Practical Slicing and Non-slicing Block-Packing without Simulated Annealing

Hayward H. Chan Igor L. Markov

Department of Electrical Engineering and Computer Science
The University of Michigan, Ann Arbor, MI 48109-2122
hhchan@umich.edu, imarkov@eecs.umich.edu

ABSTRACT

We propose a new floorplanner BloBB based on multi-level branch-and-bound. It is competitive with annealers in terms of runtime and solution quality. We empirically quantify the gap between optimal slicing and non-slicing floorplans by comparing optimal packings and best seen results. Optimal slicing and non-slicing packings for *apte*, *xerox* and *hp* are reported. We also discover that the soft versions of all MCNC benchmarks, except for *apte*, and all GSRC benchmarks can be packed with zero dead-space.

Additionally, realistic floorplans often have blocks with similar dimensions, if design blocks, such as memories, are reused. We show that this greatly reduces the complexity of black-packing.

Categories and Subject Descriptors: B.7.2 [**Integrated Circuits**]: Design Aids — *placement and routing*; G.4 [**Mathematical Software**]: Algorithm Design and Analysis; J.6 [**Computer-Aided Engineering**]: Computer-Aided Design.

General Terms: Algorithm, Experimentation.

Keywords: Floorplanning, Block-packing, Slicing, Optimal, Hierarchical, Soft blocks, Evaluation, Branch-and-bound, Large-scale.

1. INTRODUCTION

Floorplanning is increasingly important to VLSI layout as a means to manage circuit complexity and deep-submicron effects. It is also used to pack dice on a wafer for low-volume and test-chip manufacturing, where all objectives and constraints are in terms of block area and shapes [11]. Abstract formulations involve blocks of arbitrary dimensions and are commonly NP-hard, but in practice many blocks have identical or similar dimensions, and designers easily find good floorplans by aligning those blocks. Annealing-based algorithms that currently dominate the field tend to ignore such shortcuts. Moreover, research is currently focused on floorplan representations rather than optimization algorithms. *Slicing* floorplans, represented by Polish expressions and slicing trees [15], are convenient, but may not capture best solutions. Non-slicing representations include sequence-pair [12] and bounded slicing grid [13], O-Tree [6], B*-Tree [4], and TCG-S [10]. Corner block list [7] and twin binary tree [17] are proposed to represent mosaic floorplans. Interestingly, many VLSI designers and EDA tools still rely on slicing representations which lead to faster algorithms and produce floorplans with hierarchical structure, more amenable to incremental changes and ECOs.

Reported optimal branch-and-bound algorithms for floorplanning [14] run out of steam at around 6 blocks, and those for placement at 8-11 blocks [2]. Their scalability can be improved through clustering at the cost of losing optimality. However, a known algorithm that minimizes area bottom-up, by iteratively refining clusters appears very slow [16]. A top-down hierarchical framework based on annealing reported in [1] is facilitated by fixed-outline floorplanning. Their implementation is faster than a flat annealer and finds better floorplans with hundreds and thousands of blocks. It is also shown that conventional annealers fail to satisfy the fixed-outline context, and new techniques are required.

We propose a deterministic bottom-up floorplanner *BloBB* based on branch-and-bound. It is faster and more scalable than flat annealers, but produces comparable results. Unlike annealers, it takes advantage of blocks with similar dimensions and can optimally pack the three smallest MCNC benchmarks. BloBB can optimize additional objectives that can be computed incrementally, such as wirelength. Unlike annealers, it runs faster with additional constraints, e.g., the fixed-outline constraint.

Since BloBB can produce optimal packings, we can empirically quantify the gap between optimal slicing and non-slicing floorplans. To this end, [5] evaluates the sub-optimality of existing floorplanners by constructing benchmarks with zero dead-space. However, most realistic examples with hard blocks cannot be packed without dead-space, so an optimal block-packer allows one to use more realistic benchmarks for evaluating sub-optimality. We also outline how one can apply our techniques to handle multi-project reticle floorplanning [11] and soft block-packing. BloBB's extension for soft blocks is able to pack the soft versions of all MCNC benchmarks, except for *apte*, and all GSRC benchmarks with zero dead-space. Hence, the benchmarks in [5] appear less attractive.

The rest of the paper is organized as follows. Necessary preliminaries are given in Section 2. Sections 3, 4 and 5 describe our optimal non-slicing, optimal slicing and hierarchical floorplanners respectively. We discuss empirical results in Section 6 and conclude in Section 7. All proofs are omitted for brevity. Proofs and more details can be found in our technical report [3].

2. PRELIMINARIES

The Rectangle Packing Problem. Let $M = \{B_1, \ldots, B_m\}$ be a set of rigid rectangular blocks. A *packing* of M defines, for every block B_i, its orientation θ_i and planar location (x_i, y_i). No two blocks may overlap. One seeks to minimize the area of the bounding box of the floorplan. In alternative formulations [1], all blocks need to fit into a given bounding box, after which other design objectives, such as wirelength, can be minimized.

Conventions. In the rest of the paper, the term *non-slicing* means "not necessarily slicing". All sets are ordered. A permutation of order n is just an ordered n-element set, typically of blocks $\{B_1, \ldots, B_n\}$. This defines a precedence relation $\prec$ on blocks, which are often referred to by indices, e.g., 2 may denote block B_2. To know the width w_B and height h_B of block B, one needs to know its orientation. Location of B means location of its bottom-left corner.

The O-tree Representation. A rooted ordered tree with $n+1$ nodes can be represented by a bit-vector of length $2n$, which records a DFS

GLSVLSI'04, April 26–28, 2004, Boston, Massachusetts, USA.

traversal of the tree. 0 and 1 record downward and upward traversals respectively (Fig.1a). An O-Tree for n blocks is a triplet (T,π,θ) where T is a bit-vector of length $2n$ specifying the tree structure, π is a permutation of order n listing the blocks as they are visited in DFS, θ is a bit-vector of length n with block orientations (0 for "not rotated" and 1 for "rotated by $\pi/2$"). (T,π,θ) represents a packing by sequencing its blocks according to π. The x-coordinate x_B of a newly-added block B is 0 if its parent P is the root of T, or else $x_P + w_P$, the sum of the width of P (implied by θ) and its x-coordinate. The y-coordinate y_B is the smallest non-negative value that prevents overlaps between B and blocks appearing before B in π (Fig.1b).

A packing is *L-compact* (*B-compact*) iff no block can be moved left (down) while other blocks are fixed. A packing is *LB-compact* iff it is both L-compact and B-compact. The packing in Fig.1b is LB-compact. Every LB-compact packing can be represented by an O-Tree, and all packings specified by an O-Tree are obviously B-compact.

The contour data structure is central to O-Tree related representations since it allows $O(n)$ time for packing realization. A *contour* of a packing is simply a contiguous sequence of line segments that describes the shape of the upper edge of the packing. Such line segments are called *contour line segments*. Fig.1c is an example. Using this data structure while realizing an O-tree, one can find the y-coordinate for each block in amortized $O(1)$ time, facilitating the realization of an O-Tree with n blocks in $O(n)$ time [6]. No known representation achieves a smaller amount of redundancy than O-Tree and hence it is suitable for our branch-and-bound block-packer.

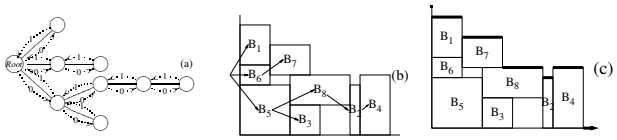


Figure 1: (a) The tree represented by $T = 0010001111001101$; (b) the packing (T,π,θ) where $\pi = \{B_5,B_3,B_8,B_2,B_4,B_6,B_7,B_1\}$; (c) the contour of U: $\{(0,0),(0,3),(3,7),(7,11),(11,12),(12,15),(15,\infty)\}$

Normalized Polish Expressions (NPEs). A *slicing floorplan* is a rectangle area recursively sliced by horizontal and/or vertical cuts into rectangular rooms [8]. A packing is *slicing* if its bounding rectangle is a slicing floorplan and each rectangular room contains exactly a block. Slicing packings can be represented by slicing trees. Each leaf node of a slicing tree represents a block and each internal node represents a horizontal or vertical cut (Fig.2). We can also consider each internal node to be a *supermodule*, consisting of the two blocks or supermodules represented by its children and merged in the way specified by itself. Given a slicing tree T, its *Polish expression* is the sequence of nodes visited in a post-order traversal of T. It is *normalized* if it does not contain consecutive $+$'s or $*$'s. For example, the expression in Fig.2c is normalized, but that in Fig.2b is not. The set of normalized Polish expressions of length $2n-1$ is in a 1-1 correspondence with the set of slicing floorplans with n blocks and hence it is non-redundant [15].

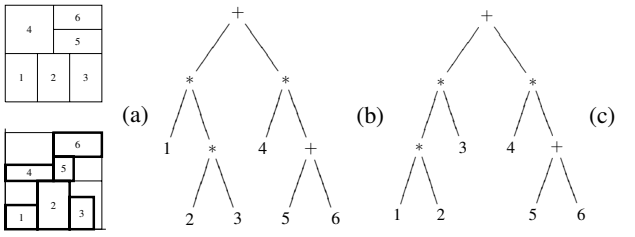


Figure 2: (a) A slicing floorplan and a slicing packing; (b) a slicing tree representing (a), its Polish expression is $123**456+*+$; (c) an equivalent slicing tree whose Polish expression is $12*3*456+*+$.

Given a slicing tree T and the orientations of the blocks, *the slicing packing of T* is a packing specified by T such that no vertical (horizontal) cuts can be moved to the left (down), and each block is placed at the bottom-left corner of the room (Fig.2a). Operators $+$ and $*$ act on the set of blocks $\{1,\ldots,n\}$ and supermodules such that $A+B$ ($A*B$) is the supermodule obtained by placing B on top of (to the right of) A. Polish expressions use the postfix notation for such operators. To evaluate a floorplan, we can simply compute the supermodule that contains all blocks by recursively merging blocks and supermodules. This procedure can be implemented in $O(n)$ time and will be explained later on.

3. OPTIMAL NON-SLICING PACKING

3.1 Branching

Branching Schedule. We adopt a branching schedule in Table 1 such that at each layer of the search tree, we define 2 bits of T, 1 block of π, or 1 bit of θ. Our basic framework is a depth-first search.

Table 1: Branching Schedule

Tree T:	1	4	7	·····	2 bits each time
Permutation π:	2	5	8	·····	1 block each time
Orientation θ:	3	6	9	·····	1 bit each time

A bit-vector identifies a rooted ordered tree iff it has equal numbers of 0's and 1's and every prefix has at least as many 0's as 1's. Hence, a partial bit-vector t with i 0's and j 1's can be extended to one representing a rooted ordered tree with n nodes iff (1) $i \geq j$ and (2) $i \leq n$. These *feasibility* conditions can be easily checked in $O(1)$ time upon every incremental change to the bit-vector. Infeasible bit-vectors are pruned, and we may get a new feasible bit-vector t at every search node of depth $4i$.

Information in a Partial Solution. Suppose (T,π,θ) is extended from (t,σ,δ). Since t has at least i 0's, the positions of all blocks in σ in T are set. Furthermore, since δ is as long as σ, the orientations of all blocks in σ are determined. The position of a block in (T,π,θ) depends only on itself and its preceding blocks in π [6]. We can then determine the locations of all blocks in σ before we explore deeper and (t,σ,δ) determines *a partial packing* (Fig.3a). By keeping a reversible contour structure that supports incremental addition and deletion, the addition and deletion of a block take amortized $O(1)$ time [6]. We say (T,π,θ) to be *extended* from (t,σ,δ) iff t, σ, and δ are prefixes of T, π, and θ respectively. It is *an extended packing* of (t,σ,δ).

3.2 Lower Bounds and Pruning

In subsequent discussions, we consider a partial packing $U = (t,\sigma,\delta)$ of i blocks and an extended packing (T,π,θ) of n blocks. Let m_k be the length of the shorter edge (min-edge) of block k for $k = 1\ldots n$. We do not distinguish between T and the *tree* presented by T. Similarly for t.

Minimum Bounding Rectangle. As the positions of the first i blocks are fixed, the bounding rectangle of U is fully contained in any extended packing. Thus, the bounding rectangle offers a lower bound for area.

Minimum Dead-Space. Once the position of a block B in σ is set, no block appearing after B in π whose x-span overlaps with that of B would lie below B. Therefore, all dead-space below every block in the partial packing is permanent. This is illustrated in Fig.3b.

Extended Dead-Space. Suppose the contour line segment above block B is shorter than $\min_{k\notin\sigma} m_k$ and has upper edge lower than its neighbors (e.g. B_2 in Fig.3c), then no unused block can rest on it, and the dead-space above B is permanent.

Maximum Min-Edge Estimation. Consider a block $A \notin \sigma$. In all extended packings, A is located above the contour of U. A lower bound for area can be produced by considering several alternative locations for A above the contour. Indeed, let A have orientation 0 in (T,π,θ) and x-coordinate x_A, such that x_A is between end-points of some contour line segment L. If A is moved left such that x_A is the beginning of L, its x and y coordinates do not increase. Hence the bounding rectangle of (t,σ,δ) with A in that location is not greater than that of (T,π,θ)

(Fig.4). Therefore, we only have to consider the cases for each contour line segment (even fewer cases need to be considered as shown in Fig.4c). The minimum of areas of all such rectangles, a_0, is a lower bound for area of complete packings with A having orientation 0. A similar lower bound a_1 corresponds to orientation 1, and leads to a lower bound $\min(a_0, a_1)$. As a trade-off between the pruning ratio and immediate computational overhead, we only consider the block whose shorter edge is $\max_{k \notin \sigma}\{m_k\}$.

Minimum Min-Edge Estimation. If t has j 0's and σ has i blocks, then $j \geq i$. If $j > i$, then we can locate the next $(j-i)$ unused blocks in T. We define *the minimum square of* σ as a square with side $\min_{k \notin \sigma}\{m_k\}$. A lower bound for area can be computed by placing $(j-i)$ minimum squares onto the partial packing according the locations specified by t (Fig.5).

LB-Compactness and O-Tree Redundancy. Some packings represented by O-Trees are not L-compact, and some of them can be specified by multiple O-Trees. To prune such O-trees we require that the y-span of each block overlap with that of its parent. Moreover, if B has overlapping y-span with multiple adjacent blocks in the left, then we require the parent of B to be the lowest of these. For example in Fig.1b, we require B_7 to have parent B_6 instead of B_1.

Dominance. The bounding rectangle of a packing can be in one of eight orientations. It suffices to analyze only one of those orientations. We formalize the notions of corner as follows. In the packing U, a block is *lower-left* iff (i) no blocks lying below have an overlapping x-span, and (ii) no blocks lying on the left have an overlapping y-span. Similarly for *lower-right*, *upper-left* and *upper-right*. A block is *a corner block* if it is one of the above. In Fig.1b, B_5 is lower-left, B_1 is upper-left, B_4 is lower-right, B_4 and B_7 are upper-right.

To facilitate pruning, observe that an LB-compact packing always contains unique lower-left, lower-right and upper-left blocks, and at least one upper-right block. We declare the rightmost upper-right block to be *the upper-right block*. In Fig.1b, B_4 is the upper-right block. To avoid dominated packings, we impose dominance-breaking constraints:

(1) the lower-left block $B_{lower-left}$ has orientation 0,

(2) $B_{lower-left} \preceq R$ for every corner block R.

It can be shown that one can transform any packing to one satisfying (1-2) without increasing area. Fig.6a-d show an example. Let $M_\sigma = \max_{k \notin \sigma}\{k\}$ and I_{lr} be the index of the current lower-right block. The

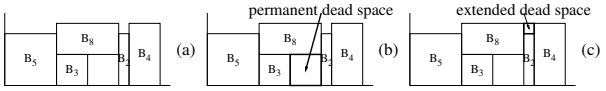


Figure 3: (a) A partial packing (t, σ, δ) with $t = 0010001111$ and $\sigma = \{B_5, B_3, B_8, B_2, B_4\}$. Fig.1b shows a compatible complete packing; (b) Every block whose x-span intersects with that of B_8 lies above B_8, hence the shown dead-space is permanent; (c) the dead-space shown is permanent since unused blocks cannot rest on B_2.

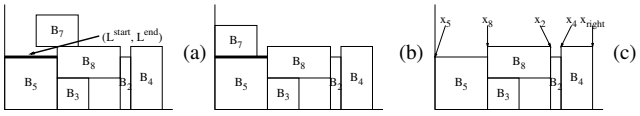


Figure 4: (a) $L^{start} \leq x_7 < L^{end}$, (b) enforcing $x_7 = L^{start}$ does not increase coordinates of B_7, (c) a lower bound can be computed from x-coordinates shown; x_8 can be ignored because the upper edge of B_5 is lower than that of B_8, and so can be x_4.

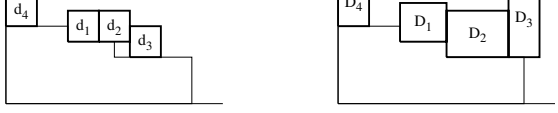


Figure 5: If the locations of next 4 blocks in t are known, we place minimum squares d_i according to t; d_i occupies the same position in t as D_i.

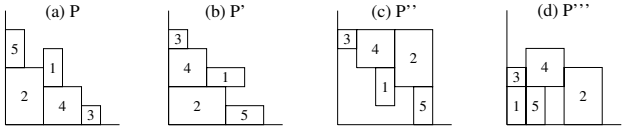


Figure 6: The packing P satisfies (3.2) but not (3.1). When we apply an α-transformation to get P', P' does not satisfy (3.2) anymore. Thus we apply a β-transformation to get P''' by flipping P' to P'' and then compacting to P'''.

index of the lower-right block is at most $I = \max(I_{lr}, M_\sigma)$. Since lower-left block in the partial packing must remain the lower-left block in any of its extended packings, we require $B_{bottom-left} \preceq B_I$. Similarly for upper-left and upper-right blocks.

Blocks with Same Height or Width. If two adjacent blocks B and B' have the same height and y-coordinate, the cluster formed by B and B' can be flipped. We break this symmetry by requiring $B \prec B'$ if B is to the left of B', and similarly, for adjacent blocks with same width and x-coordinate, e.g., B_1 and B_6 in Fig.1b. If two blocks B_i and B_j in π have the same width and height $(i < j)$, they are interchangeable and we require B_i to appear first in σ. These constraints are compatible with constraints (1-2).

4. OPTIMAL SLICING PACKING

4.1 Branching

Branching Schedule. A slicing packing of n blocks can be specified by (P, θ) where P is a Polish expression of length $2n-1$ and θ is a bit-vector of length n, storing the orientations of the blocks as described in Section 2. We maintain a growing Polish expression p and bit-vector δ.

Table 2: Branching schedule towards $(124*5+,0111)$

expression p:	1	3	5	7	8	10
orientation δ:	2	4	6		9	

We explore symbols of p one by one. If a given symbol is an operand, we explore a bit of δ, otherwise another symbol of p is explored (Table 2). We use the following characterization of Polish expression [15]. A sequence p over $\{1, \ldots, n, +, *\}$ of length $m \leq 2n-1$ can be extended to a normalized Polish expression iff (1) for every $i = 1, \ldots, n$, i appears at most once in p, (2) p has more operands than operators and (3) there are no consecutive $+$'s and $*$'s in p. The above sequences are called *partial Polish expressions*, and can be tested for in $O(1)$ time per incremental change.

Information in a Partial Solution. We maintain a series of blocks and supermodules using two stacks: *the bundle* and *the storage*.

When we push an operand and its orientation to p and δ respectively, we push the respective block (with width and height specified) into the bundle stack. When we push an operator α to p, we are guaranteed to have at least two blocks or supermodules in the bundle. We pop the two top-most blocks in the bundle, A and B, and push them in this order into the storage. We compute the supermodule formed by merging A and B in the way specified by α. When we pop an operand b and its orientation from p and θ respectively, we pop the top element of the bundle, which is necessarily b. When we pop an operator α from p, we pop the top element of the bundle, and push the two top-most blocks or supermodules from the storage to the bundle (Fig.7).

During incremental changes to p and δ, stack updates take $O(1)$ time. When we reach a leaf of the search tree, the supermodule in the bundle is the bounding rectangle specified by a complete solution (P, θ).

expression:	1	2		1	2	4		1	2	4	*
bundle:	B_1	B_2		B_1	B_2	B_4		B_1	M_{24*}		
storage:								B_4	B_2		
	(a)				(b)				(c)		

Figure 7: (a) The original configuration; (b) adding 4 to (a); (c) adding $*$ to (b); removing $*$ from (c) yields (b); removing 4 from (b) yields (a).

4.2 Lower Bounds and Pruning

For two supermodules (or blocks) M and N, we define $M \prec N$ if $B_M \prec B_N$ where B_M and B_N are the bottom-left blocks of M and N respectively. For two supermodules (or blocks) A and B, we define $A+B$ as the supermodule formed by placing B on top of A, and $A*B$ as that formed by placing B in the right of A. When we consider two partial Polish expressions, we implicitly assume that they are associated with the same bit-vector δ and hence represent two packings.

Minimum Dead-Space. The rectangles $A+B$ and $A*B$ cannot be changed after A and B are merged. Therefore, the dead-space inside $A+B$ and $A*B$ is permanent. This is illustrated in Fig.8a.

Extended Dead-Space. Let $R_1,\ldots,R_m$ be in the bundle where R_1 is at the bottom, and R_m is at the top and $m \geq 2$. The next block or supermodule M_{m-1} that R_{m-1} merges with must contain R_m. Hence the width and height of M_{m-1} are not greater than those of R_m respectively.

Similarly, $\forall\ i = 1\ldots m-1$, the next block M_i that R_i merges with must contain $R_{i+1}\ldots R_m$. Hence the width of M_i is not smaller than the maximum of widths of R_j for $j = i+1\ldots m$. Similarly for its height. In cases when both the width and height of R_i are smaller than those of M_i, we can lower-bound the dead-space when R_i merges with M_i (Fig.8b).

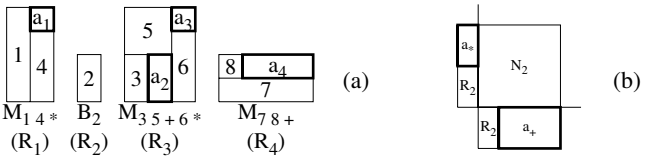


Figure 8: (a) The bundle for $14*235+6*78+$ with regions of permanent dead-space a_1, a_2, a_3 and a_4; (b) when R_2 is merged with M_2, M_2 must contain R_3 and R_4 and hence N_2; a_+ (a_*) is a lower bound for dead-space in R_2+M_2 (R_2*M_2) and hence $\min(a_+,a_*)$ is a lower bound for dead-space.

Commutativity. $A+M$ is equivalent to $M+A$, and $A*M$ to $M*A$. To break this symmetry when merging supermodules A and M, one can require $A \prec M$. We propose a better pruning mechanism below.

Suppose we are pushing the block B to the bundle, which is not empty, with the top element A. Then B must be the bottom-left block of the next supermodule M to merge with A. Hence we require $A \prec B$, implying an ascending order of blocks and supermodules in the bundle.

Abutment. Consider blocks R_1, R_2 and R_3, where $R_1 \prec R_2 \prec R_3$. If they abut horizontally or vertically, their order does not matter. For example, $(R_1+R_3)+R_2$ is equivalent to $(R_1+R_2)+R_3$. However both arrangements pass the commutativity constraint.

For chained operators of the same kind, e.g., $(R_1+R_2)+R_3$ or $(R_1*R_2)*R_3$, we require both $R_1 \prec R_3$ and $R_2 \prec R_3$. By the commutativity constraint $R_1 \prec R_2$. Therefore we only have to check if $R_2 \prec R_3$. Since an abutment of three or more blocks must be of the form $E_1E_2+E_3+\ldots+E_i+$, the abutment constraint breaks *all* symmetries of this kind.

Global Bottom-left Block and Its Orientation. We require B_1 to be the bottom-left block of all packings. This constraint is redundant because the commutativity constraint does not allow pushing B_1 to a non-empty bundle. However we can now prune hopeless partial Polish expressions much sooner. Similar to the non-slicing case, we require the orientation of B_1 to be 0.

Identical Blocks. If blocks A and B have the same dimensions, then they are interchangeable. Since the above constraints do not break all symmetries due to identical blocks, we require in that case that A appear before B in p if $A \prec B$.

5. HIERARCHICAL SLICING PACKING

In this section our optimal slicing floorplanner is extended to a scalable hierarchical slicing floorplanner which does not necessarily produce optimal solutions. The tree-structure of slicing floorplans facilitates a divide-and-conquer approach — we group blocks into clusters and pack each cluster into a supermodule. We then pack supermodules into higher-level supermodules.

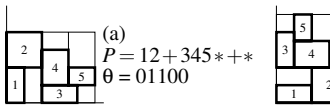


Figure 9: When the packing in (a) is flipped to (b), all operators in the Polish expression change. The packing in (a) is $(P_1\bar{P}_2+\bar{P}_3P_4P_5*+*,\theta_1\bar{\theta}_2\bar{\theta}_3\theta_4\theta_5)$ where (P_i,θ_i) or $(\bar{P}_i,\bar{\theta}_i)$ is the packing of M_i in (a).

Conquer Operations. If we flip the packing (P,θ) across a diagonal preserving the bottom-left block, the resulting packing is represented by $(\bar{P},\bar{\theta})$ where $\bar{\theta}$ is the complement of θ and $\bar{P}$ is equal to P with all pluses changed to asterisks and vice versa. This is illustrated in Fig.9.

In the rest of the paper $(\bar{P},\bar{\theta})$ denotes the flipped packing of (P,θ). We identify a supermodule by its bottom-left block, e.g., if B_2 is the bottom-left block of M, then 2 identifies B_2 and M.

Suppose we pack $\{B_1,\ldots,B_n\}$ to r supermodules $\{M_i\}$ with bottom-left blocks B_{k_i} specified by (P_i,θ_i) for $i = 1\ldots r$. We pack the r supermodules into a supermodule specified by (P,θ) (note that M_i is identified by k_i in P). Let l_i be the bit in θ that specifies the orientation of M_i. To completely specify a packing of all blocks, we substitute k_i by P_i and l_i by θ_i if $l_i = 0$, or $\bar{P}_i$ and $\bar{\theta}_i$ respectively if $l_i = 1$ (Fig.9). Note that the expanded Polish expression may not be normalized and may not satisfy all constraints in Section 4.2.

For each cluster, we find an optimal packing by branch-and-bound, subject to constraints from Section 4.2. We also limit the width and height of clusters by $L_{max} = \sqrt{A_{best}R}$, which in practice prevents supermodules with extreme aspect ratios that may not pack well at the next level. In this formula A_{best} is the area of the best packing found so far, and the constant R is termed the *aspect ratio increment*. Note that constraining aspect ratio may increase dead-space. We regulate the tradeoff between dead-space and aspect ratio by means of the *dead-space increment* constant χ. A_{best} is initialized to $A\chi$ before the first search, where A is the sum of areas of all blocks or supermodules in the cluster. If no solution is found, we increase A_{best} from $A\chi$ to $A\chi^2$ and L_{max} from $\sqrt{A_{best}R}$ to $\sqrt{A_{best}R^2}$. Such increases continue until a solution is found. We do not limit height and width at the top level of the hierarchy.

Divide Operations. While our conquer operations ensure small runtime, divide operations are responsible for solution quality. We use a greedy clustering framework from [14]. For every pair of blocks/clusters we calculate a quality metric (details below) and prioritize all pairs. The best pair is clustered if its elements have not been clustered before.

Similarity Between Blocks/Supermodules. For blocks/supermodules R_i and R_j we compute the quality metric by

$$W_{ij} = \left(\frac{\min(m_i,m_j)}{\max(m_i,m_j)}\right)^{10} + \left(\frac{\min(M_i,M_j)}{\max(M_i,M_j)}\right)^{10} \quad (1)$$

where m_i and m_j are the shorter edges (min-edges) for R_i and R_j respectively, M_i and M_j are the longer edges (max-edges) respectively. Equation (1) helps to select pairs of blocks with similar edges. Power 10 in each term emphasizes our preference for blocks with extremely similar edges, particularly useful in slicing packings. Alternatively, clustering can be based on connectivity when wirelength is minimized [14].

Similarly to Equation (1), we define the *similarity* S_{ij} of R_i and R_j by

$$S_{ij} = \left(\frac{\min(m_i,m_j)}{\max(m_i,m_j)}\right)^{2} + \left(\frac{\min(M_i,M_j)}{\max(M_i,M_j)}\right)^{2} \quad (2)$$

Clearly $0 < S_{ij} \leq 2$, and $S_{ij} = 2$ corresponds to identical blocks. We introduce the *side resolution* parameter S_{min} such that if $S_{ij} \geq S_{min}$, R_i and R_j are considered identical during branch-and-bound for symmetry-breaking purposes. In optimal packers we set $S_{min} = 2$, and smaller values trade off solution quality for better runtime.

Constraints in Cluster Formation. Suppose blocks $\{R_1,\ldots,R_r\}$ are partitioned into s clusters $C_{k_1},\ldots,C_{k_s}$. When merging clusters C_i and C_j to form a new cluster, we impose the following constraints.

(1) $t \geq \kappa^{\lfloor \log_\kappa (r-1) \rfloor}$ where κ is the *cluster base* constant and t is the number of clusters after the merger;

(2) $1 \leq |C_i| + |C_j| \leq \rho$ where ρ is the *cluster size bound*;

(3) $A_i + A_j \leq \left(\frac{A}{r}\right)\xi$ where $A_i = |C_i| A_{i,bottom-left}$, and $A_{i,bottom-left}$ is the area of the bottom-left block in C_i. Similarly for A_j. ξ is the *cluster area deviation*, and A is the total area of all blocks involved.

Constraint (1) ensures that there are enough clusters for another round of clustering. Constraint (2) limits the number of elements per cluster to guarantee that branch-and-bound finishes quickly. Constraint (3) ensures that the areas of the resulting supermodules do not differ too much. A_i is a reasonably accurate area estimate of C_i since blocks often pack into a grid-like structure. The bounds imposed in the above constraints allow our hierarchical floorplanner to adapt to problem instances.[1]

6. EXPERIMENTAL RESULTS

Our algorithms are implemented in C++ and is open-sourced under the name BloBB (Block-packing with Branch-and-Bound). BloBB and all test cases are available at [19]. All programs are compiled with `g++ 3.2.2 -O3` and evaluated on a $1.2GHz$ Linux Athlon workstation. Table 3 shows runtimes on randomly-generated test cases, which are more difficult for our block-packers as they have no symmetry and because dimensions of their blocks are all different. We use the data in Table 3 to determine the cluster base κ and the cluster size bound ρ to maximize the flexibility of the hierarchical floorplanner. Observe that dead-space (%) in optimal packings *decreases* in larger floorplans, and the difference in dead-space between slicing and non-slicing packings is within 1.5%. We also run BloBB on highly symmetrical test cases consisting of only 2 or 3 types of blocks. In these cases, the difference in area-suboptimality between slicing and non-slicing packings is within 0.3% only, and the difference decreases with block counts.[2] Since industrial examples lie between these two extremes, the average difference between slicing and non-slicing packings is expected to lie between 0.3% and 1.6%.

BloBB packs the three smallest MCNC benchmarks optimally (Fig.10 and Table 4). Such results have never been claimed before, even though solutions reported in some papers appear to be optimal. Observe that *apte* and *hp* have blocks with identical dimensions and are solved much faster than random instances of the same size.

Our hierarchical block-packer is evaluated on MCNC and larger GSRC benchmarks (Table 5). We run BloBB with the same (default) parameters for all test cases and achieve comparable results to those of Parquet [1], the TCG-S floorplanner and B*-Tree v1.0 from [18]. Parquet is a fast floorplanner based on sequence-pair, while the TCG-S floorplanner contributes many best published results for the MCNC benchmarks [10]. B*-Tree v1.0 searches in a much smaller solution space than that of our hierarchical floorplanner [17]. Based on performance results in Table 5 alone, it is difficult to claim that one floorplanner outperforms others — each floorplanner has many parameters that can be tuned further. For the MCNC and GSRC benchmarks BloBB is competitive with the TCG-S floorplanner and B*-Tree v1.0 by area, while being much faster. Notably, all competing tools produce non-slicing floorplans, while in these experiments BloBB always produces slicing floorplans, which inherits many desirable properties of slicing packings, such as simpler representation and easier incremental changes.

BloBB's adaptive nature is illustrated in Table 5, where runtime is impacted by repeated block dimensions and does not necessarily increase with block counts. To demonstrate the scalability of our floorplanner, we create the test case *n600* by merging all blocks in *n100*, *n200* and *n300*. BloBB runs faster than Parquet and B*-Tree v1.0, it also finds packings with smaller area. Fig.11 shows that most dead-space can be traced to high-level floorplans where clustering is harder. This suggests that our divide operations pack blocks into tight clusters.

7. CONCLUSIONS AND ONGOING WORK

We propose new optimal slicing and non-slicing block-packers, as well as a scalable deterministic bottom-up slicing block-packer. Our implementation BloBB is competitive with best non-slicing annealers. For small floorplans, empirical results for optimal block-packers (Table 3) confirm the perceived advantages of non-slicing floorplans. For large floorplans, data in Table 5 suggest that state-of-the-art annealers may fail to find best non-slicing floorplans reasonably quickly. Thus, slicing and hierarchical representations are competitive when runtime is limited.

While we only report results for hard blocks, we can employ the shape-curve technique from [15] for soft blocks with continuous aspect ratio. Instead of its width and height, we identify each block/supermodule by a *shape-curve*, which describes the possible dimensions it can take. Since merging blocks/supermodules corresponds to adding their shape-curves vertically or horizontally, we can apply all the techniques in Sections 4 and 5. For initial results, we pack the soft versions of all MCNC benchmarks, except *apte*, and all GSRC benchmarks with zero dead-space in less than $80s$ each. While the blocks can take aspect ratio within $[0.5,2]$, this constraint is not very restrictive. We can still achieve packings with zero dead-space in most cases when we restrict the aspect ratio to lie within $[0.59,1.70]$. The shape-curve technique can be applied to hard blocks, and we get improved experimental results. Table 6 compares BloBB's extension with MB*-Tree on the *ami49_X* benchmark suite [9]. It outperforms MB*-Tree in terms of runtime, solution quality and scalability. Fig.12 shows sample packings produced by BloBB's extension. More detailed experimental results are available in [3, 20].

Our block-packer handles additional constraints as stronger bounding criteria which often improves runtime. Fixed-outline floorplanning is an important example because annealers typically fail in this context [1]. Interestingly, our area-optimal algorithms tend to achieve aspect ratios close to 1.0 even when no fixed-outline constraints are imposed (Fig.11).

Since wirelength (HPWL) can be calculated incrementally, it can be efficiently maintained during branch-and-bound [2]. Therefore, our floorplanner can be easily extended to optimize a linear combination of wirelength and area. Alternatively, we can minimize wirelength among all min-area solutions. Another optimization strategy is to limit the wirelength by adding a constraint. We can also put highly connected blocks together during clustering.

Intriguing questions for future work include characterizing easy and difficult black-packing instances, based on block similarities. In this context our hierarchical floorplanner may be able to generate easier instances during the partitioning step. Performance may also be improved by automatically tuning key parameters at runtime.

We believe that branch-and-bound and simulated annealing can be combined in a hierarchical framework. In Fig.11, most of the dead-space results from higher-level floorplans. While our fast branch-and-bound is applied to lower-level floorplans, one may improve higher-level floorplans by simulated annealing. Another potentially useful optimization is the incremental cluster refinement algorithm from [16].

The rectangle packing problem is closely related to the 2D bin-packing problem, which has a wide range of applications such as multi-project reticle floorplanning [11]. In reticle floorplanning, slicing packings are often preferred in each reticle image, because wafers must be cut into chips by slicing lines. In BloBB, we traverse the space of slicing packings by maintaining a series of clusters. It means that any partial solution with all n blocks is a *full* slicing solution of the 2D bin-packing problem

[1] In rare cases no clusters can be formed even when $\xi > 1$. In such circumstances we recommend further increasing ξ.

[2] For example, in test cases with 10 blocks of 3 different types, optimal non-slicing packings contain 1.96% dead-space on average while slicing packings contain 2.20%. In cases with 12 blocks with 2 distinct types, optimal non-slicing packings have 0.96% dead- space while slicing packings have 1.08%.

Table 3: BloBB runtimes.

# of blocks	optimal non-slicing		optimal slicing		hierarchical	
	dead-space % / runtime (s)		dead-space % / runtime (s)		dead-space % / runtime (s)	
6	4.12	0.23	5.51	0.015	5.51	0.013
7	3.52	2.82	4.85	0.060	4.85	0.059
8	3.07	38.5	4.49	0.31	4.49	0.29
9	2.48	665	3.81	1.65	3.85	0.24
10	—	—	3.90	29.8	5.04	0.46
11	—	—	3.52	103.5	5.35	0.44
30	—	—	—	—	9.70	9.32
50	—	—	—	—	10.21	13.2
100	—	—	—	—	9.41	44.2
300	—	—	—	—	10.72	38.0
500	—	—	—	—	11.80	211.3

Average performance of BloBB on 10 randomly-generated test cases. All blocks are distinct, and their dimensions are distributed uniformly in the range 1..200. The hierarchical packer is configured with $\kappa = 8$, $\rho = 9$, $\xi = 2.00$, $R = 1.5$, $\chi = 1.5$ and $S_{min} = 1.9$.

Table 4: Optimal results for MCNC benchmarks produced by BloBB.

Test case	Block area	Non-slicing			Slicing		
		area / dead-space / runtime			area / dead-space / runtime		
apte	46.562	46.925	0.78%	2.38	46.925	0.78%	0.23
xerox	19.350	19.796	2.30%	9812	20.017	3.45%	12.8
hp	8.831	8.947	1.32%	891	9.032	2.28%	0.74

Table 5: BloBB (hierarchical) versus Parquet, TCG-S and B*-Tree v1.0.

Test case	Hierarchical		Parquet		TCG-S		B*-Tree	
	area (mm^2) / runtime (s)		area (mm^2) / runtime (s)		area (mm^2) / runtime (s)		area (mm^2) / runtime (s)	
apte	47.30	0.035	51.81	0.016	49.74	0.25	48.06	8.26
xerox	20.31	0.078	22.09	0.020	20.31	0.24	20.46	0.037
hp	9.26	0.027	9.59	0.022	9.38	0.34	11.60	25.7
ami33	1.25	1.73	1.25	0.16	1.22	4.48	1.21	14.2
ami49	38.18	3.01	38.89	0.34	38.17	18.3	36.96	15.1
n100	192234	5.62	200328	1.49	199290	143	186686	125
n100b	175263	34.7	178880	1.49	175497	144	166110	126
n200	191040	7.09	197769	6.81	198739	1286	185931	522
n200b	187824	13.34	197904	6.79	249473	847	186313	494
n300	297018	11.04	310213	16.8	324996	4889	300132	1007
n600	713775	22.3	732567	81.8	—	—	721905	3122

BloBB and Parquet are evaluated on a $1.2GHz$ Linux Athlon workstation, while TCG-S and B*-Tree v1.0 are run on $1.0GHz$ SUN Sparc workstation. Parameters of BloBB are set as in Table 3. Default parameters are used for each floorplanner. We run each of them 10 times, except that TCG-S is run once on each of GSRC benchmarks and BloBB once on all benchmarks. Minimum areas and average runtimes are reported.

Table 6: BloBB's extension, BloBB versus MB*-Tree.

Test case	BloBB's extension	BloBB	MB*-Tree
	area (mm^2) / dead-space % / runtime (s)		
ami49_40	1464 / 3.25% / 185s	1551 / 9.38% / 22s	1473 / 3.87% / 1488s
ami49_100	3652 / 3.04% / 110s	3844 / 8.45% / 9.70s	3671 / 3.57% / 3096s
ami49_200	7298 / 2.95% / 128s	7628 / 7.60% / 10.3s	7341 / 3.56% / 15372s
ami49_400	14578 / 2.54% / 414s	15633 / 10.26% / 29.1s	—
ami49_800	29251 / 3.15% / 216s	32256 / 13.75% / 42.8s	—

In test case *ami49_X*, there are 49X blocks. For example, *ami49_800* consists of 39200 blocks. BloBB and its extension are evaluated on a 1.2GHz Linux Athlon workstation while results of MB*-Tree are taken from [9], where it is evaluated on a 450MHz SUN Ultra 60 workstation.

and vice versa! To better handle the 2D bin-packing problem, BloBB's pruning can be extended with heuristics specific to bin-packing. Handling reticle floorplanning as a 2D bin-packing problem allows each reticle image to be different, and holds a potential to improve the yield.

Acknowledgments. This work was supported by the Gigascale Silicon Research Center (GSRC), an Undergraduate Summer Research Fellowship (UGSR) at the Univ. of Michigan, and an Information Technology Research (ITR) grant from the National Science Foundation (NSF). We thank Prof. Majid Sarrafzadeh from UCLA for helpful discussions.

8. REFERENCES

[1] S.N. Adya, I.L. Markov, "Fixed-outline Floorplanning: Enabling Hierarchical Design," *IEEE Trans. on VLSI* 11(6), pp. 1120-1135, 2003. http://vlsicad.eecs.umich.edu/BK/parquet/

[2] A.E. Caldwell, A.B. Kahng, and I.L. Markov, "Optimal Partitioners and End-case Placers for Standard-cell Layout," *IEEE Trans. on CAD*, 19(11), pp. 1304-1314, 2000.

[3] H.H. Chan and I.L. Markov, "Practical Slicing and Non-slicing Block-Packing without Simulated Annealing," CSE-TR-487-04, The University of Michigan, 2004.

[4] Y.-C. Chang et al., "B*-trees: A New Representation for Non-Slicing Floorplans," *DAC 2000*, pp. 458–463.

[5] J. Cong, G. Nataneli, M. Romesis, J. Shinnerl, "An Area-Optimality Study of Floorplanning," to appear in *ISPD 2004*.

[6] P.-N. Guo, C.-K. Cheng, T. Yoshimura, "An O-tree Representation of Non-Slicing Floorplan and Its Applications," *DAC 1999*, pp. 268–273.

[7] X. Hong et al., "Corner Block List: An Effective and Efficient Topological Representation of Non-Slicing Floorplan," *ICCAD 2000*, pp. 8–12.

[8] M. Lai and D. Wong, "Slicing Tree Is a Complete Floorplan Representation," *DATE 2001*, pp. 228–232.

[9] H.-C Lee, Y.-W Chang, J.-M Hsu, and H.H. Yang, "Multilevel Floorplanning/Placement for Large-Scale Modules Using B*-Trees," *DAC 2003*, pp. 812-817.

[10] J.-M. Lin and Y.-W. Chang, "TCG-S: Orthogonal Coupling of P*-admissible Representations for General Floorplans," *DAC 2002*, pp. 842–847.

[11] I. Mandoiu, "Multi-Project Reticle Floorplanning and Wafer Dicing," to appear in *ISPD 2004*.

[12] H. Murata et al., "VLSI Module Placement Based on Rectangle-Packing by the Sequence-Pair," *IEEE Trans on CAD* 15(12), pp. 1518–1524, 1996.

[13] S. Nakatake et al., "Module Placement on BSG-structure and IC Layout Applications," *ICCAD 1996*, pp. 484–491.

[14] H. Onodera, Y. Taniguchi, and K. Tamaru, "Branch-and-Bound Placement for Building Block Layout," *DAC 1991*, pp. 433–439.

[15] D. F. Wong and C. L. Liu, "A New Algorithm for Floorplan Design," *DAC 1986*, pp. 101–107.

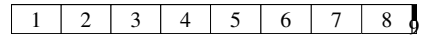
[16] J. Xu, P.-N. Guo, and C.-K. Cheng, "Cluster Refinement for Block Placement," *DAC 1997*, pp. 762–765.

[17] B. Yao et al., "Floorplan Representations: Complexity and Connections," *ACM Trans. on Design Autom. of Electronic Systems* 8(1), pp. 55–80, 2003.

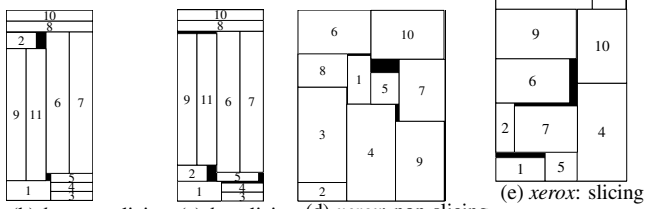
[18] http://cc.ee.ntu.edu.tw/~ywchang/research.html

[19] http://vlsicad.eecs.umich.edu/BK/BloBB/

[20] http://vlsicad.eecs.umich.edu/BK/CompaSS/

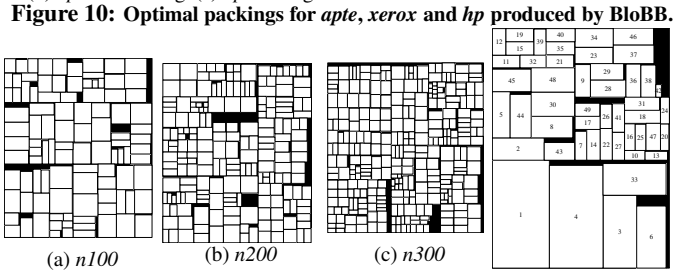


(a) *apte*: slicing and non-slicing



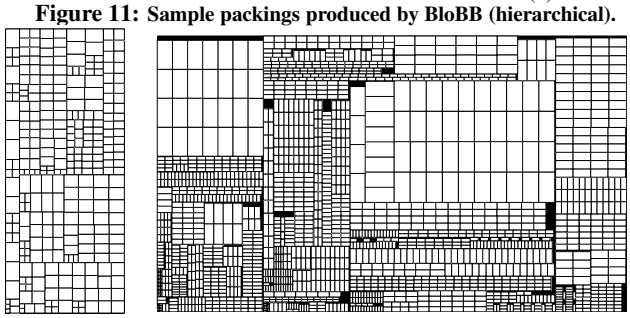
(b) *hp*: non-slicing (c) *hp*: slicing (d) *xerox*: non-slicing (e) *xerox*: slicing

Figure 10: Optimal packings for *apte*, *xerox* and *hp* produced by BloBB.



(a) *n100* (b) *n200* (c) *n300* (d) *ami49*

Figure 11: Sample packings produced by BloBB (hierarchical).



(a) *n300* (b) *ami49_40*

Figure 12: Sample packings produced by BloBB's extension. It takes (a) 82s to pack *n300* with zero dead-space and (b) 185s to pack *ami49_40* with 3.25% dead-space. *n300* consists of 300 blocks with aspect ratios within [0.63, 1.60], and *ami49_40* consists of 1960 hard blocks.