Compression Algorithms for "Dummy Fill" VLSI Layout Data

Robert B. Ellis[†], Andrew B. Kahng[‡] and Yuhong Zheng[§]

[†]Mathematics Department, Texas A&M University, College Station, TX 77843-3368
[‡]UCSD CSE and ECE Departments, La Jolla, CA 92093-0114
[§]UCSD CSE Department, La Jolla, CA 92093-0114

ABSTRACT

Dummy fill is introduced into sparse regions of a VLSI layout to equalize the spatial density of the layout, improving uniformity of chemical-mechanical planarization (CMP). It is now well-known that dummy fill insertion for CMP uniformity changes the back-end flow with respect to layout, parasitic extraction and performance analysis. Of equal import is dummy fill's impact on layout data volume and the manufacturing handoff. For future mask and foundry flows, as well as potential maskless (direct-write) applications, dummy fill layout data must be compressed at factors of 25 or greater. In this work, we propose and assess a number of lossless and lossy compression algorithms for dummy fill. Our methods are based on the building blocks of JBIG approaches - arithmetic coding, soft pattern matching, pattern matching and substitution, etc. We observe that the fill compression problem has a unique "one-sided" characteristic; we propose a technique of achieving *one-sided loss* by solving an *asymmetric cover problem* that is of independent interest. Our methods achieve substantial improvements over commercial binary image compression tools especially as fill data size becomes large.

Keywords: Dummy fill, JBIG1, JBIG2, Bi-level data compression, Fill compression, Asymmetric cover

1. INTRODUCTION

In modern VLSI manufacturing processes, material is deposited layer by layer, with interlevel dielectric (ILD) between layers. Each layer needs to be polished flat by a process known as chemical-mechanical planarization (CMP).[1] For example, a pattern of copper interconnects is deposited and silicon dioxide spun on as an insulator; they are planarized before the next layer of copper interconnects can be created.

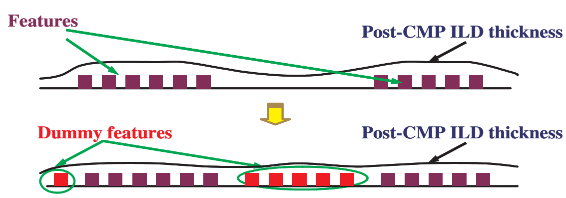

Figure 1: Insertion of dummy features to decrease post-CMP variation of ILD thickness.

The CMP result will not be flat unless the layout geometries in the previous layer of material exhibit uniform spatial density, i.e., every "window" of given size in the chip layout contains roughly the same total area of copper wiring. Therefore millions of "dummy fill" features (typically, small squares on a regular grid) are introduced into sparse regions of the layout to equalize the spatial density as shown in Figure 1. The downside is that the layout data file size is dramatically increased. A small layout data file is desired in order to quickly transmit the chip design to the manufacturing process. Compressing the dummy fill data is a straightforward solution to the problem.

Further motivation for our work stems from the increased nonrecurring cost of photomasks, which according to Sematech reaches up to $1M per mask set in the 130nm technology node. While improved mask production efficiencies – particularly in inspection and testing – may reduce this cost,[15] maskless lithography is now a requirement in the Lithography chapter of the Semiconductor Technology Roadmap.[17] Work of Dai and Zakhor[14] estimates that a compression

Corresponding author: Andrew B. Kahng (abk@ucsd.edu).

factor of 25 is required to enable direct-write maskless lithography (i.e., to transfer trillions of pixels per second onto the wafer). Indeed, Dai and Zakhor investigate several lossless compression schemes that we also study; however, their layout data is (5-bit) gray-scale and corresponds to functional layout data, while our dummy fill data is almost perfectly represented by a binary (0-1) matrix. We note that the fill data and the feature data must eventually be combined in order for the mask write to make a single pass over the chip surface. We assume that these data are compressed, transmitted and decompressed *separately*, and then combined in the mask writer. This is consistent with today's methodologies, which *separately* manage fill data as a distinct GDSII layer (cf. the concept of a "target layer"[16]), so as to avoid impact on layout hierarchy.

Symbol	Description	Symbol	Description
B	fill data file, an m by n binary matrix	$s(D)$	# of 1's in D
D	data block, a b_1 by b_2 sub-matrix of B	g	global loss ratio
R	reference block, a b_1 by b_2 binary matrix	k, f	proportional loss ratio, fixed speckle loss
C	cover block, a b_1 by b_2 binary matrix	$w(D)$	# of bits of D allowed to change from 1 to 0
$\mathcal{C}$	cover, a set of C's "close" to a set of data blocks	$\mathrm{H}(D,R)$	Hamming distance between D and R
m,n	# of rows, columns of B	c	matching ratio, D matches R iff $\mathrm{H}(D,R) \leq b_1 b_2 c$
b_1, b_2	# of rows, columns of D, R, C	O_{D}	size of the compressed D's
p	m/b_1, # of data blocks across a row of B	O_{RB}	size of the compressed R's
q	n/b_2, # of data blocks down a column of B	O_{RI}	size of the compressed I's
I	1 by pq array indexed by data blocks giving indices of matching reference blocks	I_{PM}	1 by pq array indexed by data blocks giving indices of perfectly matching reference blocks
O_{PMRI}	size of the compressed I_{PM}'s	h	Total size of the compressed items ($h = \sum_x O_x$)

Table 1: Terminology

The terminology that we use below is summarized in Table 1. A layout containing dummy fill features can be expressed as a binary (0-1) matrix.[*] Fill compression takes as input an $m \times n$ binary matrix B and outputs compressed items (D's, R's, I, etc.) with total size h. The objective of fill compression is to minimize h. The *compression ratio* is defined as $r = mn/h$. In this work, we develop a number of compression heuristics based on Joint Bi-Level Image Processing Group (JBIG) methods.[2–4] The algorithms can be lossless when fill must appear exactly as specified, or lossy when a "close" approximation of the fill pattern suffices. Different loss tolerances can be applied according to the application context. Our lossy compression algorithms allow both *proportional loss*, a prescribed upper bound on the *fraction* of 1's in a single data block which may be changed to 0's, and *fixed loss*, a prescribed upper bound on the *absolute number* of 1's in a single data block which may be changed to 0's. In the former context, for a given data block D we may change at most $w(D) = \lfloor k \cdot s(D) \rfloor$ of its 1's to 0's. All algorithms that we study have the following outline.

ALGORITHM 1 (GENERAL COMPRESSION SCHEME).

1. *Segment data matrix B into blocks D.*
2. *If lossy compression is desired,*
   (a) *generate a cover $\mathcal{C}$ for data blocks D (every D must match, modulo possible loss, a cover block $C \in \mathcal{C}$), and*
   (b) *replace B with lossy matrix B' by replacing each data block D with its matching cover block C.*
3. *Perform lossless compression on B.*

Section 2 surveys off-the-shelf software for compression and presents JBIG methods for binary data compression. JBIG1 and JBIG2 methods will be used in Step 3 of the above algorithm outline. Benchmarks of compression ratio and runtime performance are provided by off-the-shelf software. Classification of our compression heuristics is given in Section 3, and experimental results are given in Section 4. Ongoing and future research is presented in Section 5.

2. BI-LEVEL DATA COMPRESSION METHODS

Many methods have been developed for compression of bi-level data, including several familiar *off-the-shelf* commercial tools for lossless compression.[†] JBIG (Joint Bi-level Image Experts Group) algorithms have emerged as the most promising for bi-level data compression. JBIG algorithms combine arithmetic coding with context-based modeling to exploit

[*]This is true of all major commercial fill insertion tools such as Mentor Calibre, Avant! Hercules and Cadence Assura, even when operating in modes that output "tilted fill" or tiled "fill cells".

[†]Winrar, Gzip and Bzip2 are obvious examples. Gzip and Bzip2 have better performance for bi-level data compression compared to other commercial tools. The core of Gzip is Lempel-Ziv coding (LZ77).[2] Bzip2's compression mechanism is based on Burrows-

the structure of the data in order to obtain better compression ratios. We here review the two classes of JBIG algorithms, followed by brief comments on existing commercial compression tools.

2.1. JBIG*

JBIG[5] is an experts group of ISO, IEC and CCITT (JTC1/SC22/WG9 and SGVIII). Its goal is to define compression standards for bi-level image coding; in 1993 the group proposed JBIG1 as the international standard for lossless compression of bi-level images (ITU-T T.82). In 1999, JBIG developed JBIG2,[3] which is the first international standard that provides for both lossless and lossy compression of bi-level images. We use JBIG* to refer to either JBIG1 or JBIG2. Arithmetic coding and context-based statistical modeling are two key components of the JBIG* methods.

Arithmetic Coding. As the basis of most efficient binary data compression techniques, *arithmetic coding* completely bypasses the idea of replacing an input symbol with a specific code.[6] Instead, it takes a stream of input symbols and outputs a single floating-point number. Generally speaking, the interval $[0,1)$ is partitioned so that each part (subinterval) corresponds to a possible first symbol. A given subinterval is partitioned so that each part corresponds to a possible second symbol, and so on, until the desired string length is reached. The size of a symbol's subinterval relative to that of its parent interval is proportional to the probability of that symbol occurring. A symbol stream is represented by the lowest value of its innermost subinterval.

Context-Based Modeling. JBIG* performs *context-based* encoding. Context-based compression methods assume that the value of a given bit can be predicted based on the values of a given *context* of surrounding bits. The context of a bit consists of some fixed pattern of its neighboring bits; when these bits have been scanned, we know the exact context. These compression methods scan a binary matrix and for each distinct context record the actual frequencies of the corresponding bit being either 1 or 0. Thus when a given context appears in the matrix, an estimated probability is obtained for a bit to be 1. These probabilities are then sent to an arithmetic encoder, which performs the actual encoding. Figure 2(a) shows a possible 7-bit context (the bits marked "P") of the current bit "X", made up of five bits above and two on the left. The unscanned (unknown) bits are marked "?". We use the values of the seven bits as an index to a frequency lookup table. The table stores frequencies of 0's and 1's already scanned for $2^7 = 128$ different contexts.[‡]

.	.	P	P	P	P	P	.	.
.	.	P	P	**X**	?	?	?	?

(a)

	1	2	3	
4	5	6	7	8
9	10	**X**		

(b)

Figure 2: (a) a 7-bit context, (b) a 10-bit template used in JBIG1 (the numbered bits are used as the context of the bit "X")

2.2. JBIG1

In the JBIG1 algorithm, bits are coded in raster-scan order by arithmetic coding that uses probabilities estimated from the bits' contexts. The context of a bit for probability estimation consists of a number of its neighbor bits that have already been encoded. A 10-bit template is generally used, as shown in Figure 2(b). Frequencies of 1's and 0's indexed by 1024 (2^{10}) context patterns are contained in a frequency table. The probability is estimated adaptively in that the frequency of a bit's context pattern is updated after it is scanned:

for each bit of a matrix in raster-scan order **do**
  Find its context (neighboring bits)
  Find probabilities of the current bit being 0 or 1, given context
  Encode current bit using arithmetic coding
  Update table of 0-1 frequencies for this context
end for

Wheeler block-sorting text compression and Huffman coding; its performance is generally better than those of more conventional LZ77/LZ78-based compressors.[3]

‡The frequency table should be initialized to nonzero values. It seems better to initialize every table entry to have either bit 0 or 1 with a frequency of 1. When the process starts, the first bit to be scanned does not have any neighbors above it or to the left. If the context pattern of a bit does not fit inside the matrix, we assume that any context bits that lie outside the matrix are 0.

2.3. JBIG2

A properly designed JBIG2 encoder not only achieves higher lossless compression ratios than other existing standards, but also enables very efficient lossy compression with almost unnoticeable information loss. *Pattern matching and substitution* (PM&S) and *soft pattern matching* (SPM) are two compression modes of JBIG2. It is critical to note here, however, that all new lossy compression algorithms presented in this paper will not introduce loss using JBIG2, but rather by means described in Section 3.1. We present below only lossless JBIG2 compression; for lossy JBIG2 compression, the reader is referred to the literature.[3,4]

JBIG2 based on PM&S. PM&S compresses a data file by extracting repeatable patterns and encoding these patterns, their indices, and their positions in the file instead. Due to the relatively random distribution of dummy fill, we segment the input 0-1 matrix into data blocks, and then use the PM&S method for compression. The encoding procedure involves the following steps:

1. Segment input 0-1 matrix B into data blocks D
2. Construct a dictionary consisting of the set of reference blocks R
3. For each data block, search for an exactly matching reference block R in the dictionary
4. Encode the reference block index of the matching
5. Encode the dictionary of reference blocks.

JBIG2 based on SPM. JBIG2's SPM mode differs from the PM&S in that imperfect matches from the dictionary are used to guide lossless compression of data blocks. In the SPM method, we begin with the input binary matrix segmented into data blocks, and a dictionary of reference blocks of the same dimensions as the data blocks. All of the bits are encoded in raster-scan order by an arithmetic coder as in JBIG1, but using a different context to estimate the current bit. An 11-bit context template, shown in Figure 3, is generally used. Bits in the part of the context which has already been scanned are taken from the input matrix. Bits in the part of the context which has not been scanned are taken from the corresponding bits of the reference block that matches the current data block. The geometric center of the current block is aligned with that of the matching reference block. The bit numbered "7" corresponds to the bit marked "X". This is also called *refinement coding*. The pseudocode is:

```
Segment input 0-1 matrix into data blocks
Construct dictionary consisting of reference blocks
for each data block do
    Search for an "acceptable" matching reference block in the dictionary
    if there is a match
        Encode index of the matching reference block
        Encode data block using refinement coding
    else Encode data block directly
    end if
end for
```

1	2	3
4	**X**	

(a)

	5	
6	7	8
9	10	11

(b)

Figure 3. 11-bit template used in SPM coding (the numbered bits are used as the context of the bit marked "X"): (a) bits taken from the causal part of the current block, (b) bits taken from the matching reference block

3. TAXONOMY OF COMPRESSION HEURISTICS

In this section, we present details of the general compression scheme of Algorithm 1. For this model, introduction of loss and compression are kept completely separate; we first introduce loss in Step 2, and then perform lossless compression in Step 3 of Algorithm 1. Loss is introduced by replacing the data blocks by using an *asymmetric cover* consisting of *cover blocks*, where the replacement allows 1's to change to 0's but not vice versa. Section 3.1 describes the two types of asymmetric covers that will be chosen from whenever we introduce loss. Section 3.2 focuses on methods of lossless compression. The two main components of lossless compression we are interested in are *dictionary generation* and *dictionary compression*. In particular, we concentrate on the following lossless compression schemes: JBIG2 PM&S, JBIG2 SPM, JBIG1 compression of dictionaries, and JBIG1 compression of singleton data blocks (data blocks which match themselves only). The JBIG2 SPM scheme also allows loss during soft pattern matching, but for conceptual clarity we will only consider loss introduced in Step 2 of Algorithm 1, and thus only consider lossless JBIG2 SPM schemes for use in Step 3.

3.1. Asymmetric Cover

The problem of building a cover for a set of data blocks is an instance of the Set Cover Problem (SCP), which is known to be NP-hard.[7] One formulation of SCP is as follows.

Set Cover Problem (SCP). Given a ground set $\mathbf{X}$ and a collection $\mathcal{C}$ of subsets of the ground set with $\cup_{C\in\mathcal{C}}C = \mathbf{X}$, choose the smallest subset $\mathcal{C}_0 \subseteq \mathcal{C}$ such that $\cup_{C\in\mathcal{C}_0}C = \mathbf{X}$.

Our cover problem is an instance of SCP by taking $\mathbf{X}$ to be the set of distinct data blocks and by identifying a cover block C with the set of data blocks which it covers. This set of data blocks is determined by the type of asymmetric cover considered. For *proportional loss*, we allow at most a fixed percentage of the 1's in a data block to change to 0's in order for it to be covered by the cover block determined by the bit changes. For *fixed speckle loss*, we allow changing at most a fixed number of isolated 1's in a data block to 0's for it to be covered by the resulting cover block. The *radius* of a cover block C is $\max_D(\mathrm{H}(C,D))$, where the maximum is taken over all data blocks covered by C. Thus proportional loss corresponds to a *proportional-radius* cover and fixed speckle loss corresponds to a *fixed-radius* cover. In either case, loss is controlled by a *global loss ratio* g, which is the allowed global change in density of 1's. The allowed proportional loss is $k = g/p_1$, where p_1 is the fraction of bits in B that are 1's, and the allowed fixed speckle loss is $f = \lfloor g\cdot b_1 b_2\rfloor$. In either case, the maximum number of 1's changed to 0's in B will be $\lfloor g\cdot mn\rfloor$.

3.1.1. Proportional Loss

Because SCP is NP-Hard, we have little hope of obtaining an optimal set cover quickly, and so reduce our goal to achieving a reasonably good set cover. Our method for constructing $\mathcal{C}$ views the data blocks as vertices of a graph G with edges weighted according to the potential for using the same cover block to cover the two adjacent data blocks.[§] We refer to this heuristic as the proportional loss algorithm, and present the algorithm itself after some necessary terminology.

Proportional loss algorithm background. Generally, we identify a number of distinct data blocks which are clustered closely together, where closeness is roughly determined by Hamming distance. A cluster of data blocks which can be covered by the same cover block are identified, covered and set aside, until all data blocks have been covered. This scheme can be implemented by simply constructing a list of data blocks and searching through the list for the clusters, but a more efficient algorithm involves constructing a graph whose vertices are the data blocks and whose edges are a measure of potential for the two data blocks incident to the edge to be covered by the same cover block. Data blocks are clustered together in the graph by successively contracting edges and replacing a pair of data blocks with a single representative data block (which covers both original data blocks). When a vertex becomes disconnected from the graph, it is used as the cover block for all data blocks contracted into it, and then removed. The algorithm terminates when the graph is empty.

An edge $\{D_1, D_2\}$ between two data blocks is present in the graph with weight $w(D_1,D_2)$ if and only if the quantity

$$w(D_1,D_2) := \min(t(D_1) - \mathrm{H}(D_1, D_1\wedge D_2), t(D_2) - \mathrm{H}(D_2, D_1\wedge D_2)). \tag{1}$$

is nonnegative, where $t(D) = \lfloor k\cdot s(D)\rfloor$ is the total allowable loss for D, and '$\wedge$' is the bit-wise AND of two data blocks. This expression encapsulates the fact that D_1 and D_2 can be covered by the same cover block if and only if $w(D_1,D_2) \geq 0$. In particular, D_1 and D_2 can be covered by the same cover block if and only if they can both be covered by $D_1\wedge D_2$. We cluster D_1 and D_2 together by replacing both vertices with the vertex $D = D_1\wedge D_2$ and setting $t(D) = w(D_1,D_2)$. Then $w(D,D_3)$ must be updated by (1) for any vertices D_3 originally adjacent to both D_1 and D_2. If D_3 is not adjacent to both D_1 and D_2, it is impossible to cover all three with a single cover block. Given the above definitions, we now present the proportional loss algorithm. The clustering operation in Step 3 of the algorithm is illustrated in Figure 4.

[§]Several random algorithms were implemented to search for good covers, but in every case, the search space is extremely large, the complexity of randomly choosing a cover block for a given data block based on any range of statistics is prohibitive, and the resulting random covers were noncompetitive with greedy covers. Therefore we only present our best greedy cover algorithm.

Algorithm 2 (Proportional Loss Algorithm).

1. *Build graph G with* $V(G) = \{D : D \text{ a data block}\}$. *Initialize* $E(G) = \emptyset$, $\mathcal{C} = \emptyset$, *and* $t(D) = \lfloor k \cdot s(D) \rfloor$.

2. *Sort the vertices in decreasing order by weight* $s(D)$. *For all pairs of data blocks* $\{D_1, D_2\}$ *with* $\mathrm{H}(D_1, D_2) \leq \min(t(D_1), t(D_2))$, *compute edge weight* $w(D_1, D_2)$, *adding the edge to* $E(G)$ *iff* $w(D_1, D_2) \geq 0$

3. *Pick the first vertex* D_1 *in sorted order.*
   (a) *Pick the neighbor* D_2 *of* D_1 *maximizing* $w(D_1, D_2)$.
   (b) *Replace* D_1 *and* D_2 *with* $D = D_1 \wedge D_2$, *and set* $t(D) = w(D_1, D_2)$.
   (c) *Remove all edges incident to* D_1 *or* D_2.
   (d) *For all data blocks* D_3 *previously adjacent to both* D_1 *and* D_2 *in Step (a), add an edge* $\{D, D_3\}$ *with weight* $w(D, D_3)$ *iff* $w(D, D_3) \geq 0$.
   (e) *Go to Step (a) until the cluster represented by D is disconnected from the rest of the graph.*
   (f) *Add the resulting vertex to* $\mathcal{C}$, *and use it as a cover block for all data blocks in the cluster, and remove the vertex from the graph.*

4. *Repeat Step 3 until the graph has no vertices left.*

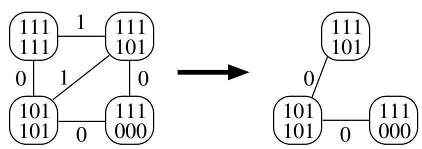

Figure 4. Step 3 of Algorithm 2 is illustrated by the clustering of data blocks 111111 and 111101, which will both eventually have cover block 111101 or 101101. Edges are labeled by weight w, and $k = 1/3$.

3.1.2. Fixed Speckle Loss

Given a data block, we may induce a gridgraph topology over the 1-bits by deleting vertices corresponding to 0-bits. A *speckle* is a connected component of 1's in this gridgraph. Notice that if we allow at most a fixed number of 1's to be lost from each data block, the resulting blocks will correspond to a fixed-radius asymmetric cover of the original blocks.[¶] We may choose the lost 1's by discarding speckles of increasing size, until the allowed fixed speckle loss f (corresponding to a given global loss bound, say, $g = 2\%$) is reached. We call this the *fixed speckle loss* method. We could also modify Algorithm 2 by setting $t(D) = f$ in Step 1, which causes the algorithm to construct a fixed-radius cover instead of a proportional-radius cover. This results in a potentially more powerful generalization of the fixed speckle loss method and remains to be investigated. We choose the fixed speckle loss method because of the much lower computational cost of searching in raster-scan order for speckles of small size.

3.2. Dictionary Construction & Compression

Dictionary construction is very important in JBIG2. A good dictionary should contain a small number of reference blocks which match a much larger number of data blocks. A small dictionary also means that reference indices (pointing from data blocks to reference blocks) are shorter. Removing singletons from the dictionary will reduce the size of reference indices without increasing the sizes of the compressed blocks (all reference and/or singleton blocks). The simplest dictionary formation approach is order-dependent, where the dictionary is generated in a sequential way. Initially, the first data block is put into the dictionary as the first reference block. All the subsequent data blocks are compared with reference blocks in the dictionary. A data block D matches a reference block R provided that $\mathrm{H}(D, R) \leq b_1 b_2 c$; assign D's reference index to the best match R minimizing $\mathrm{H}(D, R)$. A data block is put into the dictionary if it matches none of the reference blocks. Currently, each data block matches exactly one reference block.

Singleton exclusion. The dictionary may, however, contain many singletons. A singleton is a data block which is identical to its matching reference block, where the reference block matches only with that one data block. Using a dictionary does not allow for better compression of singletons, since each singleton data block must still be encoded; in addition, inclusion of singletons in the dictionary causes reference indices to be longer. A "singleton exclusion dictionary"

[¶]In other words, in terms of Algorithm 1, the "cover blocks" in $\mathcal{C}$ are just the set of distinct data blocks after speckles have been removed.

encodes only those non-singleton data blocks by using the dictionary, and compresses singletons separately using JBIG1. This prevents encoding of a reference index for a reference block that would be used only once.

Dictionary compression. The dictionary itself is compressed using JBIG1 on the reference blocks. Data blocks which match reference blocks in the dictionary are replaced with reference indices, which are encoded with multi-symbol arithmetic coding.

Combining PM&S and SPM. JBIG2 compression can be used with a combination of PM&S and SPM in the following fashion. For the PM&S part, some data blocks will perfectly match a reference block. Only an index is encoded for these data blocks. The other data blocks will only match a reference block imperfectly, and so SPM (refinement coding) is needed to encode both an index and additional bits used to record the difference between the data block and the reference block. Singletons may be removed first, if desired. Also, we note that PM&S tends to be faster than SPM due to the complexity of refinement coding.

3.3. A Component-Wise Taxonomy of New Algorithms

Table 2(a) summarizes the pieces that will be combined to construct compression algorithms. Piece "A" corresponds to compressing the entire matrix with arithmetic encoding and does not correspond to the framework of Algorithm 1. We use "A" as a comparison benchmark. Pieces "B" and "C" respectively introduce proportional and fixed speckle loss, corresponding to Step 2 of Algorithm 1; pieces "D-G" are schemes of lossless compression which correspond to Step 3. Generally, there are four constituents of the compressed data file: O_{D}, O_{RI}, O_{PMRI}, and O_{RB}. Because data blocks are fully replaced by cover blocks, Step 2 does not affect the constituents of the compressed file; however these constituents do depend on the choices made in Step 3. O_{PMRI} consists of compressed reference indices of data blocks which perfectly match reference blocks and only appears with piece "D". O_{RI} consists of compressed reference indices, denoting matches between data blocks and reference blocks, and only appears with piece "E". O_{D} consists of compressed singleton data blocks and possibly SPM refinement coding information and only appears with pieces "E" and "F". O_{RB} consists of compressed reference blocks in the dictionary and only appears with piece "G". We now list and summarize the best heuristics we tested, classified by their component pieces (see Table 2(b))[‖]

Index		Piece description
Benchmark	A	Compress matrix using JBIG1
Loss introduction	B	Proportional loss (Algorithm 2)
	C	Fixed speckle loss (§3.1.2)
JBIG2 lossless components	D	JBIG2 PM&S
	E	JBIG2 SPM (lossless)
	F	Singleton exclusion & singleton data blocks compressed by JBIG1
Compress dictionary	G	JBIG1 on reference blocks

(a)

		A1	A2.1	A2.2	A2.3	A3
Pieces	A	✓				
	B			✓		✓
	C				✓	
	D		✓	✓	✓	✓
	E		✓	✓	✓	
	F		✓	✓	✓	✓
	G		✓	✓	✓	✓
Outputs	O_{D}	✓	✓	✓	✓	✓
	O_{RI}		✓	✓	✓	✓
	O_{PMRI}		✓	✓	✓	
	O_{RB}		✓	✓	✓	✓

(b)

Table 2: (a) Description of algorithm pieces, (b) Components and output of heuristic algorithms)

- **A1 (JBIG1, lossless, Pieces: A).** The entire data file is compressed using JBIG1. In particular, dictionary construction is not required. This algorithm is used to benchmark other compression ratios against JBIG1.
- **A2.1 (JBIG2, lossless, Pieces: D, E, F, G).** The data is segmented into data blocks, and a dictionary is constructed using an order-dependent method. Singletons are excluded and compressed separately by JBIG1. Data blocks which match reference blocks perfectly are encoded using PM&S. The other data blocks are encoded using SPM. The dictionary is encoded with JBIG1.
- **A2.2 (JBIG2, proportional loss, Pieces: B, D, E, F, G).** The data blocks are replaced and loss introduced using Algorithm 2. Then proceed as in A2.1.
- **A2.3 (JBIG2, fixed loss, Pieces: C, D, E, F, G).** The data blocks are replaced using fixed speckle loss (Section 3.1.2). Then proceed as in A2.1.

‖Heuristics comprised of all combinations of pieces were investigated; but the ones listed gave the best compression ratios in nearly every test case.

- **A3 (JBIG2, proportional loss, Pieces: B, D, F, G).** The data blocks are replaced and loss introduced using Algorithm 2. Then an order-dependent dictionary is generated, where PM&S is used to generate reference indices to be encoded with multi-symbol arithmetic coding. Singletons are excluded and compressed separately by JBIG1. The other data blocks are encoded using PM&S. The dictionary is encoded using JBIG1.

General Compression Algorithm (A2-A3).

1. Segment data matrix B into blocks D.
2. If lossy compression is desired, generate an asymmetric cover $\mathcal{C}$ with either proportional loss or fixed speckle loss and replace data blocks (A2.2, A2.3, A3).
3. Perform lossless compression using order-dependent dictionary and JBIG2:
   (a) Exclude singletons (A2, A3)
   (b) PM&S on data blocks with perfectly matching reference blocks (A2, A3)
   (c) SPM on remaining data blocks (A2).
4. Compress dictionary using JBIG1 (A2,A3).

4. EXPERIMENTAL RESULTS

We report compression ratios for our lossy compression algorithms and for off-the-shelf benchmarks JBIG1 and Bzip2. Compression ratios meeting or exceeding the estimated ratio of 25 needed to enable direct-write maskless lithography[14] are obtained in many cases – especially for the larger data files. Section 4.1 motivates the choice of various parameters such as JBIG1 context size and shape, and block size and shape. Section 4.2 presents the compression ratios and running times of proportional loss and fixed speckle loss compression methods.

4.1. Parameterization of Implementation

There are several parameters that control the implementation of the "pieces". In this subsection, we explain the empirical process that was used to select reasonable values for these parameters (e.g., context size, block size, etc.).

Sensitivity of JBIG1 compression ratio to context size. Templates with different context size could be used in JBIG1 for probability estimation, as shown in Table 3. Several binary matrices are compressed using JBIG1 and a choice of templates in Table 3. The resulting compression ratios are shown in Table 4. We see that templates with more bits lead to better results at the cost of more CPU time.

1-bit	4-bit	12-bit	18-bit
c_1 x	c_1 c_2 c_3 c_4 x	c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11} c_{12} x	c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11} c_{12} c_{13} c_{14} c_{15} c_{16} c_{17} c_{18} x

Table 3: Templates with different context size in JBIG1

Index	1-bit	4-bit	12-bit	18-bit
1	40.75	51.28	59.47	**63.10**
2	8.5	16.63	22.10	**25.72**
3	0.75	1.10	1.33	**1.44**
4	7.36	14.00	16.96	**18.11**

Table 4: Compression ratios for different context size in JBIG1 for 4 test cases

Sensitivity of JBIG1 compression ratio to context shape Templates with same context size but different context shape could also be used in JBIG1 for probability estimation (see Table 5). The compression results show that using templates with bits distributed in more rows will get better results; i.e., a 3-row template is better than 2-row or 1-row template.

Sensitivity of compression ratio to matching ratio. For the purpose of lossless soft pattern matching (SPM) compression in JBIG2, a *matching ratio c* is chosen to determine whether or not a data block matches a reference block. A data block D matches a reference block R provided that $\mathrm{H}(D,R) \leq b_1 b_2 c$. Choosing a larger c will increase the range of matching reference blocks for a given data block. However, a larger c will tend to decrease the similarity between D and any given match R, and so the probability estimation used in the JBIG2 refinement coding is less accurate; this leads to

Index	ccc ccccc ccx	cccccc ccccx	cccc cccc ccx	cccc cccc ccx	cc...cx
1	59.23	57.92	59.49	**59.61**	48.87
2	**22.07**	17.10	22.04	22.06	11.33
3	**1.33**	1.27	**1.33**	**1.33**	1.03
4	**16.97**	15.95	16.90	16.90	9.94

Table 5: Compression results using different 10-bit context shapes in JBIG1 for 4 test cases

worse compression performance. Figure 5 shows compression results of a data file 50-50-10-fill-31250-2-L1 with size $b_1 \cdot b_2 = 60 \cdot 60 = 3600$ containing real dummy fill features using different matching ratios. The compression heuristics used are A2.1 (lossless) and A2.2 (proportional loss ratios $k = 0.2, 0.4$). In Figure 5, the compression ratio generally decreases with increasing matching ratio, for both lossless and lossy compression. We would rather have more accurate probability estimation than more "matching" blocks; on the other hand, a matching ratio of 0 is not ideal either. The optimal matching ratio tends to be very small (.5-2.5%), but nonzero.

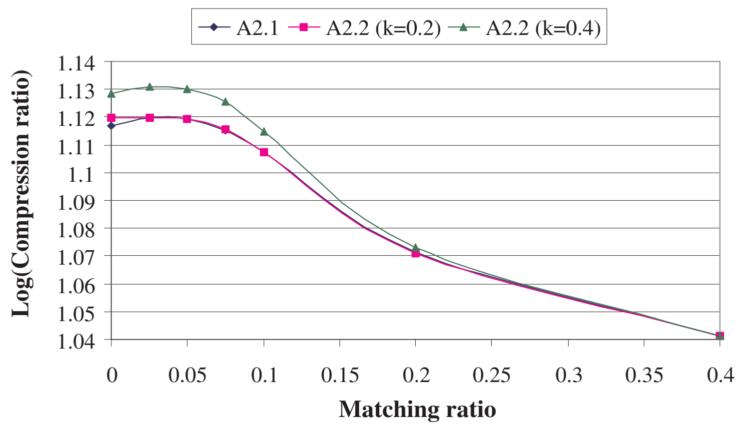

Figure 5: Sensitivity of compression ratio to matching ratio

Sensitivity of compression ratio to data block shape. Figure 6 shows compression ratios on the data file 50-50-10-fill-31250-2-L1 containing real dummy fill features using data blocks with size 3600 but different dimensions $b_1 \times b_2$. The heuristics used are A2.1, A2.2 ($k = 0.2$), and A2.2 ($k = 0.4$), with matching ratio of $c = .025$. Compression ratio is sensitive to data block shape only when the allowed proportional loss is high ($k = .4$), due to the resulting large change in the number of 1's; for the rest of the experiments, we choose blocks of varying dimensions.

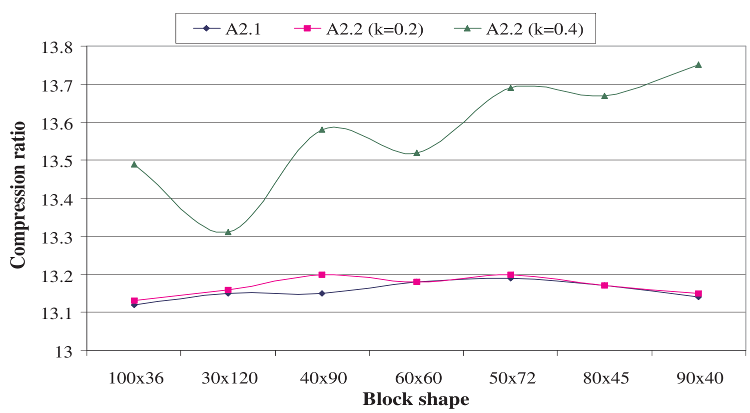

Figure 6: Sensitivity of compression ratio to data block shape for lossless (A2.1) vs. proportional loss (A2.2) heuristics

Sensitivity of compression ratio to data block size. Segmentation of the input binary matrix into data blocks appears important in JBIG2-related algorithms. The choice of the data block size is done largely empirically. Figure 9 shows compression results of the data file 50-50-10-fill-31250-2-L1 containing real dummy fill features using different data block sizes and a matching ratio of $c = .025$. The lossless compression heuristic A2.1 is used. As shown in Figure 9, the compression ratio increases sharply with the increase of the block size up to some point, after which it becomes nearly level. At the transition point, the compression ratio continues to decrease slightly. Empirically, the data block size can be chosen as the point where the increase in the compression ratio first slows to zero. The compressed files corresponding to the data points in Figure 9 have component compressed pieces whose sizes vary with data block size. Figure 8 illustrates this trade-off (total compressed file size = $O_{\rm D}$ + $O_{\rm RI}$ + $O_{\rm PMRI}$ + $O_{\rm RB}$). When the block size goes up, the dictionary of reference blocks is smaller but the entries themselves are larger, so $O_{\rm RI}$ decreases but $O_{\rm D}$ increases. The best choice of block size is large enough so that the gains in compression ratio (Fig. 9) has leveled off. Furthermore, sensitivity of compression ratio to data block size for lossless and proportional loss heuristics is studied using the same data file and matching ratio ($c = .025$) as in Figures 9 and 8. Figure **??** shows sensitivity results for compression heuristics A1 (lossless), A2.2 (proportional loss) with $k = .2$, and A2.2 with $k = 0.4$. Compression ratios for proportional loss heuristics benefit from smaller data block size by being able to change more 1's to 0's; the optimal data block size for proportional loss heuristics shifts to a smaller value than the optimal size for a lossless heuristic.

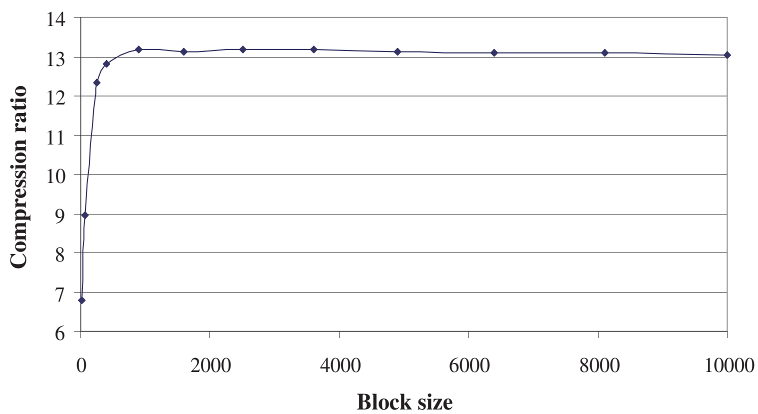

Figure 7: Sensitivity of compression ratio to data block size

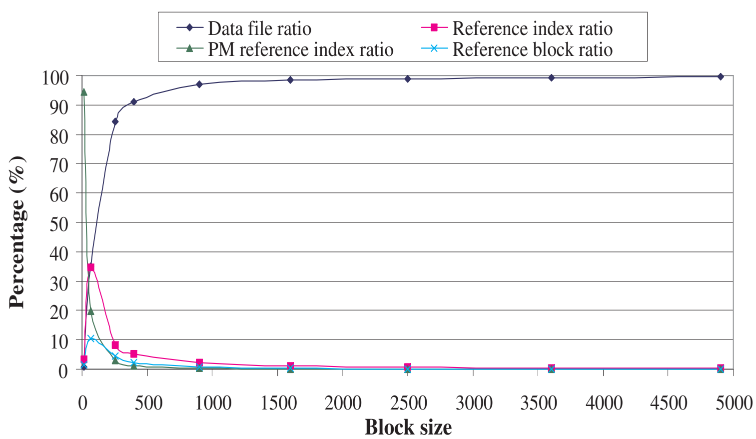

Figure 8: Variance of the four compressed output components with data block size

Sensitivity of compression ratio to proportional loss ratio. Allowing a higher *proportional loss ratio* k gives more latitude in changing 1's to 0's in generating the asymmetric cover, and thus gives better compression ratios. This is true especially since most fill data is dominated by 0's, and so decreasing the proportion of 1's suggests better compressibility simply by considering that there is an even smaller proportion of 1's to 0's. Figure 10 shows this property using proportional loss heuristics A2.2 and A3 on the data file 50-50-10-fill-31250-2-L1 with data block dimensions 60×60, matching ratio $c = .025$, and proportional loss ratios $k = 0.1, 0.2, 0.4, 0.6, 0.8$. In the figure, compression ratios increase slowly with increasing k at first, and then increase radically for larger k's. This is because the size of the proportional loss asymmetric cover decreases exponentially in k (see[11] for details), which in turn causes a much smaller dictionary to be required in the lossless compression stage. It should be mentioned here that a proportional loss heuristic is allowed to change up to

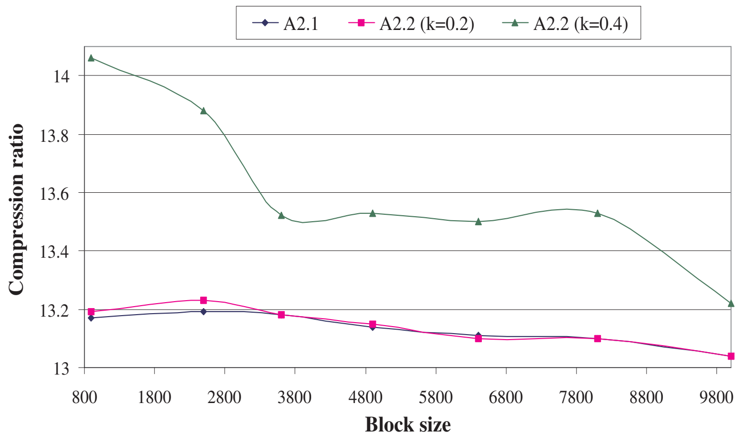

Figure 9: Sensitivity of compression ratio to data block size for lossless (A1) vs. proportional loss (A2.2) heuristics

the fraction k of 1's in each data block to 0's, but will only exploit whatever loss it determines helpful; furthermore, the possible global density change in 1's is bounded above by k times the original global density of 1's.

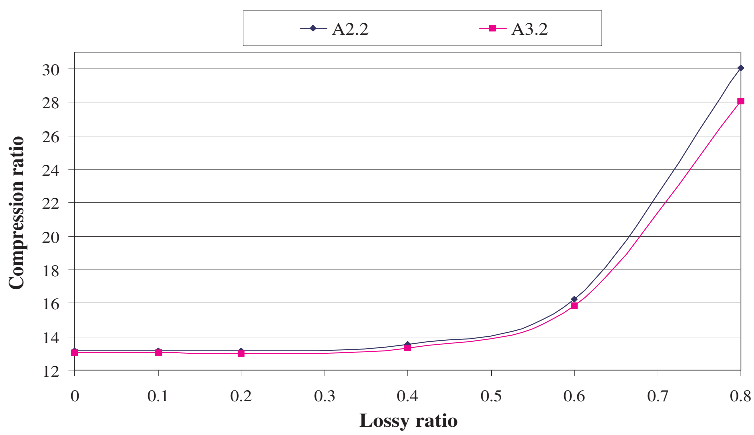

Figure 10: Sensitivity of compression ratio to proportional loss ratio k for heuristics A2.2 and A3

Experimental assumptions from sensitivity studies. All results from the above sensitivity studies are typical and suitable for all of the heuristics implemented and have also been tested for many different binary data files.

4.2. Experimental Discussion

All lossless and proportional loss methods are used to compress 15 real dummy-fill data files (cases 1-15) and 4 randomly generated data (cases 16-19). The number of binary bits is 2476×1167 in cases 1-6; 3973×4178 in cases 7,9, and 11; 4952×2333 in cases 8 and 10; 580×541 in cases 12-15; 500×600 in cases 16-17; 1000×1200 in case 18; and 2000×2500 in case 19. Resulting compression ratios and running times are listed in Tables 6(a) and (b), respectively. The system configuration is a Sun SPARC ULTRA-10 with 1GB DRAM. In all heuristics, a matching ratio of .025 is used. The block dimensions are 50×50 for case 1, 60×60 for cases 2-6, 100×100 for cases 7-11, 25×25 for cases 12-15, 30×30 for cases 16-17, 100×80 for case 18, and 40×60 for case 19.

The improvement in compression ratio r of algorithm Y over algorithm X is calculated by $(r_Y - r_X)/r_X$. We make the following observations.

- For compression ratios in lossless compression, A2.1 is best in nearly all test cases, saving 3.2%-113% vs. the best commercial software, Bzip2[**], and having an average improvement of 29.3%.
- A1 gives competitive compression ratios, saving 3.2%-109% vs. Bzip2, with average savings of 28.7%.

[**]Gzip and WinRAR were also used to test all cases, but Bzip2 performed better in every case while still having favorable running times.

- For compression ratios in lossy compression, A2.2 and A3 perform similarly in all test cases, with A2.2 on average being significantly slower but yielding slightly better compression ratios. On average, A2.2 saves 33.6% vs. Bzip2 with proportional loss ratio $k = 0.2$, and saves 54.8% on average with $k = 0.4$. On average, A3 ($k = 0.2$) saves 31.9% vs. Bzip2.
- For lossless compression, A1 is the most cost-effective method, taking only 2.7× longer than Bzip2 on average. A2.1 is nearly as cost effective, but takes 5.9× longer than Bzip2 on average.
- A3 is the most cost-effective proportional loss method, taking 3.7× longer than Bzip2 on average. The running time of A2.2 is 9.4× longer than Bzip2 on average with proportional loss ratio $k = 0.2$ and 10.3× longer with $k = 0.4$.

Case	Lossless			Lossy ($k = 0.2$)		($k = 0.4$)
	Bzip2	**A1**	**A2.1**	**A2.2**	**A3**	**A2.2**
1	24.78	26.78	45.45	101.06	68.72	100.52
2	23.78	26.83	36.52	69.48	49.12	72.24
3	25.54	27.65	30.22	76.67	59.68	80.19
4	24.95	26.85	60.66	72.06	36.96	83.54
5	24.33	27.53	47.07	66.47	40.26	73.47
6	24.88	26.69	46.14	67.99	41.24	74.23
7	13.92	154.2	272.46	531.44	165.00	548.48
8	51.96	107.14	182.02	370.46	110.29	394.57
9	10.22	153.93	350.68	609.92	166.89	629.22
10	43.29	107.17	314.04	370.68	130.62	434.49
11	12.13	154.14	479.65	600.26	188.25	623.96
12	2.14	2.92	6.09	10.63	7.61	13.91
13	1.62	2.95	6.18	9.71	7.62	14.29
14	2.95	2.90	6.16	9.13	6.00	10.85
15	4.11	2.91	6.02	9.19	6.06	11
16	2.28	2.78	1.36	5.49	4.78	5.52
17	2.37	2.79	1.19	5.99	5.30	5.99
18	10.12	11.27	15.00	17.7	13.42	17.14
19	37.63	46.83	124.15	253.25	173.49	258.02
(Ave.)	18.05	49.30	106.90	171.45	68.48	187.55

(a)

Case	Lossless			Lossy ($k = 0.2$)		($k = 0.4$)
	Bzip2	**A1**	**A2.1**	**A2.2**	**A3**	**A2.2**
1	12.20	12.24	13.19	13.23	13.17	13.88
2	10.45	12.22	17.74	17.73	17.78	17.86
3	13.04	16.98	18.35	18.31	18.33	18.53
4	5.14	10.75	10.93	11.10	11.12	11.36
5	8.88	9.67	10.20	10.19	10.17	10.41
6	8.91	12.29	12.82	12.83	12.84	13.00
7	80.00	85.47	85.39	90.52	87.43	110.27
8	23.30	22.08	23.56	23.56	23.16	24.30
9	62.50	64.52	64.50	70.29	65.50	90.62
10	12.00	19.49	19.81	19.97	19.63	20.44
11	28.65	38.76	38.82	41.23	39.07	55.76
12	1.23	1.67	1.66	1.66	1.67	2.40
13	0.95	1.33	1.33	1.33	1.34	1.93
14	1.85	2.26	2.25	2.25	2.26	2.33
15	1.27	1.57	1.56	1.56	1.57	1.56
16	40.45	77.48	70.36	70.36	75.45	71.84
17	19.23	31.35	30.89	30.89	31.04	35.05
18	2.08	2.89	2.89	3.00	3.05	4.51
19	3.98	6.09	6.06	6.07	6.10	9.52
(Ave.)	18.00	23.16	23.28	24.05	23.79	27.89

(b)

Table 6. (a) Running times using best performing lossless and proportional loss methods (unit: s), (b) Compression ratios on 19 test cases using best performing lossless and proportional loss heuristics)

Figure 11 shows compression ratios for the fixed speckle loss heuristic, A2.3. The three test cases are for the same data, block sizes, and matching ratio as the correspondingly numbered cases in Table 6; global loss ratios g = .25%, .5%, .75%, 1%, 2%, and 4% are used. For case 1, A2.3 gives compression ratio 14.31 with running time 24.63s for $g = .025$; this is 3.1% better and 75.5% faster than using A2.2 with $k = .4$, which corresponds to $g = .025$ since the percentage of 1's in the matrix of case 1 is 6.31%. For case 10, A2.3 gives compression ratio 24.48 with running time 246.74s for $g = .075$; this is 19.8% better and 43.2% faster than using A2.2 with $k = .4$, which corresponds to $g = .074$ since the matrix in case 10 is 18.4% 1's. For case 11, A2.3 gives compression ratio 46.71 with running time 398.50s for $g = .075$; this is 16.2% worse but 36.1% faster than using A2.2 with $k = .4$, which corresponds to $g = .072$ since the matrix in case 11 is 17.93% 1's.

5. SUMMARY AND FUTURE DIRECTIONS

We have implemented algorithms based on JBIG* methods in combination with PM&S, alternative dictionary generation mechanisms, and the new concept of *one-sided loss*, to compress binary data files of dummy fill features. Experimental results show that JBIG1 is quite effective, improving compression ratios by 28.7% vs. the best commercial tool Bzip2 on average, with (unoptimized) runtime penalty of approximately 2.7×. However, our new heuristics A2-A3 and the fixed speckle loss heuristic offer better compression with slower runtime, especially as data files become larger (cf. cases 7-11); data file size in real applications are expected to be at least as large as the largest cases considered here. Algorithm A2.1, based on JBIG2, improves compression ratios by 29.3% on average, with a runtime penalty of 5.9×; this lossless method may be a more effective basis for compression than JBIG1. Introduction of one-sided loss does not contribute significantly to compression performance unless the allowed proportional loss ratio is large ($>$ 40%). However, the proportional loss algorithms A2.2 and A3 are still promising in some respects, and respectively improve average compression ratios by 33.6% and 31.9% versus Bzip2. Finally, the concept of *fixed speckle loss* constitutes a fixed-radius type of asymmetric cover (just as *proportional loss* constitutes a proportional-radius type of asymmetric cover) and gives

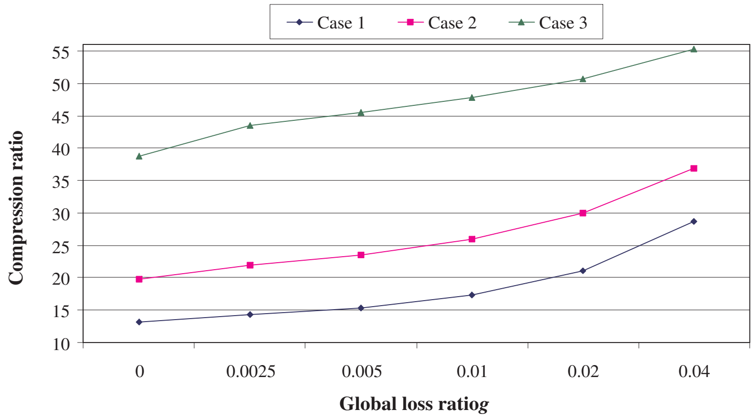

Figure 11: Compression results for fixed speckle loss heuristic A2.3 with various global loss ratios g.

exceptionally promising results, with resulting compression ratios improving by 128.6% vs. Bzip2. The fixed radius regime is reasonable in practice since feature density constraints are set by the foundry with respect to window size, and are independent of the feature area in a given design.

Our ongoing work combines compression techniques with the identification of large rectangles of contiguous fill features. Such rectangles can be removed and compressed separately: their removal improves arithmetic coding based compression by further biasing the ratio of 1s to 0's in the input. We are also investigating the generation of compressible dummy fill features. The algorithms that we discuss here may be directly connected to hierarchical VLSI layout generation by use of AREF and SREF constructs[18] to represent dummy fill features.

REFERENCES

1. G. Nanz and L. E. Camilletti, "Modeling of chemical mechanical polishing: A review," *IEEE Trans. Semiconductor. Manufacturing.*, vol. 8, no. 4, pp. 382–389, 1995.
2. "Information technology - coded representation of picture and audio information - progressive bi-level image compression," Tech. Rep. ITU-T Recommendation T.82 — ISO/IEC 11544:1993, International Telecommunications Union, 1993, (Commonly referred to as JBIG1 standard).
3. "JBIG2 final draft international standard," Tech. Rep. JTC1/SC29/WG1 N1545, ISO/IEC, December 1999.
4. P. Howard, F. Kossentini, B. Martins, S. Forchhammer, W. Rucklidge, and F. Ono, "The emerging JBIG2 standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8, no. 5, pp. 838–848, September 1998.
5. *JBIG Homepage*, http://www.jpeg.org/public/jbighomepage.htm.
6. David Salomon, *Data Compression: the complete reference*, Springer, second edition, 2000.
7. R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*, pp. 85–103. Plenum, New York, 1972.
8. P. Howard, "Lossless and lossy compression of text images by soft pattern matching," in *Proceedings of the 1996 IEEE Data Compression Conference (DCC)*, J. A. Storer and M. Cohn, Eds., March 1996, pp. 210–219, Snowbird, Utah.
9. P. Howard, "Text image compression using soft pattern matching," *Computer Journal*, vol. 40, pp. 2–3, 1997.
10. A. Moffat, T. C. Bell, and I. H.F. Witten, "Lossless compression for text and images," *Int. J. High Speed Elect. & Syst.*, vol. 8, no. 1, pp. 179–231, 1997.
11. J. N. Cooper, R. B. Ellis and A. B. Kahng, "Asymmetric binary covering codes," *J. Combin. Theory Ser. A*, vol. 100, no. 2, pp. 232-249, 2002.
12. R. Ellis, A. B. Kahng, Y. Zheng, "JBIG Compression Algorithms for Dummy Fill VLSI Layout Data," *technical report* #CS2002-0709, UCSD CSE Department, June 2002.
13. I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Morgan Kaufmann, San Mateo, CA, second edition, 1999.
14. V. Dai and A. Zakhor, "Lossless Layout Compression for Maskless Lithography Systems," *Proc. Emerging Lithographic Technologies IV*, Santa Clara, February 2000, SPIE volume 3997, pp. 467-477.
15. J. Robertson, "Vendors believe 'million-dollar reticles' can be avoided in next-generation fabs," http://www.siliconstrategies.com/story/OEG20010718S0019 .
16. F. M. Schellenberg, O. Toublan, L. Capodieci and B. Socha, "Adoption of OPC and the Impact on Design and Layout," *Proc. ACM/IEEE Design Automation Conference*, 2001, pp. 89-92.
17. *International Technology Roadmap for Semiconductors*, December 2001, http://public.itrs.net .
18. *GDSII Stream Format*, http://www.vsi.org/library/contech/gdsii.pdf.