Monte-Carlo Algorithms for Layout Density Control[*]

Yu Chen, Andrew B. Kahng, Gabriel Robins[†] and Alexander Zelikovsky[‡]

UCLA Department of Computer Science, Los Angeles, CA 90095-1596
[†]Department of Computer Science, University of Virginia, Charlottesville, VA 22903-2442
[‡]Department of Computer Science, Georgia State University, Atlanta, GA 30303
{yuchen,abk}@cs.ucla.edu, robins@cs.virginia.edu, alexz@cs.gsu.edu

Abstract— *Chemical-mechanical polishing (CMP) and other manufacturing steps in very deep submicron VLSI have varying effects on device and interconnect features, depending on local characteristics of the layout. To enhance manufacturability and performance predictability, we seek to make the layout uniform with respect to prescribed* density *criteria, by inserting "fill" geometries into the layout. We propose several new Monte-Carlo based filling methods with fast dynamic data structures and report the tradeoff between runtime and accuracy for the suggested methods. Compared to existing linear programming based approaches, our Monte-Carlo methods seem very promising as they produce nearly-optimal solutions within reasonable runtimes.*

I. Introduction

As predicted by the International Technology Roadmap for Semiconductors (ITRS) [2], VLSI technology has entered deep submicron regimes, where the manufacturing process tends to have an increasingly constraining effect on physical layout design and verification [8]. One particular requirement is to control manufacturing variation due to *chemical-mechanical polishing* (CMP) [7] [9] [12], a process in which wafers are polished using a rotating pad and slurry to achieve planarized surfaces on which succeeding processing steps can build. CMP variation can be controlled through constrains on local feature density, relative to a certain "window size" on the order of 1-3mm that depends on CMP pad material, slurry composition, and other factors [1].

Many process layers, including diffusion and thin-ox, have associated density rules that are satisfied by layout post-processing which adds *fill* geometries. Historically, only foundries or specialized TCAD tools companies performed the layout post-processing necessary to achieve uniformity. Today, however, ECAD tools for physical design and verification cannot remain oblivious to such post-processing. Without an accurate estimate of the downstream filling at the foundry, the design flow may break due to inaccurate RC extraction, timing calculations, and reliability analyses [3].

A. The Filling Problem

Layout Density Control consists of two phases: *density analysis* and *fill synthesis.* The goal of density analysis, which we have addressed in [3], is to determine the area available for filling within each window. The fill synthesis phase then generates the actual fill geometries that occupy these available areas. The corresponding problem, which is the subject of our present work, can be formulated as:

The Filling Problem. Given a design rule-correct layout in an $n \times n$ layout region, window size $w < n$ and window density upper bound U, add fill geometries to create a *filled layout* such that the *variation* in window density (maximum window density minus minimum window density) is *minimized.*

Since bounding the density in $w \times w$ windows of a fixed dissection can incur error (since other windows that are not part of the dissection could still violate the density bounds), a common industry practice is to enforce density bounds in r^2 overlapping dissections, where r determines the "phase shift" w/r by which the dissections are offset from each other. (The "common denominator" of all dissections, i.e., a square of size $\frac{w}{r} \times \frac{w}{r}$, is called a *tile.*) In other words, density bounds are enforced only for windows of the so-called *fixed* r-dissection (see Figure 1), in the hope that this would also control the density bounds of arbitrary windows.

We refer to the general case (i.e., when considering *all* possible windows) as the *floating window* regime. On the other hand, when we are concerned with only windows from some fixed dissection over the layout, we are in the *fixed-dissection* regime. Solving the Filling Problem in the fixed-dissection regime consists of two steps: (i) finding the amount of fill to be embedded in each tile, and (ii) embedding the corresponding fill amount into each tile.

B. Previous Approaches

Previous papers [3] [4] [5] [6] gave the first formulations of the *filling problem.* These works also developed a num-

[*] Research at UCLA was supported by a grant from Cadence Design Systems, Inc. and by the MARCO/DARPA Gigascale Silicon Research Center. Professor Robins was supported by a Packard Foundation Fellowship and by NSF Young Investigator Award MIP-9457412. Professor Zelikovsky was supported by GSU Research Initiation Grant.

0-7803-5973-9/00/$10.00 ©2000 IEEE.

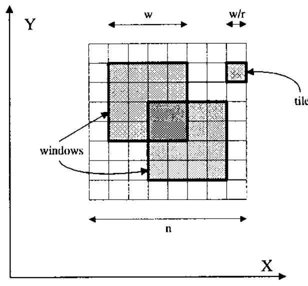

Figure 1: The layout is partitioned using r^2 ($r = 4$) distinct dissections (each with window size $w \times w$), into $\frac{nr}{w} \times \frac{nr}{w}$ *tiles*. Each $w \times w$ window (darker) consists of r^2 tiles. Note that a pair of windows from different dissections may overlap.

ber of algorithms for density analysis, along with filling synthesis algorithms in the fixed-dissection regime for flat and hierarchical designs [3] [4] [5] [6].

In [4] we proposed the first optimal solution to the min-variation filling formulation in the fixed-dissection regime. The approach, based on linear programming (LP), assumes that we can fill the slack area of each cell independently and uniformly, as is the case when the size of fill geometries is sufficiently small. For flattened designs, our LP formulation uses area, slack and filling computations for each tile T_{ij}, $i, j = 0, \ldots, \frac{nr}{w} - 1$. The indices i and j indicate that $(i \cdot \frac{nr}{w}, j \cdot \frac{nr}{w})$ is the left-bottom corner of T_{ij} (see Figure 1).

Maximize M, subject to:

$$p_{ij} \geq 0 \quad i, j = 0, \ldots, \frac{nr}{w} - 1 \tag{1}$$

$$p_{ij} \leq slack(T_{ij}) \quad i, j = 0, \ldots, \frac{nr}{w} - 1 \tag{2}$$

$$\sum_{s=i}^{i+r-1} \sum_{t=j}^{j+r-1} p_{st} \leq \alpha_{ij} \left(U \cdot w^2 - area_{ij} \right), \quad i, j = 0, \ldots, \frac{nr}{w} - r + 1 \tag{3}$$

$$M \leq \sum_{s=i}^{i+r-1} \sum_{t=j}^{j+r-1} area(T_{st}) + \sum_{s=i}^{i+r-1} \sum_{t=j}^{j+r-1} p_{st}, \quad i, j = 0, \ldots, \frac{nr}{w} - r + 1 \tag{4}$$

where $\alpha_{ij} = 0$ if $area_{ij} > U \cdot w^2$ and $= 1$, otherwise.

The constraints (1) imply that we can only add features, and cannot delete features from any tile. The slack constraints (2) are computed for each tile: if a tile T_{ij} is originally overfilled, then we set $slack(T_{ij}) = 0$. The values of p_{ij} from the LP solution indicate the fill amount to be inserted into each tile T_{ij}. The constraints (3) ensure that no window can have density more than U after filling, unless it was initially overfilled. Inequalities (4) imply that the auxiliary variable M is a lower bound on all window densities. The linear program seeks to maximize M, thus achieving the min-variation objective.

An important variant of the LP approach to the Filling Problem was recently considered in [11]. Instead of minimizing density variation over the entire layout, [11] suggests taking into account only the *range density variation* where variation is counted only for collections of neighboring windows. Furthermore, the formulation is based on a non-uniform contribution of a given feature to window densities at various distances from the feature (i.e., based on an elliptical weighting function), following the CMP model proposed in [10].

C. Contributions

In this paper we consider a new approach to the Filling Problem based on the Monte-Carlo paradigm. Our goal is to develop a method with significantly better scaling properties than the LP formulation, without incurring serious loss of solution quality. Our approach transparently extends to the non-uniform weighting model of [10], similarly to the LP approach reviewed above.

The remainder of the paper is organized as follows. Section 2 describes the Monte-Carlo method and analyzes several classes of implementation details, including (i) prioritization of the tiles to be selected for fill insertion; (ii) efficient methods for updating tile priorities; and (iii) allowable amount of fill to be added at any time into a given tile. Section 3 identifies the most promising variants based on comparisons with the optimal LP approach. Section 4 concludes with directions for future research.

II. A New Monte-Carlo Approach to Filling

The number of variables and the number of constraints in the linear program are both $O((\frac{nr}{w})^2)$. In practice, even for a large die and a user requirement of high accuracy, we might have $n = 15,000$, $w = 3,000$, and $r = 10$, which still yields a linear program of tractable size. Although such a solution is optimal, it has two drawbacks: (1) solving a very large LP is too time consuming (e.g. the runtime is $O(v^3)$, where v is the number of variables in the LP), and (2) the optimal solution for the given number r of overlapping dissections is not necessarily the optimal solution for, e.g., $2r$ overlapping dissections and in general may result in a high window density variation for the floating window regime (when all possible windows are considered).

Below we compare several Monte-Carlo methods, some of which are much faster on large layouts than the LP approach. Furthermore, the Monte-Carlo methods do not suffer from fixed-dissection vs. floating window discrepancies, since larger values of r can be handled. Our experiments show that the accuracy of the Monte-Carlo algorithms is reasonably high. In the rest of this section, we discuss several key implementation decisions for our Monte-Carlo approach (see Figure 2).

```
Monte-Carlo Filling Algorithm
Input:
    $n \times n$ layout,
    fixed $r$-dissection into tiles $T_{ij}$,
        $i, j = 0, \ldots, \frac{nr}{w} - 1$,
    $slack(T_{ij})$ = slack of tile $T_{ij}$,
    $area(W_{ij})$ = area of $w \times w$ window $W_{ij}$,
    $unit\_fill$ = unit filling area,
    $U$ = upper bound on $w \times w$ window area
Output: Filled layout
(01) For each tile $T$ initialize
(02)         $insert\_in(T) = 0$
(03)         $priority(T) = f(U, slack(T), MaxWin(T))$
(04) While the sum of tile priorities is positive do
(05)     Select a random tile $T$ according to priorities
(06)     $insert\_in(T) = insert\_in(T) + 1$
(07)     $slack(T) = slack(T) - unit\_fill$
(08)     If $slack(T) < unit\_fill$
(09)       then $priority(T) = 0$
(10)       else $priority(T) = priority(T) - unit\_fill$
(11)     For each window $W$ containing $T$ do
(12)       $area(W) = area(W) + unit\_fill$
(13)       For each tile $T' \in W$ do
(14)           update $priority(T')$ according to $area(W)$
(15) For each tile $T$
(16)     randomly perturb sequence of grid positions:
(17)     $random(i) = 1, \ldots, slack(T)/unit\_fill$
(18)     For $i = 1..insert\_in(T)$ do
(19)         Insert a unit-fill geometry
(20)         in the $random(i)^{th}$ grid position
(21) Output the filled layout
```

Figure 2: Monte-Carlo Based Filling Algorithm.

A. Priorities

The Monte-Carlo methods considered in this paper randomly choose a tile and increment its contents (i.e., area density) by a prescribed fill amount. The probability of choosing a particular tile T_{ij} is the *priority* of that tile. The priority of a tile may depend on the density of the windows containing this tile, or else be the same regardless of the window density. Note that the priority of a tile is zero if it belongs to a window which has already achieved the density upper bound U and is "locked" (see below).

We consider three different ways of computing tile fill priorities. The first method does not take into account the density of the windows containing the tile. We call this the *slack priority,* because it sets the priority to be equal to the slack of the tile. Intuitively, this means that we select the tile with probability proportional to its empty area, i.e., the choice of any available legal position of a fill geometry is uniform and independent.

In order to take into account the density of windows we consider two more alternatives:

- *maximal* priority of the tile T_{ij} is proportional to $U - MaxWin(T_{ij})$, and
- *minimal* priority of the tile T_{ij} is proportional to $U - MinWin(T_{ij})$,

where $MaxWin(T_{ij})$ and $MinWin(T_{ij})$ are respectively the maximum and minimum densities over all windows containing T_{ij}.

The intuition behind the maximal priority is to first insert fill into tiles for which the upper density U is less likely to constrain the filling. In other words, we want to insert as much fill as possible before all tiles either exhaust their slack or belong to a window with the density U. On the other hand, the minimal priority ensures preference of tiles which belong to the most underfilled windows. Thus, each insertion of a filling geometry increases the current minimum window density with high probability. The Monte-Carlo algorithm with this minimal priority scheme can be viewed as a randomized version of the greedy algorithm for solving the linear program (Equations (1)-(4)).

We may further increase the *relative* probabilities of selecting tiles with relatively higher (minimal or maximal) priorities. This is easily accomplished, e.g., by raising priorities to the power 2 or 4 before normalizing them. Raising priorities to a higher power brings the Monte Carlo algorithm even closer to the greedy algorithm (which fills tiles in a deterministic order; see Table 3).

B. Updating Priorities

Regardless of which priority scheme is chosen, it is essential to update the priority of tiles which belong to *locked* windows (i.e., windows with density U). Thus, when newly added fill causes a window to reach its maximum allowable density, all tiles in that window should be removed from the prioritization scheme, since they cannot be assigned any more fill. We propose two heuristic schedules for updating tile priorities after each fill geometry insertion. In the context of Figure 2, these are:

(H1): Update priorities of all affected tiles, i.e., execute all lines in the algorithm shown in Figure 2; and

(H2): Update priority only of tiles which belong to locked windows, i.e., in the algorithm of Figure 2, omit Line 10 and execute the loop at Lines 13-14 only if window W achieves the maximum density U.

The above discussion implies that the underlying data structures must support two distinct operations, namely, priority-based tile selection, and efficient updating of priorities. One simple way of implementing tile selection is to

(1) arrange tiles in a 1-dimensional array $T_i, i = 1 \ldots, k$; (2) create a list of sums of priorities $S_0 = 0, S_1, \ldots, S_k$, such that $S_{i+1} = S_i + priority(T_i)$; and (3) choose a random number in the range $(0, S_k)$ which will belong to some subinterval (S_{i-1}, S_i) corresponding to selection of the tile T_i. Such tile selection is very fast, but unfortunately priority updating requires $O(k)$ time on average. We suggest the *quadrisection approach* which recursively partitions the design into 4 quadrants and maintains the sum of priorities of all tiles in each quadrant. The runtime of our data structure is $O(\log k)$ per insertion.[1] Since schedule H2 updates priorities only once (i.e., when the window containing the tile is locked), the average insertion time will be much smaller for H2 than for H1 (see Table 2).

C. Filling Schedule

A third family of implementation design choices depends on how many filling geometries may be inserted into a tile per iteration. We compare two alternatives: (i) insert into a tile T_{ij} a single fill geometry per iteration, or (ii) insert the maximum possible number of fill geometries which is $\min\{U - MaxWin(T_{ij}), slack(T_{ij})\}$.

D. Slack Calculation and Fill Insertion

Finally, we address the second phase of the Filling Problem, namely fill insertion, and discuss the influence of this step on slack calculation. Fill insertion can use the Monte-Carlo approach regardless of the fill amount calculation method: randomly choose a legal fill geometry location, and iteratively insert fill at that location. The main obstacle in filling tiles is the shape of the fill geometries, which can make it infeasible to completely fill a tile up to the desirable amount specified by either the Monte-Carlo or by the linear-program approach.

Throughout this paper we avoid such problems by replacing the *area slack* calculations of [4] by *grid slack* calculations. The latter entails using an underlying *fill layout grid* to compute the maximum number of legal positions for fill geometries in each cell. This method of calculating slack is more realistic and ensures that the desirable amount of fill can actually be legally inserted into the corresponding tile. In other words, the area slack (i.e., the area of the tile that can be covered by fill) may be too optimistic, and so the prescribed LP fill solution may not correspond to a legal filling. On the other hand, the grid slack is too pessimistic and the LP solution may be slightly suboptimal but is always realizable (an exact fill amount calculation would be too time consuming).

[1]First, a random number R between 0 and the sum of all priorities is chosen. If R is greater than the priority of the first quadrant q_1, then $R = R - q_1$ and so on until $R < q_i$. We then repeat this recursively on all sub-quadrants of q_i. Finally, after repeating at most $O(\log k)$ recursive steps, we will find the tile in which to insert fill. Priority updating can be done within the same time complexity, using a bottom-up approach.

III. Experimental Results

Our experimental testbed integrates GDSII Stream input, conversion to CIF format, and internally-developed geometric processing engines, coded in C++ under Solaris. We have performed experiments using part of a metal layer extracted from an industry standard-cell layout. Benchmark L1 is the M2 layer from an 8,131-cell design, and Benchmark L1x4 is the same layout replicated four times in a 2x2 array to create a larger test case. Benchmark L2 is the M3 layer from a 20,577-cell layout. L2x4 is this layout replicated four times in a 2x2 array. Table 1 summarizes our testcases, i.e., layout dimension N, number of rectangles k, and the window size w. The window size w and value of r in the fixed r-dissection regime are shown with all experimental results. (In the given coordinate system, 40 units is equivalent to 1 micron.)

In real-world density control applications, the size of the window is prescribed by the manufacturing processes: it indicates the region containing the *effective* density which actually affects the polishing pad in a fixed position. The smaller the window size, the more variation in density is observed. The number r of fixed dissections reflects the accuracy of the prescribed approach: ideally, w/r should be equal to the size of a single filling geometry g. When r grows from 1 to w/g, the accuracy and the observed density variation both increase.

Test Cases				
testcase	L1	L2	L1x4	L2x4
layout size n	125,000	112,000	250,000	224,000
# rectangles k	49,506	76,423	198,024	305,692

Table 1: Parameters of four industry test cases.

Table 2 compares the runtimes and the original and resulting minimum window densities for the minimal, maximal, and slack priorities.[2] All run times are CPU seconds on a 300MHz Sun Ultra-5_10 with 640MB of RAM. From these results one can see a tradeoff between runtime and accuracy for different priorities: the fastest slack priority has the lowest accuracy and the minimal priority, while the slowest slack priority has the highest accuracy. The best choice seems to be maximal priority, which is almost as accurate as the minimal priority, but considerably faster.

Table 3 compares minimal and maximal priorities if they are raised into power 2 and 4 respectively. Raising into power 2 and 4 increases the accuracy of filling and has negligible effect on the runtime.

Table 4 compares the optimal results obtained by solving the linear program (Equations (1)-(4)) from [4] with

[2]All experiments assume that U equals the maximum window density of the original layout.

two fill schedule methods (see Section D). Our results show that the accuracy of the Monte-Carlo method is very high: in all our test cases the resulting variation is no more than 5% larger than the optimal obtained by the LP method. On the other hand, the the Monte-Carlo method is much faster than the LP method. When the window size is small and/or the number of fixed dissections is large, the LP method becomes impractical, while our new method is still fast.

The exhaustive comparison of different tile priorities and updating schedules shows that the minimal-priority updating is the best choice for the Monte-Carlo method (see Table 2). On the other hand, the faster filling schedule, which fills a chosen tile with the maximum possible number of filling geometries, loses in terms of performance, e.g., in some cases the window density variation does not even change (see Table 4).

The runtime advantage of the Monte-Carlo methods may be leveraged to obtain more even filling. We apply the faster Monte-Carlo method to larger number of fixed dissections, resulting in filled layouts which are more uniform than those obtainable with the LP method. For instance, the LP method applied to the layout with parameters (L1/4/4) gives a filled layout with a density variation of 15% measured for $r = 16$ fixed dissections. On the other hand, the Monte-Carlo method with the slack priority, minimal updating, and single-geometry filling schedule applied to the same layout but for $r = 16$ fixed dissections, gives a filled layout with a density variation less than 14%. Moreover, the LP method requires almost two minutes while the Monte-Carlo method takes only 10 seconds.

IV. Conclusions and Future Directions

In conclusion, we have presented a new Monte-Carlo approach for Layout Density control. We have compared several priority schemes, i.e. different ways to choose a "random" place to insert fill geometry. The Monte-Carlo method has been shown to be a scalable approach which is almost as accurate as previously known linear programming based approaches.

While we have discussed only flat algorithms, they can be efficiently integrated into existing hierarchical physical verification engines; our ongoing work pursues this integration and other extensions to current filling capability:

- Finding a more accurate slack computation method, since the grid-slack is too pessimistic, while the area-slack is too optimistic;
- Producing compact filling solutions as measured by output GDSII file size (i.e., the filling should be designed to maximally exploit AREF and SREF constructs); and
- Achieving layout density control in a truly hierarchical context, where (i) masters and not cell instances are filled, and (ii) the filling solution for a reusable IP block must be "detunable" depending on the density context within which the IP is used.

References

[1] R. R. Divecha, B. E. Stine, D. O. Ouma, J. U. Yoon, D. S. Boning, J. E. Chung, O. S. Nakagawa, and S. Y. Oh, *Effect of Fine-line Density and Pitch on Interconnect ILD Thickness Variation in Oxide CMP Process*, in Proc. CMP-MIC, Santa Clara, February 1998.

[2] http://www.itrs.net/. See also SIA, *The National Technology Roadmap for Semiconductors*, Semiconductor Industry Association, December 1997.

[3] A. B. Kahng, G. Robins, A. Singh, H. Wang, and A. Zelikovsky, *Filling and Slotting: Analysis and Algorithms*, in Proc. International Symposium on Physical Design, Monterey, CA, April 1998, pp. 95–102.

[4] A. B. Kahng, G. Robins, A. Singh, and A. Zelikovsky, *New and Exact Filling Algorithms for Layout Density Control*, in Proceedings of the 12th International Conference on VLSI Design, Goa, India, 1999, pp. 106–110.

[5] A. B. Kahng, G. Robins, A. Singh, and A. Zelikovsky, *New Multi-Level and Hierarchical Algorithms for Layout Density Control*, in Proceedings of the Asia and South Pacific Design Automation Conference, Hong Kong, China, 1999, pp. 221–224.

[6] A. B. Kahng, G. Robins, A. Singh, and A. Zelikovsky, *Filling Algorithms and Analyses for Layout Density Control*, IEEE Trans. Computer-Aided Design, 18 (1999), pp. 445–462.

[7] H. Landis, P. Burke, W. Cote, W. Hill, C. Hoffman, C. Kaanta, C. Koburger, W. Lange, M. Leach, and S. Luce, *Integration of Chemical-Mechanical Polishing into CMOS Integrated Circuit Manufacturing*, Thin Solid Films, 220 (1992), pp. 1–7.

[8] W. Maly, *Moore's Law and Physical Design of ICs*, in Proc. International Symposium on Physical Design, Monterey, CA, April 1998. special address.

[9] G. Nanz and L. E. Camilletti, *Modeling of Chemical-Mechanical Polishing: A Review*, IEEE Trans. on Semiconductor Manufacturing, 8 (1995), pp. 382–389.

[10] B. E. Stine, D. S. Boning, J. E. Chung, and L. Camilletti, *The Physical and Electrical Effects of Metal-fill Patterning Practices for Oxide Chemical-Mechanical Polishing Processes*, IEEE Transactions on Electron Devices, 45 (1998), pp. 665–679.

[11] R. Tian, D. Wong, R. Boone, and A. Reich, *Dummy Feature Placement for Oxide Chemical-Mechanical Polishing Manufacturability*, Tech. Rep. 9-19, University of Texas at Austin CS Dept., 1999.

[12] M. Tomozawa, *Oxide CMP Mechanisms*, Solid State Technology, (1997), pp. 169–175.

			Heuristic I				**Heuristic II**					
Layout Info			**Min_Pri**		**Max_Pri**		**Min_Pri**		**Max_Pri**		**SLK_Pri**	
T/W/r	Max	Min	Min	CPU	Min	CPU	Min	CPU	Min	CPU	Min	CPU
L1/31/2	0.20201	0.10548	0.19354	3.30	0.19336	2.21	0.19345	1.01	0.19327	1.00	0.19254	0.97
L1/31/3	0.20712	0.09683	0.19186	9.96	0.19102	5.96	0.19148	1.33	0.19093	1.31	0.19571	1.32
L1/31/4	0.21248	0.09369	0.19870	26.29	0.19811	15.18	0.19778	1.67	0.19660	1.69	0.19505	1.67
L1/31/5	0.21449	0.09097	0.19950	57.08	0.19871	32.35	0.19874	2.08	0.19847	2.07	0.19678	2.08
L1x4/31/2	0.21075	0.08739	0.15132	13.38	0.15132	10.08	0.15124	4.72	0.15044	4.66	0.14948	4.64
L1x4/31/3	0.21511	0.07808	0.14765	38.00	0.14765	25.10	0.14765	5.78	0.14765	5.72	0.14762	5.66
L1x4/31/4	0.21489	0.10775	0.19192	90.59	0.19101	54.50	0.19027	6.59	0.18977	6.55	0.19002	6.49
L1x4/31/5	0.21462	0.10103	0.18454	187.16	0.18445	109.58	0.18336	7.67	0.18307	7.65	0.18164	7.57
L2/28/2	0.18076	0.05065	0.11353	4.70	0.11327	3.98	0.11301	1.67	0.11411	1.68	0.11305	1.67
L2/28/3	0.22651	0.05125	0.14774	20.98	0.14538	15.98	0.14527	2.87	0.14612	2.86	0.14944	2.87
L2/28/4	0.21827	0.08072	0.17866	49.30	0.17796	35.05	0.17810	3.30	0.17814	3.29	0.17912	3.28
L2/28/5	0.23764	0.07203	0.17121	100.84	0.16703	78.74	0.16535	3.92	0.16582	3.91	0.16830	3.99
L2x4/28/2	0.22327	0.05011	0.17217	44.54	0.16472	34.63	0.16712	13.88	0.16450	13.90	0.16129	13.59
L2x4/28/3	0.20957	0.05087	0.12437	90.25	0.12514	69.53	0.12286	13.12	0.12234	13.19	0.12364	12.99
L2x4/28/4	0.22412	0.05010	0.17105	242.11	0.16867	176.95	0.16887	17.41	0.16908	17.33	0.16407	17.18
L2x4/28/5	0.23771	0.05005	0.16841	516.86	0.16482	373.06	0.16538	21.36	0.16285	21.32	0.16037	21.00

Table 2: Monte-Carlo methods with varying tile selection priorities and updating schedules. **Notation:** *T/W/r:* Layout/ window size / r-dissection; *Max in Layout Info:* the original maximum window density in the layout; *Min in Layout Info:* the original minimum window density in the layout; *Max_Pri* = maximal priority; *Min_Pri* = minimal priority; *SLK_Pri* = slack priority. The columns *Min* and *CPU* correspond to minimum window density of the filled layout and the runtime in CPU seconds, respectively. The maximum window density in the filled layout is the same as in the original layout.

Layout Info			*Min_Pri*	$(Min\_Pri)^2$	$(Min\_Pri)^4$	*Max_Pri*	$(Max\_Pri)^2$	$(Max\_Pri)^4$
T/W/r	MaxDen	MinDen	MinDen	MinDen	MinDen	MinDen	MinDen	MinDen
L1/31/2	0.20201	0.10548	0.19393	0.195461	0.194632	0.19345	0.19271	0.18247
L1/31/3	0.20712	0.09683	0.19223	0.19265	0.19245	0.19205	0.19251	0.19216
L1/31/4	0.21248	0.09369	0.19814	0.19893	0.19661	0.19643	0.19655	0.19592
L1/31/5	0.21449	0.09097	0.19869	0.19961	0.19855	0.19793	0.19895	0.19664
L2/28/2	0.18076	0.05065	0.11150	0.11198	0.11213	0.11437	0.11400	0.11673
L2/28/3	0.22651	0.05125	0.14697	0.15168	0.15715	0.14625	0.14816	0.15261
L2/28/4	0.21827	0.08072	0.17688	0.17777	0.17800	0.17677	0.17737	0.17799
L2/28/5	0.23764	0.07203	0.16627	0.16671	0.16842	0.16675	0.16539	0.16889

Table 3: Monte-Carlo method with power 2 and 4 priorities. **Notation:** $(Min\_Pri)^2$*:* using the power 2 of minimal priority as the priority of tile. Similarly, $(Max\_Pri)^4$*:* using the power 4 of maximal priority as the priority of tile.

			LP Method		**Heuristic I**				**Heuristic II**			
Layout Info					**Filling_1**		**Filling_2**		**Filling_1**		**Filling_2**	
T/W/r	Max	Min	Min	CPU	Min	CPU	Min	CPU	Min	CPU	Min	CPU
L1/31/2	0.20201	0.10548	0.20119	0.11	0.19354	3.30	0.18036	0.11	0.19345	1.01	0.17834	0.17
L1/31/3	0.20712	0.09683	0.20026	0.23	0.19186	9.96	0.18763	0.12	0.19148	1.33	0.18218	0.14
L1/31/4	0.21248	0.09369	0.20084	0.57	0.19870	26.29	0.19176	0.35	0.19778	1.67	0.19164	0.22
L1/31/5	0.21449	0.09097	0.20328	1.75	0.19950	57.08	0.19212	0.56	0.19874	2.08	0.19634	0.49
L1/8/2	0.26966	0.02080	0.15968	1.52	0.15868	6.32	0.13249	0.20	0.15868	2.29	0.14865	0.36
L1/8/3	0.27043	0.03151	0.17174	11.54	0.17162	14.78	0.16882	1.14	0.17162	2.54	0.16635	1.12
L1/8/4	0.27375	0.03362	0.18261	39.66	0.18282	39.97	0.17834	3.76	0.18282	3.53	0.17763	4.54
L1/8/5	0.27213	0.02766	0.14901	60.80	0.14827	67.45	0.13564	9.21	0.14827	3.90	0.13678	1.42
L1/4/2	0.28250	0.00544	0.16734	24.47	0.16771	7.31	0.11763	0.92	0.16771	2.79	0.11143	3.97
L1/4/3	0.27807	0.00911	0.13792	67.61	0.13229	14.31	0.13102	2.99	0.13229	3.17	0.11784	2.72
L1/4/4	0.28250	0.00950	0.16914	395.95	0.16452	44.07	0.14987	9.57	0.16452	5.28	0.15212	38.36
L1/4/5	0.28237	0.00390	0.12928	335.0	0.12385	88.91	0.11373	22.70	0.12385	8.58	0.11234	45.82
L1x4/31/2	0.21075	0.08739	0.15845	35.9	0.15132	13.38	0.09343	0.23	0.15124	4.72	0.0902	0.87
L1x4/31/3	0.21511	0.07808	0.15082	378.9	0.14765	38.00	0.09188	2.34	0.14765	5.7	0.10465	1.52
L1x4/31/4	0.21489	0.10775	0.19812	1864.3	0.19192	90.59	0.11473	2.37	0.19027	6.5	0.11274	3.59
L1x4/31/5	0.21462	0.10103	N/A	N/A	0.18454	187.16	0.11241	3.22	0.18336	7.67	0.10945	4.84

Table 4: Optimal LP filling compared with the Monte Carlo approach using different filling schedules. **Notation:** *Filling_1:* inserting a single filling geometry into a tile per iteration; *Filling_2:* inserting maximum possible filling geometries into a tile per iteration. The columns *Min* and *CPU* correspond to minimum window density of the filled layout and the runtime in CPU seconds, respectively. The maximum window density in the filled layout is the same as in the original layout. We did not fill all positions in the table because of the huge running time for the LP method.