A recursive cost-based approach to fracturing

Shangliang Jiang, Xu Ma, and Avideh Zakhor

Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA, 94720, USA
Email: {sjiang,maxu,avz}@eecs.berkeley.edu

ABSTRACT

In microlithography, mask patterns are first fractured into trapezoids and then written with a variable shaped beam machine. The efficiency and quality of the writing process is determined by the trapezoid count and external slivers. Slivers are trapezoids with width less than a threshold determined by the mask-writing tool. External slivers are slivers whose length is along the boundary of the polygon. External slivers have a large impact on critical dimension (CD) variability and should be avoided. The shrinking CD, increasing polygon density, and increasing use of resolution enhancement techniques create new challenges to control the trapezoid count and external sliver length. In this paper, we propose a recursive cost-based algorithm for fracturing which takes into account external sliver length as well as trapezoid count. We start by defining the notion of Cartesian convexity for rectilinear polygons. We then generate a grid-based sampling as a representation for fracturing. From these two ideas we develop two recursive algorithms, the first one utilizing a natural recurrence and the second one a more complex recurrence. Under Cartesian convexity conditions, the second algorithm is shown to be optimal, but with a significantly longer runtime than the first one. Our simulations demonstrate the natural recurrence algorithm to result in up to 60% lower external sliver length than a commercially available fracturing tool without increasing the polygon count.

Keywords: Fracture, mask data preparation, Variable Shaped Beam mask writing, sliver

1. INTRODUCTION

In optical lithography, light emitted from an illumination system is transmitted through the mask to generate an image on the wafer. Mask writing is a significant step affecting the fidelity of the printed image on the wafer and critical dimension (CD) control. The mask is typically printed using a Variable Shaped Beam (VSB) mask writing machine. As a first step the mask pattern is fractured into numerous non-overlapping trapezoids. Subsequently these trapezoids are exposed by the VSB mask writing machine onto the mask.

The use of a VSB mask writing system places requirements on the resulting fracturing solution. These requirements are either hard constraints that cannot be violated, or soft constraints that complicate the mask writing process and should preferably be avoided.[1,2] More specifically,

1. The mask pattern must be partitioned into a set of non-overlapping basic trapezoids.
2. The maximum linear size of a shot produced by current VSB mask writing machines is between $2\mu m$ and $3\mu m$ on the mask scale.
3. To have reasonable critical dimension uniformity, the cumulative length of external slivers should be minimized.
4. In order to minimize the shot count and hence writing time, the resulting number of trapezoids should be minimized.

The first two requirements are hard constraints as VSB writing technology cannot produce arbitrary shapes and sizes. Furthermore, the basic trapezoids must be non-overlapping as overlapped regions would result in excess exposure leading to increased variability. The last two requirements are soft; both the number of trapezoids and external sliver length should preferably be minimized to improve mask write time and CD uniformity.

Optical Microlithography XXIV, edited by Mircea V. Dusa, Proc. of SPIE Vol. 7973,
79732P · © 2011 SPIE · CCC code: 0277-786X/11/$18 · doi: 10.1117/12.879583

M. Bloecker et al. have evaluated many metrics to quantify fracturing quality;[5] they proposed and demonstrated that shoreline or external sliver length is a suitable metric for evaluating fracture quality as it closely correlates with manufacturability while being fast to evaluate. Features with width below a certain ϵ, as determined by the VSB mask-writing tool, are called slivers. Slivers whose length is along the boundary of the layout polygon are called external slivers. As the shot size becomes smaller, the current density becomes more steep adversely influencing the shot placement. Thus slivers result in large size variability, adversely affecting CD control.[3] Fig. 1 demonstrates two possible fracturing options and a possible tradeoff for a given mask pattern: Fig. 1(a) has fewer shots, and one external sliver while Fig. 1(b) has an extra shot as compared to Fig. 1(a) but no slivers. In practice, depending on the requirements, a different fracturing should be employed.

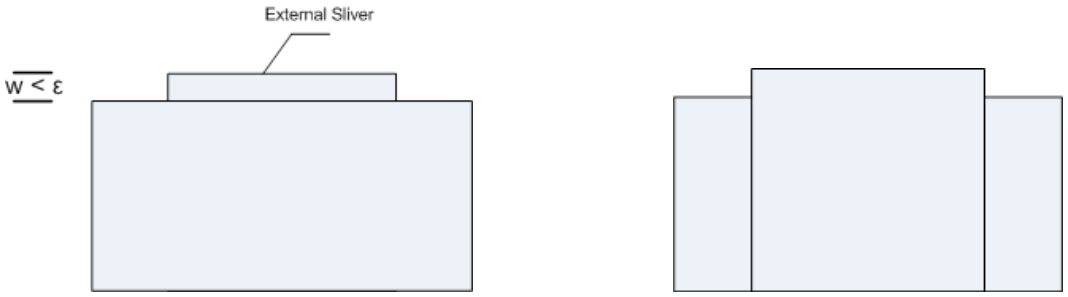


Figure 1: Two methods of fracturing the same polygon: (a) fracture solution with two polygons and one external sliver; (b) fracture solution with three polygons and zero external slivers.

C. Spence et al. have demonstrated that the number of shots is directly correlated with the write-time.[4] Furthermore, it is reasonable to assume that the shot count is approximately equal to the number of post-fracture figures, namely the resulting trapezoids. This is because for small critical feature sizes, few polygons if any, would require more than one shot. From this it follows that the write-time is directly proportional to the number of post-fracture trapezoids and thus the latter needs to be minimized.

In this paper, we restrict a fracturing to consist only of rectangles. With this constraint, we formulate fracturing as an optimization problem by choosing a cost function that is the weighted sum of the number of rectangles and the external sliver length. We develop a new grid-based representation for rectilinear polygons and propose two recursive algorithms to solve the optimization problem. A rectilinear polygon is a polygon whose corners form right angles. When the rectilinear polygon is restricted to be convex in the Cartesian directions, one of our proposed algorithms guarantees optimality. The other one is shown to be faster, and empirically matches the quality of the solutions resulting from the optimal algorithm.

In Section 2, we review results on fracturing that support our formulation choice. In Section 3, we describe our cost function, and define Cartesian convexity for rectilinear polygons. In Section 4, we describe our proposed algorithm. Section 5 includes experimental results for our proposed algorithms and their comparison with those of a commercial solver. We conclude with possible directions for future work in Section 6.

2. BASICS OF FRACTURING

In this section we review prior results in fracturing. In Section 2.1 we discuss partitioning a rectilinear polygon into the minimum number of rectangles. Section 2.2 states the requirements of a valid fracturing in terms of the interior points. We combine these results in Section 4 to develop the input representation and recurrence algorithm for our proposed fracturing method.

2.1. Minimum Cardinality Rectangle Partition

T. Ohtsuki has proposed an exact $O(n^{5/2})$ algorithm for partitioning a rectilinear polygon into the minimum number of rectangles[6] . The algorithm is based upon restricting partitioning rays to start only from concave corners and selecting the maximum disjoint set of chords. Chords are partitioning rays that start and terminate

at concave rays. The maximum disjoint set of chords can be selected by finding a maximum independent set for a bipartite graph, which is equivalent to finding the maximum matching of the same bipartite graph, from König's Theorem.[8] This is achievable in $O(n^{5/2})$ by solving the bipartite matching problem with the Hopcroft-Karp algorithm,[9] where n is the number of vertices.

The above results suggest a two step process to minimize the total number of rectangles: first, the rays forming the partition should be restricted to those that start from concave corners. Second, the number of chords selected in the partitioning needs to be maximized. In the next section we use this to show that only a finite number of interior points need to be checked to confirm the validity of the fracturing.

2.2. Valid Fracturing

In this section we discuss the requirements for a valid fracturing to occur. In particular, we examine the constraints that fracturing places upon the interior points. Interior points are those residing within the boundary of the input polygon. We combine these constraints with the results from Section 2.1 to demonstrate that only a select subset of the interior points need to be checked to show the validity of a fracturing. The results here are used in Section 4.1 to form the representation for our proposed fracturing algorithm.

Kahng et al. have shown that a valid fracturing of a rectilinear polygon results in all interior points of the polygon satisfying 1 of 3 constraints shown in Fig 2.[1,2] A valid fracturing of a rectilinear polygon is defined to be a partition of the polygon that consists only of rectangles. We define a line segment as part of a line that is bounded by two end points. The 3 validity constraints on the interior points of the polygon are:

- The point has 0 line segments entering it, as shown in Fig. 2(a).
- The point has 2 line segments entering that are in the same orientation; either both horizontal or both vertical. The vertical example is shown in Fig. 2(b).
- The point has 3 line segments entering it as shown in Fig. 2(c).

(a) (b) (c)

Figure 2: Valid internal points.

These three interior point constraints are the only valid configurations. The remaining two possibilities are invalid and can be described as follows:

- The point has 1 line segment leaving it, as shown in Fig. 3(a). This cannot exist within a valid fracturing as this point and line segment cannot be part of a rectangle.
- The point has 2 line segments entering with different orientations, one horizontal and the other vertical. This configuration is invalid as the concave side cannot be part of a rectangle. An example is shown in Fig. 3(b).

As stated in Section 2.1, to minimize the total number of rectangles resulting from the fracturing of a rectilinear polygon, only rays from concave corners should be used. In that case, all interior points of the polygon can be split into three groups. The first group consists of interior points that do not lie on the path of

(a) (b)

Figure 3: Invalid interior points.

any ray. These points never have any line segments passing through them. They always satisfy the constraint shown in Fig. 2(a), and do not need to be checked to confirm the validity of the fracturing.

The second group consists of the points that lie on the path of two rays. These interior points are marked by the intersections of the rays and need to be checked for the fracturing constraints in Fig. 2. However, there are only $O(n^2)$ such interior points, where n is the number of concave corners. This follows since these interior points exist as the intersections of two rays from concave corners. These interior points are the only interior points upon which a line segment may terminate. A line segment must terminate upon a boundary to result in a valid fracturing; the boundary may be part of the polygon boundary or an earlier placed line segment. As interior points do not reside on the polygon boundary, only interior points at the intersection of two rays can be the location of a line segment termination.

Finally, the third group consists of the interior points that lie along the path of only one ray. As line segments can only terminate at the points of intersections of two rays, these interior points cannot correspond to a terminated line segment. Therefore all interior points along the path of one ray must either satisfy the constraint in Fig. 2(a) i.e. no line segments passing through them or the constraint in Fig. 2(b) i.e. a line segment passing through and continuing in one of the Cartesian directions. Thus, with the restriction that line segments only terminate upon the intersection of two rays, these interior points do not need to be considered as they always satisfy the validity requirements for fracturing.

To summarize, the points for which validity of the fracturing is a concern are the $O(n^2)$ points that lie on the intersections of two rays, where n is the number of concave corners. This motivates our proposed representation in Section 4.1.

3. FRACTURING PROBLEM FORMULATION

In this section we consider two aspects of our problem: first, translating the soft constraints in fracturing to a cost function; second, examining convexity and how it relates to our problem. The VSB mask writing tool characteristics suggest two possibly conflicting goals in fracturing: the first is to minimize the number of resulting rectangles to limit shot count, and the second is to minimize the external sliver length for CD control. To capture both goals, we consider the following cost function:

$$\#(rectangles) + \lambda_L L(external\ slivers) \tag{1}$$

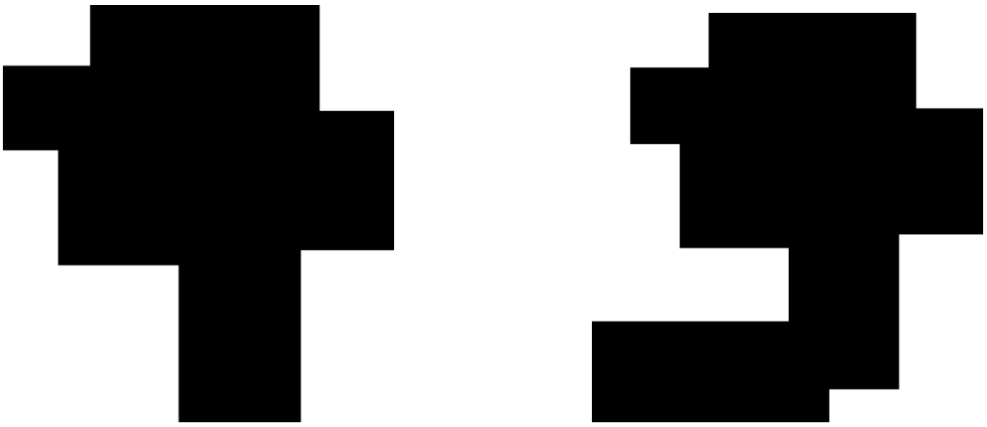
where λ_L is a parameter chosen by the user to weigh the cost of external sliver length against the rectangle count and $L(external\ slivers)$ refers to the length of the external slivers. We have empirically found that $\lambda_L \approx \frac{1\text{nm}}{\epsilon}$ results in reasonable fracturing performance by providing improvements in external sliver length without increasing the rectangle count.

A rectilinear polygon, with more than 4 corners, must have concave vertices and cannot be a convex polygon. Nevertheless, we propose a definition of a Cartesian convexity for rectilinear polygons as a rectilinear polygon

that is convex in the Cartesian directions. Mathematically, this can be written as:

$$(x_1, y) \text{ and } (x_2, y) \in POLYGON \text{ Then: } \forall t \in [0,1], ((1-t)x_1 + tx_2, y) \in POLYGON$$
$$(x, y_1) \text{ and } (x, y_2) \in POLYGON \text{ Then: } \forall t \in [0,1], (x, (1-t)y_1 + ty_2) \in POLYGON$$

The above equations state that if any two points that lie in a Cartesian convex rectilinear polygon can be joined by a line segment in a Cartesian direction, then all points in the line segment also lie within the polygon. This is equivalent to the property that any line in a Cartesian direction intersects the boundaries that are perpendicular to the line twice. Fig. 4 gives examples of a Cartesian convex and a non-convex rectilinear polygon.



(a) Convex (b) Non-Convex

Figure 4: Comparison of Cartesian convex and non-convex rectilinear polygons.

To summarize, the importance of this property is that any line in a Cartesian direction that lies within a Cartesian convex rectilinear polygon intersects the boundaries twice. This motivates our proposed recurrences in Sections 4.3 and 4.4.

4. PROPOSED ALGORITHM

Our proposed algorithms in this paper are applicable to rectilinear polygons. For convenience, we use the term polygon to refer to rectilinear polygons in the remainder of this paper. Our algorithms apply 3 steps to generate a fracturing. Section 4.1 shows how to transform the polygon into a rectangular gridded array. Section 4.2 describes the costs for the internal rectangles. Sections 4.3 and 4.4 include recursive algorithms used to find the lowest cost fracturing for Cartesian convex polygons. In Section 4.5 we describe the way dynamic programming is used to maintain a polynomial runtime. In Section 4.6 we extend our algorithms to non-Cartesian convex polygons.

4.1. Representation of Rectilinear Polygons

As stated in Section 2.1, rays should only be generated from concave corners to minimize rectangle count. Furthermore, in Section 2.2 we have shown that not all points within a polygon are relevant to the validity of a fracturing. The only relevant points are points where the constraints on the interior points may be violated. These points are the corners, the rays from each corner, and intersections of the rays with the boundary and with one another. We call the grid formed by these points the basic grid and use it to represent the polygon.

We propose the following representation for dicretizing the problem. First, we embed the input polygon with a rectangle. Next, we generate all the relevant points by including all possible partitioning rays from every corner and noting the intersections. For the resulting basic grid we number the points with coordinates $(1,1)$ to (n_1, n_2) as shown in Fig. 5.

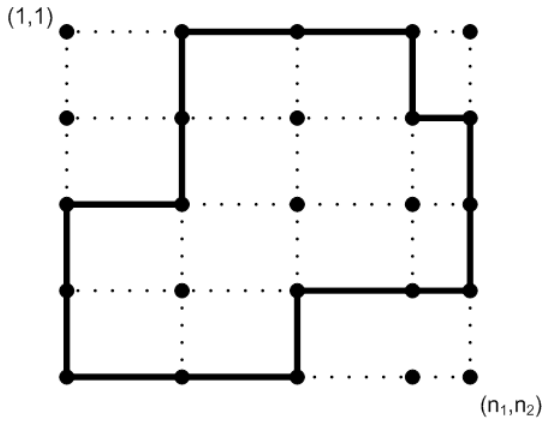


Figure 5: Embedding the polygon within a rectangle and generating the basic grid.

The polygon may be represented by a binary array of size $n_1 \times n_2$ where an entry is 0 if its corresponding point is exterior to the polygon and 1 if the point is interior. n_1 corresponds to the number of discrete coordinates in the x-direction for all vertices and n_2 to the number of discrete coordinates in the y-direction. To represent a rectangular selection of the basic grid a pair of coordinates $[(a, b), (c, d)]$ may be used; the first coordinate (a, b) refers to the top left corner while the second coordinate refers to the bottom right corner. This is shown in Fig. 6 where the marked rectangle is $[(2, 3), (5, 5)]$.

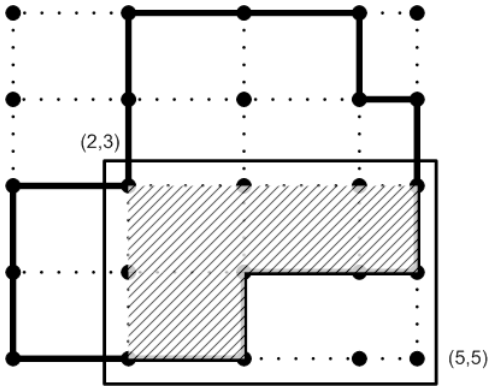


Figure 6: A representation of a sub-rectangle.

We define a sub-rectangle to be a rectangle whose vertices lie along the basic grid as shown in Fig. 6. The purpose of the sub-rectangle is to represent the subsection of the polygon embedded within it. For example, in Fig. 6, the sub-rectangle [(2,3),(5,5)] refers to the shaded portion of the polygon. This representation allows us to enumerate all possible sub-rectangles for a given rectilinear polygon. Specifically, the number of sub-rectangles is $O(n^4)$, where n is the number of vertices in the original polygon. This is because the grid has $O(n^2)$ points and a sub-rectangle requires two points from the grid.

The motivation behind this representation is that the cost function of any given sub-rectangle can be recursively computed by considering all possible decompositions of the given sub-rectangle into smaller sub-rectangles. At each step we select the decomposition that results in the lowest cost. A naive implementation results in an exponential runtime; however, in Section 4.5 we describe ways of applying the decomposition and storing costs to arrive at a tractable polynomial runtime.

4.2. Generating Costs for Rectangles within a Fracture

To solve the optimization problem we need to generate costs for each sub-rectangle. We use Equation 1 to assign to each sub-rectangle the cost of the partition of the subsection of the polygon enclosed by the sub-rectangle. For instance, the sub-rectangle, $[(2,3),(5,5)]$ in Fig. 7 has the cost of the polygon enclosed with vertices: $[(2,3),(5,3),(5,4),(3,4),(3,5),(2,5)]$. More generally, the four possible cases and resulting costs for sub-rectangles are described below:

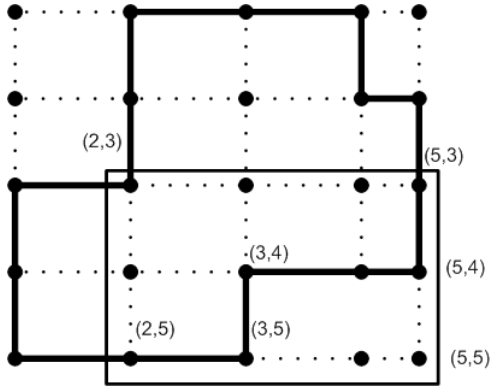


Figure 7: A representation of a sub-rectangle.

1. Sub-rectangles are entirely exterior to the input polygon – Cost 0.
2. Sub-rectangles are entirely interior to the polygon and are not external slivers – Cost 1.
3. Sub-rectangles are entirely interior to the polygon and are external slivers – Cost $1 + \lambda_L L(\textit{exposed edge})$.
4. Sub-rectangles consisting of both exterior and interior points of the polygon – Determined recursively by partitioning the sub-rectantangle

The sub-rectangles that satisfy the first case have zero cost as they do not contain any portion of the input polygon and have no bearing on the resulting fracture. Their existence comes about as an artifact of the representation. An example of such a subrectangle in Fig. 6 is $[(1,1),(2,2)]$. In the second and third cases, the sub-rectangle exists completely within the polygon and corresponds to a possible rectangle in the fracture. Therefore their costs contribute to the metric in the following way: 1 to represent the increase in shots and $\lambda_L L(\textit{exposed edge})$ to represent an external sliver. These values have been chosen according to the metric in Equation (1). Finally, the fourth case has no immediately obvious cost. The subrectangle marked in Fig., 7 falls into the fourth case. We solve for this cost by using the recursive algorithms described in the following two sections.

4.3. Natural Recurrence for Cartesian Convex Polygons

For an input polygon with Cartesian convexity, the first step is to form the basic grid and assign a cost to all the sub-rectangles that are readily available. The costs of the sub-rectangles for case 4 of Section 4.2 are determined recursively.

To develop this recurrence we first define a guillotine cut for a given Cartesian convex polygon as a ray that starts from a concave corner and continues until it reaches a boundary of the polygon. Such a ray splits the polygon into two pieces. We also define a rectangle cut to be a cut in a Cartesian direction that splits a rectangle into two smaller rectangles. Finally, a guillotine cut in the polygon is equivalent to a rectangle cut in the embedding rectangle, as shown in Fig. 8.

We now describe the recurrence that finds the lowest cost fracturing formed with only guillotine cuts: we take advantage of the fact that a guillotine cut for a Cartesian convex polygon is equivalent to a rectangle cut. The recurrence relationship for the sub-rectangles in case 4 of Section 4.2 can be written as:

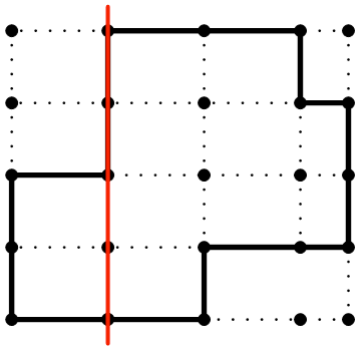


Figure 8: A guillotine cut in the polygon is equivalent to a rectangle cut.

$$
\begin{aligned}
F([(a,b),(c,d)]) = \min\{ & \\
& \min_{j\in\{a,\ldots,c\}} [F([(a,b),(j,d)]) + F([(j,b),(c,d)])], \\
& \min_{k\in\{b,\ldots,d\}} [F([(a,b),(c,k)]) + F([(a,k),(c,d)])] \\
& \}
\end{aligned}
\tag{2}
$$

where $[(a,b),(c,d)]$ is a sub-rectangle and $F([(a,b),(c,d)])$ refers to the optimal fracturing of the subsection of the input polygon embedded by the sub-rectangle defined by $[(a,b),(c,d)]$. The equation states that the optimal fracturing of the sub-rectangle $[(a,b),(c,d)]$ is found by finding the cut that minimizes the cost. This cut is found be sweeping through all horizontal, $j \in \{a,...,c\}$, and vertical, $k \in \{b,...,d\}$, cuts. The recurrence is demonstrated in Fig. 9, where Figs. 9(a), 9(b), and 9(c) show the sweeping of the vertical guillotine cuts and Figs. 9(d), 9(e), and 9(f) show the sweeping of the horizontal guillotine cuts. Each sub-figure in Fig. 9 demonstrates one of the possible decompositions. The algorithm selects the decomposition with the minimal cost. In order to do so, the algorithm may repeat the recurrence for the resulting components of the decomposition.

The above recurrence, by construction, solves for the best fracture that is formed only with guillotine rays. Using dynamic programming the algorithm may be implemented in $O(n^5)$; this is because there are $O(n^4)$ possible sub-rectangles for each of the sub-rectangles and there are $O(n)$ possible cuts that must be considered.

4.4. Optimal Recurrence for Cartesian Convex Polygons

In this section we show that the natural recurrence is not always optimal. Fig. 10 shows a possible fracturing that cannot be found by the natural recurrence as no guillotine cuts exist in the fracture.

Nevertheless, in Appendix A we show that this configuration is the only possible one in which guillotine cuts fail to fracture a Cartesian convex polygon and use this to develop a new fracturing algorithm. The first step is to again form the basic grid and assign costs to all the sub-rectangles that are readily available. The remaining costs are solved recursively.

Fig. 11 shows the only two possible ways for a decomposition to occur with no guillotine cuts. Both cases are generated by first selecting the inner rectangle, $[(j,k),(l,m)]$, and then partitioning the remaining shape without guillotine cuts. Fig 11(a) shows case 1, in which the cuts are selected to form a pinwheel rotating clockwise, while Fig 11(b) shows case 2 with a pinwheel rotating counter-clockwise. Again, this implies a recurrence, given below:

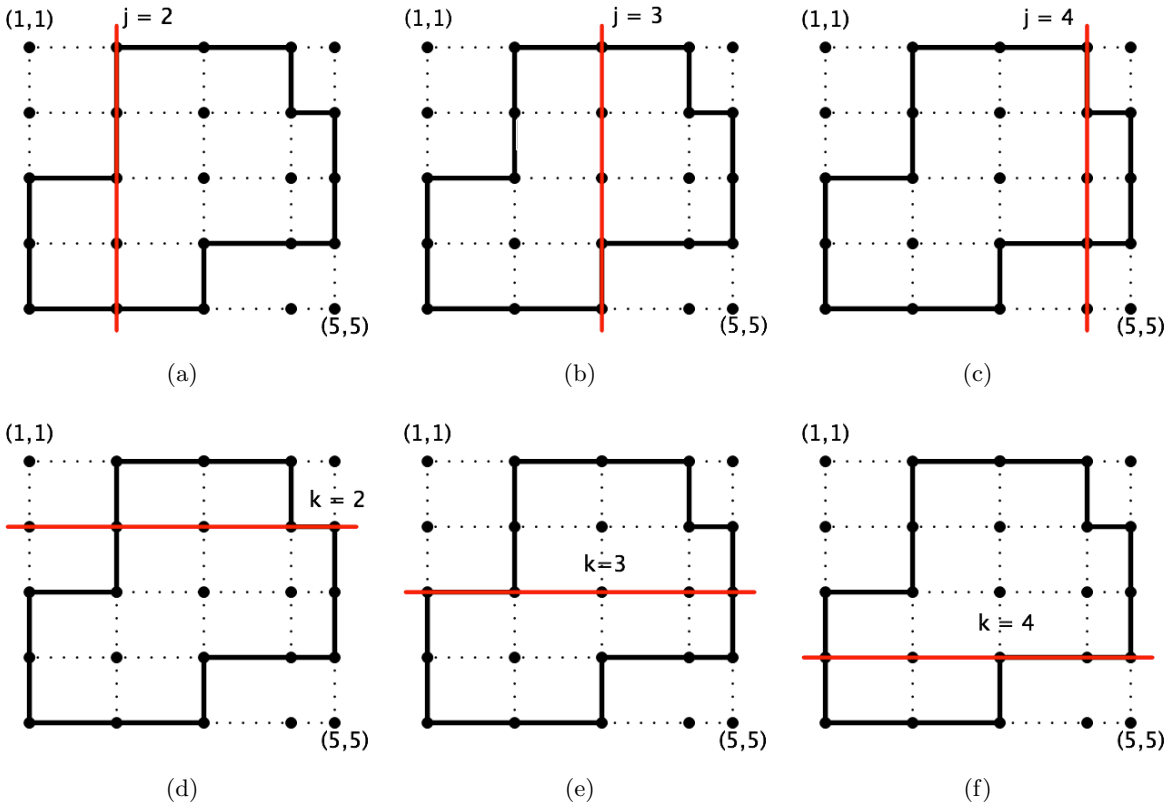


Figure 9: Decomposition example; vertical guillotine cuts for (a) j=2; (b) j = 3; (c) j = 4 and horizontal guillotine cuts for (d) k = 2; (e) k = 3; (f) k = 4.

$$
\begin{aligned}
F([(a,b),(c,d)]) = \min_{a\le j\le l\le c, b\le m\le k\le d}\{ & \\
& \min[F([(a,b),(l,k)]) + F([(l,b),(c,m)]) + \\
& \quad F([(a,k),(j,d)]) + F([(j,m),(c,d)]) + \\
& \quad F([(j,k),(l,m)])], \\
& \\
& \min[F([(a,b),(j,m)]) + F([(j,b),(c,k)]) + \\
& \quad F([(a,m),(l,d)]) + F([(l,k),(c,d)]) \\
& \quad F([(j,k),(l,m)])] \\
& \}
\end{aligned}
\tag{3}
$$

where $[(a,b),(c,d)]$ is a sub-rectangle and $F([(a,b),(c,d)])$ refers to the optimal fracturing of the subsection of the input polygon embedded by the sub-rectangle defined by $[(a,b),(c,d)]$. The first term in Eqn. (3) corresponds to the cost associated with case 1, shown in Fig. 11(a) and the second term to that of case 2 in Fig. 11(b).

Equation 3 states that the optimal fracturing of the sub-rectangle $[(a,b),(c,d)]$ is found by finding all possible decompositions and selecting the one with minimal cost. A decomposition is generated by first selecting a sub-rectangle, $[(j,k),(l,m)]$, that resides within the initial rectangle. Next, the algorithm generates the two pinwheel recurrences with the remaining shape, as shown in Fig 11. This forms one decomposition. The decomposition process is repeated by selecting all sub-rectangles $[(j,k),(l,m)]$ that consist only of points on the basic grid belonging to the strict subset of the input sub-rectangle $[(a,b),(c,d)]$. Upon selecting $[(j,k),(l,m)]$, the same pinwheel recurrence can be generated. After generating all possible decompositions the one with minimal cost

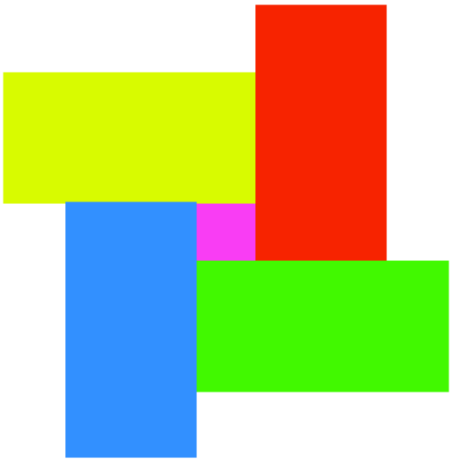


Figure 10: Natural recurrence failing.

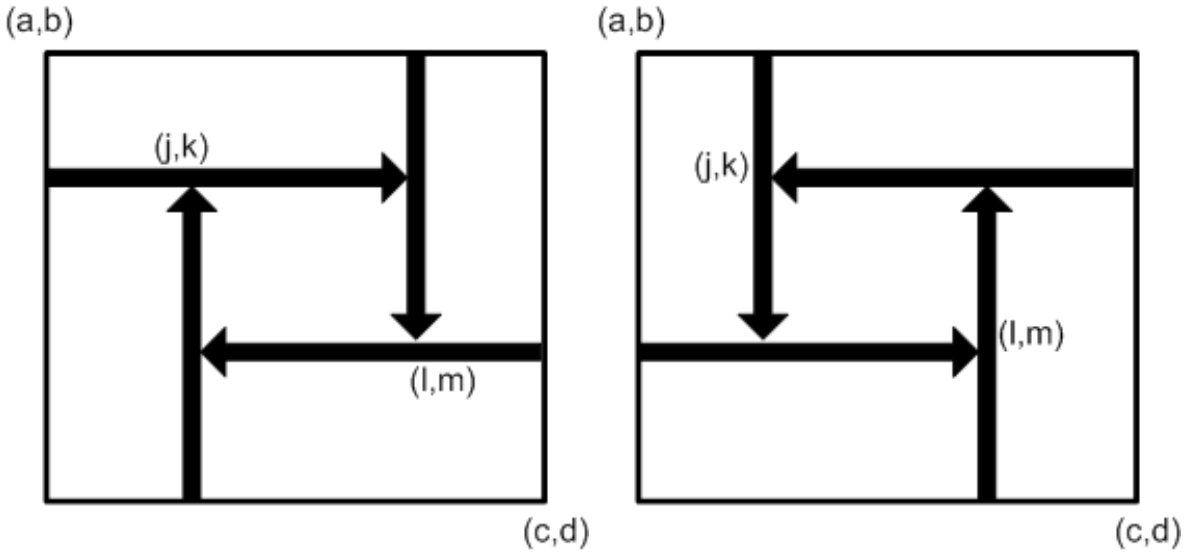


Figure 11: Pinwheel fracturing decomposition with no guillotine cuts; (a) Case 1; (b) Case 2.

is stored as the optimal cost of sub-rectangle $[(a, b), (c, d)]$. The algorithm may need to repeat the recurrence for each resulting decomposition. An example is shown in Fig. 12 with a few decompositions.

Equation 3 describes a recurrence that does not rely on guillotine cuts. In Appendix A we show that this recurrence is optimal for Cartesian convex polygons and place bounds on the runtime. As explained in Appendix A, the runtime for this recurrence is $O(n^8)$, where n is the number of vertices in the input polygon.

4.5. Implementation of Recurrences using Dynamic Programming

The algorithms in Sections 4.3 and 4.4 are recursive, and as such, a naive implementation would result in an exponential runtime. This is because the cost of each sub-rectangle is computed multiple times. An alternative implementation is via dynamic programming, whereby a complex problem is broken into smaller overlapping sub-problems.[10] In particular, dynamic programming requires the problem to have an optimal structure and

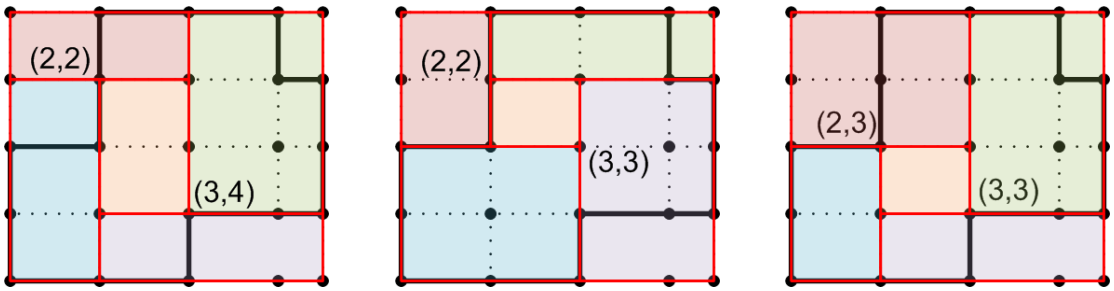


Figure 12: Three cases of a pinwheel decomposition example with [(j,k),(l,m)] equal to: (a) [(2,2),(3,4)]; (b) [(2,2),(3,3)]; (c) [(2,3), (3,4)].

the sub-problems to have natural ordering and optimal structure. The first requirement means that the solution to the problem may be given by a combination of optimal solutions for the sub-problems. For the fracturing problem this is clearly the case since combining a series of optimal fracturing solutions is equivalent to forming a union of optimal fractures.

The second requirement is that the sub-problems have a natural ordering so that solving a sub-problem requires only the solutions to the smaller sub-problems. The fracturing problem satisfies this property in that the optimal fracturing of a sub-rectangle only requires the optimal fracturing of sub-rectangles that are of smaller size. For example, the optimal solution to a sub-rectangle of size $[3 \times 3]$ requires the optimal fracturing of all smaller sub-rectangles: $\{[1 \times 1], [2 \times 1], [3 \times 1], [1 \times 2], [2 \times 2], [3 \times 2], [1 \times 3], [2 \times 3]\}$.

In Sections 4.3 and 4.4 we have described two recurrence expressions to break the fracturing problem into several smaller fracturing sub-problems. Each of these sub-problems correspond to a sub-rectangle of the basic grid. Next, we solve the sub-problems starting with the smallest so as to use the solutions of the smaller sub-problems to solve the larger sub-problems. After finding the solution to a sub-problem, we store the value in a table. This technique is called memoization.[10] When the cost of a sub-rectangle is needed again, the cost is simply looked up within the table. This limits the runtime to be the product of the number of sub-rectangles with the total number of possible decompositions for each sub-rectangle; as each are polynomial bounded with respect to the number of vertices, the two recursive algorithms can be computed in polynomial time.

4.6. Nonconvex Polygons

The two algorithms in Sections 4.3 and 4.4 are designed for Cartesian convex polygons. The second algorithm guarantees an optimal fracturing for a Cartesian convex input polygon. In general a layout polygon may not be convex. For the non-convex case we suggest two possible approaches, built upon the above algorithms.

The first approach is the inclusion of a pre-processing step that makes appropriate decisions to split each non-convex polygon into smaller convex polygons. This may be done based on rules such as selecting chords or partitioning rays that do not introduce external slivers. For the remaining convex polygons we can run either of the two proposed algorithms. However, the initial decision process may result in a sub-optimal solution.

The second approach is to simply run one of the algorithms in Sections 4.3 and 4.4 on the non-convex polygon. The algorithm still generates a fracturing but there are no longer any guarantees on optimality. In Section 5, we show results based on this approach for non-convex polygons.

5. EXPERIMENTAL RESULTS

In this section we describe simulation results based on our proposed natural recurrence algorithm of Section 4.3. We have tested our algorithm on two different test chips: an SRAM test chip with CD = 45nm, and a random logic circuit with CD = 90nm. For the SRAM test chip we apply an OPC recipe using the Calibre^{TM} edge-based OPC software to the Poly and Metal 1 layers. For all the resulting post-OPC layers we run fracturing with the sliver parameter, $\epsilon = 15nm$ and $25nm$. We run the proposed natural recurrence of Section 4.3 on the OPC

layers and set $\lambda_L = 0.04$ based on empirical observations. The resulting total rectangle count and external sliver length are compared with the Calibre^{TM} fracture package for Poly and Metal 1 layers and are shown in Tables 1 and 2,[11] respectively.

For both tables the first column contains the ϵ parameter chosen, the second column contains the number of polygons tested for each setting, and the third column contains the metric of interest, rectangle count or external sliver length. The rectangle count is the total number of rectangles and the total external sliver length is the sum of the external sliver lengths. The fourth and fifth columns compare the resulting metrics for the Calibre^{TM} fracture package with our proposed algorithm, and the sixth column shows the corresponding percentage improvement.

Table 1: Comparison of rectangle count and external sliver length between Calibre^{TM} software and the proposed algorithm for Poly layer with varying sliver parameter ϵ.

Sliver Parameter ϵ	Polygons Tested	Fracturing Metrics	Calibre^{TM} Software	Proposed Algorithm	Percentage Reduction
15nm	2102	Rectangle count	28548	28548	0
15nm	2102	Total external sliver length	2052	1692	18
25nm	2102	Rectangle count	28643	28551	12
25nm	2102	Total external sliver length	2727	2400	12

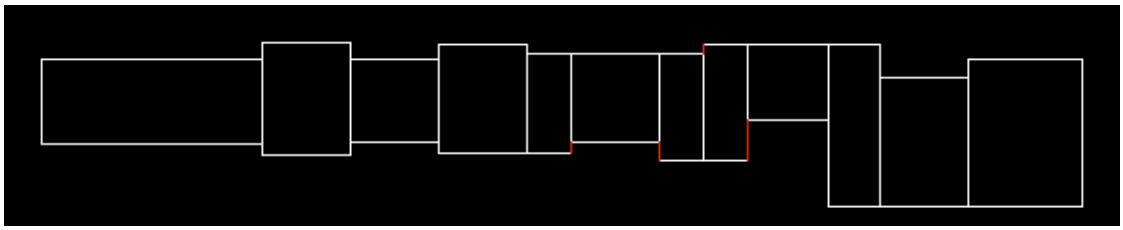
Table 2: Comparison of rectangle count and external sliver length between Calibre^{TM} software and the proposed algorithm for Metal 1 layer with varying sliver parameter ϵ.

Sliver Parameter ϵ	Polygons Tested	Fracturing Metrics	Calibre^{TM} Software	Proposed Algorithm	Percentage Reduction
15nm	1348	Rectangle count	19495	16529	15
15nm	1348	Total external sliver length	10208	7038	31
25nm	1348	Rectangle count	19661	16547	15
25nm	1348	Total external sliver length	12570	9468	25

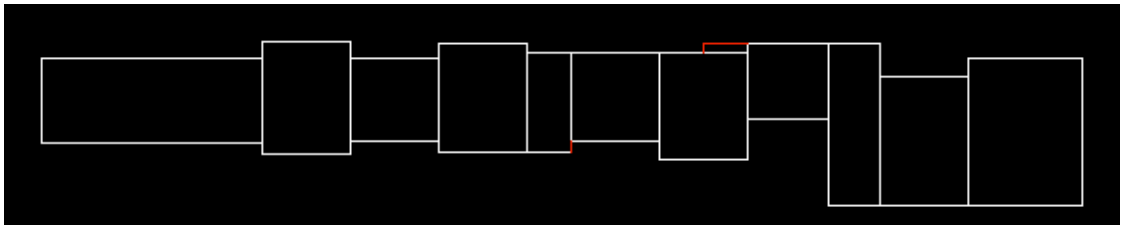
As seen, our proposed algorithm outperforms Calibre^{TM} with respect to both metrics in all cases. For the Poly layer our algorithm results in 12-18% improvement to external sliver length while keeping rectangle count constant. Meanwhile, for Metal 1 our proposed algorithm results in up to a 31% improvement in external sliver length while actually decreasing the rectangle count by up to 15%. This shows that our algorithm generates a sizable improvement to the external sliver length.

A pictorial example of the performance comparison for Poly between Calibre^{TM} and the proposed algorithm is illustrated in Fig. 13; Fig. 13(a) shows the fracture solution of Calibre ^{TM} with 12 rectangles and 43nm of external sliver length, and Fig. 13(b) depicts the fracture solution of the proposed algorithm with 12 rectangles and 35nm of external sliver length.

Fig. 14 compares Calibre^{TM} and the proposed algorithm for Metal 1 layer; Fig. 14(a) shows the fracture solution of Calibre ^{TM} with 41 rectangles and 20nm of external sliver length, and Fig. 14(b) depicts the fracture solution of the proposed algorithm with 41 rectangles and no external sliver length. In both figures, the red lines represent the external slivers.

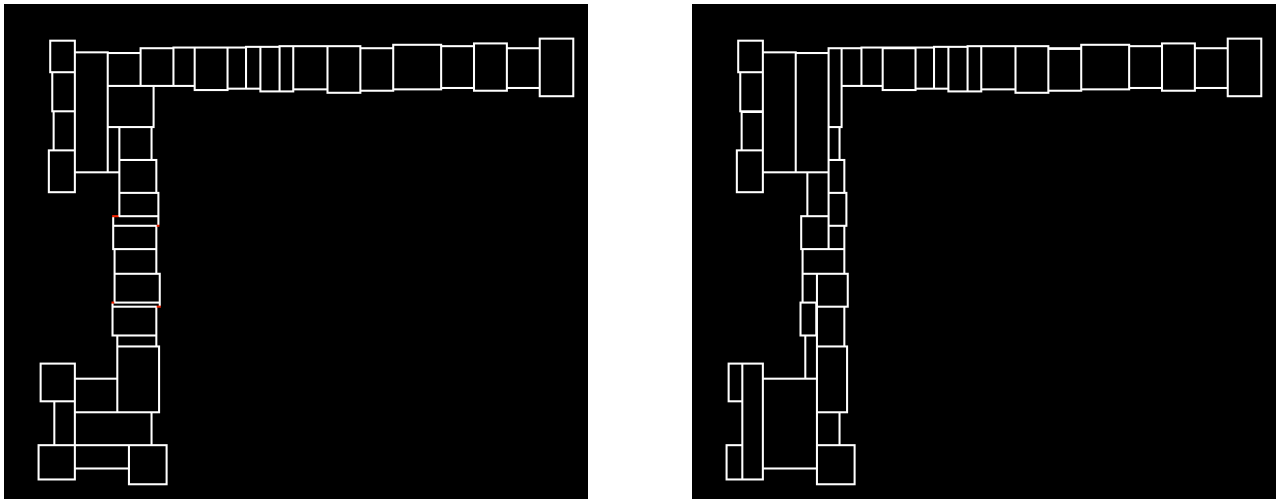


(a)



(b)

Figure 13: A performance comparison of Calibre^{TM} and the proposed algorithm for Poly: (a) Fracture solution generated by Calibre^{TM} with 12 rectangles and 43nm of external sliver length; (b) Fracture solution generated by the proposed algorithm with 12 rectangles and 35nm of external sliver length.



(a) (b)

Figure 14: A performance comparison of Calibre^{TM} and the proposed algorithm for Metal 1: (a) Fracture solution generated by Calibre^{TM} with 41 rectangles and 20nm of external sliver length; (b) Fracture solution generated by the proposed algorithm with 41 rectangles and no external sliver length.

Next, we test our algorithm on a random logic circuit with CD = 90nm with application of an OPC recipe on the Poly and Metal 1 layers. For this layout we vary the levels of OPC aggressiveness by changing the fragmentation parameter using the Calibre^{TM} edge-based OPC software. For all the resulting post-OPC layers we set $\epsilon = 25nm$, one quarter of the CD. We run the proposed natural recurrence of Section 4.3 on the OPC layers and set the weight, $\lambda_L = 0.04$, which has empirically been shown to result in good performance. The total resulting rectangle count and external sliver length for Poly and Metal 1 layers are compared with the Calibre^{TM} fracture package and shown in Tables 3 and 4.[11] The columns of Tables 3 and 4 are similar to that of Tables 1 and 2. The only difference is that column 1 of Tables 3 and 4 refer to the OPC aggressiveness rather than the sliver parameter ϵ.

Table 3: Comparison of rectangle count and external sliver length between Calibre^{TM} software and the proposed algorithm for Poly layer with varying OPC fragmentation parameter and $\epsilon = 25nm$.

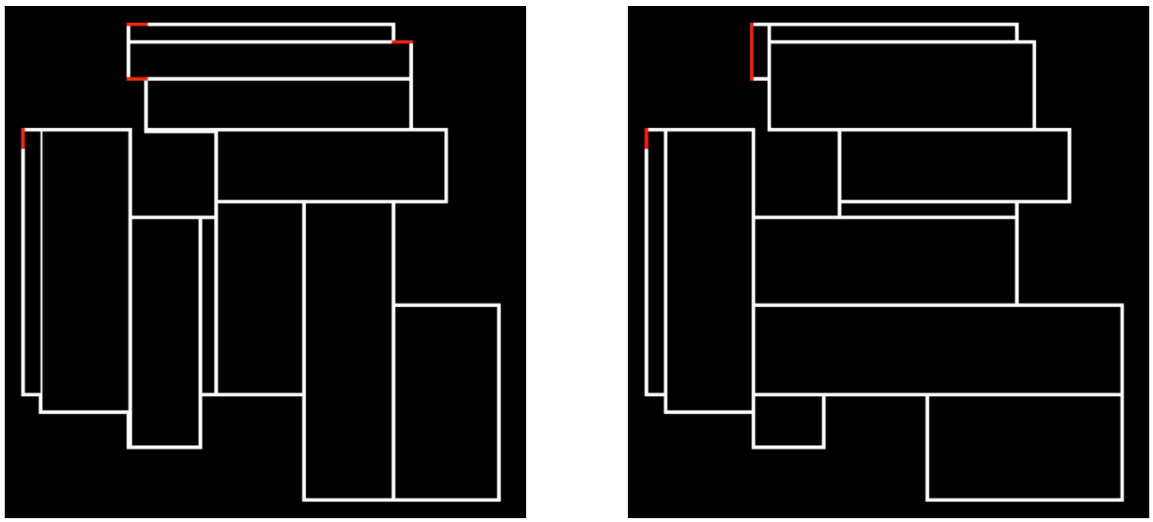
Fragmentation Parameters	Polygons Tested	Fracturing Metrics	Calibre^{TM} Software	Proposed Algorithm	Percentage Reduction
30nm	310	Rectangle count	10183	10033	1
30nm	310	Total external sliver length	7610	6610	13
50nm	325	Rectangle count	6614	6556	1
50nm	325	Total external sliver length	625	240	62
70nm	320	Rectangle count	5727	5651	1
70nm	320	Total external sliver length	240	120	50

Table 4: Comparison of rectangle count and external sliver length between Calibre^{TM} software and the proposed algorithm for Metal 1 layer with varying OPC fragmentation parameter and $\epsilon = 25nm$.

Fragmentation Parameters	Polygons Tested	Fracturing Metrics	Calibre^{TM} Software	Proposed Algorithm	Percentage Reduction
30nm	165	Rectangle count	5022	4983	1
30nm	165	Total external sliver length	1925	1460	24
50nm	389	Rectangle count	11371	11285	1
50nm	389	Total external sliver length	3680	3070	17
70nm	201	Rectangle count	6429	6363	1
70nm	201	Total external sliver length	4830	2960	39

Again, our proposed algorithm outperforms Calibre^{TM} for all cases with respect to both metrics. For the Poly layer, when the OPC fragmentation parameter is 30nm, our algorithm results in a 13% improvement to external sliver length. For larger values of the OPC fragmentation parameter, the external sliver length is decreased by 62% and 50% for the OPC fragmentation parameter set to 50nm and 70nm, respectively. For Metal 1, our proposed algorithm results in 24-39% improvement in external sliver length. In all three OPC fragmentation settings there is decreased external sliver length and polygon count.

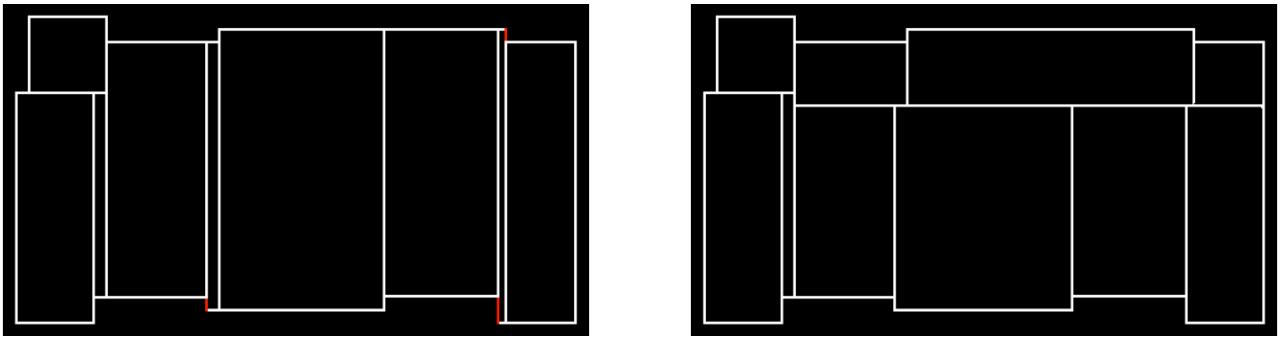
Fig. 15 shows a performance comparison between Calibre^{TM} and the proposed algorithm for Poly layer with fragmentation parameter set to 30nm; Fig. 15(a) shows the fracture solution of Calibre ^{TM} with 11 rectangles and 70nm of external sliver length, and Fig. 15(b) shows the fracture solution of the proposed algorithm with 11 rectangles and 40nm of external sliver length. A pictorial example of the Metal 1 performance comparison between Calibre^{TM} and the proposed algorithm for layer with OPC fragmentation parameter set to 30nm is illustrated in Fig. 16; Fig. 16(a) shows the fracture solution of Calibre ^{TM} with 9 rectangles and 40nm of external sliver length and Fig. 16(b) shows the fracture solution of the proposed algorithm with 10 rectangles and no external sliver length. In both cases the red lines represent the external slivers.



(a) (b)

Figure 15: A performance comparison of Calibre^{TM} and the proposed algorithm for Poly: (a) Fracture solution generated by Calibre^{TM} with 11 rectangles and 70nm of external sliver length; (b) Fracture solution generated by the proposed algorithm with 11 rectangles and 40nm of external sliver length.

Even though our algorithm has been designed to minimze sliver length and polygon count, it does not greatly increase sliver count. In fact, total sliver count has barely increased, from 651 to 718 for Poly layer and has



(a) (b)

Figure 16: A performance comparison of Calibre^{TM} and the proposed algorithm for Metal 1: (a) Fracture solution generated by Calibre^{TM} with 9 rectangles and 40nm of external sliver length; (b) Fracture solution generated by the proposed algorithm with 10 rectangles and no external sliver length.

decreased from 506 to 479 for Metal 1. This is tested for the 90nm random logic circuit for all OPC fragmentation parameters, 30nm, 50nm, and 70nm.

Finally, we test our proposed recursive algorithm against the theoretical minimum rectangle count by running it with $\lambda_L = 0$. For the theoretical result we compute the number of concave corners and maximum number of disjoint chords to obtain the theoretical minimum number of rectangles needed to properly partition a rectilinear polygon. This is done for the 90nm layout on Poly and Metal 1 layers, with the OPC fragmentation parameter set to 50nm. It is only run on the first 30 polygons for each layer. For all polygons our proposed algorithm achieves the theoretical minimum as compared with the disjoint chord method.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have developed a novel fracturing algorithm based on recursion and implemented it using dynamic programming. First, the polygon is embedded by a rectangle and all relevant points for testing fracture validity are generated. Second, we compute the cost for all sub-rectangles consisting only of interior points to the polygon. Finally, we compute the cost of the remaining sub-rectangles using recursion and memoization. We have shown that for convex rectilinear polygons, a version of the algorithm generates an optimal solution. Experimentally, the results for the non-convex case show improvement versus commercially available software tools such as Calibre^{TM}. In fact, simulations show that our algorithm significantly reduces total external sliver length by up to 31% while not impacting the rectangle count for Metal 1. In addition, by controlling λ_L, the algorithm is able to achieve the theoretical minimum rectangle count. Both of these are promising in that the algorithm can both achieve the minimum rectangle count and a decreased external sliver length by controlling a user specified parameter.

In our future work we plan to expand our cost function to better model fracturing requirements. In particular, we need to include factors such as aspect ratio of rectangles, area of rectangles, and maximum shot size. Second, we need to incorporate certain heuristics to improve runtimes and examine the resulting changes to the cost and runtime. Finally, we plan to generalize our algorithm, currently only applicable to rectangles, to incorporate trapezoids.

7. ACKNOWLEDGMENTS

This research is conducted with funding from Intel. The authors wish to acknowledge the contributions of the students, faculty, and sponsors of the Berkeley Wireless Research Center as well as the wafer fabrication donation of STMicroelectronics. We would also like to thank Yunsup Lee and Professor Krste Asanovic for providing the 90nm SRAM layouts tested in this project. Without the layouts from STMicroelectronics and Professor Krste Krste we could not have tested the algorithms proposed. Finally, we wish to thank Yuri Granik, Emile Sahouria, and Mentor Graphics for the critical help in understanding the Calibre^{TM} software package.

APPENDIX A. PROOF OF OPTIMALITY FOR THE PINWHEEL RECURRENCE

In this appendix we show that the pinwheel recurrence described in Section 4.4 is optimal for Cartesian convex polygons. First, we define a boundary segment as a line that either is a guillotine cut or part of the polygon boundary. The only possibility for a convex rectilinear polygon to be fractured without a guillotine cut is if the ray A does not terminate on a boundary segment, B. However, if B is a non-boundary segment then it must terminate at another non-boundary segment C. Similarly, C must terminate at a non-boundary segment, D. Because each of these rays involve a 90° turn, D must terminate at A. This implies only the following two options, shown in Fig. 11, exist if a convex rectilinear polygon is not fractured by boundary rays.

Consider temporarily that the rays did not form this sort of pinwheel structure. There are two possibilities for this to occur. The first possibility is if a ray, A, needs to be terminated elsewhere, shown in Fig. 17. However, as shown, ray B is no longer be part of the pinwheel structure and could be replaced with C. This means the proposed counter-example is identical to the pinwheel recurrence.

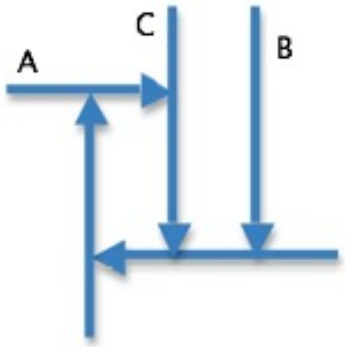


Figure 17: Possible incorrect counterexample 1.

The other possibility is shown in Fig. 18. In this case A could not exist as it does not start at a concave corner. This fact is only true for convex rectilinear polygons. The diagram implies B either passes through the vertex that A starts or goes through a point between A and the starting vertex. In the first case, one of the two rays is unneeded, as both serve to remove the same vertex while in the second case, A is no longer connected to a vertex and thus unneeded.

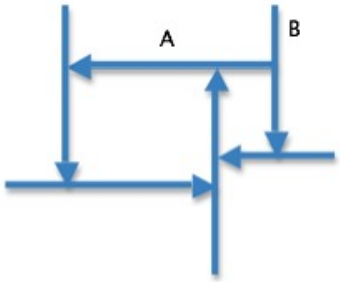


Figure 18: Possible incorrect counterexample 2.

This shows that the pinwheel recurrence is optimal for the Cartesian convex case. It can be calculated in $O(n^8)$: there are $O(n^4)$ possible sub-rectangles and each sub-rectangle has $O(n^4)$ possible decompositions. Thus to calculate the cost for all sub-rectangles requires $O(n^8)$ runtime.

REFERENCES

[1] A. B. Kahng, X. Xu, and A. Zelikovsky, "Yield- and cost- driven fracturing for variable shaped-beam mask writing," in *Proc. 24th BACUS Symposium on Photomask Technology and Management*, pp. 360–371, Sep.(2004)

[2] A. B. Kahng, X. Xu, and A. Zelikovsky, "Fast yield driven fracture for variable shaped beam mask writing" *Proc. SPIE Int. Soc. Opt. Eng. 6283*, 62832R (2006)

[3] N. Hiroomi, K. Moriizumi, K. Kamiyama, "A new figure fracturing algorithm for high quality variable-shaped EB exposure-data generation", *Proc. SPIE, Kawasaki 1996*, 2793:398-409.

[4] C. Spence, S. Goad, P. Buck, R. Gladhill, R. Cinque, J. Preuninger, U. Griesinger, M. Bloecker, "Mask data volume: historical perspective and future requirements, *Proc. SPIE Int. Soc. Opt. Eng. 6281*, 62810H (2006)

[5] M. Bloecker, R. Gladhill, P. D. Buck, M. Kempf, D. Aguilar, R. B. Cinque, "Metrics to Assess Fracture Quality for Variable Shaped Beam Lithography" *Proc. of SPIE Vol. 6349*, 63490Z (2006)

[6] T. Ohtsuki, M. Sato, M. Tachibana and S. Torii, "Minimum fracturing of composite rectangular region," *Trans. Inf. Process* **24**, pp. 647–653, (1983)

[7] X. Cheng, D. Du, J. Kim, L. Ruan, "Optimal Rectangular Partitions", *Handbook of Combinatorial Optimization.* Ed. Ding-Zhu Du and Panos M. Pardalos. Springer US, (2006). E-Book

[8] D. Kõnig, "Gráfok és alkalmazásuk a determinánsok és a halmazok elméléter" *Matematikai és Természettudományi Értesítõ* **34:** 104-119 (1916)

[9] J. E. Hopcroft, R. M. Karp, "An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs", *SIAM Journal of Computing* **2** (4) 225-231. (1973)

[10] R. Bellman, *Dynamic Programming*, Princeton University Press. (1957)

[11] http://www.mentor.com/