Compressible Area Fill Synthesis

Yu Chen, Andrew B. Kahng, Gabriel Robins, *Member, IEEE*, Alexander Zelikovsky, and Yuhong Zheng

***Abstract*—Control of variability and performance in the back end of the VLSI manufacturing line has become extremely difficult with the introduction of new materials such as copper and low-k dielectrics. To improve manufacturability, and in particular to enable more uniform chemical–mechanical planarization (CMP), it is necessary to insert *area fill* features into low-density layout regions. Because area fill feature sizes are very small compared to the large empty layout areas that need to be filled, the filling process can increase the size of the resulting layout data file by an order of magnitude or more. To reduce file transfer times, and to accommodate future maskless lithography regimes, data compression becomes a significant requirement for fill synthesis. In this paper, we make the following contributions. First, we define two complementary strategies for fill data volume reduction corresponding to two different points in the design-to-manufacturing flow: *compressible filling* and *post-fill compression*. Second, we compare compressible filling methods in the fixed-dissection regime when two different sets of compression operators are used: the traditional GDSII array reference (AREF) construct, and the new Open Artwork System Interchange Standard (OASIS) repetitions. We apply greedy techniques to find practical compressible filling solutions and compare them with optimal integer linear programming solutions. Third, for the post-fill data compression problem, we propose two greedy heuristics, an exhaustive search-based method, and a smart spatial regularity search technique. We utilize an optimal bipartite matching algorithm to apply OASIS repetition operators to irregular fill patterns. Our experimental results indicate that both fill data compression methodologies can achieve significant data compression ratios, and that they outperform industry tools such as *Calibre* V8.8 from Mentor Graphics. Our experiments also highlight the advantages of the new OASIS compression operators over the GDSII AREF construct.**

***Index Terms*—Dummy fill, fill data compression, GDSII AREF, greedy method, OASIS repetitions, VLSI manufacturability.**

I. Introduction and Background

CHEMICAL–MECHANICAL planarization (CMP) and other manufacturing steps in nanometer-scale VLSI processes have varying effects on device and interconnect features, depending on the local characteristics of the layout. To improve manufacturability and performance predictability, foundry rules require that a layout be made uniform with respect to prescribed density criteria, through the insertion of *area fill features*. Currently, area fill is added by physical verification tools (such as Mentor Graphics' Calibre) in the form of a flat "target layer" [26], which is eventually merged with the actual layout features at the mask data preparation step of the manufacturing handoff. Interconnect layers above M1 have little natural hierarchy that can be exploited, and contexts for instantiations of IP blocks may be different; this typically leads to a flat filling solution. According to the 2002 International Technology Roadmap for Semiconductors [20], the fractured (MEBES format) layout data volume for a single critical layer will reach hundreds of gigabytes during the transition between 130 nm and 90 nm technologies [2]. To alleviate file transfer times, and to accommodate future regimes of maskless lithography (e.g., direct-write requires transfer of terabytes of layout data per second[1]), layout data must be compressed as much as possible (required compression factors have been estimated at 20× or more [15]).

The basic area fill feature is typically the same across the entire layout (with the most common fill shape being square or rectangular). Moreover, filling patterns exhibit a high degree of spatial regularity across the layout [22]. Area fill feature dimensions scale with the underlying technology, since microloading and other mechanisms of process variability are exacerbated by large variations in feature dimensions. Thus, the number of fill features per layer is expected to scale at approximately 2× per technology node (ignoring the impact of reticle enhancement techniques such as OPC). The filling process tends to increase the size of a GDSII file by an order of magnitude, due to the small size of the area fill features relative to the large empty layout areas that must be filled.[2] Higher data volumes lead to increased read/write times and prevent the leveraging of hierarchical data processing, among other concerns. Thus, fill data compression becomes a requirement for effective fill synthesis.

Off-the-shelf data compression techniques such as the Joint Bi-Level Image Processing Group (JBIG), Ziv-Lempel (LZ77) and ZIP cannot directly be used inside the standard GDSII stream data format, and such techniques are, therefore, of limited use in today's design-to-manufacturing flows. However, for a direct-write maskless lithography system, a data processing system architecture and three compression algorithms are compared in [15] and an interesting alternative compression is

Manuscript received April 29, 2003; revised February 6, 2004. This work was supported by a Packard Foundation Fellowship, by the MARCO Gigascale Silicon Research Center, by a National Science Foundation (NSF) Young Investigator Award MIP-9457412, by NSF Grant CCR-9988331, and by a grant from Cadence Design Systems, Inc. This paper was recommended by Associate Editor T. Yoshimura.

Y. Chen was with the Department of Computer Science, University of California, Los Angeles, CA 90095-1596 USA. He is now with Blaze-DFM, Inc., Sunnyvale, CA 94089 USA (e-mail: yuchen@blaze-dfm.com).

A. B. Kahng is with the Department of Computer Science and Engineering, and Department of Electrical and Computer Engineering, University of California at San Diego, La Jolla, CA 92093-0114 USA (e-mail: abk@ucsd.edu).

G. Robins is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903-2442 USA (e-mail: robins@cs.virginia.edu).

A. Zelikovsky is with the Department of Computer Science, Georgia State University, Atlanta, GA 30303 USA (e-mail: alexz@cs.gsu.edu).

Y. Zheng is with the Department of Computer Science, University of California at San Diego, La Jolla, CA 92093-0114 USA (e-mail: yzheng@cs.ucsd.edu).

Digital Object Identifier 10.1109/TCAD.2005.850859

[1]New proposed standards will allow mask write without decompression of data [9], [10].

[2]Advanced OPC tends to decrease area fill contribution to total layout data volume. However, fill is most often used with metal layers, where OPC is used sparingly or not at all, so the contribution of fill to data volume remains significant for these layers.

0278-0070/$20.00 © 2005 IEEE

suggested in [17]. Ueki *et al.* in [29] propose a data compaction algorithm for mask data processing in vector scan electron beam writing systems, where 'array' and 'cell' constructs are used to represent the data.

This paper discusses the application of fill compression at the following two phases of design-to-manufacturing flow: (1) right before the design is transferred to manufacturing (where fill generation is modified to produce compressible fill), and (2) after a filled design is produced (where compression is applied to the already generated fill). We give the two corresponding optimization formulations for Compressible Filling and for Post-Fill Compression, propose several algorithms for solving these problems, and compare the compression ratios achieved by the GDSII AREF constructs as well as the new Open Artwork System Interchange Standard (OASIS) repetition operators.

In the background below, we briefly summarize area fill methods in fixed-dissection regimes and analyze compression ratios achievable by the GDSII AREF constructs and OASIS repetition operators. Section II formally introduces the Compressible Filling and Post-Fill Compression problems. For the Compressible Filling problem, Section III proposes linear programming-based methods and greedy methods based on the GDSII AREF and OASIS operators. For the Post-Fill Compression problem, Section IV develops an efficient spatial regularity detection algorithm combined with an optimal bipartite matching algorithm. Section V describes our experimental results, and Section VI concludes with future research directions.

A. Fill Generation and Fixed-Dissection Regimes

Existing methods for the synthesis of area fill are all based on discretization: the layout is partitioned into *tiles*, and filling constraints or objectives (e.g., minimizing the maximum density variation) are enforced for square *windows* of size $w \times w$, each consisting of $r \times r$ tiles. The possible locations for each filling geometry are then determined within each tile, based on the corresponding design rules. The size of dummy fill features is normally prescribed by the foundries. For example, common design rules for 0.13-μm technology suggest rectangular fill features 3 μm in length. Since manufacturers suggest regular square or rectangular fill patterns, and since such fill shapes are typically generated by commercial fill synthesis tools, we assume throughout this paper that all fill features are rectangles of the same fixed size. Note that this assumption also accommodates mixed-size fill features, as long as they are arranged into "fill cells" (e.g., as with tilted fill), since these fill cells also have a rectangular shape.

Thus, to practically control layout density in *arbitrary* windows, density bounds are enforced only within a *finite* set of windows. More precisely, foundry design rules and EDA physical verification and layout tools attempt to enforce density bounds within *fixed dissections* of the layout. Each fixed dissection is a partition of the layout into nonoverlapping $w \times w$-windows. The density constraints are enforced within r^2 overlapping dissections, where r determines the "phase shift" w/r by which the dissections are offset from each other.

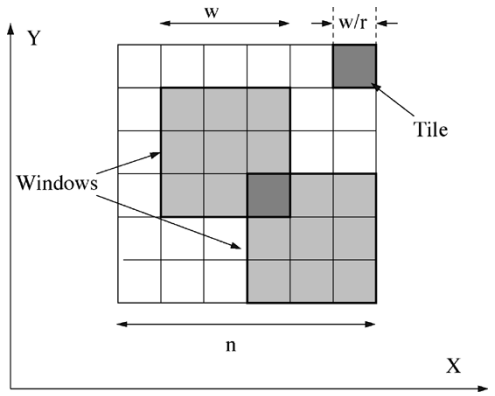

Fig. 1. Fixed r-dissection regime with parameters $r = 3$, $w = 3$ and the layout size 6×6. The layout is partitioned by $r^2 = 9$ distinct 6×6 overlapping wrap-around dissections with the phase shift $w/r = 1$. There are a total of 36 windows (each tile is a left-bottom tile of a window).

A more accurate layout density control requires a greater value of r. The resulting *fixed r-dissection* (see Fig. 1) partitions the layout into $n \times n$ tiles T_{ij}, where $n = (\text{layout_size}/w) \cdot r$. Thus, the layout is covered by $w \times w$-windows W_{ij}, $i, j = 0, \ldots, (nr/w) - 1$. Note that windows are "wrapped around" the layout, e.g., a window that overlaps with the upper edge of the layout also contains tiles on the bottom of the layout (and similarly for opposite sides of the layout). This is not only convenient for simplifying the counting, but also models the fact that the layout density at the edge of one die may affect the manufacturing of the die's immediate neighbors on the wafer. Each window W_{ij} consists of r^2 tiles T_{kl}, $k = i, \ldots, i + r - 1$, $l = j, \ldots, j + r - 1$.[3]

The fill generation problem in the fixed-dissection regime seeks a number of area fill features to be inserted into each tile. Two main filling objectives have been addressed in the recent literature:

- the *Min-Var Objective*, where the *variation* in window density (i.e., maximum window density minus minimum window density) is minimized, subject to the constraint that no window density exceeds a given upper bound U;
- the *Min-Fill Objective*, where the number of inserted area fill features is minimized, subject to the constraint that each window density remains within a given range $[L, U]$.

Methods for area fill synthesis in the fixed-dissection context include:

- Linear Programming (LP) methods based on rounding the solution to a relaxation of a corresponding integer linear program [22], [27], [28];
- Greedy and Monte Carlo (MC) methods which iteratively find a best or random tile in which the next fill feature should be added into the layout [5], [6], [28];
- Iterated Greedy (IGreedy) and Iterated Monte Carlo (IMC) methods that improve the solution quality by alternating area fill insertion and deletion phases, while optimizing the density variation [6].

[3]In typical 0.13-μm design rules, the window size is 200 μm, the tile size (or step size) is 100 μm, and the dummy fill feature size is 3 μm. For example, in a 2 cm $\times$ 2 cm layout, there are 200 $\times$ 200 overlapping windows that need to be checked and 6666 $\times$ 6666 possible dummy fill feature positions.

TABLE I
SYNTAX OF THE EIGHT OASIS REPETITION TYPES. THE PARAMETERS x-DIMENSION, y-DIMENSION, x-SPACE, y-SPACE, DIMENSION, n-DIMENSION, AND m-DIMENSION ARE ALL INTEGERS, WHILE DISPLACEMENT, n-DISPLACEMENT, AND m-DISPLACEMENT ARE ALL g-DELTAS. M AND N ARE THE NUMBERS OF ROWS AND COLUMNS IN THE REPETITION

TYPE	FORMAT	DESCRIPTION	# of integers
1	**x-dimension y-dimension [x-space] [y-space]**	$M \times N$ ($N > 0, M > 0$) matrix with uniform orthogonal spacing between elements	4
2	**x-dimension x-space**	$1 \times N$ ($N > 1$) vector with uniform orthogonal spacing between elements	2
3	**y-dimension y-space**	$M \times 1$ ($M > 1$) vector with uniform orthogonal spacing between elements	2
4	**x-dimension x-space$_1$...x-space$_{N-1}$**	$1 \times N$ ($N > 1$) vector with non-uniform orthogonal spacing between elements	N
5	**y-dimension y-space$_1$...y-space$_{M-1}$**	$M \times 1$ ($M > 1$) vector with non-uniform orthogonal spacing between elements	M
6	**n-dimension m-dimension [n-displacement] [m-displacement]**	$N \times M$ ($N > 0, M > 0$) repetition with uniform and (possibly) non-orthogonal displacement between the elements	4, 5, or 6
7	**dimension displacement**	a P-element ($P > 1$) repetition with uniform and (potentially) diagonal displacements between the elements.	2 or 3
8	**dimension displacement$_1$ ... displacement$_{P-1}$**	P-element ($P > 1$) repetition with (possibly) non-uniform and (possibly) non-orthogonal displacements between the elements	$2 \cdot P - 1$

B. Compression Operators in GDSII and OASIS Formats

A new, recently proposed standard layout data representation format called the *Open Artwork System Interchange Standard* (OASIS) has as a primary objective the enabling of significant data compression. This new standard is expected to be simple enough to be widely supported by all EDA suppliers and CAD groups with prior knowledge of GDSII. OASIS is capable of representing the commonly used GDSII data types, which will enable a smooth migration from GDSII to the new and more efficient OASIS format. The key OASIS concepts most relevant to our work are as follows.

- An OASIS *unsigned integer* is an N-byte integer value, where $N > 0$, and is implementation-dependent. The integer byte length is variable, and integers are represented as byte continuations, where the upper bit of each byte except the last in the chain is set; the remaining seven bits in each byte are concatenated to form the actual integer value.
- An OASIS *g-delta construct* represents a general displacement, with the following two forms:
  - — An unsigned integer, representing an orthogonal or 45° displacement.
  - — A pair of unsigned integers, which represents a general (x, y) displacement.
- OASIS supports primitive geometric shapes such as *rectangles, polygons, paths, trapezoids, circles*, and *X-geometries*.
- A *placement* record identifies an instantiation of a *cell*. A cell may in turn contain a number of basic structures, including other cells.
- A *repetition* is an array of either placements, geometries or texts. There are eight *repetition types* (see Table I), as illustrated (for a standard fill feature) in Fig. 2. We also define the *starting point* of the repetition to be the left bottom point of the left bottom element of the repetition.

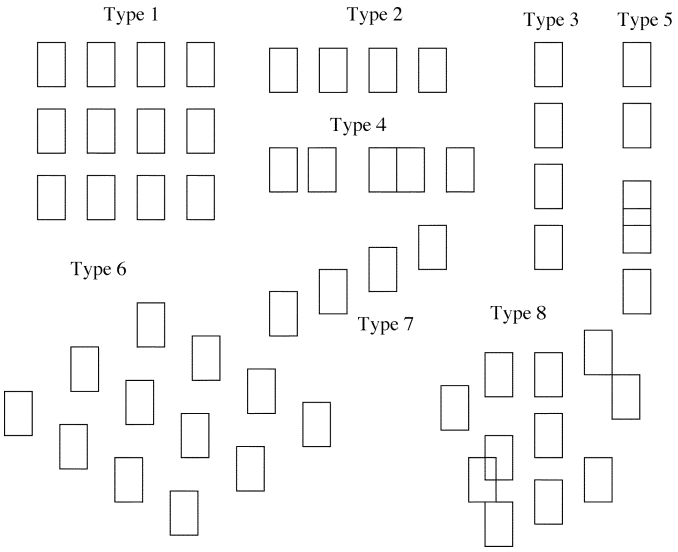

Fig. 2. Examples of the eight types of OASIS repetitions. These are more formally defined in Table I.

The *full OASIS format* allows the use of repetition Types 1–7 (Type 8 repetition is subsumed by the other repetition operators, and thus has no significant effect on single-level data compression). The GDSII AREF construct can represent only the first three OASIS repetition types. We, therefore, define the *restricted OASIS format* as one that may use only repetition Types 1–3. To compare the compression capabilities of different operators, we define the following measure of compression efficacy.

Definition 1: The compression ratio R_c is the ratio of the size of a flat OASIS file (S_{orig}) to the size of its compressed version ($S_{\text{compressed}}$)

$$R_c = \frac{S_{\text{orig}}}{S_{\text{compressed}}}. \quad (1)$$

We now estimate the data size requirements of the different repetition types, which are all based on *unsigned integers* and the *g-delta* constructs. Table I summarizes the different types of repetitions and provides estimates of the number of unsigned integers required to represent each type. Here, Type 6 repetitions may have different sizes, depending on the displacement direction: if one or both of the directions are multiples of orthogonal or 45° displacements, then their size may be either 4, 5, or 6 integers. Similarly, the size of a Type 7 repetition may also vary between 2 and 3 integers, depending on the displacement direction being either 90° or 45°.

Consider an $M \times N$ array of *rectangles*, and let A be the number of integers required to represent a single record without any repetitions. Let R be the number of integers required to store the additional information when using repetitions. Then, the total number of integers required to store $M \times N$ independent rectangles would be $M \cdot N \cdot A$, while the total number of integers required to store these rectangles as an array with repetition would be $A + R$. The estimated *compression ratio* is, therefore, given by

$$R_c = \frac{(M \cdot N \cdot A)}{(A + R)}. \quad (2)$$

Since R is always smaller than $(M \cdot N - 1) \cdot A$, there will be a reduction in data volume when using repetitions.

An area fill feature may be a *rectangle*, a *polygon*, a *trapezoid*, a *circle*, or other shape, and OASIS can define different records to represent these shapes. The following is an example of estimating the size of a rectangle when used as a fill feature:

$$[\text{Record}-\text{header}][\text{layer}-\text{number}][\text{datatype}-\text{number}] \times [\text{width}][\text{height}][x][y][\text{repetition}].$$

The size "A" of a single rectangle record is seven integers, with these integers representing the record header, layer number, datatype number, x, y, width, and height. Different fill feature types may have different record sizes, e.g., a polygon requires seven integers, a circle needs six, and a trapezoid requires nine integers. Assuming that representing a single fill feature requires seven integers, Table II gives the estimated compression ratio for each repetition type.

We conclude from Table II that repetition Types 1 and 6 are the most powerful compression operators, followed by Types 2, 3, and 7, while repetition Types 4 and 5 are the least powerful compression operators. Again, Type 8 is subsumed by the other repetition operators and thus has no significant effect on single-level data compression; however, it may still be useful in hierarchical cases, similar to the SREF operator in GDSII.

TABLE II
COMPRESSION RATIOS FOR THE DIFFERENT TYPES OF RECTANGLE REPETITIONS

Type	Compression Ratio R_c	Asymptotic behavior R_c
1	$7MN/11$	MN
2	$7N/9$	N
3	$7M/9$	M
4	$7N/(7+N)$	7
5	$7M/(7+M)$	7
6	$7MN/11 \sim 7MN/13$	MN
7	$7P/9 \sim 7P/10$	P
8	$7P/(6+2P)$	3.5

II. Problem Formulations

Fill compression can be applied either during fill generation, or after the fill for a design has been produced. In the first case, area fill is generated using the GDSII AREF construct or OASIS repetition operators directly to satisfy the layout density requirements.

Fig. 3 illustrates the potentially significant reduction in data volume when compressible fill is generated. Assume that given the layout shown, the fill synthesis phase prescribed the insertion of 84 fill features into the layout (i.e., we now need to succinctly represent 84 fill rectangles). If we use the GDSII AREF construct or the OASIS repetition operators, only nine AREFs or OASIS repetition operators are needed (denoted by dark rectangles in Fig. 3), resulting in a significant data volume reduction. Significant data volume reduction can also result from using the GDSII AREF construct or the OASIS repetition operators to encode already generated fill. This is illustrated in Fig. 4, where, e.g., only seven GDSII AREF constructs or OASIS repetition operators are necessary to represent 61 existing fill features.

Below, we give the two corresponding optimization formulations for Compressible Filling and Post-Fill Compression Problems.

A. Compressible Filling Problem

One approach to reducing the fill data volume is to use the GDSII AREF or OASIS repetition constructs for fill pattern compression. This can be formulated as follows.

Compressible Filling Problem (CFP): Given a design rule-correct layout, generate a minimum number of GDSII AREF or OASIS operators to represent area fill features that keep window density variation within the given bounds (L, U).

We first address the CFP formulation in the fixed-dissection regime. In Section I we have described several methods for computing the number of area fill features that should be inserted into each tile. Let F_{ij} be the required number of area fill features to be inserted into tile T_{ij}. We represent each tile as an array of sites corresponding to the possible positions for inserting area fill features. Some sites are forbidden with respect to fill insertion, since they are already occupied by existing layout features.

Compressible Filling Problem in Fixed-Dissection Regimes: Given a design rule-correct layout consisting of $m \times n$ tiles T_{ij}, and fill requirements F_{ij} for each tile, generate a minimum number of GDSII AREF or OASIS operators to represent area fill, such that each tile T_{ij} contains exactly F_{ij} area fill features.

If we wish to address the Min-Var and Min-Fill filling problems together, we can obtain a *feasible range*, instead of an exact

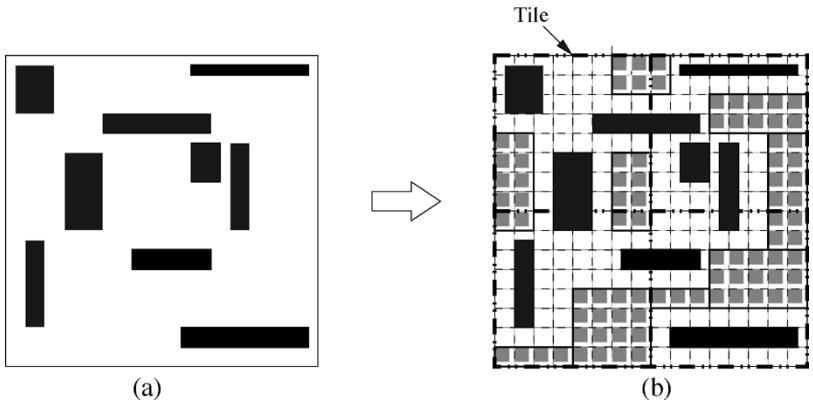

Fig. 3. Example of compressible filling. (a) Original layout. (b) Filled layout with 84 area features in nine AREFs.

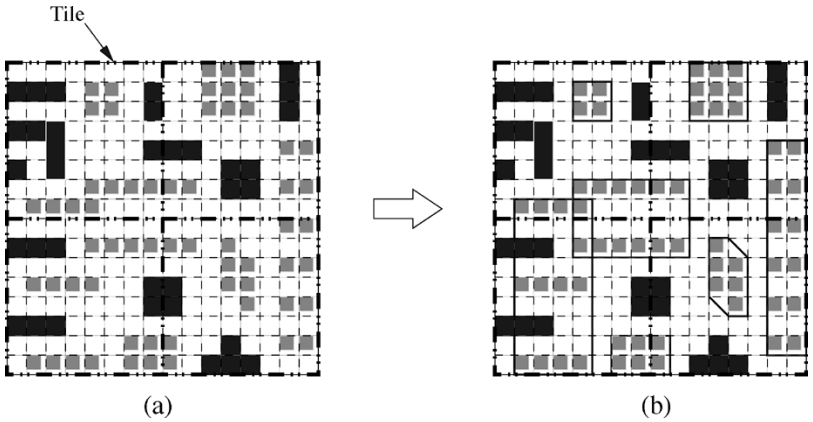

Fig. 4. Example of post-fill compression. (a) Layout divided into four tiles with 61 inserted area fill features. (b) Compressed layout with 61 area fill features using seven OASIS repetitions.

value F_{ij}, for the number of area fill features to be inserted into each tile while satisfying the density requirements. The corresponding optimization formulation is as follows.

Ranged Compressible Filling Problem in Fixed-Dissection Regimes: Given a design rule-correct layout consisting of $m \times n$ tiles, generate a minimum number of GDSII AREF or OASIS operators to represent area fill, such that each tile T_{ij} contains a number of area fill features within the given range (LB_{ij}, UB_{ij}).[4]

B. Post-Fill Compression Problem

An alternative stage in the design-to-manufacturing flow where the fill data volume can be reduced is after the fill for the entire layout has been generated. Then, the GDSII AREF construct or OASIS repetition operators are applied without changing the filled design.

Post-Fill Compression Problem (PFCP): Given a layout containing area fill features, represent these area fill features using the GDSII AREF construct or OASIS repetition operators in a way that minimizes the overall data volume.

In order to detect compressible fill patterns, we represent the fixed-dissection layout region as a binary $m \times n$ matrix $M(= [m_{ij}], 1 \leq i \leq m, 1 \leq j \leq n)$, where 1's correspond to area fill features, and 0's denote empty areas or original features. The intersection of a row (i) and a column (j) in the matrix M is denoted by m_{ij}. Each nonzero element of the input matrix corresponds to a basic fill feature. A 0-1 matrix representation of fill layout is possible for the outputs of all major commercial fill insertion tools (e.g., Mentor's Calibre, Synopsys' Hercules and Cadence's Assura), even when operating in modes that output "tilted fill" or tiled "fill cells". In Section IV we describe a greedy method for solving PFCP based on exhaustive search of all repetitions, and then we give a computationally efficient greedy algorithm based on searching for regularities in the planar pointset.

[4]The given range (LB_{ij}, UB_{ij}) is the feasible range for the number of area fill features for each tile. It is determined by the fill generation process and does not depend on the neighboring bounds.

III. Approaches to Compressible Filling

A. Approaches Based on GDSII Compression Operators

The Integer Linear Programming (ILP) method for inserting area fill is the most accurate (see the Appendix), but cannot be applied to large layouts. Therefore, we propose greedy heuristics to determine the minimum number of AREFs for the required number of area fill features in each tile. We describe a strict greedy (albeit inefficient) algorithm for multiple-tile ranged fill generation using either overlapping AREFs or nonoverlapping AREFs; we then describe two faster practical variants. Single-tile and fixed fill are special cases of multiple-tile and ranged fill, respectively.

Strict Greedy Approach:

1. Get site set G of $M \times N$ multiple-tile consisting of tiles T_{ij}, $i = 1 \ldots M$, $j = 1 \ldots N$;
2. **For** each valid *AREF* A_α in the multiple-tile **Do**
   Initialize S_α (number of unfilled free sites in A_α), and
   $S_{\alpha,ij}$ (number of unfilled free sites in A_α in each tile (i, j));
3. Pick the *AREF* A^*_α for which $S^*_\alpha = max\{S_\alpha \mid S_{\alpha,ij} \leq U_{ij}\}$;
4. Update the fill requirements: $U_{ij} = U_{ij} - S^*_{\alpha,ij}$, $L_{ij} = L_{ij} - S^*_{\alpha,ij}$;
5. Update G, S_α and $S_{\alpha,ij}$ of each *AREF*;
6. **If** $L_{ij} \leq 0$ in each single tile (i, j) **Then** exit **Else** go to Step 3.

Fig. 5. Strict Greedy Algorithm for multiple-tile ranged filling.

Greedy Speedup Approach 1:

1. Get site set G of $M \times N$ multiple-tile consisting of tiles T_{ij}, $i = 1 \ldots M$, $j = 1 \ldots N$;
2. **For** the largest *AREF* A_α originating from each original free site **Do**
   Initialize S_α (number of unfilled free sites in A_α), and
   $S_{\alpha,ij}$ (number of unfilled free sites in A_α in each tile (i, j));
3. Pick *AREF* A^*_α that fills maximum number of unfilled free sites, where $S^*_\alpha = max\{S_\alpha \mid S_{\alpha,ij} \leq U_{ij}\}$ **If** exists;
   Else select *AREF* A^*_α where $S^*_\alpha = max\{S_\alpha\}$, and pick its sub-*AREF* $A^*_{\alpha_sub}$ by doubling x or y until $S^*_{\alpha,ij} \leq U_{ij}$;
4. Update the fill requirements: $U_{ij} = U_{ij} - S^*_{\alpha,ij}$, $L_{ij} = L_{ij} - S^*_{\alpha,ij}$;
5. Update G, S_α and $S_{\alpha,ij}$ of each largest *AREF*;
6. **If** $L_{ij} \leq 0$ in every single-tile (i, j) **Then** exit **Else** go to Step 3.

Fig. 6. Greedy Speedup Approach 1.

Strict Greedy Approach: The *strict greedy approach* repeatedly adds an AREF that fills the maximum number of unfilled free sites in multiple tiles, yet does not overfill any tile. It iterates until the filling requirements in all the tiles are satisfied. In the ranged filling context, the fill requirements in each tile are changed from a fixed F_{ij} to a certain range (L_{ij}, U_{ij}), where U_{ij} is used to control overfill, and L_{ij} is used to satisfy the minimum fill requirements.

Since each AREF contains different combinations of coordinates, sizes, and steps, checking all the AREFs has a time complexity of $O(n^3)$, where n is the total number of sites in the layout. In the strict greedy approach, the status of all the valid AREFs is checked and updated during each iteration, resulting in an overall time complexity[5] of $O(n^4)$. Our implementation of this algorithm provides good solutions, but may be impractical due to its high running time. Letting $L_{ij} = U_{ij} = F_{ij}$ in Fig. 5 solves the fixed fill problem, and using a single-tile as a multiple-tile solves the single-tile filling problem.

Greedy Speedup Approach 1: We propose speedups of the basic greedy approach that offer tradeoffs between compression performance and run time (see Fig. 6). Our first greedy speedup heuristic finds the largest AREFs originating from each free site, and picks an AREF that fills the maximum number of unfilled free sites without overfilling any tiles (if such an AREF exists). Otherwise, it selects the maximum AREF from all the largest AREFs, and finds one of its sub-AREFs which does not overfill the tiles. This process is iterated until all of the tile filling requirements are satisfied. Note that the solution is generated from all largest AREFs (or their feasible sub-AREFs) starting from all originally free sites (i.e., sites which were free in the original layout) rather than from all valid AREFs, as in the Strict Greedy algorithm.

For this *Greedy Speedup Approach 1*, the sets of largest AREFs originating from the original free sites are different for the single-tile option and for the multiple-tile option; due to runtime considerations, our heuristics (Step 3) will not necessarily choose the best sub-AREF. Rather, a sub-AREF is selected by doubling the step sizes (x and y) of the original AREFs, i.e., using $2 \times x$ and $2 \times y$ as the steps for the required sub-AREF. Thus, we cannot guarantee better behavior with the multiple-tile option than with the single-tile option. For example, for test case $T2$ and $s = 500$ or 250 in Table IV, the results of Greedy Speedup Approach 1 with the multiple-tile option are worse than those obtained with the single-tile option, in terms of both the number of AREFs and the run times. Finding the largest AREF originating from each original free site has a time complexity of $O(n^3)$. Selecting the maximum AREF (or its sub-AREF within the set of the largest AREFs) requires $O(n^2)$ time. The time complexity of the algorithm is, therefore, improved to $O(n^3)$.

Greedy Speedup Approach 2: An even more efficient approach can be realized by picking *acceptable AREFs* originating from each free site, instead of *maximum* AREFs (see Fig. 7). An acceptable AREF is an AREF that fills the maximum number of unfilled free sites but does not overfill the tiles among all the AREFs originating from the same free site whose sizes are

[5]A faster implementation of the Strict Greedy Approach maintains a priority queue with invalid candidates being permanently removed.

Greedy Speedup Approach 2:

1. Get site set G of $M \times N$ multiple-tile consisting of tiles T_{ij}, $i = 1 \ldots M$, $j = 1 \ldots N$;
2. **For** each free site in G in the scan order **Do**
3. Calculate S_α (number of unfilled free sites in A_α) of the *AREF*s originating from each free site in G, where $S_{\alpha,ij} \le U_{ij}$, $w \le Kx$, and $h \le Ly$ ($S_{\alpha,ij}$: number of unfilled free sites in A_α in each tile (i, j))
4. Pick the *AREF* A^*_α, for which $S^*_\alpha = \max\{S_\alpha\}$;
5. Update the fill requirements: $U_{ij} = U_{ij} - S^*_{\alpha,ij}$, $L_{ij} = L_{ij} - S^*_{\alpha,ij}$;
6. Update G;
7. If $L_{ij} \le 0$ in every single-tile (i, j) then Stop;

Fig. 7. Greedy Speedup Approach 2.

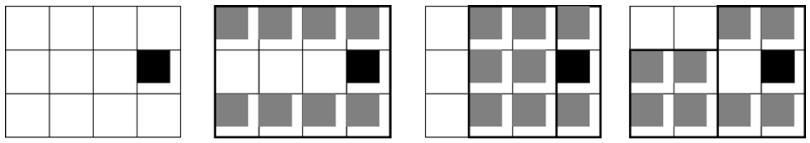

Fig. 8. Comparison of compressible filling using different approaches (Fill requirement $= 8$). (a) Original tile, the dark solid square shows the occupied original site. (b) Strict Greedy Approach. (c) Greedy Speedup Approach 1. (d) Greedy Speedup Approach 2 ($K = 2, L = 2$).

smaller than $K \cdot L$ (here K and L define a prescribed upper bound on the size of AREFs used in compression). Our second greedy speedup heuristic repeatedly adds an acceptable AREF originating from each free site and iterates until the tile filling requirements are satisfied. Finding an acceptable AREF originating from a free site (i.e., an AREF using a free site as its starting (reference) point), requires $O(K \cdot L \cdot n)$ time. The time complexity of the algorithm, therefore, improves to $O(K \cdot L \cdot n^2)$. Moreover, this algorithm is very efficient on actual benchmarks, where $K \cdot L \ll n$ and AREFs of size $K \cdot L$ yield adequate compression. Intuitively, larger AREFs have starting sites at the bottom left corner of the tile; therefore, our Greedy Speedup Approach 2 scans free sites from the bottom left corner and considers the largest AREFs starting from such sites.

Fig. 8 illustrates the difference between the strict greedy algorithm, the Greedy Speedup 1 algorithm and the Greedy Speedup 2 algorithm, using an example of inserting eight fills into a tile. In Fig. 8(a), the dark solid square shows the occupied original site. The strict greedy approach searches all the valid AREFs and one AREF is picked to fill eight free sites as shown in Fig. 8(b). In the Greedy Speedup 1 approach, only the largest AREFs originating from each original free site are checked. Since the largest AREF originating from the bottom-left free site contains nine free sites and the fill requirement is eight, it will not be picked in the first round and two AREFs are selected as shown in Fig. 8(c). In the Greedy Speedup 2 approach, AREFs whose sizes are no larger than 2×2 are searched. Fig. 8(d) shows the two selected AREFs.

B. Approaches Based on OASIS Operators

The *greedy algorithm* described above for compressible filling based on the GDSII AREF construct repeatedly fills the maximum number of unfilled free sites in multiple tiles, yet does not overfill any tile. It iterates until the filling requirements in all of the tiles are satisfied. The implementation in this section replaces the greedy objective, which maximizes coverage of unfilled free sites with one which instead maximizes the compression ratio. Compared to the GDSII AREF construct, the OASIS repetition operators have an additional degree of freedom. Therefore, checking the status of all the valid repetitions and updating them at each iteration results in a time complexity of $O(n^5)$.

To speed up this greedy algorithm, our implementation sacrifices strict greediness, i.e., instead of finding a repetition with the largest compression ratio, we find a maximal repetition with sufficiently large R_c (i.e., larger than the given lower bound L_{R_c}. The constant L_{R_c} is an experimentally derived lower bound on the compression ratio for acceptance of repetitions of Type {1, 2, 3, 6, 7}. All our algorithms use $L_{R_c} = 5.0$). The time complexity thus improves to $O(K \cdot L \cdot n^3)$, where $K \cdot L$ upper bounds the number of elements in any repetition. In Fig. 9, S_α is the number of unfilled free sites covered by A_α; letting $L_{ij} = U_{ij} = F_{ij}$ will solve the fixed fill problem, and using a single-tile instead of a multiple-tile will solve the single-tile filling problem.

IV. Approaches to Post-Fill Compression

This section will discuss approaches for the post-fill compression problem in the fixed-dissection regime. We can apply the greedy method from the previous section to the post-fill compression where free sites correspond to the fill features. The key problem is to rapidly find a repetition with the largest, or sufficiently large, compression ratio. The resulting algorithms utilize the following priority scheme to search for repetitions.

1) Find a repetition of Type 1, 2, 3, 6, or 7 with maximum compression ratio R_c (this priority in identifying compression operators is based on the compression ratio analysis from Table I).

Greedy Approach for CFP based on OASIS operators

1. Input site set G of an $M \times N$ multiple-tile (i, j), $i = 1 \ldots M$, $j = 1 \ldots N$;
2. **For** each free site in G in the scan order **Do**
3. Find a fill repetition A^*_α among Type {1, 2, 3, 6, 7} that has the maximum R_c but does not overfill tiles;
4. **If** $R_c > L_{R_c}$ (given lower bound of R_c)
5. Update the fill requirements: $U_{ij} = U_{ij} - S^*_{\alpha,ij}$, $L_{ij} = L_{ij} - S^*_{\alpha,ij}$;
6. Update G, S_α and $S_{\alpha,ij}$ of each repetition;
7. **While** ($L_{ij} > 0$) **Do**
8. Find a fill repetition A^*_α among Type {4, 5} that has the maximum R_c but does not overfill tiles;
9. Update the fill requirements: $U_{ij} = U_{ij} - S^*_{\alpha,ij}$, $L_{ij} = L_{ij} - S^*_{\alpha,ij}$;
10. Update G, S_α and $S_{\alpha,ij}$ of each repetition;

Fig. 9. Greedy Algorithm using OASIS operators.

Exhaustive Search-Based Single-level Greedy Algorithm for PFCP

1. Input fill matrix B;
2. **Do**
3. Find next unmarked and unvisited site in B in scan order;
4. Search exhaustively for maximum repetitions of Types {1, 2, 3, 6, 7} originating from the site;
5. Pick the repetition type having the maximum R_c assuming that $R_c > L_{R_c}$ (given lower bound on R_c);
6. Mark all sites covered by the repetition;
7. **Until** all fill sites are marked or used as starting points;
8. Use bipartite matching to find a minimum number of repetition Types {4, 5} covering remaining unvisited sites;

Fig. 10. Exhaustive Search-Based Single-level Greedy (ESBG) approach for fill data compression.

2) If $R_c > L_{R_c}$, output the repetition and update the fill data. The constant L_{R_c} is an experimentally derived lower bound on the compression ratio for acceptance of repetitions of Type 1, 2, 3, 6, or 7. Our algorithms all use $L_{R_c} = 5.0$.
3) Repeat 1 and 2 until no repetition exists with $R_c > L_{R_c}$.
4) Find a minimum number of Type 4 or 5 repetitions to cover the remaining fill geometries using bipartite matching.

Step 4 of the algorithm above can be implemented optimally, since each repetition of Type 4 (respectively, Type 5) covers an entire row (respectively, column). The problem of covering all the remaining points with the minimum number of repetitions of Types 4 and 5 is, therefore, equivalent to the well-known problem of finding a minimum vertex cover in a bipartite graph $H = (V = R \cup C, E)$ (where vertices correspond to the set R of rows and the set C of columns, and edges connect columns and rows if and only if there is a 1 at their intersection in the 0-1 matrix). Applying Konig's Theorem, a minimum vertex cover in a bipartite graph H can be derived from a maximum matching, which can be found within time $O(N \cdot \sqrt{m+n})$ where N is the number of 1's in the matrix [13].

A. Exhaustive Search-Based Single-Level Greedy Method (ESBG)

A straightforward but inefficient way to represent fill features with the GDSII AREF construct or OASIS repetition operators is to perform exhaustive search for each repetition type. In our Exhaustive Search-Based Single-Level Greedy Method (ESBG), each 1-element of the input 0-1 matrix is treated as the bottom-left corner of a potential maximal parallelogram or rectangle (the latter being a special case of the former) with all the corners of the parallelogram containing 1's (Types 1, 2, 3, 6, and 7). If a repetition type originating from a 1-element with maximum compression ratio satisfies the given minimum compression ratio requirements, it is then saved and all of the sites covered by this repetition are marked as *visited*. This process is repeated, each time using the remaining unvisited 1-elements in the matrix as potential starting points, until all of the 1-elements have been either included in repetitions or tried as starting points. Finally, any remaining 1-elements that have not been marked as visited are covered by repetition Types 4 and 5 using a bipartite perfect matching-based minimum vertex cover method, described at the end of Section IV-B. Since type 1, 2, 3, 6, and 7 repetitions involve different combinations of coordinates, sizes, steps and directions, the time complexity of the algorithm is $O(N^4)$. Fig. 10 gives a formal description of our algorithm.

B. Regularity Search-Based Single-Level Greedy Algorithm (RSBG)

The exhaustive search-based greedy method described above may be computationally inefficient for large layouts. We can improve on this approach by using a regularity detection technique for planar pointsets [21]. Fig. 11 illustrates the difference between the ESBG and the RSBG methods, using an example with fill features at sites 1, 3, 4, 5, 6, and 7. Using the ESBG approach, the originating site for repetition types is selected along the scan order. The ESBG algorithm first chooses a repetition

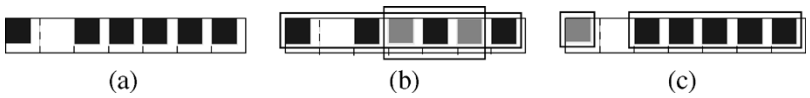

Fig. 11. Comparison of ESBG and RSBG algorithms. (a) Original fill data with fill features at sites 1, 3, 4, 5, 6, and 7. (b) OASIS repetitions extracted using the ESBG algorithm. (c) OASIS repetitions extracted using the RSBG algorithm.

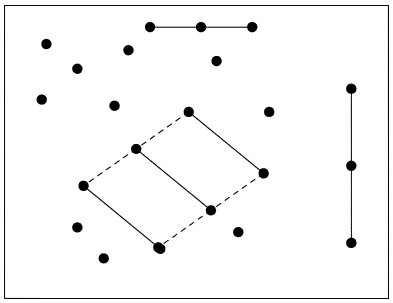

Fig. 12. Finding regularities in a pointset.

with fill features at sites 1, 3, 5, and 7, and then covers the fill features at sites 4 and 6 with another repetition. However, the RSBG method chooses a repetition with fill features at sites 3, 4, 5, 6, and 7, and then covers the fill feature at site 1.

1) Finding Regularities in a Pointset: The work of [21] addresses two problems for a given set of N points (Fig. 12):

- finding subsets of equally spaced collinear points (i.e., points that lie on the same line are regularly spaced along that line);
- finding subsets of regularly spaced parallelogram cells.

The following method for solving the first problem (see Fig. 13) is based on finding all maximal arithmetic progressions in a set of numbers. First, we obtain the set of all equally spaced number triples. Next, we use these triples to construct a *Regularity Graph*, where each equally spaced triple (p_a, p_b, p_c) induces two vertices (a, b) and (b, c) and one edge $((a, b), (b, c))$. Finally, the connected components of this graph correspond to all maximal equally spaced collinear subsets in the original pointset. The total time required for this computation is $O(N^2)$.

This technique generalizes to solve the second problem as follows (see Fig. 14). Each vertex of the regularity graph now represents a parallelogram, with edges in the graph representing adjacencies among neighboring parallelograms. The maximum degree of each vertex in this graph is four, and the size of this graph is bounded by $O(N^3)$, where N is the number of points. Similarly to the first problem, the connected components of this regularity graph correspond to maximal regularly spaced subsets of points in the original configuration. Fig. 14 summarizes these techniques [21].

2) Greedy Method: Our proposed *greedy single-level* fill compression algorithm uses 1-elements of the input matrix to represent points in the plane, and 0-elements of the input matrix to denote intermediate spaces. Three levels of granularity among the elements are considered during the search in order to find maximal repetition types:

- *Points*;
- *Line Segments* formed by pairs of points;
- *Parallelograms* formed by sets of four distinct points.

Our heuristic finds a group of congruent adjacent cells[6] and selects repetition types which contain maximal numbers of unvisited points located among cells of this group. The proposed greedy method (Fig. 15) starts with the set of points determined by the input 0-1 matrix B, and forms all possible segments over point pairs. Next, these segments are sorted in nondecreasing order of their length, as projected onto the axes. Parallelograms are then formed from pairs of segments having the same length and slope; this is accomplished by scanning the sorted segment list in a left-right and bottom-up order. Next, a graph is constructed where each graph vertex corresponds to a cell, and edges correspond to geometric adjacencies between neighboring cells (i.e., congruent cells that share a common edge). For each connected component of this graph, we search for maximal repetitions of Types 1, 2, 3, 6, and 7, as detailed in Steps (14)–(20) of the algorithm in Fig. 15. This method for identifying maximal repetitions extends the method of [21].

The largest of all maximal repetitions is thus identified, and the corresponding points are marked as visited. The same process is repeated for the remaining set of points, until no feasible new repetitions nor unvisited points remain. Note that the resulting repetitions may overlap with each other, and thus some points may be covered by more than one repetition (i.e., some fill elements may be represented redundantly). Finally, Step (21) of Fig. 15 *optimally* covers the remaining points with repetitions of Types 4 and 5.

The overall time complexity of this algorithm is $O(N^3)$, which may still be prohibitive for millions of fill elements. We may, therefore, partition the layout into k blocks to speed up the runtime, with the time complexity of this modified variant being $O(N^3/k^2)$. Note that since the minimum vertex cover can be found much faster, it is possible to employ a coarser partition into blocks (or forgo altogether the partition into blocks) in Step (19). Alternatively, we can initially run an optimal algorithm for finding a minimum covering with repetitions of Types 4 and 5. Then, each candidate repetition may be selected only if it reduces the size of the minimum covering.

V. Computational Experience

All of our experiments were performed on metal layers extracted from industry standard-cell layouts (Table III). Our experimental testbed integrates GDSII Stream input and internally developed geometric processing engines, coded in C++ under Solaris 2.8. We use CPLEX version 7.0 [14] as the linear programming solver. All runtimes are reported in CPU seconds on a 300 MHz Sun Ultra-10 with 1 GB of RAM.

[6]A cell is defined by a pair of distinct segments having the same length and slope. The offset between two segments S_i and S_j is defined by horizontal width $w(S_i, S_j)$ and vertical height $h(S_i, S_j)$.

Finding subsets of equally spaced collinear points:

1. Sort the input pointset by its x-coordinates in time $O(N \log N)$;
   This yields a sorted list of points $p_1, p_2, \ldots, p_N$;
2. Let A be a pointer to the leftmost point p_1 in the sorted list;
3. Let B and C be two other pointers (to p_2 and p_3, resp.) in the sorted list;
4. **If** $x(p_B) - x(p_A) = x(p_C) - x(p_B)$, record the *basic cell* consisting of the equally-spaced triple (p_A, p_B, p_C);
5. **If** $x(p_B) - x(p_A) \leq x(p_C) - x(p_B)$, increment B, i.e., point to the next p in the sorted list;
   Else increment C;
6. Repeat Steps 3 – 5 until $B \geq N$;
7. Increment A;
8. Repeat Steps 3 – 7 until $A \geq N - 1$.

Fig. 13. Finding subsets of equally spaced collinear points.

Finding All Regularly Spaced Parallelogram Cells in a Planar Pointset:

1. A set of n points defines $\binom{N}{2}$ line segments.
2. Sort line segments in increasing order of the following tuple:
   (length, slope, and x-coordinate of left endpoint).
3. Match equal-length segment pairs to form parallelogram cells.
4. Form maximal contiguous sets of congruent adjacent cells to obtain all maximal regularly-spaced patterns.

Fig. 14. Finding all maximal sets of regularly spaced parallelogram-shaped cells in a pointset.

A. Compressible Filling Results

We first give compressible filling results using the GDSII AREF operator. Table IV compares the data volumes of uncompressed fill results against compressed fill results (due to the infeasibility of running the LP solver on large test cases, only compression results from the Greedy Speedup 1 and Greedy Speedup 2 approaches are reported here). The data volume increase due to area fill insertion becomes significant when the fill feature sizes are small (e.g., more than 100 MB for the single layer $T2$ when $s = 250$). The Greedy approach can achieve very large compression ratios for the resulting GDSII files, especially when the fill features are small. For example, in test case $T3$, the fill data volumes are reduced by about 30× (from 73.8 MB to 2.5 MB) when the fill feature size is $s = 250$; the compression ratio is only 1.5× for $s = 1500$. Furthermore, the runtimes of the Greedy approaches make them feasible in practice. The tradeoff between runtime and solution quality is also apparent from Table IV by comparing the single-tile results with the multiple-tile results.

The average compression ratio of the Greedy Speedup 1 algorithm is 10.6 for the ranged single-tile fill and 8.7 for the ranged multiple-tile fill. The average compression ratio of the Greedy Speedup 2 algorithm is 7.1 for the ranged single-tile fill versus 8.2 for the ranged multiple-tile fill. The average runtime of the Greedy Speedup 1 algorithm is 1929 CPU seconds for the ranged single-tile fill and 4749 CPU seconds for the ranged multiple-tile fill. The average runtime of the Greedy Speedup 2 algorithm is 12 s for the ranged single-tile fill versus 20 s for the ranged multiple-tile fill. We can achieve a smaller number of AREFs by using the Multiple-Tile approach, although that would require longer runtimes. As expected, the Greedy Speedup 2 approach is much faster than the Greedy Speedup 1 approach, with only a small degradation in solution quality.

Table V compares the ILP method, the Greedy Speedup 1 method, and the Greedy Speedup 2 method. Since the runtimes of ILP-based methods make them infeasible for multiple-tile fill compression, we only report results for single-tile fill compression. We observe improvements in terms of both the number of AREFs as well as the runtimes for the ranged fill-compression approaches. Our experiments also indicate that the Greedy method yields results comparable to the optimal ILP method, yet offers a significant runtime advantage (i.e., a decrease from several hours to less than one second).

We assess the OASIS-based compressible filling approaches by inserting low density fill as well as high density fill into the layout.[7] Table VI reports the compressible filling results. The average compression ratio of low density compressible filling is 6.8 using the restricted OASIS repetitions, 10.4 using the full OASIS repetitions, 10.6 using Type 1, 2, 3, 4, and 5 repetitions, and 6.6 using Type 4 and 5 repetitions. The average compression ratio of the high density compressible filling is 4.6 using the restricted OASIS repetitions, 8.5 using the full OASIS repetitions, 8.7 using Type 1, 2, 3, 4, and 5 repetitions, and 6.7 using Type 4 and 5 repetitions.

The results indicate that the compression ratios using the full OASIS format are on average twice those using the restricted OASIS Format. This confirms an advantage of the OASIS compression operators over those of GDSII.[8] Table VI also shows the performance of different types of repetition combinations.

[7]The high density fill was produced by fill synthesis with the Min-Var objective, while the low density fill was produced by fill synthesis with the Min-Fill objective.

[8]Full OASIS compression will work well on specific styles of fill, such as "tilted fill" using repetition Types 6 and 7.

Regularity Search-Based Single-level Greedy Algorithm for PFCP

Input: binary matrix

Output: compressed OASIS file

1. Add segments between all pairs of points into segment array R by scanning the input binary matrix;
2. Sort all segments in R by its y-distance then x-distance using bucket sort; Build adjacency list G (i.e., list of congruent adjacent cells);
3. **For** each subset Q in R where each segment has the same x and y **Do**
4. $G = (V, E) = (\emptyset, \emptyset)$;
5. **For** $k_1 = 1 : s(Q) - 2$ **Do**
6. $k_2 = k_1 + 1$; $k_3 = k_1 + 2$;
7. **While** $k_3 \leq s(Q)$ **Do**
8. $(A, B, C) = (k_1, k_2, k_3)$;
9. **If** $w(A,B) = w(B,C)$ and $h(A,B) = h(B,C)$
10. $V = V \cup \{(A,B),(B,C)\}$;
$E = E \cup \{((A,B),(B,C))\}$;
11. **Else If** $h(A,B) > h(B,C)$ or $w(A,B) > w(B,C)$,
$k_3 = k_3 + 1$;
12. **Else** $k_2 = k_2 + 1$;
13. Sort all vertices in the adjacency list G by its w and h using bucket sort;
// Find repetitions in G
14. **For** each subset P of G where cells have same w and h **Do**
15. **For** each vertex $v \in P$ **Do**
16. Search the repetition Types 1, 2, 3, 6 and 7 originating from v;
17. Save into the set S the repetition with the maximum compression ratio bigger than L_{R_c} (based on the number of unvisited points);
18. Select from S a repetition with the maximum compression ratio, and mark the points contained in the selected repetition as visited;
19. Update compression ratios of the repetitions in S, remove the picked repetition and the repetitions whose compression ratios are smaller than L_{R_c};
20. Go to Step 18 until $S = \emptyset$;
21. Use a bipartite matching algorithm to find the minimum number of instances of repetition Types 4 and 5 that cover the remaining unvisited points;
22. Output all saved repetitions.

Fig. 15. Regularity Search-Based Single-level Greedy method for fill data compression.

TABLE III
PARAMETERS OF SEVEN INDUSTRY TEST CASES

Testcase	T1	T2	T3	T4	T5	T6	T7
layout size	819,200	819,200	819,200	125,000	112,000	112,000	522,060
# rectangles	32,258	142,585	78,293	49,506	76,423	133,201	133,872

Since the sizes of repetition Types 1, 2, 3, 6, and 7 are independent of the number of 1's covered, and the sizes of repetition Types 4 and 5 are dependent on the number of 1's, using repetition Types 4 and 5 achieves better compression results if the fill requirements are small. Otherwise, if the fill requirements are large, using Types 1, 2, 3, 6, and 7 repetitions will achieve better compression results. We also find that full OASIS compression is slightly worse than using only Types 1, 2, 3, 4 and 5. The reason is that using Types 6 and 7 repetitions may break large instances of Types 1, 2, and 3, yet may fail to decrease the number of Type 1, 2, and 3 repetitions since these are all independent of the number of 1's.

Finally, Table VII compares the Greedy Speedup 2 algorithm with the off-the-shelf data compression tool GZIP. GZIP has

TABLE IV
DATA COMPRESSION. **NOTATION**: *RANGED SINGLE-TILE*: RANGED SINGLE-TILE FILL COMPRESSION APPROACHES; *RANGED MULTIPLE-TILE*: RANGED MULTIPLE-TILE FILL COMPRESSION APPROACHES; *UNCOMP*: FILL SOLUTION WITHOUT COMPRESSION; *GS-1* GREEDY SPEEDUP APPROACH 1; *GS-2* GREEDY SPEEDUP APPROACH 2; $T/W/r$: LAYOUT/WINDOW SIZE/R-DISSECTION; s: SITE SIZE; *DATA*: FILE SIZE INCREASE IN KILOBYTES DUE TO FILL FEATURES (REDUCTION FACTOR RELATIVE TO *UNCOMP*); *CPU*: RUNTIME (IN SECONDS)

			Ranged Single-Tile				Ranged Multiple-Tile			
Testcase		**Uncomp**	**GS-2**		**GS-1**		**GS-2**		**GS-1**	
T/W/r/	s	Data(K)	Data	CPU	Data	CPU	Data	CPU	Data	CPU
T1/80K/4	1500	1538	1036 (1.48 ×)	0.23	1016 (1.51 ×)	3.07	1004 (1.53 ×)	0.26	1001 (1.54 ×)	8.87
T1/80K/4	1000	2303	1043 (2.20 ×)	0.60	1013 (2.27 ×)	17.10	1010 (2.28 ×)	0.73	993 (2.32 ×)	57.14
T1/80K/4	500	6156	1062 (5.80 ×)	3.27	1016 (6.06 ×)	357.36	1035 (5.95 ×)	4.47	1008 (6.10 ×)	1329.84
T1/80K/4	250	22256	1099 (20.25 ×)	20.75	1016 (21.90 ×)	11976.27	1061 (20.98 ×)	31.64	1027 (21.67 ×)	29081.66
T2/80K/4	1500	1672	1242 (1.35 ×)	0.20	1188 (1.41 ×)	1.56	1135 (1.47 ×)	0.22	1161 (1.44 ×)	4.63
T2/80K/4	1000	4846	2105 (2.30 ×)	0.75	1777 (2.73 ×)	8.52	1949 (2.49 ×)	1.06	1776 (2.73 ×)	30.64
T2/80K/4	500	27405	5015 (5.46 ×)	7.59	3671 (7.49 ×)	178.50	4731 (5.79 ×)	14.67	3925 (6.98 ×)	693.64
T2/80K/4	250	106664	7527 (14.17 ×)	64.17	3071 (34.73 ×)	5525.80	7460 (14.30 ×)	133.02	4251 (25.09 ×)	14166.58
T3/80K/4	1500	1448	993 (1.46 ×)	0.16	978 (1.48 ×)	1.07	960 (1.51 ×)	0.10	962 (1.51 ×)	2.99
T3/80K/4	1000	2867	1142 (2.51 ×)	0.48	1073 (2.67 ×)	7.06	1031 (2.78 ×)	0.50	1068 (2.68 ×)	21.83
T3/80K/4	500	18842	2719 (6.93 ×)	5.44	2583 (7.29 ×)	156.53	1946 (9.68 ×)	7.11	2957 (6.37 ×)	523.87
T3/80K/4	250	73825	3477 (21.23 ×)	39.68	1968 (37.51 ×)	4917.56	2512 (29.39 ×)	50.04	2888 (25.56 ×)	11066.06
Average			(7.1 ×)	12	(10.6 ×)	1929	(8.2 ×)	20	(8.7 ×)	4749

TABLE V
PERFORMANCE OF THE FILL COMPRESSION METHODS. **NOTATION**: *FIXED FILL-COMPRESSION*: FIXED FILL COMPRESSION APPROACHES (WHERE THE NUMBER OF FILL FEATURES IN EACH TILE IS FIXED); *RANGED FILL-COMPRESSION*: RANGED FILL COMPRESSION APPROACHES (WHERE THE NUMBER OF FILL FEATURES IN EACH TILE IS RANGED); *ILP*: ILP-BASED APPROACH; *GS-1*: GREEDY SPEEDUP APPROACH 1; *GS-2*: GREEDY SPEEDUP APPROACH 2. *#AREF*: NUMBER OF AREFS IN THE FILL SOLUTION; *CPU*: RUNTIME (IN SECONDS)

		Fixed Fill-Compression						Ranged Fill-Compression					
Testcase		**ILP**		**GS-1**		**GS-2**		**ILP**		**GS-1**		**GS-2**	
T/W/r	s	#AREF	CPU	#AREF	CPU	#AREF	CPU	#AREF	CPU	#AREF	CPU	#AREF	CPU
T1/80K/4	1500	1187	11918	1307	2.04	1548	0.22	1151	7931	1159	2.01	1323	0.21
T2/80K/4	1500	1210	6932	1269	0.99	1429	0.10	1167	1154	1193	0.95	1371	0.09
T3/80K/4	1500	464	4502	456	0.62	552	0.09	381	759	384	0.58	483	0.10
T4/12K/4	200	850	30922	891	1.63	1317	0.15	787	4215	796	1.58	1272	0.16
T5/12K/4	200	3787	23002	4043	4.61	6712	0.50	3694	18126	3927	4.49	6403	0.47
T6/12K/4	200	1249	13813	1320	1.60	1727	0.15	1230	11466	1304	1.58	1666	0.17
Average		1458	15182	1548	1.92	2214	0.20	1402	7275	1461	1.87	2086	0.20

been applied to the GDSII file containing only the generated fill. The average compression ratio is 5.51 using GZIP, 8.39 using the Greedy Speedup 2 method, and 50.57 using the Greedy Speedup 2 plus GZIP method. The average runtime is 13.32 s using GZIP, 2.46 s using the Greedy Speedup 2 method, and 3.79 s using the Greedy Speedup 2 approach combined with the GZIP method. The data indicates that the compression ratio of our Greedy Speedup 2 algorithm is on average significantly larger than that of GZIP. Moreover, the Greedy Speedup 2 approach is also considerably faster than GZIP. The last two columns of Table VII show that the compression ratios of Greedy Speedup 2 and GZIP are independent of each other, i.e., the compression ratio of the Greedy Speedup 2 followed by GZIP is about equal to the product of the compression ratios of the two standalone methods. This implies that neither GZIP nor our own methods can replace each other, but rather they complement each other to produce larger compression ratios than either is able to produce by itself.

B. Post-Fill Compression Results

For the fill data compression problem, we implemented the greedy algorithms based on regularity detection (RSBG, Fig. 15), and the exhaustive search (ESBG, Fig. 10). To compare fill compression results for fill data from different filling methods, we used four fill generation methods to insert fill features into the same industry test cases: Iterated Monte Carlo (IMC) as described in [6], the Mentor Graphics Calibre tool,

TABLE VI
COMPRESSION RATIOS OF LOW/HIGH DENSITY COMPRESSIBLE FILL. **NOTATION**: *I*: RESTRICTED OASIS; *II*: FULL OASIS; *III*: USING TYPE 1, 2, 3, 4, AND 5 REPETITIONS; *IV*: USING TYPE 4 AND 5 REPETITIONS

Testcase		**Low Density Fill**				**High Density Fill**			
T/W/r/	s	**I**	**II**	**III**	**IV**	**I**	**II**	**III**	**IV**
T2/32K/2	1500	6.9	11.0	11.0	6.4	3.4	8.5	8.6	7.4
T2/32K/2	1000	7.2	11.8	11.9	6.7	4.0	8.4	8.6	6.8
T2/32K/2	500	13.0	16.7	16.5	6.9	7.6	10.8	11.4	6.9
T2/32K/4	1500	2.6	6.1	6.1	6.0	2.7	7.3	7.3	6.5
T2/32K/4	1000	3.4	7.4	7.5	6.7	3.5	7.8	8.0	6.8
T2/32K/4	500	8.0	11.3	11.5	7.0	6.7	10.3	10.5	6.9
T7/32K/2	1500	7.5	9.8	10.3	6.5	3.4	6.7	6.8	5.6
T7/32K/2	1000	6.2	10.2	10.5	6.5	3.7	7.6	7.8	6.3
T7/32K/2	500	11.0	14.4	15.1	6.9	6.3	9.2	9.7	6.8
T7/32K/4	1500	3.7	7.2	7.3	6.2	3.2	7.3	7.3	6.3
T7/32K/4	1000	6.2	9.6	9.8	6.9	5.6	9.0	9.3	6.8
T7/32K/4	500	6.2	9.6	9.8	6.9	5.6	9.1	9.3	6.8
Average		6.8	10.4	10.6	6.6	4.6	8.5	8.7	6.7

TABLE VII
COMPARISON OF COMPRESSION RATIOS AND RUNTIMES OF FILL SYNTHESIS, GZIP, THE GREEDY SPEEDUP 2 AND THE GREEDY SPEEDUP 2 FOLLOWED BY GZIP

Testcase		**Fill Synthesis[3]**	**GZIP**		**Greedy Speedup 2**		**Greedy Speedup 2+GZIP**	
T/W/r	s	CPU(sec)	comp_ratio	CPU(sec)	comp_ratio	CPU(sec)	comp_ratio	CPU(sec)
T1/80k/4	1500	34.07	5.24	8.63	12.90	5.04	71.59	5.64
T2/80k/4	1500	40.44	5.44	12.36	4.04	5.62	25.94	7.57
T3/80k/4	1500	36.11	5.89	10.73	10.48	0.39	61.07	1.16
T4/12k/4	200	13.61	5.51	10.48	7.99	1.29	49.75	2.34
T5/12k/4	200	31.80	5.44	31.92	6.49	0.54	42.91	3.53
T6/12k/4	200	18.13	5.55	5.78	8.44	1.86	52.18	2.49
Average		29.03	5.51	13.32	8.39	2.46	50.57	3.79

the Cadence Assura tool, and our compressible filling method using either the *restricted OASIS* or *full OASIS* formats.

Table VIII and Table IX present fill compression results for low and high density fill data using the ESBG and RSBG methods. The full OASIS format results are on average 1.4× smaller than the restricted OASIS format results using the ESBG method; the compression ratio gain is about 2× on average when using the RSBG method. The average compression ratio of the exhaustive search-based single-level greedy algorithm is 5.0 using the restricted OASIS repetitions and 6.8 using the full OASIS repetitions. The average compression ratio of the regularity search-based single-level greedy algorithm is 3.9 using the restricted OASIS repetitions and 7.7 using the full OASIS repetitions. The average runtime of the exhaustive search-based single-level greedy algorithm is 75 s using the restricted OASIS repetitions and 18 790 s using the full OASIS repetitions. The average runtime of the regularity search-based single-level greedy algorithm is 536 s using the restricted OASIS repetitions and 3988 s using the full OASIS repetitions. The separate averages of compression ratio and runtime for low and high density fill data are listed in Table VIII and Table IX.

Comparing the ESBG and RSBG methods, our experiments show that using the ESBG algorithm achieves 29.5% average improvement in compression ratio with the restricted OASIS format, and runs about 7.2× faster than the RSBG algorithm. However, the RSBG algorithm achieves 13.4% average improvement in compression ratio for the full OASIS format, and runs about 4.7× faster than the ESBG algorithm. This is due to the fact that ESBG searches a larger space than RSBG: repetitions of Types 1, 2, and 3 found by the ESBG algorithm tend to be larger than those found by the RSBG algorithm, and thus ESBG uses fewer repetitions with respect to the restricted OASIS format. Under full OASIS, more repetitions of Types 6 and 7 are found using ESBG than using RSBG; this will not reduce the number of Type 1, 2, and 3 instances, and so the ESBG algorithm will tend to generate more repetition types under the full OASIS format.

TABLE VIII
FILL COMPRESSION RATIOS FOR LOW DENSITY FILL DATA. **NOTATION**: *CPU*: RUNTIME (IN SECONDS)

Low Density Fill data generated by Iterated Monte Carlo									
Testcase		**Exhaustive Search**		**Regularity Search**		**Exhaustive Search**		**Regularity Search**	
T/W/r/	s	**Types 1, 2, 3**	**CPU**	**Types 1, 2, 3**	**CPU**	**All Oasis Types**	**CPU**	**All Oasis Types**	**CPU**
T2/32K/2	1500	3.6	1	3.3	16	5.5	342	6.0	3
T2/32K/2	1000	3.6	4	3.3	106	6.2	3134	6.9	30
T2/32K/2	500	4.4	35	3.5	1078	6.5	23945	7.8	664
T2/32K/4	1500	2.7	1	2.6	27	5.4	1188	6.0	3
T2/32K/4	1000	3.3	5	2.8	180	5.6	6292	6.9	71
T2/32K/4	500	4.7	91	3.8	2151	6.1	71189	7.9	5393
T7/32K/2	1500	4.6	1	4.1	3	6.0	116	7.2	6
T7/32K/2	1000	4.1	2	3.8	56	6.7	810	7.7	171
T7/32K/2	500	4.7	42	3.6	816	7.1	9694	8.2	2383
T7/32K/4	1500	3.0	1	2.6	4	4.9	305	5.5	1
T7/32K/4	1000	3.1	1	2.9	79	5.5	1836	6.3	15
T7/32K/4	500	4.5	35	3.7	968	6.1	22389	7.7	3673
Low Density Fill data generated by Compressible Filling									
Testcase		**Exhaustive Search**		**Regularity Search**		**Exhaustive Search**		**Regularity Search**	
T/W/r/	s	**Types 1, 2, 3**	**CPU**	**Types 1, 2, 3**	**CPU**	**All Oasis Types**	**CPU**	**All Oasis Types**	**CPU**
T2/32K/2	1500	5.4	0.3	5.1	11	7.6	459	7.8	6
T2/32K/2	1000	5.9	2	4.8	96	8.0	1335	8.2	223
T2/32K/2	500	9.2	110	4.6	934	9.6	6106	8.6	2048
T2/32K/4	1500	3.0	1	3.0	30	5.5	1095	6.2	6
T2/32K/4	1000	3.8	5	3.4	210	5.9	6170	7.3	132
T2/32K/4	500	6.8	516	5.1	1507	7.0	27915	9.3	8401
T7/32K/2	1500	6.5	0.3	6.5	4	7.9	149	9.5	13
T7/32K/2	1000	6.0	3	4.5	57	8.6	842	8.3	247
T7/32K/2	500	9.2	45	4.6	730	9.7	4377	8.6	2968
T7/32K/4	1500	3.3	0.3	3.3	4	5.7	223	6.3	1
T7/32K/4	1000	3.9	1	3.6	37	6.0	1841	6.8	33
T7/32K/4	500	6.1	39	4.6	956	7.1	13463	8.5	2811
Average		4.8	39	3.9	419	6.7	8550	7.5	1221

On occasion, the ESBG method using full OASIS types is a little worse than the ESBG method with restricted OASIS types. In the exhaustive search-based greedy approach, we determine the originating site for repetition types according to the scan order. This nonoptimal order makes it possible for a repetition Type 6 or 7 chosen for one site, to cover some sites from where larger repetitions can originate. In other words, Type 6 and 7 repetitions may break large instances of Types 1, 2, and 3, yet may fail to decrease the number of Type 1, 2, and 3 repetitions. As a result, in some cases the compression ratio from a search with full OASIS types can be worse than that from a search with restricted OASIS types (i.e., 1, 2, 3).

Regarding the relative compressibility of fill from different filling methods, we expect that compressible fill should enable better compressibility than IMC-generated fill. We can directly compare compression of low and high density fill data generated using the two different approaches. Since high density fill covers almost all free sites in the layout, compressible filling in a high-density scenario does not offer much advantage, as expected. However, low density fill produced by the compressible filling method always yields better compression results than fill generated by randomized methods such as IMC (Table VIII and IX).

Table X reports fill compression results for fill data generated by the Mentor Graphics Calibre tool (V8.8) and the Cadence Assura tool (V2.0). Our method yields better compression than Calibre does on its own output data. When analyzing the number of different repetition types used in fill compression, we have observed that repetitions of Types 6 and 7 are not ideal choices, unless there are many Type 6 and 7 patterns in the layout. We further observe that repetitions of Type 4 and 5 clearly provide full OASIS with additional compression capability as compared to restricted OASIS.

C. Perfect Matching versus Greedy Approach

After extracting repetition Types 1, 2, 3, 6, and 7, we applied both the perfect matching-based minimum vertex cover method,

TABLE IX
FILL COMPRESSION RATIOS FOR HIGH DENSITY FILL DATA. **NOTATION**: *CPU*: RUNTIME (IN SECONDS)

High Density Fill data generated by Iterated Monte Carlo									
Testcase		**Exhaustive Search**		**Regularity Search**		**Exhaustive Search**		**Regularity Search**	
T/W/r/	s	**Types 1, 2, 3**	**CPU**	**Types 1, 2, 3**	**CPU**	**All Oasis Types**	**CPU**	**All Oasis Types**	**CPU**
T2/32K/2	1500	4.0	6	3.3	107	7.1	5137	8.0	1246
T2/32K/2	1000	4.9	54	3.4	344	6.8	23621	7.8	5719
T2/32K/2	500	7.8	540	4.6	2118	7.7	146729	9.1	22586
T2/32K/4	1500	3.7	4	3.2	57	6.8	5762	7.4	188
T2/32K/4	1000	4.9	51	3.3	331	6.8	17820	7.7	5003
T2/32K/4	500	7.8	536	4.8	6145	7.7	127891	9.4	296
T7/32K/2	1500	3.8	1	3.7	19	6.5	1479	7.0	99
T7/32K/2	1000	4.4	4	3.7	83	6.6	3918	7.6	366
T7/32K/2	500	6.7	51	4.4	583	7.1	38416	8.4	3099
T7/32K/4	1500	3.3	1	3.0	10	5.7	1284	5.8	11
T7/32K/4	1000	4.4	3	3.7	83	6.6	3211	7.6	581
T7/32K/4	500	6.7	51	4.4	583	7.1	20468	8.4	3720
High Density Fill data generated by Compressible Filling									
Testcase		**Exhaustive Search**		**Regularity Search**		**Exhaustive Search**		**Regularity Search**	
T/W/r/	s	**Types 1, 2, 3**	**CPU**	**Types 1, 2, 3**	**CPU**	**All Oasis Types**	**CPU**	**All Oasis Types**	**CPU**
T2/32K/2	1500	4.0	2	3.6	139	7.5	4184	8.5	626
T2/32K/2	1000	4.9	47	3.3	576	6.8	7607	7.8	5464
T2/32K/2	500	7.9	572	4.5	973	7.8	117108	9.0	36297
T2/32K/4	1500	3.5	2	3.3	102	6.4	3607	7.5	309
T2/32K/4	1000	4.8	42	3.6	768	6.8	12278	8.0	7008
T2/32K/4	500	7.3	516	4.8	1307	7.7	112567	9.3	42196
T7/32K/2	1500	3.8	1	3.8	19	6.4	759	7.0	255
T7/32K/2	1000	4.4	5	3.7	82	6.7	2948	7.5	285
T7/32K/2	500	6.8	72	4.4	577	7.1	18074	8.4	13167
T7/32K/4	1500	3.3	1	3.3	11	5.7	782	6.3	51
T7/32K/4	1000	4.4	5	3.7	82	6.6	2877	7.6	283
T7/32K/4	500	6.7	72	4.4	577	7.1	18179	8.4	13292
Average		5.2	110	3.8	653	6.9	29029	7.9	6756

as well as the greedy algorithm for finding repetition instances of Types 4 and 5. Table XI lists the number of instances of Types 4 and 5 used by these two methods. The minimum vertex cover method wins 3.8% over greedy in ESBG, and 0.7% over greedy in RSBG.

VI. CONCLUSION

In this paper, we introduced two compression strategies to reduce the data volume due to area fill synthesis for layout density control. First, we explored *compressible filling* strategies for fixed-dissection regimes which exploit the GDSII array reference record (AREF) construct and the new OASIS repetition operators. We applied greedy and linear programming-based optimization techniques, and obtained practical compressed fill solutions.

The experiments show that for the compressible fill problem, our Greedy Speedup 2 method yields results comparable to the optimal ILP method, while offering a significant runtime improvement. Second, we proposed a post-fill data compression method based on spatial regularity detection and optimal bipartite matching. Our experimental results help quantify the prospective advantages of the OASIS compression operators, and our compression algorithms outperform leading industry tools. For the fill data compression problem, our smart spatial regularity search technique yields substantial compression ratio improvements and runs much faster than the exhaustive search technique when using the full OASIS format. Finally, our experimental results illustrate the superiority of the OASIS compression operators over the corresponding GDSII operators; additional improvements can be expected when OASIS is used hierarchically.

Based on our experimental results, two interesting potential modifications to the OASIS *repetition* operator may enhance future compression effectiveness.

- A new *irregular array* construct equivalent to the combination of Type 4 and 5 repetitions may be added to pro-

TABLE X
FILL DATA COMPRESSION RESULTS FOR FILL DATA GENERATED BY CALIBRE AND ASSURA. **NOTATION**: *TI/T/F*: TOOL/TESTCASE/FILE, TOOL IS MENTOR GRAPHIC CALIBRE (MGC) OR CADENCE ASSURA (ASSURA)

Test	Compression Ratio	Exhaustive Search		Regularity Search	
TI/T/F	R_c	**Types 1, 2, 3 only**	**All Oasis Types**	**Types 1, 2, 3 only**	**All Oasis Types**
MGC/T1/1	1.64	6.58	7.28	3.69	8.03
MGC/T2/1	2.30	8.07	8.60	4.45	8.60
MGC/T2/2	1.86	6.09	8.05	4.18	8.15
MGC/T2/3	2.02	5.52	7.83	4.89	7.83
MGC/T2/4	2.19	4.86	6.02	3.41	8.00
AssuraT2	N/A	3.63	5.51	2.46	5.81
Average	2.00	5.79	7.22	3.85	7.74

TABLE XI
NUMBERS OF TYPE 4 AND 5 REPETITIONS, i.e., THE IRREGULAR ROWS AND COLUMNS, RESULTING FROM USING PERFECT MATCHING VS. A GREEDY APPROACH FOR COMPRESSING HIGH DENSITY FILL. **NOTATION**: $T/W/r$: TESTCASE/WINDOW SIZE/r-DISSECTION; *MVC*: PERFECT MATCHING METHOD FOR MINIMUM VERTEX COVER; *GREEDY*: THE GREEDY APPROACH

Low Density Fill

Testcase		Iterated Monte Carlo				Compressible Fill Generation			
		Exhaustive Search		Regularity Search		Exhaustive Search		Regularity Search	
T/W/r/	s	MVC	Greedy	MVC	Greedy	MVC	Greedy	MVC	Greedy
T2/32K/2	1500	267	272	269	277	133	134	157	159
T2/32K/2	1000	626	633	627	654	537	577	600	626
T2/32K/2	500	1262	1303	1278	1322	1189	1310	1274	1293
T2/32K/4	1500	373	378	373	383	375	387	383	384
T2/32K/4	1000	639	641	639	641	632	633	638	638
T2/32K/4	500	1242	1324	1278	1290	1196	1260	1279	1279
T7/32K/2	1500	90	90	90	90	51	53	53	53
T7/32K/2	1000	358	365	364	369	259	261	295	300
T7/32K/2	500	928	945	955	970	783	796	897	909
T7/32K/4	1500	89	89	89	89	70	70	69	70
T7/32K/4	1000	463	468	464	469	401	402	408	410
T7/32K/4	500	977	1025	993	1034	965	989	976	996

High Density Fill

Testcase		Iterated Monte Carlo				Compressible Fill Generation			
		Exhaustive Search		Regularity Search		Exhaustive Search		Regularity Search	
T/W/r/	s	MVC	Greedy	MVC	Greedy	MVC	Greedy	MVC	Greedy
T2/32K/2	1500	372	375	400	402	412	414	413	416
T2/32K/2	1000	594	596	631	631	590	591	631	631
T2/32K/2	500	1113	1263	1273	1275	1123	1259	1279	1279
T2/32K/4	1500	365	366	373	374	403	403	413	413
T2/32K/4	1000	584	585	631	631	591	592	634	634
T2/32K/4	500	1125	1289	1273	1273	1289	1300	1272	1272
T7/32K/2	1500	292	298	293	297	289	296	299	302
T7/32K/2	1000	483	517	495	515	485	523	496	516
T7/32K/2	500	911	926	993	995	931	960	994	995
T7/32K/4	1500	273	281	273	281	280	283	278	280
T7/32K/4	1000	484	515	495	517	485	534	495	518
T7/32K/4	500	924	946	995	997	924	950	994	996

vide effective compression. More formally, this new repetition type is an $M \times N$ array with possibly nonuniform spacing between elements along the x- and y-directions. The two motivations for adding this new "Type 9" repetition to OASIS are: 1) enabling a potentially unbounded compression ratio $O(MNA/(M+N+A))$, possibly higher than that of existing types 4, 5, and 8 repetitions, and 2) enabling less uniform, yet highly compressible fill. By varying the x and y spacings, it is possible to achieve fill distributions closer to that generated by efficient Monte Carlo methods [4].

- Including a standard pseudorandom number generator in the OASIS format will substantially simplify the application of Monte Carlo methods for CMP layout density control. When generating compressible fill (see [7]), large amounts of feasible sites can be described using a small number of repetitions, and a built-in pseudorandom number generator can thus be used to reproducibly and completely specify the filling of a prescribed number of sites.

Appendix
Linear Programming-Based Methods for Compressible Filling

ILP approaches for the Fill Compression problem seek to minimize the number of AREFs for the given number of area fill features, while obeying constraints which prescribe the exact number of area fill features to be inserted into each tile. We use the following definitions:

- $S_{ij,pq} \equiv$ site in position (p, q) in a tile, where the tile itself is in position (i, j) in the overall layout. Every empty site is a possible position where an area fill feature may be inserted.
- $A_\alpha \equiv$ feasible AREF in the layout, where α consists of the following eight parameters: starting site coordinate (ij, pq), width w, height h, horizontal step x, and vertical step y (see Fig. 16).

Single-Tile Integer Linear Program

To insert exactly the prescribed number of area fill features into each tile, a straightforward method for fill compression is to consider the problem independently in each tile. We call this approach *single-tile compression*. For each tile T_{ij} consisting of $k \times l$ sites, we define the variables

$$s_{pq} \equiv \begin{cases} 1, & S_{ij,pq} \text{ is covered by some AREF} \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

$$a_\alpha \equiv \begin{cases} 1, & \text{AREF } A_\alpha \text{ is chosen} \\ 0, & \text{otherwise} \end{cases}. \tag{4}$$

We then seek the minimum number of AREFs in the slack sites of the tile T_{ij}. The total number of slack sites covered by these AREFs must be equal to the prescribed number of area fill features. The corresponding ILP is as follows.

Minimize :

$$\sum_{\text{all feasible AREFs}} a_\alpha \tag{5}$$

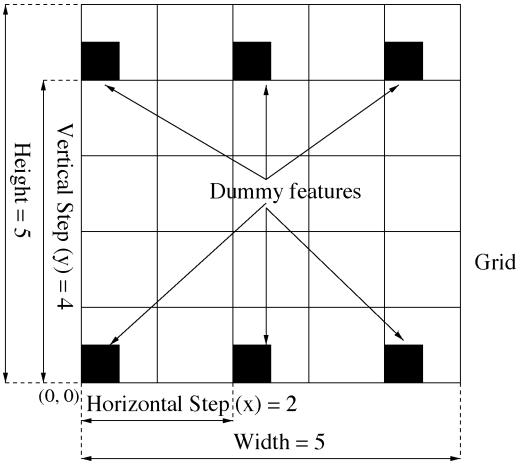

Fig. 16. Illustration of AREF $a_{i,j;0,0;5,5;2,4}$. in tile (i, j).

Subject to :

$$F_{ij} = \sum_{p=0}^{k-1} \sum_{q=0}^{l-1} s_{pq} \tag{6}$$

$$s_{pq} \geq a_\alpha \quad \text{if } s_{pq} \text{ is covered by } a_\alpha \qquad p = 0, \ldots, k-1, \quad q = 0, \ldots, l-1 \tag{7}$$

$$s_{pq} \leq \sum_{\text{all AREFs covering } s_{pq}} a_\alpha \qquad p = 0, \ldots, k-1, \quad q = 0, \ldots, l-1 \tag{8}$$

$$s_{pq} = 0 \quad \text{if } S_{ij,pq} \text{ is occupied by original features} \qquad p = 0, \ldots, k-1, \quad q = 0, \ldots, l-1 \tag{9}$$

- Constraints (6) imply that the total number of covered slack sites is equal to the number of area fill features.
- Constraints (7) imply that once an AREF is chosen, all the sites it covers will be filled.
- Constraints (8) imply that if no AREF covering site $S_{ij,pq}$ is chosen, then the site $S_{ij,pq}$ cannot be filled.
- Constraints (9) imply that we cannot fill any site $S_{ij,pq}$ which is already covered by an original layout feature.

Constraints (7) construct an inequality for each site covered by each AREF. However, we can replace these with the following set of inequalities, which significantly decreases the overall number of constraints:

$$n_{pq} \cdot s_{pq} \geq \sum_{\text{all AREFs covering } s_{pq}} a_\alpha \qquad p = 0, \ldots, k-1, \quad q = 0, \ldots, l-1 \tag{10}$$

where n_{pq} is the number of AREFs which cover site $S_{ij,pq}$ in tile T_{ij}, and $\sum a_\alpha$ is the sum of all AREF variables which cover the site $S_{ij,pq}$. To decrease the ILP problem size, we determine all *valid* AREFs and then construct the ILP formulations based only on these valid AREFs. We call an AREF *valid* if all the sites in it are empty and its indexes have reasonable physical meaning (e.g., $a_{i,j;0,0;2,1;0,0}$ in tile (i, j) is invalid since there is no AREF with width $= 2$ and horizontal step 0). Constraints (9) are necessary only when using the feasible AREFs. When using only valid AREFs, constraints (9) are not required.

Multiple-Tile Integer Linear Program

Ideally, we should seek AREFs for fill features with respect to the entire layout, rather than for each tile independently. However, the large number of tiles and possible AREFs across the entire layout make such a global strategy intractable. Instead, we propose a multiple-tile compression approach, which offers a tradeoff between solution quality and runtime, as follows:

- partition the layout into groups consisting of $A \times B$ tiles; and
- solve each group separately.

Thus, instead of finding nonoverlapping AREFs in one tile, we seek AREFs for the empty spaces within an $A \times B$ array of contiguous tiles. The difference between the Multiple-Tile ILP formulation and the Single-Tile ILP formulation is that the prescribed numbers of fill features for each tile must be *simultaneously* achieved. That is, we replace the constraints (6) with

$$F_{ij} = \sum_{p'=i\cdot k}^{(i+1)\cdot k-1} \sum_{q'=j\cdot B}^{(j+1)\cdot l-1} s_{p'q'} \qquad i = 0, \ldots, A-1, \quad j = 1, \ldots, B-1. \quad (11)$$

Here, constraints (11) imply that the total number of covered slack sites in each tile is equal to its prescribed number of fill features.

Ranged Compressible Filling

As noted in [6], excess fill features can be deleted without affecting the density variation to meet the Min-Fill objective. In other words, we can exploit the allowed range of fill features for each tile to relax the LP constraints in order to decrease the LP solver's runtime. The constraints (6) and (11) can thus be respectively rewritten as

$$LB_{ij} \le \sum_{p=0}^{k-1} \sum_{q=0}^{l-1} s_{pq} \le UB_{ij} \quad (12)$$

$$LB_{ij} \le \sum_{p'=i\cdot k}^{(i+1)\cdot k-1} \sum_{q'=j\cdot B}^{(j+1)\cdot l-1} s_{p'q'} \le UB_{ij} \qquad i = 0, \ldots, A-1, \quad j = 1, \ldots, B-1. \quad (13)$$

Here, UB_{ij} is the upper bound for the number of fill features for tile (i, j) which can, e.g., be taken from the normal Monte Carlo fill result in [8], and LB_{ij} is the lower bound for the number of fill features for tile (i, j) which can, e.g., be obtained from the deletion phase in [6].

Rounded LP Relaxation of Integer Linear Programming

If we round the fractional LP relaxation of the above ILPs, then a solution may not be feasible since the number of fill features may be unequal to the F_{ij}'s. Therefore, after rounding we use a greedy algorithm (see the next section) to add or remove AREFs in order to add exactly F_{ij} fill features into each tile T_{ij}.

Nonoverlapping AREFs

In the above formulations, the resulting AREFs may overlap with each other. This means that some area fill features may be represented multiple times in a GDSII file. The need for a nonoverlapping version of this formulation arises from practical concerns and our experimental data.

From our experimental data, the number of area fill features represented multiple times by AREFs can be large. In the actual design however, each of these multiply represented features actually occurs only once. To determine a minimum number of *nonoverlapping* AREFs for the slack sites of the tile T_{ij}, constraints (8) may be modified as follows:

$$s_{pq} \le \sum_{\text{all AREFs covering } s_{pq}} a_\alpha \le 1 \qquad p = 0, \ldots, k-1, \quad q = 0, \ldots, l-1. \quad (14)$$

By rounding the result, we can, therefore, remove some AREFs from a tile, if necessary, so that the number of fill features becomes less than or equal to F_{ij}. Then, we use a greedy algorithm (see Section III) to add exactly F_{ij} fill features into each tile T_{ij}.

REFERENCES

[1] D. Boning, B. Lee, T. Tugbawa, and T. Park, "Models for pattern dependencies: capturing effects in oxide, STI, and copper CMP," presented at the Semicon/West Tech. Symp.: CMP Tech. for ULSI Manuf., San Francisco, CA, Jul. 2001.

[2] P. Buck, private communication, Jul. 2001.

[3] Y. Chen, A. B. Kahng, G. Robins, and A. Zelikovsky, "Area fill synthesis for uniform layout density," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 21, no. 10, pp. 1132–1147, Oct. 2002.

[4] ——, "Smoothness and uniformity of filled layout for VDSM manufacturability," in *Proc. Int. Symp. Physical Design*, Apr. 2002, pp. 137–142.

[5] ——, "Hierarchical dummy fill for process uniformity," in *Proc. ASP-DAC*, Jan. 2001, pp. 139–144.

[6] ——, "Practical iterated fill synthesis for CMP uniformity," in *Proc. Design Automation Conf.*, Los Angeles, CA, Jun. 2000, pp. 671–674.

[7] Y. Chen, A. B. Kahng, G. Robins, A. Zelikovsky, and Y. H. Zheng, "Data volume reduction in dummy fill generation," in *Proc. Design Automation and Test in Europe (DATE) Conf.*, Munich, Germany, Mar. 2003, pp. 868–873.

[8] Y. Chen, A. B. Kahng, G. Robins, and A. Zelikovsky, "New Monte Carlo algorithms for layout density control," in *Proc. ASP-DAC*, 2000, pp. 523–528.

[9] N. Cobb and E. Sahouria, "Hierarchical GDSII based fracturing and jobdeck system," *Proc. SPIE*, vol. 4562, pp. 734–762, 2001.

[10] N. Cobb and W. Zhang, "High performance hierarchical fracturing," *Proc. SPIE*, vol. 4754, pp. 91–96, 2002.

[11] J. Cong, L. Hagen, and A. B. Kahng, "Net partitions yield better module partitions," in *Proc. 29th Design Automation Conf.*, Jun. 1992, pp. 47–52.

[12] W. J. Cook and A. Rohe. Blossom IV—A minimum weighted perfect matching solver. [Online]. Available: http://www.or.uni-bonn.de/home/rohe/matching.html

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 2001.

[14] *Cplex7.0 User's Manual*, ILOG, Mountain View, CA, 2000.

[15] V. Dai and A. Zakhor, "Lossless layout compression for maskless lithography systems," in *Proc. Emerging Lithographic Technologies IV*, vol. SPIE 3997, Santa Clara, CA, Feb. 2000, pp. 467–477.

[16] R. R. Divecha, B. E. Stine, D. O. Ouma, J. U. Yoon, and D. S. Boning *et al.*, "Effect of fine-line density and pitch on interconnect ILD thickness variation in oxide CMP process," presented at the 3rd Int. Chemical Mechanical Polish for ULSI Multilevel Interconnection Conf. (CMP-MIC), Santa Clara, CA, Feb. 1998.

[17] R. Ellis, A. B. Kahng, and Y. H. Zheng, "JBIG Compression algorithms for dummy fill VLSI layout data," Comput. Sci. Eng. Dept., Univ. California, San Diego, Tech. Rep. CS2002-0709, Jun. 2002.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: Freeman, 1978.

[19] F. Harary, *Graph Theory*. Reading, MA: Addison-Wesley, 1994.

[20] International Technology Roadmap for Semiconductors (2002). [Online]. Available: http://public.itrs.net

[21] A. B. Kahng and G. Robins, "Optimal algorithms for extracting spatial regularity in images," *Pattern Recognit. Lett.*, vol. 12, pp. 757–764, 1991.
[22] A. B. Kahng, G. Robins, A. Singh, H. Wang, and A. Zelikovsky, "Filling algorithms and analyzes for layout density control," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 18, no. 4, pp. 445–462, Apr. 1999.
[23] G. Nanz and L. E. Camilletti, "Modeling of chemical-mechanical polishing: a review," *IEEE Trans. Semicond. Manuf.*, vol. 8, no. 4, pp. 382–389, Nov. 1995.
[24] New Standards Specification for Open Artwork System Interchange Standard (2002, Dec. 11). [Online]. Available: http://www.semi.org/web/wcontent.nsf/url/stds_blueballot
[25] D. Ouma, D. Boning, J. Chung, G. Shinn, L. Olsen, and J. Clark, "An integrated characterization and modeling methodology for CMP dielectric planarization," in *Proc. IEEE Int. Interconnect Technology Conf.*, San Francisco, CA, Jun. 1998, pp. 67–69.
[26] F. M. Schellenberg, L. Capodieci, and B. Socha, "Adoption of OPC and the impact on design and layout," in *Proc. Design Automation Conf.*, Las Vegas, Jun. 2001, pp. 89–92.
[27] R. Tian, D. Wong, and R. Boone, "Model-based dummy feature placement for oxide chemical mechanical polishing manufacturability," in *Proc. Design Automation Conf.*, Jun. 2000, pp. 667–670.
[28] R. Tian, X. Tang, and D. F. Wong, "Dummy feature placement for chemical-mechanical polishing uniformity in a shallow trench isolation process," in *Proc. Int. Symp. Physical Design*, Apr. 2001, pp. 118–123.
[29] S. Ueki, I. Ashida, and H. Kawahira, "Effective data compaction algorithm for vector scan EB writing system," presented at the 20th Annu. BACUS Symp. Photomask Technology, Monterey, CA, Sep. 2000.

Yu Chen received the M.S. degree in computer science and engineering from Zhejiang University, China, and the Ph.D. degree in computer science from the University of California at Los Angeles, in 1998 and 2003, respectively.

He is currently a Senior Technical Staff Member with Blaze-DFM, Inc., Sunnyvale, CA. His research interests include VLSI physical design, performance analysis, combinatorial optimization, and computational commerce.

Andrew B. Kahng received the A.B. degree in applied mathematics (physics) from Harvard College, Cambridge, MA, and from June 1983 to June 1986 was with Burroughs Corporation Micro Components Group, San Diego, CA, where he worked in device physics, circuit simulation, and CAD for VLSI layout. He received the M.S. and Ph.D. degrees in computer science from the University of California at San Diego. He joined the computer science faculty at University of California at Los Angeles (UCLA) in July 1989, and is currently Professor as well as Vice-Chair for graduate studies. From April 1996 through September 1997, he was on sabbatical leave and leave of absence from UCLA, as a Visiting Scientist at Cadence Design Systems, Inc. He resumed his duties at UCLA in Fall 1997. His interests include VLSI physical layout design and performance analysis, combinatorial and graph algorithms, and stochastic global optimization.

Prof. Kahng has received National Science Foundation (NSF) Research Initiation and Young Investigator Awards, and a DAC Best Paper Award. He was the founding General Chair of the 1997 ACM/IEEE International Symposium on Physical Design, and defined the physical design roadmap as a member of the Design Tools and Test working group for the 1997 renewal of the SIA National Technology Roadmap for Semiconductors. He is currently a member of the EDA Council's EDA 200X task force, and the Design Tools and Test working group for the 1999 SIA NTRS renewal.

Gabriel Robins (M'91) received the Ph.D. degree in computer science from the University of California at Los Angeles (UCLA) in 1992.

He is Professor of computer science in the Department of Computer Science at the University of Virginia, Charlottesville. His research interests include VLSI CAD, physical design, computational biology, and bioinformatics. He co-authored a book on high-performance routing as well as over 80 refereed papers, including a Distinguished Paper at ICCAD. Professor Robins served on the U.S. Army Science Board, and is an alumni of the Defense Science Study Group, an advisory panel to the U.S. Department of Defense. He also served on panels of the National Academy of Sciences and the National Science Foundation, as well as an expert witness in major IP litigations.

Prof. Robins received an IBM Fellowship and a Distinguished Teaching Award at UCLA. He received a Packard Foundation Fellowship, a National Science Foundation Young Investigator Award, a University Teaching Fellowship, an All-University Outstanding Teaching Award, a Faculty Mentor Award, and the Walter N. Munster Endowed Chair at the University of Virginia. He was General Chair of the 1996 ACM/SIGDA Physical Design Workshop, and a co-founder of the 1997 International Symposium on Physical Design. Professor Robins also served on the technical program committees of several other leading conferences, on the Editorial Board of the IEEE Book Series, and as Associate Editor of IEEE TRANSACTIONS ON VLSI. He is a member of ACM, SIGDA, and SIGACT.

Alexander Zelikovsky received the Ph.D. degree in computer science from the Institute of Mathematics of the Belorussian Academy of Sciences, Minsk, Belarus, in 1989.

He worked at the Institute of Mathematics, Kishinev, Moldova, from 1989 to 1995.. Between 1992 and 1995, he visited Bonn University and the Institut fur Informatik in Saarbrueken, Germany. He was a Research Scientist at the University of Virginia, Charlottesville, from 1995 to 1997 and a Postdoctoral Scholar at the University of California at Los Angeles from 1997 to 1998. Since 1999, he has been an Associate Professor in the Computer Science Department, Georgia State University, Atlanta. He is the author of more than 90 refereed publications. His research interests include VLSI physical layout design, discrete algorithms, ad hoc wireless networks, and computational biology.

Yuhong Zheng received the B.S. degree in precision instruments from Tsinghua University, China, and the M.S. degree in computer science and engineering from the University of California at San Diego in 1995 and 2004, respectively.

Her research interests include VLSI CAD, algorithm design and methodology development for physical design and design-manufacturing interface.