented on a `Xilinx Spartan-3E XC3S500E` FPGA board. It only required 187 slice flip flops and 236 four-input Look-Up Tables (LUT) which corresponds to 2% of the overall FPGA resources. Among the 236 four-input LUT, only 104 of them were used for logic while the remaining 132 of them were used for dual port RAMs. The system was tested at a 100 MHz clock rate, which accommodates the critical

Device	`Xilinx Spartan-3E XC3S500E`
Number of slice flip flops	187 / 9,312 (2%)
Number of 4 input LUTs	236 / 9,312 (2%)
System clock rate	100 MHz

Table 4.1: FPGA synthesis result of the `Corner2` decoder

data path of the system after the synthesis, which had a delay of 9.55 ns.

Considering an FPGA implementation of an MQ-Decoder - a variation of arithmetic decoder - requires 813 slice flip flops out of 33,792 and 1,329 LUT out of 67,584 on a `Xilinx Virtex II XC2V6000-6` FPGA board at a maximum system clock rate of 146.5MHz [44], the effect of the `Corner2` decoder excluding the arithmetic decoder is small. Since the `Corner2` decoder circuit was designed using a `C-to-FPGA` conversion tool, we expect that the FPGA circuit and its ASIC design can both be improved, and hence the `Corner2` decoder is suitable for delivering mask layout data to the maskless lithography system.

As we have shown in Chapter III, the `Corner2` compression algorithm has many advantages over `Block C4` [27]; it has a better compression ratio and a higher decoding throughput while requiring less decoder memory. We have shown here that the `Corner2` decoder can be implemented as an FPGA circuit. The FPGA implementation suggests that the `Corner2` decoder only requires a small silicon area while having high throughput. We expect its ASIC design would work fast enough to improve the data delivery throughput and would be small enough to be added on to the lithography writer.

CHAPTER V

Improving Corner2 Frequent Pattern Discovery

In Chapter III, we have introduced the `Corner2` algorithm. The `Corner2` algorithm utilizes dictionary-based compression to handle repeated circuit components and applies a transform which is specifically tailored for circuit layout images to deal with irregular circuit components. It obtains high compression ratios and fast encoding/decoding times while requiring limited decoder memory on the decoder hardware. Moreover, we have shown in Chapter IV, the entire decompression is simple so that it could be implemented in a hardware add-on to the lithography writer.

However, there is some room for improvement in how the dictionary is built to handle frequent circuit patterns because the `Corner2` patterns are extracted from the layout description given in the GDSII or OASIS formats and not from the layout image itself. Because the rasterization process can produce different bitmap image patterns for the polygons that have the same layout description depending on the rasterizing grid, polygon size, and polygon displacement, the patterns that `Corner2` utilizes are far from perfect.

In this chapter, we will introduce an algorithm that discovers the polygons that could be used as frequent patterns for the `Corner2` encoding process. Moreover, we will develop an algorithm to limit the number of patterns that can form the frequent pattern dictionary to keep the dictionary small while replacing as many patterns as

possible for efficient compression.

This chapter consists of four sections: First, we point out the problem of `Corner2` in Section 5.1. Second, we will introduce the candidate pattern generation algorithm in Section 5.2. Third, we illustrate how we can choose the optimized frequent pattern list from the candidate patterns in Section 5.3. In Section, 5.4 we show experimental results on how the new patterns improve the `Corner2` compression performance.

5.1 Problems of `Corner2` Pattern Discovery Algorithm

The detailed frequent pattern discovery algorithm of `Corner2` is shown in Figure 5.1. The layout description (GDSII) of the entire circuit and the layer number of the targeting layout image is input to the algorithm. Then the encoder analyzes the entire circuit to find the frequent substructures and rasterizes the corresponding substructures at the specified layer.

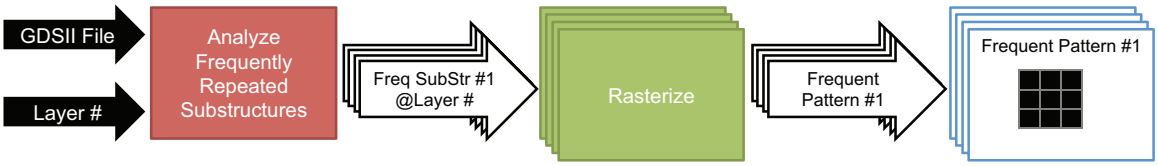

Figure 5.1: Frequent Pattern Discovery in `Corner2`

During the generation of binary layout images an image could be truncated if there is a mismatch between the pixel grid and the GDSII grid. Considering the pixel grid is much coarser than the GDSII grid, a pattern could be truncated when it is realized as a binary image; i.e., a substructure could be repeated within the GDSII domain but the corresponding pattern may not repeatedly match within the binary image domain.

The example in Figure 5.2 shows how the rasterization process can produce such results. In the example, two 5 $nm\times$ 5 nm squares are defined in the layout description (GDSII). When we rasterize the image in a 4 nm grid (Figure 5.2. right), we are

actually gathering a 4×4 block as a single pixel from the 1 nm grid (Figure 5.2. left) and fill the pixel if the number of filled pixels in the block is no less than 8. The rasterized image shown in Figure 5.2. right has two different polygons, but they came from the same polygon description. This pattern truncation problem occurs in general because the layout description grid and the rasterizing grid do not perfectly match, and this causes a performance deterioration in `Corner2`.

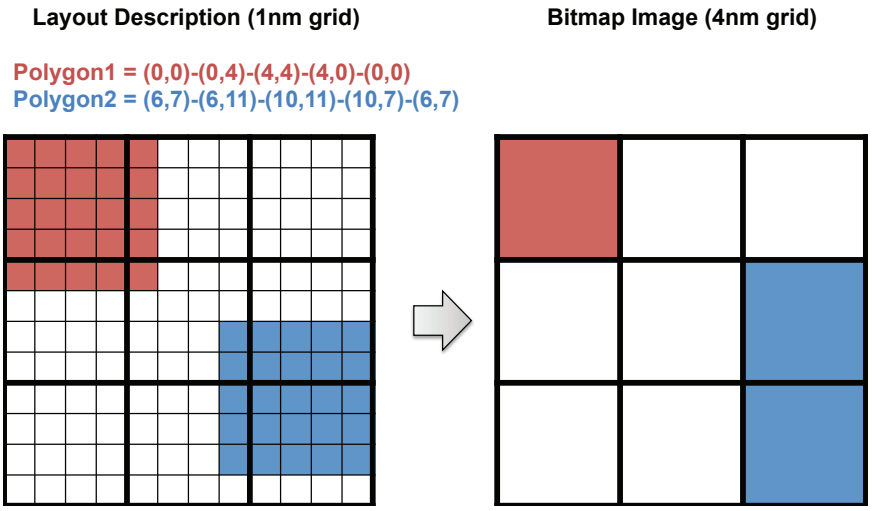

Figure 5.2: Example of pattern mismatch due to rasterization

5.2 Candidate Pattern Generation Algorithm

Unlike the `Corner2` approach, we extract the patterns from the layout image itself without the help of the layout description (GDSII). By searching the layout images, we generate a list of candidate patterns and from the candidate patterns we later determine which one should be included in the frequent pattern list. In this section, we will illustrate the candidate pattern generation algorithm. As we explained in Section 3.1.2, during the FPR process we match the pattern by isolating it; i.e., surrounding the pattern with empty rows on the top and bottom and columns on the left and right. Because of this property, the patterns that can be used as the frequent patterns should follow the following conditions:

1. the patterns should be defined in a rectangular region,

2. the patterns should never overlap with other patterns, and

3. the patterns should be isolated.

By isolating the patterns and avoiding pattern overlap, we prevent a polygon from being partially covered by patterns. This is not required by the decoder, but if there is partial pattern coverage, then the pattern usage will be less efficient and will harm the compression performance. Therefore, we want to prevent this. Moreover, it also helps to reduce the complexity of the candidate pattern generation algorithm.

We will explain how the candidate patterns are generated below. In the following subsection, we determine which candidate patterns will be included in the frequent pattern dictionary. The candidate pattern generation algorithm is shown in Algorithm 4 [45]. In the algorithm, x is the column index $[1, \cdots, C]$ of the image and y is the row index of the image $[1, \cdots, R]$.

The algorithm starts by picking a pixel from the layout image in raster order, i.e., from top to bottom and then from left to right. If the pixel is filled (1), then we define a rectangular region $(x_0, y_0) - (x_1, y_1)$ so that all of the filled pixels that are connected to (x, y) are covered by it (Line 6). Here, we say pixel (x_i, y_i) is *connected* to pixel (x_j, y_j) if both pixels are filled and $|x_i - x_j| \leq 1$ and $|y_i - y_j| \leq 1$. We then make the rectangular region $(x_0, y_0) - (x_1, y_1)$ as the candidate pattern `Pattern` (Line 7) and search the list of candidate patterns `PatternList` (Line 8) to see whether the pattern was already in the list or not. If `Pattern` was already in the `PatternList`, we increase its frequency by 1 (Line 10). Otherwise, we put `Pattern` to the `PatternList` and initialize its frequency to 1 (Line 12). Finally, we make sure that the region is not searched again marking the region in `Checked` (Lines 14–18), and this prevents the patterns from overlapping.

Algorithm 4 Candidate Pattern List Generation

Input: Layout image `IN` $\in \{0,1\}^{C \cdot R}$
Output: List of patterns `PatternList`
Intermediate: List of patterns `Checked` $\in \{0,1\}^{C \cdot R}$

```
1: Initialize Checked(x, y) = 0, ∀x, y.
2: Initialize P = 0.
3: for y = 1 to R do
4:   for x = 1 to C do
5:     if Checked(x, y) = 0 and IN(x, y) = 1 then
6:       (x0, y0, x1, y1) = DefinePatternRegion(x, y)
7:       Pattern = MakePattern(x0, y0, x1, y1)
8:       p = PatternList.Find(Pattern)
9:       if p ≠ NOT_FOUND then
10:        PatternList[p].frequency += 1
11:      else
12:        PatternList.Insert(Pattern, 1)
13:      end if
14:      for yy = y0 to y1 do
15:        for xx = x0 to x1 do
16:          Checked(xx, yy) = 1
17:        end for
18:      end for
19:    end if
20:  end for
21: end for
```

While the patterns Algorithm 4 produces are isolated polygons, some of the patterns that are extracted from the GDSII description in Figure 5.1 were groups of isolated polygons. We found that for the case the candidate patterns generated by Algorithm 4 were small compared to the patterns extracted from the GDSII layout description and deteriorated the compression performance when they were used solely. Therefore, along with the candidate patterns generated by Algorithm 4, we use the frequent patterns discovered from the original `Corner2` algorithm and use the entire pattern list as the candidate patterns in order to ensure that the new algorithm always outperform the original `Corner2` algorithm.

To incorporate the GDSII extracted patterns with the Algorithm 4 patterns, we first run the FPR algorithm of the original `Corner2` algorithm using the GDSII ex-

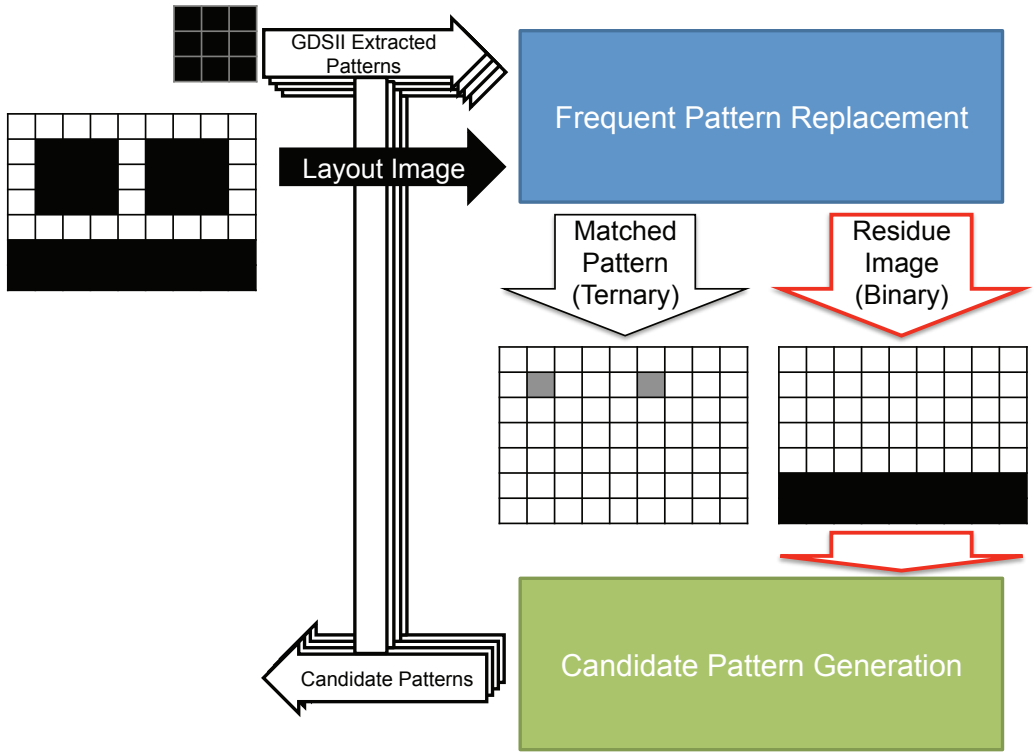

Figure 5.3: Candidate Patterns for `Corner2-BIP`

tracted patterns only as in Figure 5.3. Then we apply Algorithm 4 to the residue image which is the image region that has not been matched by the FPR process. By combining both the GDSII extracted patterns and the patterns generated using Algorithm 4, we obtain the final candidate pattern list.

5.3 Pattern Optimization

Once the candidate patterns are discovered, we analyze them in order to decide which patterns to keep and which patterns to discard. There are the two parameters that we consider to make this decision. The first is *gain*, which provides information on what improvement we should expect by using the pattern for the FPR process. Since compression is related to the corners we remove by patterns as well as the frequency of patterns, we define the gain of pattern p as :

$$Gain_p = [N(C_p) - 1] \times N(F_p),$$

where $N(C_p)$ is the number of corners of pattern p and $N(F_p)$ is the frequency of pattern p.

The second parameter is *cost* which shows how much decoder memory is required to keep the pattern in the decoder memory. For the pattern p whose dimension is $w_p \times h_p$, the decoder usually needs $w_p \times h_p$ bits of memory to store the pattern. However, we can reduce the cost when the pattern is fully filled (except the top row and leftmost column). For this case, since we already know all pixels are filled all we need to store is the pattern dimensions. We allocate 16 bits for each dimension and 1 bit flag to specify whether or not it is a fully filled pattern. Therefore the cost of pattern p with dimension $w_p \times h_p$ is defined as follows:

$$Cost_p = \begin{cases} 33, & \text{if pattern } p \text{ is fully filled} \\ 33 + w_p \times h_p, & \text{otherwise} \end{cases}$$

To reduce the complexity of the optimization problem we initially discard the candidate patterns whose gains were less than a preset threshold $Threshold$.

To choose the frequent patterns, we want to

$$\text{maximize } \sum_p Gain_p \cdot x_p \text{ such that } \sum_p Cost_p \cdot x_p \leq P_{size}, \tag{5.1}$$

where x_p is a binary number indicating whether pattern p should be used as a frequent pattern (1) or not (0), and P_{size} is the decoder memory in bits that can be used to store the frequent pattern dictionary.

Optimization (5.1) is an instance of a standard Binary Integer Programming (BIP)

problem [46]

$$\begin{aligned} &\text{maximize} && \mathbf{c}^T\mathbf{x} \\ &\text{subject to} && \mathbf{a}^T\mathbf{x} \leq b \\ &\text{and} && \mathbf{x} \in \{0,1\} \end{aligned} \tag{5.2}$$

by setting $\mathbf{c} = [Gain_1\ Gain_2\ \cdots\ Gain_G]$, $b = P_{size}$, and $\mathbf{a} = [Cost_1\ Cost_2\ \cdots\ Cost_G]$.

Here G is the number of candidate patterns which are passed onto the BIP solver after thresholding. By applying a widely used BIP solver [47], we are able to choose the optimal frequent patterns from the generated candidate pattern list. In order to keep the BIP solver efficient, we adaptively incremented the *Threshold* so that the number of patterns the BIP solver has to consider is limited.

5.4 Experimental Results

We tested how the improved frequent pattern discovery algorithm affects the results on the benchmark circuits we introduced in Section 2.4.

In our experiment, we ran the `Corner2` algorithm with two frequent pattern dictionaries. The frequent patterns of `C2-ORG` were extracted from the layout description (GDSII) as in the original `Corner2` algorithm in Chapter III while those of `C2-BIP` (or `Corner2-BIP`) were extracted by generating pattern candidates as in Section 5.2 and were chosen by solving the binary integer programming problem (5.1). Most of `C2-ORG` and `C2-BIP` was written in C/C++. Only the binary integer programming part was written in MATLAB using the MATLAB function `bintprog` [47]. All of the experiments ran on a laptop computer having a 2.53 GHz Intel Core 2 Duo CPU and 4 GB RAM.

5.4.1 Memory

For the memory circuit there were relatively few candidate patterns because of the circuit regularity, and hence, adaptive thresholding did not take place to discard

any of candidate patterns before the BIP solver. We set P_{size} for `C2-BIP` so that the total decoder memory matches that of `Block C4` while P_{size} for `C2-ORG` was set by the layout description and cannot be controlled. As shown in the table, `C2-BIP` outperforms `C2-ORG` and `Block C4` on every layer.

Compression Ratio (x)			
Layer	`C2-BIP`	`C2-ORG`	`Block C4`
1	22,201	18,658	148
2	1,289	1,216	80
3	22,201	18,658	148
4	151	86	54
5	920	920	133
6	920	920	133
7	84	84	32
8	90	67	24
9	171	170	38
10	114	57	25
11	442	442	76
12	11,059	3,169	122
13	28,606	9,616	141
Average	261	192	57

Table 5.1: `Corner2-BIP` Compression Ratio - Memory

Table 5.1 shows the compression ratio of the algorithms for the memory circuit. On average `C2-BIP` achieved 35.8% better compression than `C2-ORG`, which was already 3.3 times better than `Block C4`. We can also see that `C2-BIP` outperforms both `C2-ORG` and `Block C4` in all layers.

Moreover, as we can see from Table 5.2, even though the entire process was more complex, the encoding process of `C2-BIP` is only 3.3% slower than that of `C2-ORG` because Algorithm 4 was only applied to the region where `C2-ORG` failed to match and that region was relatively small. Moreover, the BIP solver was efficient especially when the candidate pattern set was small. Therefore, most of the encoding was absorbed by the FPR algorithm of `C2-ORG`.

`C2-BIP` was also able to decode the compressed image 3.3% faster than `C2-ORG` as

Encoding Times (s)			
Layer	C2-BIP	C2-ORG	Block C4
1	18.89	4.29	1,875
2	26.48	26.36	1,891
3	18.36	4.29	1,788
4	49.02	48.93	1,815
5	25.26	26.85	1,831
6	25.28	26.86	1,796
7	23.07	25.27	1,846
8	97.32	109.97	1,885
9	25.37	30.18	1,817
10	84.16	78.49	1,917
11	70.68	80.22	1,795
12	6.23	4.16	1,775
13	6.52	4.16	1,762
Total	485.45	470.05	23,793

Table 5.2: `Corner2-BIP` Encoding Times - Memory

Decoding Times (s)			
Layer	C2-BIP	C2-ORG	Block C4
1	2.40	2.53	55.44
2	3.16	3.34	54.99
3	2.42	2.53	52.90
4	2.96	3.20	53.70
5	2.95	3.05	53.55
6	2.90	3.05	54.48
7	3.21	3.23	54.63
8	3.21	3.51	54.79
9	2.79	3.05	54.83
10	2.98	3.44	54.07
11	2.78	3.04	56.43
12	2.95	2.49	51.50
13	2.94	2.48	51.88
Total	37.67	38.93	703.18

Table 5.3: `Corner2-BIP` Decoding Times - Memory

shown in Table 5.3. This is mainly due to the increase in FPR and the decrease in corner transformation. Since the application is compress-once-and-decode-multiple-times, the decreased decoding time makes the algorithm more efficient.

5.4.2 BFSK

Tables 5.4 – 5.6 show the experimental results for the BFSK circuit. The BFSK circuit had a large number of candidate patterns, and we had to discard some of them to offer a small enough input to the BIP solver using adaptive thresholding. As shown in the table, `C2-BIP` outperforms `C2-ORG` and `Block C4` on every layer. Similarly to the memory circuit, the encoding process of `C2-BIP` is slower than that of `C2-ORG`. Most of the complexity is induced by the circuit irregularity. Because the BFSK circuit was irregular Algorithm 4 took longer to run. Since most of the candidate patterns had relatively small gains only a few patterns were passed on to the BIP solver resulting in a slowdown of the entire encoding process by only 13.7% over `C2-ORG`.

Compression Ratio (x)			
Layer	`C2-BIP`	`C2-ORG`	`Block C4`
1	20,874	9,226	151
2	7,432	6,760	150
3	1,596	1,186	137
4	3,745	3,270	147
5	1,125	1,033	59
6	609	580	128
7	399	390	111
8	173	167	81
9	474	452	122
10	460,192	195,365	153
11	206	200	86
12	1,082	1,006	138
13	209	203	89
14	1,170	1,093	139
15	236	230	92
16	1,368	1,296	141
17	8,019	6,656	150
18	26,467	19,579	152
19	20,773	9,173	151
Average	537	515	113

Table 5.4: `Corner2-BIP` Compression Ratio - BFSK

Encoding Times (s)			
Layer	C2-BIP	C2-ORG	Block C4
1	1.74	2.41	451
2	16.33	17.88	3,791
3	15.47	18.31	3,800
4	15.11	17.21	3,760
5	26.62	28.07	12,468
6	26.28	26.43	4,485
7	54.74	37.09	4,476
8	15.36	17.91	4,588
9	58.11	35.56	4,393
10	15.93	20.73	4,866
11	17.56	17.90	4,414
12	42.59	33.20	4,428
13	15.00	17.88	4,651
14	50.16	34.38	4,413
15	17.29	17.80	4,364
16	53.06	38.54	4,381
17	14.30	15.74	3,756
18	13.21	14.26	3,049
19	1.73	2.42	454
Total	470.59	413.75	80,987

Table 5.5: `Corner2-BIP` Encoding Time - BFSK

On average `C2-BIP` achieved 4.5% better compression than `C2-ORG`, which was already 4.6 times better than `Block C4`. Since the difference between `C2-BIP` and `C2-ORG` is on how `C2-BIP` detects the circuit regularity, we expected to achieve only a marginal improvement for irregular circuits such as the BFSK circuit. Finally, `C2-BIP` was also able to decode the compressed image 3.4% faster than `C2-ORG`.

Decoding Time (s)			
Layer	C2-BIP	C2-ORG	Block C4
1	0.80	0.84	29.50
2	6.77	8.22	242.07
3	6.63	8.24	237.42
4	6.57	8.22	237.35
5	7.29	8.35	280.67
6	7.26	8.33	274.15
7	12.08	9.74	282.03
8	9.71	9.68	289.53
9	10.91	9.67	275.56
10	7.61	9.41	293.04
11	10.10	9.64	281.57
12	10.89	9.57	278.34
13	10.32	9.62	290.37
14	10.46	9.57	276.35
15	9.33	9.60	277.88
16	9.34	9.59	275.98
17	6.49	8.21	236.50
18	5.23	6.47	190.71
19	0.79	0.84	27.68
Total	148.58	153.81	4,576.71

Table 5.6: `Corner2-BIP` Decoding Time - BFSK

CHAPTER VI

Tailoring Corner2 for Multiple Electron Beam Direct Write Systems

Based on the data delivery system introduced by Dai and Zakhor [40], we developed in Chapter III a lossless compression algorithm, `Corner2` that has better compression performance than `Block C4` [27] for both regular and irregular circuits. In Chapter V, we improved the frequent pattern discovery algorithm of `Corner2` by using isolated polygons, i.e., polygons that are separated from each other, as candidate patterns and by solving an integer programming problem. The result shows that the `Corner2-BIP` algorithm obtains high compression ratios and fast encoding/decoding times while requiring limited decoder cache on the decoder hardware. Moreover, we have shown that the entire decompression is simple so that it could be implemented as a hardware add-on to the electron beam writer in Chapter IV.

However, neither `Corner2` nor `Corner2-BIP` were optimized for the MEB systems that are currently under development. `Corner2` and `Corner2-BIP` assume the decoder can write in a row-by-row fashion with a raster order, i.e. from top to bottom and from left to right, but neither MAPPER [8], IMS [9], nor REBL [10] utilizes raster writing. In fact, REBL writes a 4096×248 block at a time to produce a large pixel with multilevel electron beam dosage, while MAPPER and IMS have electron beam writers positioned in a lattice formation allowing each electron beam writer to write

a designated block in a zig-zag order.

In this chapter we introduce `Corner2-MEB` [48], which is a modified `Corner2-BIP` algorithm suitable for MAPPER systems. We redesigned the algorithm to support both block processing and a zig-zag writing order. The new algorithm can also be applied to IMS systems, but it does not appear to perform well on them because of their extremely small block size (16×16). The experimental results show that for MAPPER systems there are performance deteriorations because of block processing, but the `Corner2-MEB` algorithm still attains a high performance.

The rest of this chapter consists of three parts; in Sections 6.1 and 6.2 we describe the `Corner2-MEB` encoding and decoding processes. We show experimental results in Section 6.3.

6.1 The Compression Algorithm

Figure 6.1 shows an overview of the `Corner2-MEB` compression algorithm. First, the layout image is separated into blocks so that each block is written by a single electron beam writer. Second, we detect the frequent patterns within the blocks. In order to do that, we first extract the frequent patterns from the GDSII description of the entire layer image as in Section 3.1.2, generate candidate patterns from the individual image blocks and choose the optimized frequent pattern list from all of the candidate patterns as in Section 5.2. Third, each block goes through a forward transformation process which replaces the frequent patterns from the image blocks and applies the corner transform to the unmatched parts. Note this process is similar to the FPR and corner transform processes in the original `Corner2` algorithm in Section 3.1, but it has been revised so that each image block is later reconstructed by the decoder in a zig-zag order. Fourth, the pixels of the encoded image blocks are flattened to form a pixel stream so that the decoder can at each time simultaneously reconstruct a pixel for each block. Finally, the flattened pixel stream is compressed

to a bit stream using entropy coding technology - RLE, EOB coding, and arithmetic coding - as in the original `Corner2` algorithm shown in Section 3.1.5. In the following subsections, we will explain each step in detail.

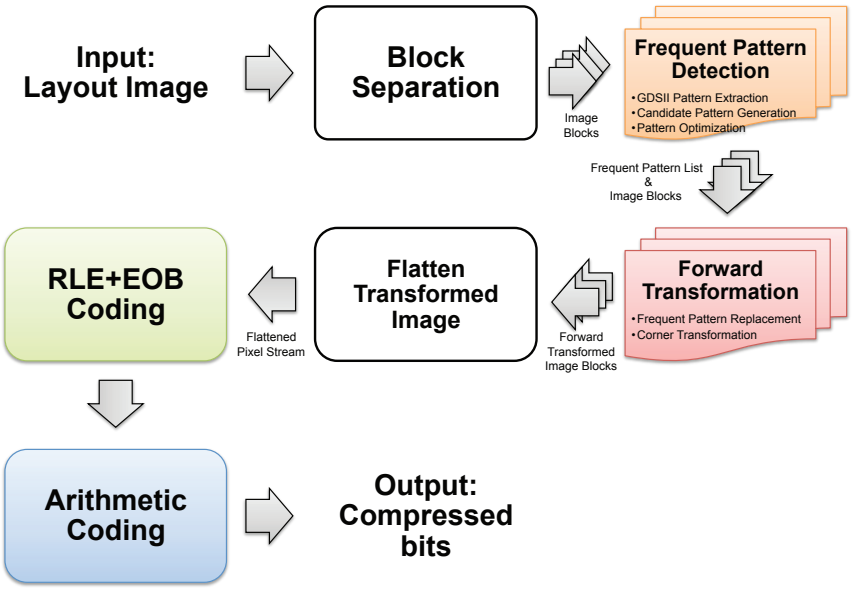

Figure 6.1: `Corner2-MEB` Compression Algorithm Overview

In Figure 6.1, the Frequent Pattern Detection and Forward Transformation algorithms refer to processes that are applied to individual image blocks as opposed to the full layout image. Similarly, the input/output of those two components is an image block and not the full image. For example, the forward transformation process is applied independently to each image block to produce the corresponding transformed image block.

6.1.1 Block Separation

Since each electron beam writer has a limited writing region, we need to make sure the image that we pass to each electron beam writer corresponds to its writing region, which is a block. Therefore, we segment the layout image into blocks. Figure 6.2 shows the writing strategy of the MAPPER lithography system [8]. In the right part of the figure, the green pillars show the electron beam positions. While the electron

beams are moving in a horizontal zig-zag order and turned on or off depending on the control signal, the stage where the wafer is fixed is moving vertically so that the electron beams write a tall rectangular stripe region on the wafer. Each electron beam writes a region which is 2 μm wide and the 26 mm tall in a horizontal zig-zag order. Once the electron beams write 26 mm tall blocks, they continuously repeat writing the blocks until they reach the end of wafer. That is, each electron beam writes a 2 μm wide stripe with the height of the wafer region that it covers.

The left part of the figure shows how the movement of the stage affects the writing strategy on the wafer. By moving the stage in a vertical zig-zag order, the MAPPER system with 13,000 electron beams writes a 26 mm wide stripe during a single vertical scan. By writing these 26 mm wide stripes on the wafer in a vertical zig-zag order, we are actually printing an upside down copies of the circuit layout image for the stripes written from the bottom to the top. This could be a problem if a circuit layout image has to be covered by multiple stripes. However, the 26 mm width of a stipe is wide enough to cover most circuits; for example, the previous generation 45 nm Intel Core i7-920 process had a die size of 263 mm^2 [49].

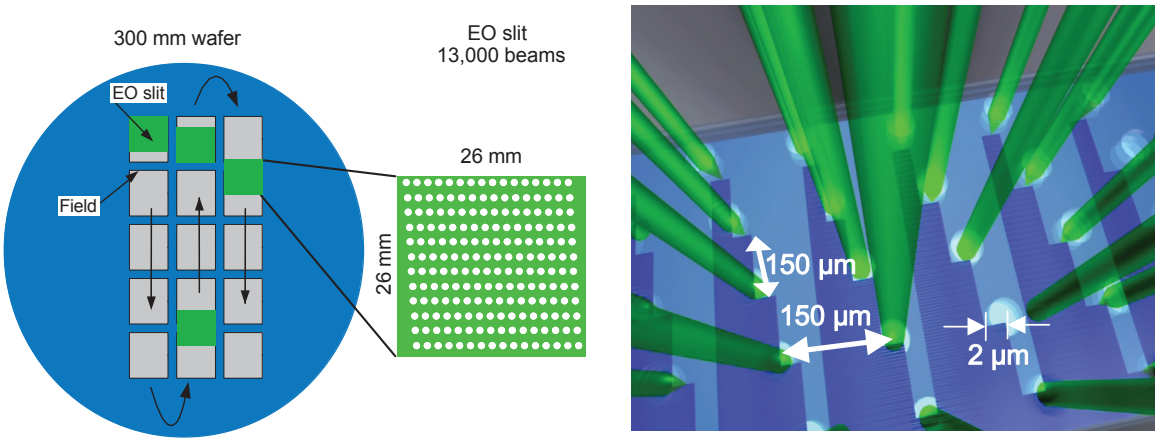

Figure 6.2: The MAPPER writing strategy

Observe that since MAPPER staggers its writing regions in the vertical direction by 150 μm as shown in the right part of Figure 6.2, we cannot simply partition the

image into blocks and pass each block to the corresponding electron beam writer. The writing strategy of the electron beam writers to create the circuit layout image on the wafer is shown in Figure 6.3. Here, the starting position of each electron beam in each writing is marked with a black square and the writing region of each electron beam writer is illustrated as a tall rectangular block where the rows are successively written from the top row to the bottom row in a horizontal zig-zag order. Whenever an electron beam writer finishes writing a block, it repeats writing the same block until it reaches the end of the wafer region. In the example of Figure 6.3, each electron beam writer writes three copies of the blocks (represented by "Writings #1 – #3"), and two copies of the circuit layout image (represented by "Circuits #1 and #2") are printed on the wafer.

In order to emphasize that some blocks can contribute to two different circuit layout image copies, we used different coloring (light and dark) for the corresponding blocks. For example, block #2 of Writing #2 consists of two parts block 2-2-1 and block 2-2-2. While block 2-2-1 is part of Circuit #1 along with the second part of block #2 of Writing #1 (block 2-1-2), block 2-2-2 is part of Circuit #2 along with the first part of block #2 of Writing #3 (block 2-3-1). Finally, note that in Figure 6.3 all blocks with the same color are identical. That is, block 2-1-1, block 2-2-1, and block 2-3-1 are identical and block 2-1-2, block 2-2-2, and block 2-3-2 are identical.

Therefore, the block separation process reinterprets the circuit layout image (or Circuit) in the order of the Writings of Figure 6.3. Figure 6.4 shows how this process changes the circuit layout image. The top of Figure 6.4 shows a "Circuit" of Figure 6.3, which is a circuit layout image. To obtain the block separated image, we first partition the circuit layout image. Then, we apply circular shifts to the blocks that contribute to two different circuit layout image copies in the Figure 6.3 – the corresponding blocks are blocks #2 – #4 and #6 – #8 – so that each block has its electron beam starting point at the top left corner of the block, as in the second part

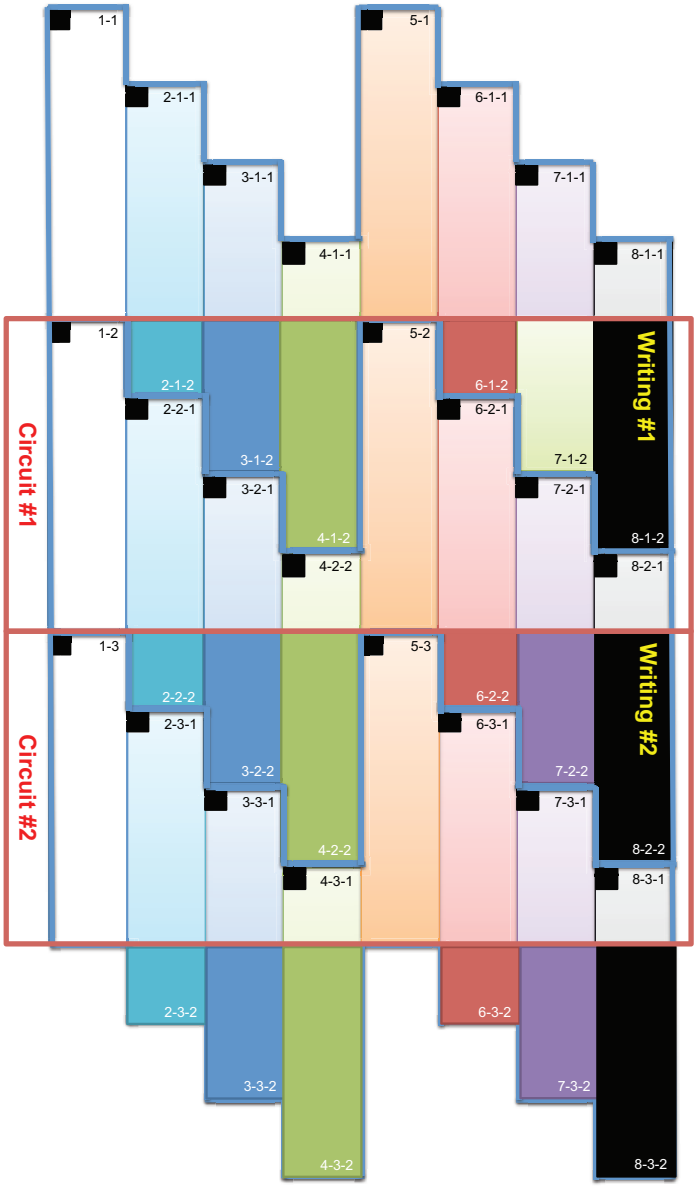

Figure 6.3: The application of the MAPPER writing region to the wafer

of Figure 6.4. Note that the second part of Figure 6.4 corresponds to a "Writing" of Figure 6.3. Throughout the discussion of the processes in Figure 6.1, when we refer to image blocks we are referring to the blocks in the second part of Figure 6.4. The reason that we concentrate on a "Writing" instead of a "Circuit" in the lower part of Figure 6.4 is to emphasize the flow of decompressed data to the electron beams.

The frequent pattern discovery algorithm which is described in Section 6.1.2.1 searches each block to discover the frequent patterns used for all blocks. Then each

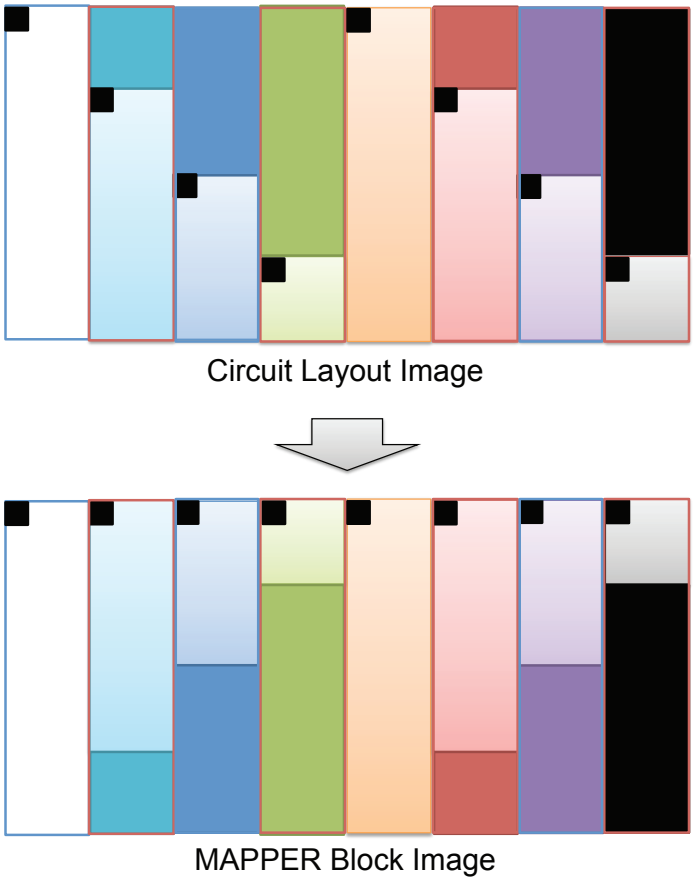

Figure 6.4: The effects of block separation

block is a separate input to the forward transformation process. Since it is important to understand the FPR process in order to illustrate the frequent pattern discovery process, in the next subsection we begin by explaining the forward transformation process.

6.1.2 Forward Transformation

Once the blocks are separated and the frequent patterns are discovered, each block separately goes through the forward transformation process. During the forward transformation process, we first search the image block looking for embeddings of predetermined frequent patterns. Whenever there exists an embedding of a frequent pattern, we replace the embedding with a simple representation. After all of the

frequent pattern embeddings are replaced with simple representations, we apply the corner transformation to the remaining image where no frequent pattern embeddings were found. Throughout this process, we handle the regular circuit parts with FPR and the irregular circuit parts with corner transformation.

In the following subsections, we will explain in detail the FPR and corner transformation processes.

6.1.2.1 Frequent Pattern Replacement

Figure 6.5 offers an overview of FPR. The inputs to the FPR encoder are an image block and the frequent pattern list, and the FPR encoder outputs a matched pattern image and a binary residue image. The FPR process is applied to each image block with the frequent pattern list fixed for all image blocks. The FPR encoder seeks the patterns within the block. Whenever a pattern is matched within the image block, the encoder will replace the *first point* of the pattern embedding with a pattern symbol and will replace the rest of filled pixels that have been matched with "0"s (or empty). Note that because of the zig-zag writing the first point of the pattern embedding could be either the top-left corner or the top-right corner of the pattern depending respectively on whether the first row of the pattern is odd or even. For the example in Figure 6.5, the first 3×3 square pattern begins at the first row of the image. Since the first row is written from left to right, the top left corner of the pattern is replaced with the pattern symbol (gray pixel). Similarly, the second 3×3 square pattern is found at the second row of the image and since the second row is written from right to left, the top right corner of the pattern is replaced with the pattern symbol.

Note the output matched pattern can have symbols from the set $\{0, S_1, S_2, \cdots, S_P\}$, where S_i is the symbol used to represent frequent pattern i and P is the size of the frequent pattern list. Finally note that when the FPR encoder seeks a pattern, it

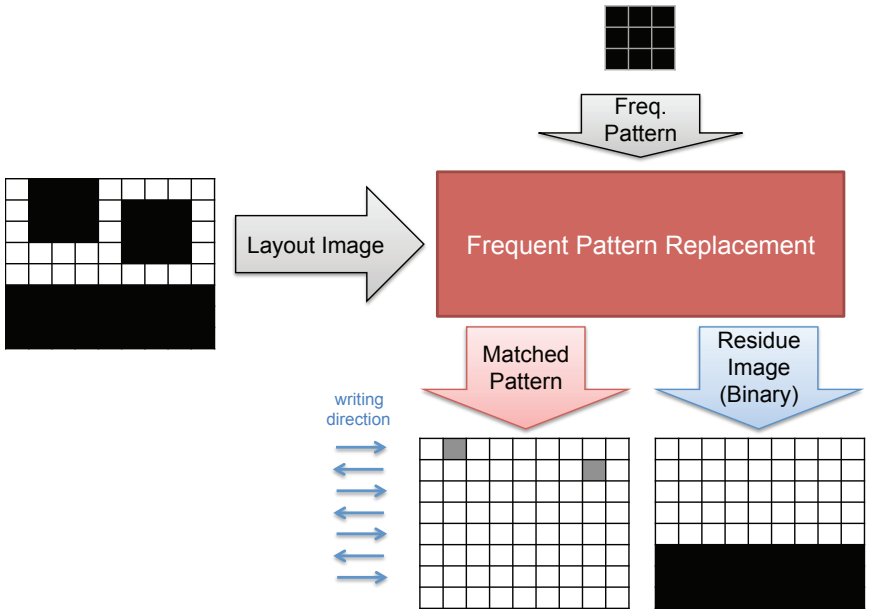

Figure 6.5: Frequent Pattern Replacement

more precisely searches for the pattern surrounded by empty rows and columns so that the pattern embedding is isolated, i.e., not connected to other polygons. This is necessary to avoid interference with corner transformation and to prevent partial pattern matching which could result in performance deterioration. Otherwise, the 3×9 rectangle at the bottom of Figure 6.5 can be represented by three 3×3 squares, which is not desirable.

Once the FPR process is applied, the residue image block is passed to the corner transformation process.

6.1.2.2 Corner Transformation

The input to the corner transformation process is the residue image block from the FPR encoder, and the output is the corner transformed image block. As we indicated earlier, like the FPR encoder the corner transformation process is only processing a block of the layout image at a time.

Figure 6.6 illustrates the corner transformation process. First, we expand the input image block by introducing two empty columns (shown as gray grids in the

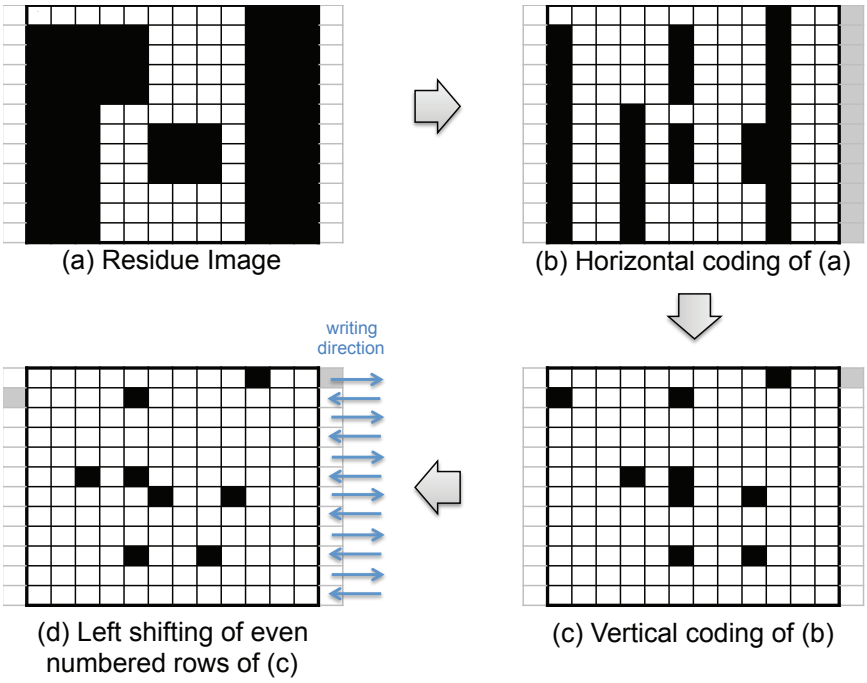

Figure 6.6: Corner transformation process of `Corner2-MEB`

figure) to surround the input image block in order to handle the zig-zag writing order. Second, we apply horizontal and vertical bi-transitional encoding on the expanded image block. This bi-transitional encoding marks the pixels (black) where the current pixel value is different from the previous pixel read in the encoding direction; i.e., left to right for horizontal encoding and top to bottom for vertical encoding. Next, we *left-shift* the even-numbered rows; i.e., the rows which are written from right to left. During this left-shift, the pixels from the surrounding right column could get inside the image area. Finally, we discard the pixels in the expanded columns. By applying this corner transformation, we represent the residue image block using its *transitional corners*. We call these points transitional corners because they are similar to polygon corners, but they are extracted by transitional coding. Similar to the FPR process, the corner transformation is separately applied to each image block.

The algorithm is summarized in Algorithm 5. In the algorithm, $x \in [1, \cdots, C]$ is the column index of the image block, $y \in [1, \cdots, R]$ is the row index of the image

block, and we will assume all odd rows are written from left to right and all even rows are written from right to left. Note that $R \times C$ is the dimension of the block and is predefined by the MEB DW system.

Algorithm 5 Corner Transformation Algorithm

Input: Layer image $\texttt{IN} \in \{0,1\}^{R \cdot C}$
Output: Corner image $\texttt{OUT} \in \{0,1\}^{R \cdot C}$

```
 1: Initialize OUT(x, y) = 0, ∀x, y.
 2: for y = 1 to R do
 3:   for x = 1 to C do
 4:     if y is odd then
 5:       if IN(x − 1, y − 1) = IN(x, y − 1) and IN(x − 1, y) ≠ IN(x, y) then
 6:         OUT(x, y) = 1
 7:       end if
 8:       if IN(x − 1, y − 1) ≠ IN(x, y − 1) and IN(x − 1, y) = IN(x, y) then
 9:         OUT(x, y) = 1
10:       end if
11:     else
12:       if IN(x − 1, y − 1) = IN(x, y − 1) and IN(x − 1, y) ≠ IN(x, y) then
13:         OUT(x − 1, y) = 1
14:       end if
15:       if IN(x − 1, y − 1) ≠ IN(x, y − 1) and IN(x − 1, y) = IN(x, y) then
16:         OUT(x − 1, y) = 1
17:       end if
18:     end if
19:   end for
20: end for
```

In Section 3.1.3, we introduced a one-step corner transformation algorithm where pixel (x, y) is processed as a function of the input pixels $(x-1, y)$, $(x, y-1)$, and $(x-1, y-1)$. We modified that algorithm so that it applies a left-shift for the even-numbered rows in Lines 12–16 of Algorithm 5.

Observe that the transitional corners can only appear at a polygon corner, right of a polygon corner, left of a polygon corner, below the polygon corner, to the bottom-right of a polygon corner, and to the bottom-left of a polygon corner. By matching isolated patterns during FPR, we guarantee that the transitional corners produced by Algorithm 5 do not overlap with any of the pattern symbols in the matched pattern

image block. We sum the pattern matched image block and the corner transformed image block to form the *forward transformed image block*. By choosing the FPR symbols (S_i) to not overlap with corner symbols $\{0, 1\}$, we can always separate the matched pattern image block and the corner transformed image block from the forward transformed image block.

Finally, after all blocks have been forward transformed, they are input to the Flatten Pixel Stream process which will be explained in Section 6.1.4 to produce a one-dimensional (encoded) pixel stream.

6.1.3 Frequent Pattern Discovery

We next discuss the generation of the frequent pattern list. This process contains three sub-processes: 1) pattern extraction from the GDSII layout description, 2) candidate pattern generation, and 3) selection of the optimized pattern list. In the following subsections we will offer a detailed description of these procedures.

6.1.3.1 GDSII Pattern Extraction

In Section 3.1.2, we extracted patterns that are frequently used in the entire layout image by seeking the frequently used substructures in the original GDSII layout descriptions since the layout image is rasterized from the GDSII representation. This procedure, which can be effective, is illustrated in Figure 6.7.

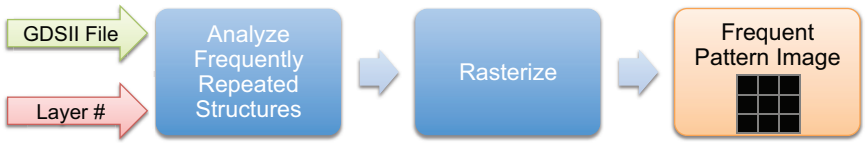

Figure 6.7: Frequent Pattern Discovery from GDSII Layout Description

However, as illustrated in Figure 6.8, this method has shortfalls because some substructures could result in different image patterns depending on the rasterization grid. In Figure 6.8, two 5 $nm\times$ 5 nm squares are defined in the layout description

(GDSII). When we rasterize the image in a 4 nm grid (Figure 6.8. right), we are actually quantizing a 4×4 block from the 1 nm grid as a single pixel (Figure 6.8. left) and we fill the pixel if the number of filled pixels in the original block is at least 8. The rasterized image shown in Figure 6.8. right has two different polygons, but they came from the same layout description.

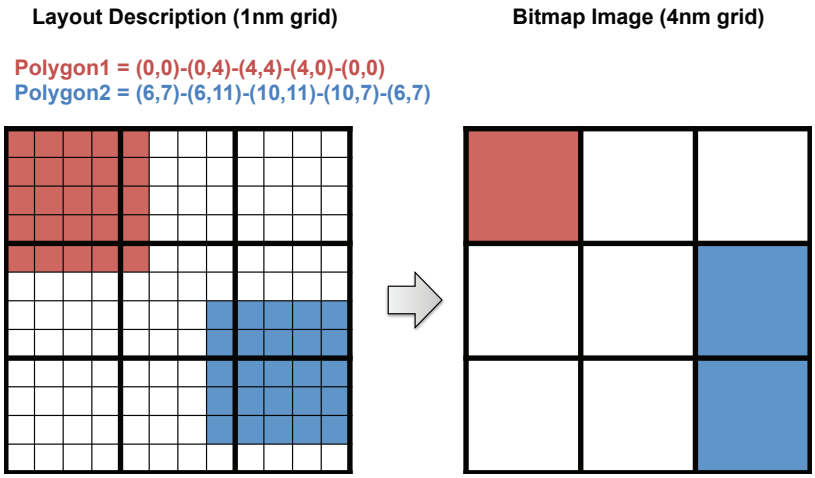

Figure 6.8: Example of pattern mismatch due to rasterization

Furthermore, since we have partitioned the image into blocks, the patterns extracted from the GDSII representation may not have matches. We therefore need to search for frequent patterns within the block images.

6.1.3.2 Candidate Pattern Generation

In order to extract patterns that are frequently used for entire image blocks, we search the blocks. We then generate a list of candidate patterns and later determine which among these should be included in the frequent pattern list. In this section, we discuss the candidate pattern discovery algorithm which is mainly described in Section 5.2. As we explained in Section 6.1.2.1, during the FPR process we match isolated patterns; i.e., we first surround the pattern with empty rows on the top and bottom and empty columns to the left and right. We require that the frequent patterns satisfy the following conditions:

1. the patterns should be defined in a rectangular region,

2. the patterns should never overlap with other patterns, and

3. the patterns should be isolated.

First, we will explain how the candidate patterns are generated. Later we determine which ones will be included in the frequent pattern dictionary. The candidate pattern generating algorithm is shown in Algorithm 6. In the algorithm, $x \in [1, \cdots, C]$ is the column index of the image block and $y \in [1, \cdots, R]$ is the row index of the image block. Once again note that $R \times C$ is the block dimension which is predetermined by the MEB DW system.

Algorithm 6 Candidate Pattern Generation

Input: Layout image `IN` $\in \{0,1\}^{R \cdot C}$
Output: List of patterns `PatternList`
Intermediate: List of patterns `Checked` $\in \{0,1\}^{R \cdot C}$

```
Initialize Checked(x, y) = 0, ∀x, y.
for y = 1 to R do
  for x = 1 to C do
    if Checked(x, y) = 0 and IN(x, y) = 1 then
      (x0, y0, x1, y1) = DefinePatternRegion(x, y)
      Pattern = MakePattern(x0, y0, x1, y1)
      p = PatternList.Find(Pattern)
      if p ≠ NOT_FOUND then
        PatternList[p].frequency += 1
      else
        PatternList.Insert(Pattern, 1)
      end if
      for yy = y0 to y1 do
        for xx = x0 to x1 do
          Checked(xx, yy) = 1
        end for
      end for
    end if
  end for
end for
```

The algorithm starts by picking a pixel from the image block in raster order, i.e., from top to bottom and then from left to right. If the pixel is filled (1), then

we define a rectangular region $(x_0, y_0) - (x_1, y_1)$ so that all of the filled pixels that are connected to (x, y) are covered by it (Line 5). We then make the rectangular region $(x_0, y_0) - (x_1, y_1)$ as the candidate pattern `Pattern` (Line 6) and search the list of candidate patterns `PatternList` (Line 7) to see whether or not the pattern was already in the list. If `Pattern` was already in the `PatternList`, we increase its frequency by 1 (Line 9). Otherwise, we put `Pattern` into the `PatternList` and initialize its frequency to 1 (Line 11). Finally, we make sure that the region is not searched again by marking the region in `Checked` (Lines 13–17), and this prevents the patterns from overlapping.

Once again, note that this frequent pattern discovery algorithm is applied separately to each image block. By augmenting the discovered `PatternList` after processing each block image, we obtain the final candidate pattern list. Moreover, to incorporate the GDSII extracted patterns in Section 6.1.3.1 with the Algorithm 6 patterns, we first run the FPR algorithm of Section 6.1.2.1 on each image block using only the GDSII extracted patterns. We next apply Algorithm 6 to the residue image blocks which are the block image regions that have not been matched by the FPR process. By combining both the GDSII extracted patterns and the patterns generated using Algorithm 6, we obtain the final candidate pattern list.

6.1.3.3 Pattern Optimization

Once the candidate patterns are discovered, we analyze them in order to decide which patterns to keep and which patterns to discard. The final patterns will be used as the frequent patterns of the FPR encoder for every image block. As discussed in Section 5.3, there are the two parameters that we consider to make this decision. The first parameter is *gain*, which provides information on what improvement we should expect by using the pattern for the FPR process. Since compression is related to the corners we remove by patterns as well as to the frequency of patterns, we define the

gain of pattern p as :

$$Gain_p = [N(C_p) - 1] \times N(F_p),$$

where $N(C_p)$ is the number of corners of pattern p and $N(F_p)$ is the frequency of pattern p.

The second parameter is *cost*, which shows how much decoder memory is required to keep the pattern in the decoder memory. For the pattern p whose dimension is $w_p \times h_p$, the decoder usually needs $w_p \times h_p$ bits of memory to store the pattern. However, we can reduce the cost when the pattern is fully filled. For this case, since we already know that all pixels are filled all we need to store are the pattern dimensions. We allocate 16 bits for each dimension and a 1 bit flag to specify whether or not the pattern is fully filled. Therefore the cost of pattern p with dimension $w_p \times h_p$ is defined as follows:

$$Cost_p = \begin{cases} 33, & \text{if pattern } p \text{ is fully filled} \\ 33 + w_p \times h_p, & \text{otherwise} \end{cases}$$

To reduce the complexity of the optimization problem we initially discard the candidate patterns whose gains were less than a preset threshold $Threshold$.

To choose the frequent patterns, we want to

$$\text{maximize } \sum_p Gain_p \cdot x_p \text{ such that } \sum_p Cost_p \cdot x_p \leq P_{size}, \tag{6.1}$$

where x_p is a binary number indicating whether pattern p should be used as a frequent pattern (1) or not (0), and P_{size} is the decoder memory in bits that can be used to store the frequent pattern dictionary.

Optimization (6.1) is an instance of a standard Binary Integer Programming (BIP)

problem [46]

$$\begin{aligned} &\text{maximize} && \mathbf{c}^T\mathbf{x} \\ &\text{subject to} && \mathbf{a}^T\mathbf{x} \leq b \\ &\text{and} && \mathbf{x} \in \{0,1\} \end{aligned} \tag{6.2}$$

by setting $\mathbf{c} = [Gain_1\ Gain_2\ \cdots\ Gain_G]$, $b = P_{size}$, and $\mathbf{a} = [Cost_1\ Cost_2\ \cdots\ Cost_G]$.

Here G is the number of candidate patterns. By applying a widely used BIP solver [47], we are able to choose the optimal frequent patterns from the generated candidate pattern list.

Finally, note that this optimal frequent pattern set is used during the FPR process of every image block. If we instead use a different frequent pattern set for each image block, we can only assign a fraction of P_{size} for the FPR of each image block in order to sustain the same decoder memory requirement. That results in discarding some of the large patterns that are widely used in multiple image blocks, and hence, in deteriorating the compression performance.

6.1.4 Flatten Pixel Stream

Once the forward transformed image blocks are generated, we "flatten" the forward transformed image blocks into a one-dimensional pixel stream so that the MAPPER system receives an encoded pixel for each block at a time to match the current placement of all of the electron beams. We obtain this stream by gathering the transformed image blocks and reading one pixel from each block in a horizontal zig-zag order.

An example of this permutation is shown in Figure 6.9 where each block has dimension 5×4 and each block is written in zig-zag order. For the example, we first read the first (or top-left) pixels of each block in the order of the blocks. Then, we read the second pixels in zig-zag order of each block and continue that until the last pixels in zig-zag order of each block are read. The order in which the pixels are read

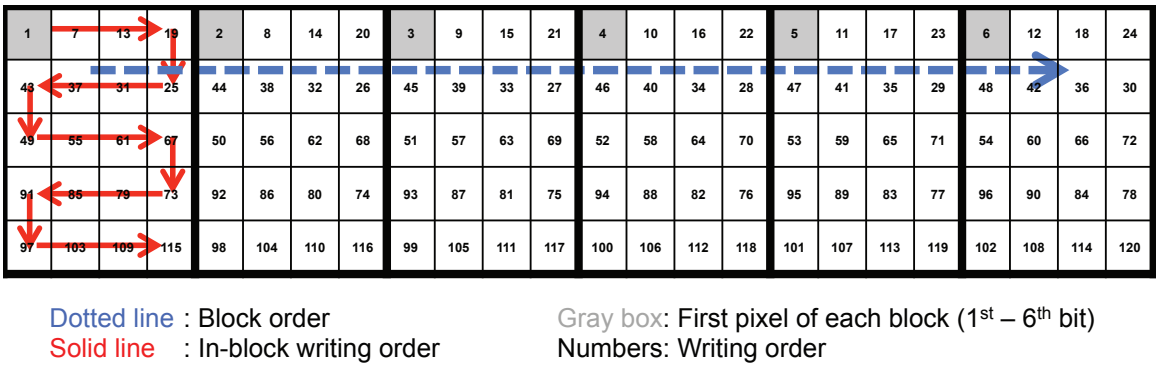

Figure 6.9: Permute pixels corresponding to the writing strategy

in this example is marked in the image.

Finally note that this permutation operation inputs the forward transformed image blocks and outputs a long one-dimensional stream to which the following entropy coding schemes will be applied.

6.1.5 Entropy Encoding

We use the final entropy coding scheme of Section 3.1.5 to compress the final flattened stream of pixels. Since the forward transformed images are very sparse, the flattened stream contains long run of zeroes making RLE and EOB coding efficient to compress it. Moreover, we also find long run of EOBs as the forward transformed images are very sparse. We use an N-ary representation to encode runs of EOBs and an M-ary representation to encode run of zeroes as in Section 3.1.5. Observe that the output of RLE+EOB coding can have up to $P + 1 + M + N$ symbols, where P is the number of frequent patterns after the pattern optimization process in Section 6.1.3.3 and one symbol is needed for representing the transitional corners shown in Section 6.1.2.2.

This $P+1+M+N$ symbol string is compressed using arithmetic coding [33, 35] for further compression. We followed the implementation of arithmetic coding provided by Witten et al. [35]; for the implementation the decoder requires four bytes per

alphabet symbol, and since we used $P + 1 + M + N$ symbols, $32(P + 1 + M + N)$ bits were required for arithmetic decoding.

6.2 The Decompression Algorithm

The `Corner2-MEB` decoder decompresses the compressed bit stream to write the circuit layout image using the electron beam arrays. Because of the memory constraints of the decoder, the same compressed bit stream is repeatedly retransmitted to the decoder during each "Writing" of an electron beam.

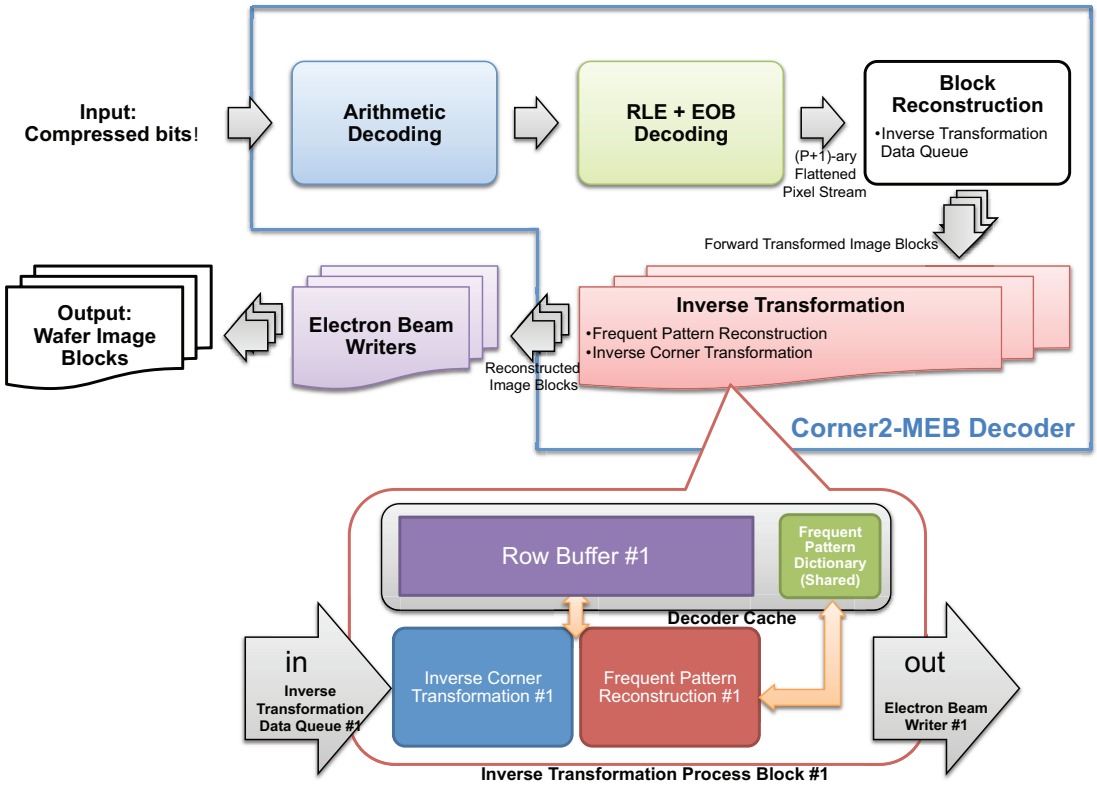

Figure 6.10: `Corner-MEB` Decompression Algorithm Overview

The `Corner2-MEB` decompression process is illustrated in Figure 6.10 and consists of four major blocks: arithmetic decoding, run length and end-of-block decoding, block reconstruction, and an inverse transformation process block for each electron beam which is directly connected to it. The entire decoder is fabricated on the same

silicon as the electron beam writer (controller) array.

The first two steps, arithmetic decoding and run length and end-of-block decoding, reverse the corresponding encoding procedures and output the $(P+1)$-ary flattened pixel stream of Section 6.1.4. In the block reconstruction process the input pixel stream is separated into multiple pixel streams which each correspond to a forward transformed image block. Each such block is converted back into a reconstructed image block which is passed to an electron beam writer that writes the decoded block onto the wafer as the stage moves. The different image blocks are processed and written in parallel on the wafer by independent inverse transformation processes and the corresponding electron beam writers.

The decoder requires $\lceil \log_2(3) \times width \rceil$ bits of cache for each inverse transformation process block to keep intermediate information for the row-by-row decoding of each block, P_{size} bits of memory for the frequent pattern dictionary, and $32(P+1+M+N)$ bits for arithmetic decoding, where $width$ is the width of the image block, P is the number of total frequent patterns, and M/N are RLE/EOB parameters.

6.2.1 Block Reconstruction

Since arithmetic and RLE+EOB decoding are well-understood in data compression, we will next discuss the reconstruction of the forward transformed image blocks of Section 6.1.2 from the flattened pixel stream of Section 6.1.4. As illustrated in Figure 6.11, we can achieve this by wiring each symbol of the flattened pixel stream to the corresponding inverse transformation data queue.

In the example shown in Figure 6.11, we are assuming there are six electron beam writers and six inverse transformation processes working simultaneously for the system. The information stored in the inverse transformation data queue is passed onto the inverse transformation process where each row of a forward transformed image block is reconstructed and decoded as a row of the reconstructed image block.

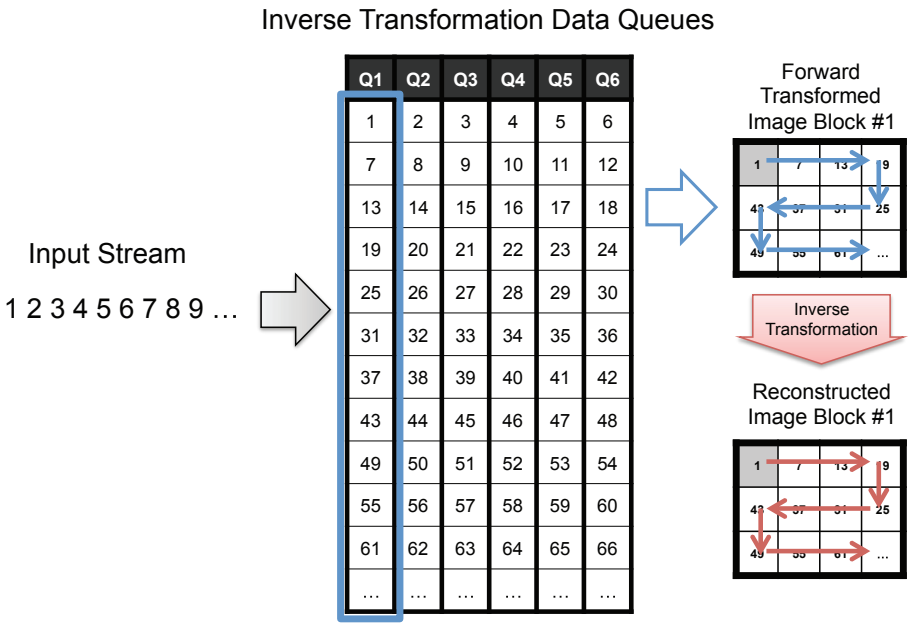

Figure 6.11: Reconstructing the forward transformed image blocks from the flattened stream

Since each inverse transformation process block is connected to the corresponding electron beam writer, each reconstructed row is passed to the electron beam writer to be written on the wafer.

6.2.2 Inverse Transformation

Inverse transformation is applied to each forward transformed image block to reconstruct the original image block. The reconstructed image block is forwarded to the associated electron beam writer. The inverse transformation performs Frequent Pattern Reconstruction and Inverse Corner Transformation. Recall that the frequent patterns are encoded using symbols $S_1, \cdots, S_P$ while the transitional corners are marked by the symbol 1.

The bottom half of Figure 6.10 shows the detailed hardware architecture of an inverse transformation block. As illustrated in the figure, both the inverse corner transformation and the frequent pattern reconstruction processes require a row buffer

to keep track of the previous row's status. The frequent pattern reconstruction process also requires a frequent pattern dictionary. Note that this frequent pattern dictionary is shared among all of the inverse transformation blocks.

Observe that since the electron beams write a pixel of each image block at the same time, synchronization is needed among the inverse transformation processes. Here we have the inverse transformation processes produce a row of each reconstructed image block in parallel.

We next explain the operation of the inverse corner transformation and the frequent pattern reconstruction processes for an image block.

6.2.2.1 Inverse Corner Transformation

The inverse corner transformation uses pixels from the previous row and column to decode the current pixel. We designed the decoder to decode the input corner transformed image block in a row-by-row manner instead of in its entirety in order for this process to be compatible with the restricted memory available to the hardware decoder. The inverse corner transformation process is as follows: First, the decoder reads the input corner transformed image block in a zig-zag order. The zig-zag order is the same as raster order except it reads the even rows from right to left. Second, the decoder processes the current pixel by checking the status of the row buffer (`BUFF`). The row buffer is used to store the status of the previous (decoded) row. It uses two symbols, 0 and 1, to represent its status, and hence, the buffer requires *width* bits of memory. "0" means 'no transition' while "1" means 'transition' and indicates the starting/ending point of a vertical line. Third, whenever the read symbol is a transitional corner point ("1"), the decoder starts reconstructing a horizontal line by setting the horizontal line fill flag (`Fill`) until it reads another transitional corner point from the input row and resets the horizontal line fill flag. For the even rows, this line is written from right to left, while it is written from left to right for the odd

rows. Fourth, for every new horizontal line created by the transitional corners, the row buffer (`BUFF`) is updated so that the decoder can take the status of the current row into account while reconstructing the next row.

Note that because we are dealing with "transitional corners", a horizontal line starts from a transitional corner point and ends one pixel before the pairing transitional corner point. In Figure 6.6, step (d) is applied to sustain the same decoding rule for the even rows. If the encoder did not left shift the even rows, then in order to reconstruct the horizontal lines of those rows the decoder has to start a horizontal line one pixel after a transitional corner point and end it at the pairing transitional corner point. However, there could be a problem reconstructing the horizontal line of the even rows when the first pixel of the row (i.e., the rightmost pixel) has to be filled. Since there is no pixel to the right of the rightmost pixel, the reverse line from right to left can not be reconstructed using this decoding rule. By inserting an empty right column and allowing the transitional corners to appear there and applying left shifts to the even rows as in step (d) of Figure 6.6, we can always reconstruct horizontal lines by starting from a transitional corner point and ending it one pixel before the corresponding transitional corner point following the row direction.

Because the inverse corner transformation rule is independent of the row direction, we have to make sure that the column index matches the row direction. Algorithm 7 describes the inverse corner transformation process. We assume the inverse corner transformation decoder can randomly access the entire row buffer and the input corner transformed image block is read in a zig-zag order. Note that the $\bigoplus$ operation is a binary `XOR` operation, and is only applied to binary summands.

In Algorithm 7, the horizontal fill flag (`Fill`) is initialized for every row (Line 4). Then we check the row number and update the column index (Lines 6–10). If the row is odd numbered, then the column index starts from 1 to C (Lines 6–7), while the order is reversed (from C to 1) if the row number is even. We update the column index

Algorithm 7 Inverse Corner Transformation

Input: Corner transformed image block $\texttt{IN} \in \{0,1\}^{R \cdot C}$
Output: Reconstructed image block $\texttt{OUT} \in \{0,1\}^{R \cdot C}$
Intermediate: Row Buffer $\texttt{BUFF} \in \{0,1\}^{R}$

```
 1: Initialize BUFF(x) = 0, ∀x.
 2: Initialize OUT(x, y) = 0, ∀x, y.
 3: for y = 1 to R do
 4:    Fill = 0
 5:    for x = 1 to C do
 6:       if y is odd then
 7:          x' = x
 8:       else
 9:          x' = C + 1 − x
10:       end if
11:       if BUFF(x') = 1 then
12:          OUT(x', y) = 1
13:       end if
14:       if IN(x', y) = 1 then
15:          Fill = Fill ⊕ 1
16:       end if
17:       OUT(x', y) = OUT(x', y) ⊕ Fill.
18:       BUFF(x') = BUFF(x') ⊕ Fill.
19:    end for
20: end for
```

if the row number is even (Lines 8–10). Lines 11–13 process the buffer. If the buffer is filled, i.e., if there is a vertical fill, then the corresponding pixel is filled. If the input pixel (read in zig-zag order) is a transitional corner ("1"), the decoder changes the status of the horizontal fill flag (Lines 14–16). Finally, depending on the horizontal fill flag (`Fill`), the output pixel (Line 17) and the buffer (Line 18) are updated if necessary. If the input pixel is "0", the decoder makes no horizontal/vertical changes to the image, but it fills the output pixels and updates the buffer according to the fill status.

6.2.2.2 Pattern Reconstruction

If the decoder finds symbols $\{S_1, \cdots, S_P\}$ within the forward transformed image block, it starts the pattern reconstruction process. First, it reconstructs the first

row of the corresponding pattern according to its writing order. If the decoder is processing an odd row, it will reconstruct the pattern row from left to right and otherwise it will reconstruct the pattern row from right to left. Second, it updates the corresponding buffer with the following string:

[$ Pattern# Row#]

Here $ is a special symbol used to indicate the start of FPR. If pattern p of dimension $w_p \times h_p$ is to be reconstructed, then Pattern# is a $\lceil \log_2(P) \rceil$-bit binary representation of p and Row# is a $\lceil \log_2(h_p - 1) \rceil$-bit binary representation of one less than the remaining number of pattern p rows the decoder has to reconstruct. Because Pattern# determines which frequent pattern the decoder needs to reconstruct (and its dimension), and Row# determines which row of the corresponding pattern it has to reconstruct, the decoder can reconstruct the pattern. We could alternatively use base-3 logarithms, but that would complicate the decoding hardware.

Note that this [$ Pattern# Row#] stream is updated depending on the writing direction of the next row. An example of the frequent pattern reconstruction process is demonstrated in Figure 6.12. When the first row is decoded, we update the row buffer in the reverse direction starting at the 4th column because the next row is written from right to left and the pattern involves the second, third, and fourth columns. The [$ Pattern# Row#] stream is represented as [$ 1][1] in the reverse direction because there is only one frequent pattern defined in the dictionary making $\lceil \log_2(P) \rceil = 0$. In the next row, the decoder has to process two more rows of the pattern. When the second row is decoded, since the writing order of the next row is from left to right and since we need to write one more row to the first pattern, the buffer of the first pattern is updated to [$ 0] in the forward direction. Similarly, the buffer of the second pattern is updated to [$ 1]. Finally, when the third row is decoded, we re-initialize

[1] In Figure 6.12, the $'s are represented by red pixels and the 1's are represented by black pixels.

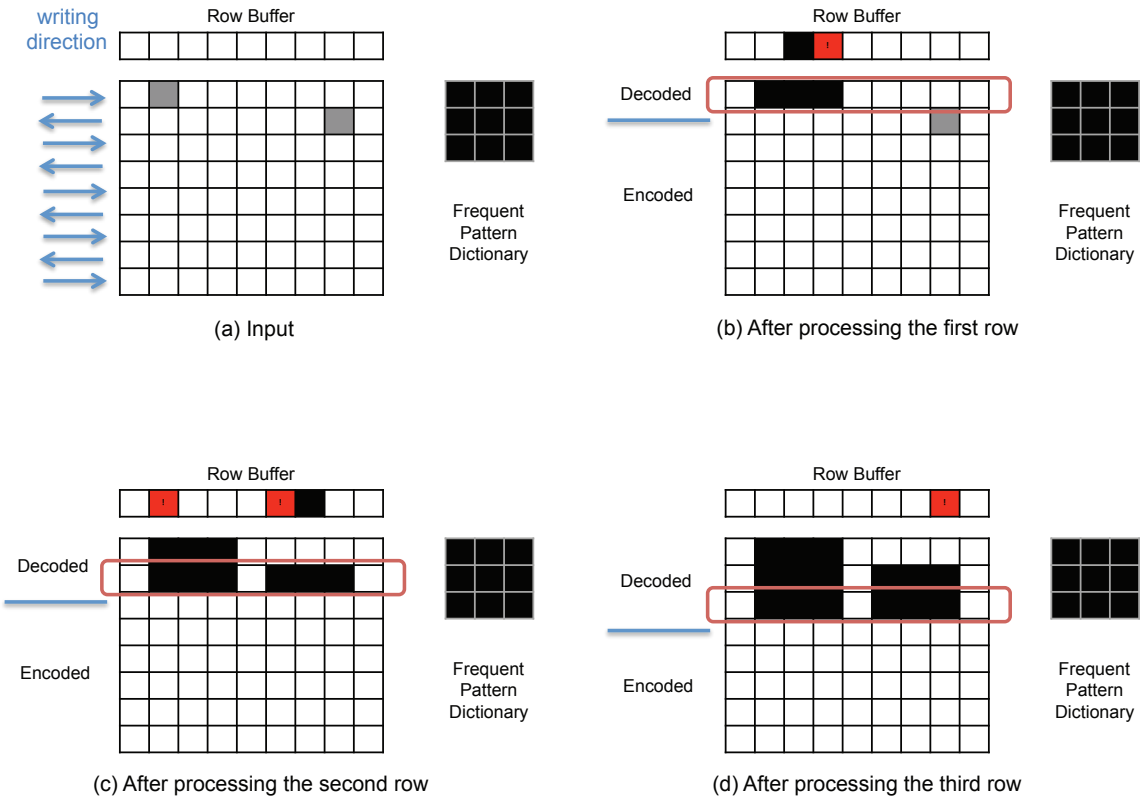

Figure 6.12: Frequent Pattern Reconstruction Example

the buffer of first pattern to [0 0] to indicate that the frequent pattern reconstruction is complete.

In order to combine this frequent pattern reconstruction and the inverse corner transformation, the inverse transformation decoder requires $\lceil \log_2(3) \times C \rceil$ bits for the row buffer for each inverse transformation block and P_{size} bits to store the pattern dictionary. Given that the forward transformed image is a $(P+2)$-ary array with dimension $R \times C$, this buffer requirement is comparatively small.

6.3 Experimental Results

We tested the algorithm on two benchmark circuits introduced in Section 2.4. Most parts of `Corner2-MEB` were written in C/C++ with OpenMP support for processing the forward and inverse transformation processes in parallel. For the decod-

ing process, the inverse transformation processes are synchronized so that the same "Writing" row of each was processed in parallel. Because `Corner2-MEB` blocks can be processed independently, we applied parallel processing for faster encoding / decoding speed. The binary integer programming part was written in MATLAB using the MATLAB function `bintprog` [47]. All of the experiments ran on a laptop computer having a 2.53 GHz Intel Core 2 Duo CPU and 4 GB RAM.

We applied `Corner2-MEB` (C2-MEB) [48] and compared the results with `Corner2-BIP` (C2-BIP) [45], `Corner2` (C2-ORG) [42], and `Block C4` (BC4) [27]. While `Corner2-BIP`, `Corner2`, and `Block C4` were applied to the entire layout image, `Corner2-MEB` was applied in a block fashion. We set the MAPPER block size to be $888 \times height$ because the MAPPER writing region is a width-2 μm stripe on a 2.25 nm grid [8]. Here, *height* is the height of the circuit layer image which was 17,816 for the memory circuit and 31,624 for the BFSK circuit.

In the following subsections we show the compression results for both the memory and BFSK circuits on the MAPPER system. In the following subsections, we defined the compression ratios to be

$$\frac{\texttt{Input File Size}}{\texttt{Compressed File Size}}.$$

Note that the total compression ratio is not the average of the preceding compression ratios, but the "net average" which is defined as

$$\frac{\texttt{Total Input File Size}}{\texttt{Total Compressed File Size}}.$$

To compare the result with `Corner2-BIP`, we used the same M, N, and P_{size} setting as in Section 5.4.

Layer	C2-MEB	C2-BIP	C2-ORG	BC4
1	11,074	22,201	18,658	147.78
2	457	1,289	1,216	79.80
3	11,074	22,201	18,658	147.78
4	83	151	86	54.12
5	640	920	920	133.22
6	640	920	920	133.22
7	104	84	84	31.58
8	41	90	67	24.40
9	89	171	170	38.17
10	96	114	57	25.31
11	204	442	442	75.70
12	2,837	11,059	3,169	121.69
13	5,971	28,606	9,616	141.02
Average	164	261	192	57.36

Table 6.1: Compression Ratio (x) - Memory Array (Block size : $888 \times 17,816$)

6.3.1 Memory Circuit

Tables 6.1–6.3 show the experimental results for the memory circuit when the MAPPER block separation was applied. Because of the block partitioning, C2-MEB had 37.3% and 14.9% smaller compression ratio than C2-BIP and C2-ORG respectively, which operated on the full layout image. However, C2-MEB was still 2.9 times better than BC4 as shown in Table 6.1. The block partitioning resulted in fewer pattern matches and introduced more corners in the block boundaries. Moreover, the flattening process also deteriorated the RLE+EOB encoding by segmenting long run of zeroes into smaller pieces. However, because `Corner2`-variations have better layout image modeling, C2-MEB still resulted in better compression performance than BC4.

On average, C2-MEB encoding was 3% (or 0.3%) faster than C2-BIP (or C2-ORG) while 50.8 times faster than BC4 as shown in Table 6.2. While C2-MEB generated more candidate patterns due to block truncation which slowed down the encoding process, it became faster than C2-BIP and C2-ORG because of parallel processing. However, C2-MEB encoding could be further accelerated by using more paralleliza-

Layer	C2-MEB	C2-BIP	C2-ORG	BC4
1	23.12	18.89	4.29	1,875
2	30.84	26.48	26.36	1,891
3	23.74	18.36	4.29	1,788
4	43.47	49.02	48.93	1,815
5	15.40	25.26	26.85	1,831
6	15.55	25.28	26.86	1,796
7	25.88	23.07	25.27	1,846
8	92.07	97.32	109.97	1,885
9	32.57	25.37	30.18	1,817
10	68.83	84.16	78.49	1,917
11	64.58	70.68	80.22	1,795
12	15.64	6.23	4.16	1,775
13	17.20	6.52	4.16	1,762
Total	468.87	485.45	470.05	23,793

Table 6.2: Encoding Time (s) - Memory Array

tion. Since the Intel Core 2 Duo CPU only supports running 4 threads simultaneously, we were not able to take advantage of processing all blocks in parallel. Distributed computing over the grid or an application of General Purpose GPU (GP-GPU) based parallel processing might enable this.

Layer	C2-MEB	C2-BIP	C2-ORG	BC4
1	2.81	2.40	2.53	55.44
2	3.57	3.16	3.34	54.99
3	2.81	2.42	2.53	52.90
4	3.55	2.96	3.20	53.70
5	3.22	2.95	3.05	53.55
6	3.24	2.90	3.05	54.48
7	3.52	3.21	3.23	54.63
8	3.87	3.21	3.51	54.79
9	3.44	2.79	3.05	54.83
10	3.58	2.98	3.44	54.07
11	3.40	2.78	3.04	56.43
12	3.46	2.95	2.49	51.50
13	3.58	2.94	2.48	51.88
Total	44.04	37.67	38.93	703.18

Table 6.3: Decoding Time (s) - Memory Array

On average, C2-MEB decoding was 17% (or 13%) slower than C2-BIP (or C2-ORG) while 16.0 times faster than BC4 as shown in Table 6.3. Our decoding time results do not include the time to reverse the flattening process because this procedure would be unnecessary in a hardware implementation. The process was only used to verify that the reconstructed image matches with the input image. Furthermore, similarly to the encoding time, we expect the decoding time to be further reduced when full parallelization is applied. Considering this is a compress-once-and-decode-multiple-times application, C2-MEB is more realistic than C2-BIP and C2-ORG in that it takes advantage of multiple electron beams, and it offers better compression than BC4.

6.3.2 BFSK Circuit

Layer	C2-MEB	C2-BIP	C2-ORG	BC4
1	12,126	20,874	9,226	151
2	6,275	7,432	6,760	150
3	1,016	1,596	1,186	137
4	2,942	3,745	3,270	147
5	925	1,125	1,033	59
6	476	609	580	128
7	297	399	390	111
8	122	173	167	81
9	361	474	452	122
10	179,212	460,192	195,365	153
11	139	206	200	86
12	848	1,082	1,006	138
13	142	209	203	89
14	943	1,170	1,093	139
15	167	236	230	92
16	1,070	1,368	1,296	141
17	5,635	8,019	6,656	150
18	19,687	26,467	19,579	152
19	12,005	20,773	9,173	151
Average	385	537	515	113

Table 6.4: Compression Ratio (x) - BFSK Circuit (Block size : $888 \times 31,624$)

Tables 6.4–6.6 show the experimental results for the BFSK circuit when the MAPPER block separation was applied. Like the memory circuit, C2-MEB had a 28.4% smaller compression ratio than C2-BIP and a 25.2% smaller compression ratio than C2-ORG. This is mainly due to the new corners introduced and the shortened runs of zeroes due to block separation. However, C2-MEB still offered a 3.4 times better compression ratio than BC4. On average, C2-MEB was 58% slower than C2-BIP and 79% slower than C2-ORG, while 109.0 times faster than BC4. Compared to the memory circuit result, the improvements in encoding speeds were because more candidate patterns had to be generated even though not so many of them were used. Similarly, C2-MEB was 18% slower than C2-BIP and 14% slower than C2-ORG, while 26.1 times faster than BC4. However, this was because we only could run 4 inverse transformation processes in parallel which could be further improved with the help of massively parallel computing.

Although the compression performance of `Corner2-MEB` was inferior to that of `Corner2-BIP` due to block processing, it is better suited to MEB DW systems because it incorporated the actual writing strategy and because the inverse transformation process can be parallelized for higher throughput. Moreover, the performance of `Corner2-MEB` was still far better than that of `Block C4`. Hence handling regular circuit parts using dictionary-based compression and irregular circuit parts using corner-based (or vertex-based) representation as in `Corner2` is far more effective than handling regular circuit parts using LZ-based compression and irregular circuit parts using prediction-based compression as is done in `Block C4`.

The decoder needs to be implemented in hardware, and we showed that the `Corner2` inverse transformation decoder along with the RLE+EOB decoder required only 2% of a `Xilinx Spartan 3E` FPGA board even when using 5 kbytes of decoder cache in Chapter IV. The `Corner2-MEB` decoder consists of an arithmetic decoder, RLE+EOB decoder, and 13,000 inverse transformation blocks with each inverse trans-

Layer	C2-MEB	C2-BIP	C2-ORG	BC4
1	6.29	1.74	2.41	451
2	38.87	16.33	17.88	3,791
3	45.60	15.47	18.31	3,800
4	35.70	15.11	17.21	3,760
5	41.80	26.62	28.07	12,468
6	40.45	26.28	26.43	4,485
7	63.57	54.74	37.09	4,476
8	35.40	15.36	17.91	4,588
9	59.26	58.11	35.56	4,393
10	38.57	15.93	20.73	4,866
11	37.33	17.56	17.90	4,414
12	53.19	42.59	33.20	4,428
13	34.73	15.00	17.88	4,651
14	53.20	50.16	34.38	4,413
15	37.22	17.29	17.80	4,364
16	55.64	53.06	38.54	4,381
17	32.80	14.30	15.74	3,756
18	27.90	13.21	14.26	3,049
19	5.90	1.73	2.42	454
Total	743.41	470.59	413.75	80,987

Table 6.5: Encoding Time (s) - BFSK Circuit

formation block smaller than those we introduced in Section 4.2. Each inverse transformation block consists of simple operations such as comparisons, branches, XORs, and binary arithmetic. Since a GPU core can handle these operations and more advanced operations, we argue that the dedicated inverse transformation core will be much smaller than a GPU core. The row buffer requirement for each inverse transformation block is $\lceil \log_2(3) \cdot B_{width} \rceil$ bits, where B_{width} is the block width. Therefore, the entire `Corner2-MEB` system supporting 13,000 electron beams will require less than 3 MB of cache. Since current GPUs have over 3,000 cores [50] and previous generation Intel CPUs have 8 MB of cache [49], the entire `Corner2-MEB` decoder can be built using current silicon technologies.

One major concern is that we do not have an efficient hardware implementation for the arithmetic decoder making the final arithmetic encoding less suitable for the

Layer	C2-MEB	C2-BIP	C2-ORG	BC4
1	1.09	0.80	0.84	29.50
2	9.35	6.77	8.22	242.07
3	9.20	6.63	8.24	237.42
4	9.23	6.57	8.22	237.35
5	9.76	7.29	8.35	280.67
6	9.66	7.26	8.33	274.15
7	11.37	12.08	9.74	282.03
8	10.91	9.71	9.68	289.53
9	11.34	10.91	9.67	275.56
10	10.47	7.61	9.41	293.04
11	10.80	10.10	9.64	281.57
12	11.05	10.89	9.57	278.34
13	10.79	10.32	9.62	290.37
14	11.11	10.46	9.57	276.35
15	10.81	9.33	9.60	277.88
16	11.26	9.34	9.59	275.98
17	9.08	6.49	8.21	236.50
18	7.24	5.23	6.47	190.71
19	1.05	0.79	0.84	27.68
Total	175.58	148.58	153.81	4,576.71

Table 6.6: Decoding Time (s) - BFSK Circuit

purpose. We can tackle this issue by using other standard compression techniques and trading off part of the compression ratio for an efficient hardware decoder; since `Corner2-MEB` has about 3 times better compression performance than `Block C4`, there is some room for this trade-off.

CHAPTER VII

Conclusion and Future Works

MEB lithography has many hurdles to overcome before it can establish itself as an alternative to conventional photolithography. In this thesis, we have examined one of its challenges, designing a high throughput data delivery system.

As the technology develops, we fabricate circuits with smaller features on the wafer. In order for the circuits to have smaller features using the MEB lithography systems, the electron beam spot size has to reduce requiring more pixels over the same wafer area. In order to write these pixels, a massive amount of data is required to control the electron beam writers; For example, Dai [3] suggested 735 Terapixels are required for a 300 mm wafer using 45 nm technology. Since this control data can only be stored at a storage disk with limited throughput, we need a novel approach for delivering such data.

The data delivery issue can be solved by adopting the data delivery architecture Dai [3] proposed which was demonstrated in Figure 1.9.(e); storing the compressed data on a processing board, transmitting the compressed data to the writer circuits, and decoding the compressed data on-the-fly at the hardware decoder add-on to the writer circuits. In order for this data delivery architecture to work, there are two requirements that the compression algorithm should satisfy. First, the compression algorithm should always obtain high compression ratio. In fact, the compression ratio

should always be no less than

$$\frac{\text{Transfer rate of Decoder to Writer}}{\text{Transfer rate of Processor Board to Decoder}}$$

for all layer images. Second, the compression algorithm should have a simple decompression process requiring small cache so that the decoder can be implemented in hardware with small area and high throughput.

In this thesis, we have introduced a family of `Corner2` compression algorithms that attains significantly better compression than `Block C4` while requiring low decoding complexity. The `Corner2` compression is based on the observation that circuit layout images have highly repetitive parts where a block of image is repeated over a certain area and irregular parts that consists of Manhattan polygons. In order to deal with the repetitive parts, we introduced a dictionary-based compression technique which is widely used for text compression while we designed a corner transformation to represent the irregular circuit parts using their polygon corners. By applying these two schemes, we obtained sparse images which can be later compressed efficiently using a series of entropy coding techniques such as RLE, EOB, and arithmetic coding.

We introduced two ways to extract frequent patterns from the layer images. The first ways is to extract the frequent substructures from the layout description (such as GDSII) rather than the layer image itself. While this technique alone gave impressive performance, we improved it by analyzing the input layer image and generating candidate patterns and choosing the optimized pattern set satisfying the restricted memory requirement. For the candidate patterns, we used rectangular regions which cover all the polygons inside them and for optimization, we solved a binary integer programming (BIP) problem with the pattern cost, pattern gain, and the decoder memory restriction. While there is room for improvement in choosing the pattern gain function as the gain function we described in Section 5.3 is loosely related to the

compressed file size, this pattern detection algorithm could be used for compressing any binary images given that the patterns have similar restrictions.

In order to decode the `Corner2` compressed layout images, the decoder requires $\log_2(3) \cdot width$-bit cache for the row-by-row inverse corner transformation and frequent pattern reconstruction, and P_{size} bytes of memory to contain the frequent pattern dictionary. The decoder operation was standard and simple making it suitable for hardware implementation. In fact, our proof-of-concept `Corner2` decoder required only 2% of the `Xilinx Spartan-3E` FPGA board [41].

`Corner2` always outperformed `Block C4` in compression ratio, encoding time, decoding time, and decoder memory requirements showing that we have improved the way to model layout images compared to that of family of `C4` compression algorithms. However, there is still room for improvement: We need to test the algorithm on more advanced (sub-65 nm) circuit layout images as well as on real control data that is used for the corresponding MEB tool. Because most advanced circuit layout data and their MEB control data are proprietary, we were not able to access these. However, since our approach improved the basic model of circuit layout images, we can argue that our approach (maybe with modifications) will also provide better results for other data types.

Finally, we need to implement the `Corner2` decoder in an ASIC to really test its performance. In order to satisfy the throughput requirement, we need to make sure that the Corner2 decoder has enough throughput. In `Corner2` decoding, the arithmetic decoder has the highest complexity and the remainder has much lower complexity than that of the `Block C4` decoder. Considering the simplicity of the decoding algorithm (except the arithmetic decoder), we argue that the `Corner2` decoder will always have better performance than the `Block C4` decoder. In the case where the arithmetic decoding becomes too complex, we can always trade-off the encoder complexity and the compression ratio by using a less complex entropy encoder such

as Huffman coding [25]. Because our average compression ratio is 4.6–4.8 times better than that of `Block C4`, we have a wide trade-off range.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] B. J. Lin, *Optical Lithography: Here is Why.* Bellingham: SPIE Press, 2010. [Online]. Available: http://spie.org/x648.html?product_id=821000

[2] ——, "Future of multiple-e-beam direct-write systems," in *Proceedings of SPIE 8323*, 2012, p. 832302. [Online]. Available: http://dx.doi.org/10.1117/12.919747

[3] V. Dai, "Data compression for maskless lithography systems: Architecture, algorithms, and implementation," Ph.D. dissertation, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 2008. [Online]. Available: http://www-video.eecs.berkeley.edu/papers/vdai/phd-thesis.pdf

[4] N. Chokshi, Y. Shroff, and W. G. Oldham, "Maskless extreme ultraviolet lithography," *Journal of Vacuum Science Technology B*, vol. 17, no. 6, pp. 3047–3051, 1999. [Online]. Available: http://dx.doi.org/10.1116/1.590952

[5] L. Pan, Y. Park, Y. Xiong, E. Ulin-Avila, Y. Wang, L. Zeng, S. Xiong, J. Rho, C. Sun, D. B. Bogy, and X. Zhang, "Maskless plasmonic lithography at 22 nm resolution," *Scientific Reports*, vol. 1, no. 175, pp. 1–6, 2011. [Online]. Available: http://dx.doi.org/10.1038/srep00175

[6] C. Jahnert and S. Fritsche, "High quality mask storage in an advanced logic-waferfab," in *Proceedings of SPIE 8352*, 2012, p. 83520Q. [Online]. Available: http://dx.doi.org/10.1117/12.923053

[7] T. R. Groves, D. Pickard, B. Rafferty, N. Crosland, D. Adam, and G. Schubert, "Maskless electron beam lithography: prospects, progress, and challenges," *Microelectronic Engineering*, vol. 61-62, pp. 285–293, 2002. [Online]. Available: http://dx.doi.org/10.1016/S0167-9317(02)00528-2

[8] E. Slot, M. J. Wieland, G. de Boer, G. F. ten Berge, A. M. C. Houkes, R. Jager, T. van de Peut, J. J. M. Peijster, S. W. H. K. Steenbrink, T. F. Teepen, A. H. V. van Veen, B. J. Kampherbeek, and P. Kruit, "MAPPER: High throughput maskless lithography," in *Proceedings of SPIE 6912*, 2008, p. 69211P. [Online]. Available: http://dx.doi.org/10.1117/12.771965

[9] E. Platzgummer, C. Klein, and H. Loeschner, "eMET POC: Realization of a proof-of-concept 50 kev electron multibeam mask exposure tool,"

in *Proceedings of SPIE 8166*, 2011, p. 816622. [Online]. Available: http://dx.doi.org/10.1117/12.895523

[10] P. Petric, C. Bevis, A. Brodie, A. Carroll, A. Cheung, L. Grella, M. McCord, H. Percy, K. Standiford, and M. Zywno, "REBL nanowriter: Reflective Electron Beam Lithography," in *Proceedings of SPIE 7271*, 2009, p. 727107. [Online]. Available: http://dx.doi.org/10.1117/12.817319

[11] C. Mack, *Fundamental Principles of Optical Lithography: The Science of Microfabrication*. New York: Wiley, 2007. [Online]. Available: http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470018933.html

[12] A. N. Broers, A. C. F. Hoole, and J. M. Ryan, "Electron beam lithography - resolution limits," *Microelectronic Engineering*, vol. 32, pp. 131–142, 1996. [Online]. Available: http://dx.doi.org/10.1016/0167-9317(95)00368-1

[13] N. Silvis-Cividjian, C. W. Hagen, P. Kruit, M. A. J. v.d. Stam, and H. B. Groen, "Direct fabrication of nanowires in an electron microscope," *Applied Physics Letters*, vol. 82, no. 20, pp. 3514–3516, 2003. [Online]. Available: http://dx.doi.org/10.1063/1.1575506

[14] G. H. Jansen, "Coulomb interactions in particle beams," *Journal of Vacuum Science Technology B*, vol. 6, no. 6, pp. 1977–1983, 1988. [Online]. Available: http://dx.doi.org/10.1116/1.584148

[15] J. Ruan, "Electron beam lithography throughput and resolution enhancement with innovative blanker design," Ph.D. dissertation, University at Albany, State University of New York, 2010. [Online]. Available: http://gradworks.umi.com/3422427.pdf

[16] N. W. Parker, A. D. Brodie, and J. H. McCoy, "High throughput ngl electron beam direct-write lithography system," in *Proceedings of SPIE 3997*, 2000, pp. 713–720. [Online]. Available: http://dx.doi.org/10.1117/12.390042

[17] G. Cramer, H.-I. Liu, and A. Zakhor, "Lossless compression algorithm for REBL direct-write e-beam lithography system," in *Proceedings of SPIE 7637*, 2010, p. 76371L. [Online]. Available: http://dx.doi.org/10.1117/12.845506

[18] K. Chang, S. Pamarti, K. Kaviani, E. Alon, X. Shi, T. J. Chin, J. Shen, G. Yip, C. Madden, R. Schmitt, C. Yuan, F. Assaderaghi, and M. Horowitz, "Clocking and circuit design for a parallel I/O on a first-generation CELL processor," in *International Solid-State Circuit Conference 2005 (ISSCC 2005)*, 2005, pp. 526–527, 615. [Online]. Available: http://dx.doi.org/10.1109/ISSCC.2005.1494101

[19] S. M. Rubin, *Computer Aids for VLSI Design*, 2nd ed. Boston: Addison-Wesley, 1987, ch. Appendix C. [Online]. Available: http://www.rulabinsky.com/cavd/text/chapc.html

[20] Y. Chen, A. B. Kahng, G. Robins, A. Zelikovsky, and Y. Zheng, "Evaluation of the new OASIS format for layout fill compression," in *Proceedings of the 2004 11th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2004)*, 2004, pp. 377–382. [Online]. Available: http://dx.doi.org/10.1109/ICECS.2004.1399697

[21] A. J. Reich, K. H. Nakagawa, and R. E. Boone, "OASIS vs. GDSII stream format efficiency," in *Proceedings of the SPIE 5256*, 2003, pp. 163–173. [Online]. Available: http://dx.doi.org/10.1117/12.518271

[22] F. Yesilkoy, K. Choi, M. Dagenais, and M. Peckerar, "Implementation of e-beam proximity effect correction using linear programming techniques for the fabrication of asymmetric bow-tie antennas," *Solid-State Electronics*, vol. 54, no. 10, pp. 1211–1215, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.sse.2010.05.009

[23] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977. [Online]. Available: http://dx.doi.org/10.1109/TIT.1977.1055714

[24] V. Dai and A. Zakhor, "Lossless layout compression for maskless lithography systems," in *Proceedings of the SPIE 3997*, 2000, pp. 467–477. [Online]. Available: http://dx.doi.org/10.1117/12.390085

[25] D. A. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings of the I. R. E.*, 1952, pp. 1098–1102. [Online]. Available: http://dx.doi.org/10.1109/JRPROC.1952.273898

[26] L. Oktem and J. Astola, "Hierarchical enumerative coding of locally stationary binary data," *Electronics Letters*, vol. 35, no. 17, pp. 1428–1429, 1999. [Online]. Available: http://dx.doi.org/10.1049/el:19990969

[27] H.-I. Liu, V. Dai, A. Zakhor, and B. Nikolic, "Reduced complexity compression algorithms for direct-write maskless lithography systems," in *Proceedings of SPIE 6151*, 2006, p. 61512B. [Online]. Available: http://dx.doi.org/10.1117/12.656844

[28] ——, "Reduced complexity compression algorithms for direct-write maskless lithography systems," *J. Micro/Nanolith. MEMS MOEMS*, vol. 6, no. 1, p. 013007, 2007. [Online]. Available: http://dx.doi.org/10.1117/1.2435202

[29] S. W. Golomb, "Run-length encodings," *IEEE Transactions on Information Theory*, vol. 12, no. 3, pp. 399–401, 1966. [Online]. Available: http://dx.doi.org/10.1109/TIT.1966.1053907

[30] J. Yang and S. A. Savari, "A lossless circuit layout image compression algorithm for maskless lithography systems," in *Proceedings of the 2010 Data Compression Conference*, 2010, pp. 109–118. [Online]. Available: http://dx.doi.org/10.1109/DCC.2010.17

[31] ——, "Transform-based lossless image compression algorithm for electron beam direct write lithography systems," in *Recent Advances in Nanofabrication Techniques and Applications*, B. Cui, Ed. InTech, 2011, pp. 95–110. [Online]. Available: http://dx.doi.org/10.5772/21896

[32] F. Krecinic, S.-J. Lin, and J. J. H. Chen, "Data path development for multiple electron beam maskless lithography," in *Proceedings of SPIE 7970*, 2011, p. 797010. [Online]. Available: http://dx.doi.org/10.1117/12.881010

[33] A. Moffat, R. M. Neal, and I. H. Witten, "Arithmetic coding revisited," *ACM Transactions on Information Systems*, vol. 16, no. 3, pp. 256–294, 1998. [Online]. Available: http://dx.doi.org/10.1145/290159.290162

[34] A. Gu and A. Zakhor, "Lossless compression algorithms for hierarchical IC layout," *IEEE Transactions on Semiconductor Manufacturing*, vol. 21, no. 2, pp. 285–296, 2008. [Online]. Available: http://dx.doi.org/10.1109/TSM.2008.2000282

[35] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Communications of the ACM*, vol. 30, pp. 520–540, June 1987. [Online]. Available: http://dx.doi.org/10.1145/214762.214771

[36] M. Peon, R. R. Osorio, and J. D. Bruguera, "A VLSI implementation of an arithmetic coder for image compression," in *23rd EUROMICRO Conference '97 New Frontiers of Information Technology*, 1997, pp. 591–598. [Online]. Available: http://dx.doi.org/10.1109/EURMIC.1997.617380

[37] (2007) JBIG official website. [Online]. Available: http://www.jpeg.org/jbig

[38] M. Kuhn. (2008) JBIG-KIT. [Online]. Available: http://www.cl.cam.ac.uk/~mgk25/jbigkit/

[39] (2009) LibTIFF library. [Online]. Available: http://www.libtiff.org

[40] V. Dai and A. Zakhor, "Lossless compression of VLSI layout image data," *IEEE Transactions on Image Processing*, vol. 15, no. 9, pp. 2522–2530, 2006. [Online]. Available: http://dx.doi.org/10.1109/TIP.2006.877414

[41] J. Yang, S. A. Savari, and X. Li, "Hardware implementation of Corner2 lossless compression algorithm for maskless lithography systems," in *Proceedings of SPIE 8323*, 2012, p. 83232O. [Online]. Available: http://dx.doi.org/10.1117/12.917581

[42] J. Yang and S. A. Savari, "Lossless circuit layout image compression algorithm for maskless direct write lithography systems," *J. Micro/Nanolith. MEMS MOEMS*, vol. 10, no. 4, p. 043007, 2011. [Online]. Available: http://dx.doi.org/10.1117/1.3644620

[43] (2012) Impulse CoDeveloper by Impulse Accelerated. [Online]. Available: http://www.impulseaccelerated.com

[44] D. Lucking and E. Balster, "An increased throughput FPGA design of the JPEG2000 binary arithmetic decoder," in *2010 International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, 2010, pp. 400–405. [Online]. Available: http://dx.doi.org/10.1109/DICTA.2010.74

[45] J. Yang and S. A. Savari, "Improvements on Corner2, a lossless layout image compression algorithm for maskless lithography systems," in *Proceedings of SPIE 8352*, 2012, p. 83520K. [Online]. Available: http://dx.doi.org/10.1117/12.923205

[46] L. A. Wolsey, *Integer Programming*. New York: Wiley-Interscience, 1998.

[47] (2012) bintprog by MathWorks. [Online]. Available: http://www.mathworks.com/help/toolbox/optim/ug/bintprog.html

[48] J. Yang, S. A. Savari, and H. R. Harris, "Data delivery system for multiple electron beam lithography systems using image compression," *J. Micro/Nanolith. MEMS MOEMS*, 2012, submitted.

[49] (2011) Intel Core i7-920 Processor Specifications. [Online]. Available: http://ark.intel.com/products/37147/Intel-Core-i7-920-Processor-(8M-Cache-2_66-GHz-4_80-GTs-Intel-QPI)

[50] (2012) Nvidia GeForce GTX 690 GPU Specification. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-690/specifications