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

Hayward H. Chan and Igor L. Markov
hhchan@umich.edu, imarkov@eecs.umich.edu

May 6, 2004

Abstract

We propose a new block-packer 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. Most ongoing work deals with non-slicing packings, and implicitly assumes that best slicing packings are highly sub-optimal. Contrary to common belief, we show that the gap in optimal slicing and non-slicing packings is very small. Optimal slicing and non-slicing packings for *apte*, *xerox* and *hp* are reported.

We extend BloBB to the block-packer CompaSS, that handles soft blocks. Optimal slicing packings for soft versions of *apte*, *xerox* and *hp* are reported. We discover that the soft versions of all MCNC benchmarks, except for *apte*, and all GSRC benchmarks can be packed with zero dead-space. Moreover, the aspect ratio bound [0.5,2] turns out to be not very restrictive, when area is concerned. Our heuristic slicing block-packer is able to pack with zero dead-space in most cases when we restrict the aspect ratio bound to [0.57,1.70]. CompaSS improves the best published results for the *ami49_X* benchmarks suite, outperforming the leading multilevel annealer in runtime, solution quality and scalability.

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.

Contents

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 [10]. 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 [11] and bounded slicing grid [12], O-Tree [5], B*-Tree [3], and TCG-S [9]. Corner block list [6] 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 [13] 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 block-packer 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, [4] 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. BloBB is extended to a soft block-packer, CompaSS. Similar to BloBB, it handles large instances hierarchically and produces near-optimal packings. Despite its sub-optimality, it 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 [4] appear less attractive. Moreover, we empirically show that the aspect ratio bound $[0.5, 2.0]$ is not very restrictive by producing zero-dead-space packings under tighter aspect ratio constraints. We also outline how one can apply our techniques to handle multi-project reticle floorplanning [10].

We make the following 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. Given

a set of rigid rectangular blocks $M = \{B_1, \ldots, B_m\}$, 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. The *rectangle packing problem* is 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. As we will see, although we design block-packers for the former formulation, the algorithms can be extended to address the latter easily.

The rest of the paper is organized as follows. Sections 2, 3, 4 and 5 describe our optimal non-slicing, optimal slicing, hierarchical and soft-block packers respectively. We discuss empirical results in Section 6 and conclude in Section 7. The appendix provides proofs and more details about slicing floorplans.

2 Optimal Non-slicing Packing

In this section, we build a block-packer that explores the space of partial packings by adding and removing blocks one by one. It maintains a partial packing at a given point of time, and returns the one with the smallest bounding box as the result.

2.1 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 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 [5].

We choose the O-Tree representation because no known representation achieves a smaller amount of redundancy. A partial O-Tree defines a partial packing that can be extended to complete packings, and this property facilitates effective pruning. While B*-trees are equivalent to O-Trees in some sense, we prefer O-trees because of their convenient bit-vector representation. According to [3], the two potential disadvantages of O-Trees are (i) the varying numbers of children per node, and (ii) the use of constraint graphs to compact non-L-compact packings. However, in our work we do not explicitly track children through parents and do not use compaction.

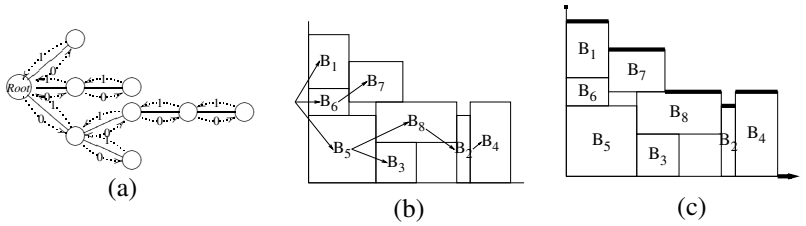


Figure 1: The O-Tree Representation.
(a) The tree represented by $T = 0010001111001101$; (b) the packing (T,π,θ) where $\pi = \{B_5,B_3,B_8,B_2,B_4,B_6,B_7,B_1\}$; (c) the contour of U: $\{(0,0),(0,3),(3,7),(7,11),(11,12),(12,15),(15,\infty)\}$

2.2 Branching

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 $4i$.

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 [5]. We can then determine the locations of all blocks in σ before we explore deeper and (t,σ,δ) determines *a partial packing* (Fig.2a). By keeping a reversible contour structure that supports incremental addition and deletion, the addition and deletion of a block take amortized $O(1)$ time [5]. 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).

2.3 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. For each partial packing, we apply the following dead-space estimations.

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.

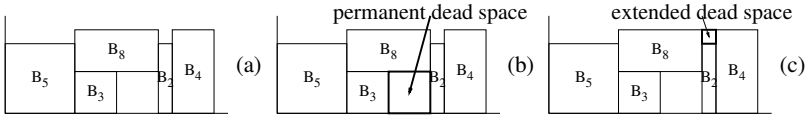


Figure 2: Minimum and Extended Dead-Space.
(a) A partial packing (t,σ,δ) with $t = 0010001111$ and $\sigma = \{B_5, B_3, B_8, B_2, B_4\}$. Fig.1b shows a compatible complete packing; (b) Every block whose x-span intersects with that of B_8 lies above B_8, hence the shown dead-space is permanent; (c) the dead-space shown is permanent since unused blocks cannot rest on B_2.

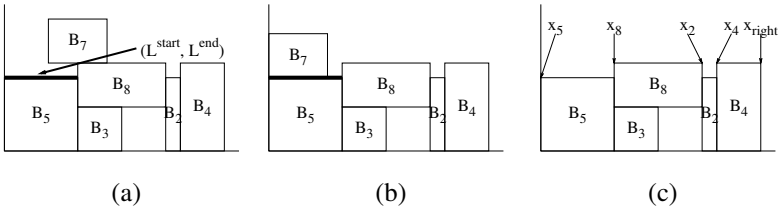


Figure 3: Maximum Min-edge Estimation.
(a) $L^{start} \leq x_7 < L^{end}$, (b) enforcing $x_7 = L^{start}$ does not increase coordinates of B_7, (c) a lower bound can be computed from x-coordinates shown; x_8 can be ignored because the upper edge of B_5 is lower than that of B_8, and so can be x_4.

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.2b.

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.2c), 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.3). Therefore, we only have to consider the cases for each contour line segment (even fewer cases need to be considered as shown in Fig.3c). 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.4). This method is justified in Appendix A.

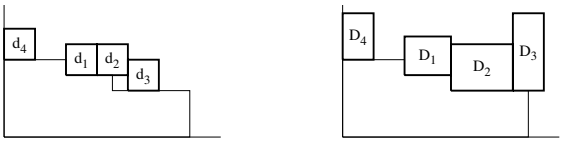


Figure 4: Minimum Min-edge Estimation.
If the locations of next 4 blocks in t are known, we place minimum squares d_i according to t; d_i occupies the same position in t as D_i.

Symmetry-breaking dramatically shrinks the size of the solution space. While symmetry-breaking is often not suitable for local search algorithms [14], it provides stronger bounding criteria for our branch-and-bound algorithm.

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. The soundness of this pruning criterion is proved in Appendix A.

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.

Appendix A proves that one can transform any packing to one satisfying (1-2) without increasing area. Fig.5a-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 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. We can impose even stronger criteria in the following special cases.

- When there are more than one block, the upper-left and lower-right blocks are distinct. Hence, we can require $B_{lower-left} \preceq S$ where S has the second largest index among the unpacked blocks.
- When the current contour is a straight line, we can flip/rotate the partial packing. Therefore, we can require $B_{lower-left}$ to have index not greater than all corner blocks of *this partial packing*.

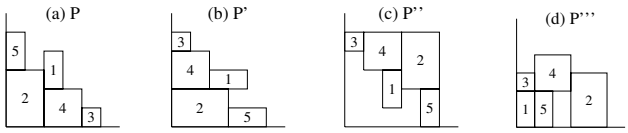


Figure 5: Dominance-Breaking.
The packing P satisfies (2.2) but not (2.1). When we apply an α-transformation to get P', P' does not satisfy (2.2) anymore. Thus we apply a β-transformation to get P''' by flipping P' to P'' and then compacting to P'''. Precise definitions for α and β-transformations are given in Appendix A.

- When there is only one block in the contour, we can flip/rotate the partial packing under that block. Similarly to the above, we require $B_{lower-left}$ to proceed all corner block of that partial packing. Moreover, we require $B_{lower-left}$ to have index smaller than the top-most block.

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). since the index of lower-left block does not grow while those of other corner blocks do not decrease after flips introduced above.

3 Optimal Slicing Packing

In this section, we build an optimal block-packer that contains slicing packings only. Unlike the non-slicing packer, it maintains a series of slicing sub-floorplans, and consistently merging or dissembling them.

3.1 Normalized Polish Expressions (NPEs)

A *slicing floorplan* is a rectangle area recursively sliced by horizontal and/or vertical cuts into rectangular rooms [7]. 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.6). 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.6c is normalized, but that in Fig.6b 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].

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.6a). 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

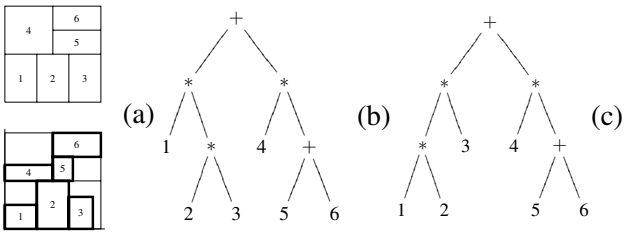


Figure 6: Slicing Floorplan, Slicing Packing, Slicing Trees and Polish Expressions.
(a) A slicing floorplan and a slicing packing; (b) a slicing tree representing (a), its Polish expression is $123**456+*+$; (c) an equivalent slicing tree whose Polish expression is $12*3*456+*+$.

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. Note that we do not keep track of the locations of the blocks throughout the search, but realize the packing after it. The realization procedure for n blocks can be implemented in $O(n)$ time (Appendix B). We compute the locations of the blocks once after the branch-and-bound process and the computation takes negligible time in practice.

3.2 Branching

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 2.1. 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.

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).

3.3 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).

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.

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: Incremental Changes with *Bundle* and *Storage*.
(a) The original configuration; (b) adding 4 to (a); (c) adding $*$ to (b); removing $*$ from (c) yields (b); removing 4 from (b) yields (a).

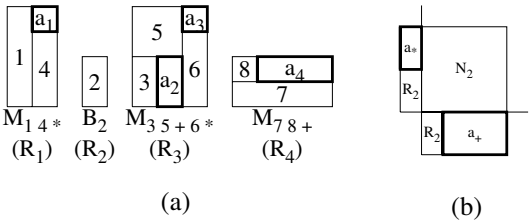


Figure 8: Minimum and Extended Dead-Space.
(a) The bundle for $14 * 235 + 6 * 78+$ with regions of permanent dead-space a_1, a_2, a_3 and a_4; (b) when R_2 is merged with M_2, M_2 must contain R_3 and R_4 and hence N_2; a_+ (a_*) is a lower bound for dead-space in $R_2 + M_2$ ($R_2 * M_2$) and hence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

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 $A \prec B$. Note that commutativity and abutment constraints do not break all symmetries by identical blocks. Fig.9 is an example

(a) $15*6*23+4*+$ (b) $14*6*23*5*+$

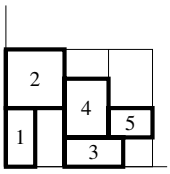
Figure 9: Symmetry with Identical Blocks.
Two normalized Polish expressions that satisfy the commutativity and abutment constraints, but that in (a) does not satisfy the symmetry-breaking constraint for identical blocks.

4 Hierarchical Slicing Packing

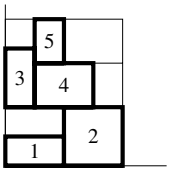
In this section our optimal slicing block-packer is extended to a scalable hierarchical slicing block-packer BloBB which does not necessarily produce optimal solutions. The tree-structure of slicing floorplans facilitates a divide-and-conquer approach — we group blocks into clusters and pack each cluster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

4.1 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.10. 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.

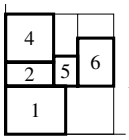


(a) $P = 12+345**+$, $\theta = 01100$

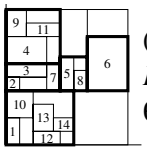


(b) $\bar{P} = 12*345+**+$, $\bar{\theta} = 10011$

Figure 10: Effect of Flipping a Slicing Packing.
When the packing in (a) is flipped to (b), all operators in the Polish expression change.



(a)
$P = 124+5*6*+$
$\theta = 01000$



(b)
$P_{expanded} = P_1\bar{P}_2P_4 + P_5 * P_6$
$\theta_{expanded} = \theta_1\bar{\theta}_2\theta_4\theta_5\theta_6$

Figure 11: Merging Clusters into a Higher-level Cluster.
Concatenation of sub-packings into a complete packing. $P_{expanded} = P_1\bar{P}_2P_4 + P_5 * P_6$ and $\theta_{expanded} = \theta_1\bar{\theta}_2\theta_4\theta_5\theta_6$ where (P_i, θ_i) or $(\bar{P}_i, \bar{\theta}_i)$ describe the packing of M_i in (a).

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.10). Note that the expanded Polish expression may not be normalized and may not satisfy all constraints in Section 3.3. Fig.11 shows an example.

For each cluster, we find an optimal packing by branch-and-bound, subject to constraints from Section 3.3. 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.

4.2 Divide Operations

While our conquer operations ensure small runtime, divide operations are responsible for solution quality. We use a greedy clustering framework from [13]. 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.

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 [13].

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

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

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.

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 block-packer to adapt to problem instances.[1]

5 Packing Soft Blocks

Slicing packing can handle soft blocks very easily. We extend BloBB to a block-packer CompaSS, that handles soft blocks. In the optimal mode, CompaSS explores the space of slicing sub-floorplans, similar to BloBB, except that it uses a curve to denote a sub-floorplan, instead of its dimensions. As we will see, using a curve allows us to consider soft blocks, and improves the quality of packing hard blocks, since each sub-floorplan can take many shapes when it merges with another.

5.1 The Shape-Curve Representation

Consider a soft block B, whose width and height can vary. The shapes that B can take are characterized by a shape-curve. Given a block/supermodule B, its *shape-curve* records the set of dimensions the bounding box of B can take. A rectangle can contain B iff its upper right corner lies above the shape-curve of B (Fig.12a). Exact shape-curves are hard to deal with, so we use piecewise linear curves to approximate them. The approximation can be made arbitrarily precise, by using more line segments, at the cost of efficiency. Given a shape-curve for block B, we can determine the

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

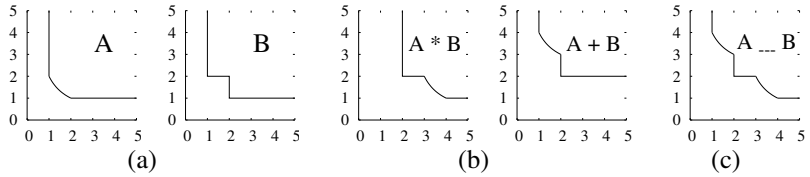


Figure 12: The Shape-Curve Representation for Soft Blocks/Supermodules.
The shape-curves of (a) a soft block A, hard block B, (b) supermodules $A*B$ and $A+B$ and (c) the supermodule $A \sqcup B$. As an example, a rectangle can contain A iff its upper right corner lies above the shape-curve of A.

dimensions of its smallest bounding rectangle (*min-box*) by examining the curve point by point. For blocks A and B, we add the shape-curves of A and B horizontally to get the shape-curve of $A*B$, and vertically for $A+B$ (Fig.12b). This simple curve-arithmetic, whose soundness is proved in Appendix A, allows efficient computation for the shape-curves of supermodules. Note that hard blocks are just a special types of soft blocks that can take 1 or 2 orientations (aspect ratios), and hence all our algorithms below can handle hard blocks, or a mix of soft and hard blocks directly.

5.2 Optimal Slicing Soft Block Packing

Given a set of blocks, specified by shape-curves, all resultant Polish expressions correspond to a unique shape-curve. We adopt the branching schedule in Section 3.2, to search for a normalized Polish expressions, whose min-box is smallest. Instead of keeping a series of rectangles, CompaSS maintains a series of shape-curves. Note that we do *not* need the orientation bit-vector, since the orientation information is captured in the shape-curve. We use the same data structure (buffer and storage) to keep track of the merging and dissembling information. It supports $O(d)$ time merging and $O(1)$ time dissembling of shape-curves, where d is the number of line segments in the shape-curve of the merged supermodule. For a block or supermodule M, the min-box of its shape-curve gives a lower bound in area. Similar to packing hard blocks, the shape-curves of $A+B$ and $B+A$ are the same. Therefore, we can apply the commutativity and abutment constraints to break symmetries.

5.3 Hierarchical Soft Block Packing

Similarly to BloBB, CompaSS groups blocks into clusters and pack each cluster into a supermodule. The supermodules are packed into higher-level supermodules, until a single supermodule is formed. While BloBB records the dimensions of the clusters, CompaSS records their shape-curves. It allows CompaSS to have many alternatives to fit that cluster into higher-level packings. We employ the techniques in Section 4.2 to group blocks/supermodules into clusters. For each cluster, we look for an optimal packing, subject to adjustable constraints. The details about the parameters can be found CompaSS' manual, available at [20].

5.4 Post-processing

Given a Polish expression, it turns out that we can reassign the operators to minimize the area of the min-box very efficiently. For example for the Polish expression $AB+C*$, we first identify the operator slots, and get $AB \sqcup C \sqcup$. We can determine whether we should assign $+$ or $*$ to each of the

$\sqcup$ slots to get a possibly smaller packing. The effect of this technique is more pronounced when there are more blocks.

Consider blocks/supermodules A and B, with shape-curves C_A and C_B, we define $A \vee B$ to be the supermodule that has shape-curve $C_{A \vee B}$, where the region above $C_{A \vee B}$ is the union of the regions above C_A and C_B (Fig.12c). $A \vee B$ is the supermodule that can either take some shape of A, or some shape of B. We define $A \sqcup B$ to be $(A + B) \vee (A * B)$, or in postfix notation, $AB\sqcup = AB + AB * \vee$. Given a Polish expression P, we can substitute all the operators with $\sqcup$. The min-box of the resultant curve gives the smallest area, among all Polish expressions with operators in the same positions as P. To determine the locations of the blocks, we reassign $+$ and $*$'s to each $\sqcup$ slots. Appendix B provides more details in the realization algorithm.

6 Experimental Results

Our algorithms are implemented in C++ and are open-sourced under the names BloBB (Block-packing with Branch-and-Bound) and CompaSS (Compacting Soft and Slicing Packings). BloBB is available at [19] and CompaSS at [20]. All programs are compiled with `g++ 3.2.2 -O3` and evaluated on a 1.2GHz Linux Athlon workstation. All parameters and runtime summaries about BloBB can be found at [19], while those about CompaSS at [20]. *Dead-space %* refers to the ratio of amount of dead-space to the total area of the blocks.

6.1 Optimal Block-Packing

We evaluate BloBB on a suite of randomly-generated test cases. In the randomly-generated test cases, the blocks have integer dimensions distributed uniformly in the range 1..200. All blocks are distinct to eliminate the effect of instance-specific symmetries. We use these cases to simulate the worse-case that BloBB is responsible for, since in realistic instances, the blocks tend to have similar dimensions and not have extreme aspect ratios (especially in small instances with less than 20 blocks). On the other hand, we construct highly symmetrical instances in which there are only 2 or 3 types of blocks. We empirical show that the existence of identical blocks improve the quality of the optimal slicing or non-slicing packing. Table 3 shows the average dead-space % in optimal slicing and non-slicing packings and BloBB's runtime. From Table 3, we observe that:

- Presence of identical blocks significantly improves the quality of the optimal slicing and non-slicing packings.
- In each test cases suite (random, 3-type and 2-type), the deadspace % *decreases* with block counts. It occurs in both optimal slicing and non-slicing packings, and their gap is small. In all cases, the average differences in dead-space % of optimal slicing and non-slicing packings are no more than 1.5%. It contradicts with common belief that slicing packings are highly suboptimal for packing hard blocks.
- In test cases of the same number of blocks, the gap between optimal slicing and non-slicing packings *decreases* when there are more identical blocks. For example, in the 9-block random instances, the average gap between slicing and non-slicing packings is 1.33%, that in 9-block 3-type instances is 0.26% and that in 9-block 2-type instances is 0.23%.

BloBB packs the three smallest MCNC benchmarks optimally (Table 5 and Fig.13). 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. The gaps between optimal slicing and non-slicing

packings are less than 1.2% in all cases.

We also run CompaSS of the soft versions of the MCNC benchmarks (Table 6), where all the blocks have aspect ratio within $[0.5, 2]$. We find the optimal slicing solutions for *apte*, *xerox* and *hp*. Interestingly, CompaSS is able to find solutions with zero dead-space for *ami33* and *ami49* respectively. Therefore, we optimally pack *xerox*, *hp*, *ami33* and *ami49* subject to the aspect ratio constraint.

6.2 Hierarchical Block-Packing

BloBB and CompaSS are evaluated on MCNC and larger GSRC benchmarks (Table 7). The parameters for CompaSS depend on the number of blocks, and since CompaSS is designed to handle a large variety of instances, with block counts ranging from 10 to 100K, it is difficult and impractical to set certain parameters of CompaSS as default. The parameters for CompaSS in each cases are available at [20]. We, however, run BloBB with the same (default) parameters for all test cases. BloBB and CompaSS 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 [9]. B*-Tree v1.0 searches in a much smaller solution space than that of our hierarchical block-packer [17]. Based on performance results in Table 7, 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.

The adaptive nature of BloBB and CompaSS is illustrated in Table 7, where their runtimes are impacted by repeated block dimensions and do not necessarily increase with block counts. To demonstrate the scalability of our block-packers, 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. On the other hand, CompaSS produces better packings than BloBB, at the cost of runtime. In packings produced by BloBB, most dead-space can be traced to high-level floorplans where clustering is harder (Fig.14). This suggests that BloBB's divide operations pack blocks into tight clusters. On the other hand, there are less dead-space is resulted from high-level floorplans by CompaSS, since for each cluster, BloBB keeps track of its width and height while CompaSS keeps track of its shape-curve, that records many possible packing for the cluster. Therefore when CompaSS perform higher level packing, it has a lot of alternatives to choose from for each cluster. It explains why CompaSS appears more competent than BloBB when dead-space is concerned.

For large scale block-packing (up to 40K blocks), we compare BloBB and CompaSS with MB*-Tree [8], since other block-packers above are designed to handle smaller instances (up to 500 blocks). We evaluate BloBB and CompaSS on the *ami49_X* benchmarks proposed in [8]. Each instances consists of copies of *ami49*, for example *ami49_40* consists of 40 copies of *ami49*. Table 8 shows our results for BloBB and CompaSS with those reported in [8]. CompaSS finds solutions with smaller area more quickly than MB*-Tree. It confirms that the slicing packings are competitive with non-slicing packings when the number of blocks is large. Typical multi-level block-packers lose either solution quality (e.g. BloBB) or speed (e.g. MB*-Tree) when the number of blocks increases. Interestingly, CompaSS is robust in both solution quality and runtime with respect to the number of blocks. It is because CompaSS takes advantage of the size of the prob-

lem. When there are more blocks, CompaSS produces better low-level floorplans since the blocks within each cluster tend to be very similar, and produces better high-level floorplans since it keeps track of many alternatives for each cluster. The runtime does not exhibit dramatic increase with block count since we can achieve good dead-space ratio with fewer blocks (lower-level clusters) in each cluster. It simplifies the packing problem at each level considerably, and offsets the effect that the shape-curves have more points. Moreover, the post-processing of optimizing the operator is more effective and there are more blocks. Fig.14 shows some of the results by CompaSS.

CompaSS' is also evaluated on soft version of the MCNC and GSRC benchmarks. Table 9 shows that CompaSS is able to produce zero-dead-space packings in most cases, where the blocks have aspect ratio within $[0.5, 2]$. Therefore, the dead-space produced by any floorplanner in these cases reveal its sub-optimality in area. Moreover, we show that this constraint in aspect ratio is not restrictive by providing zero-dead-space packings subject to stricter aspect ratio constraints. Most cases can be packed with zero dead-space if we restrict the aspect ratio to lie within $[0.63, 1.60]$. It suggests that the problem of packing soft blocks is easier than that of packing hard blocks.

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, and extend it to one that handles soft blocks. Our implementations BloBB and CompaSS are 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 7 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.

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.14). In general, new features and constraints may increase or decrease the efficiency of symmetry-breaking.

Since wirelength (HPWL) can be calculated incrementally, it can be efficiently maintained during branch-and-bound [2]. Therefore, our block-packer 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 block-packers may be able to generate easier instances during the partitioning step. Performance may also be improved by the following.

- Key parameters can be tuned statically and/or dynamically. Before packing, the block-packers can set the global parameters after considering the information of the input, such as number of blocks, and dimensions of the blocks. For example, a smaller cluster-base κ can be used if there are many blocks (such as more than 1000), since each high-level cluster can take many possible shapes. Parameters can also change dynamically at runtime. For example, a large cluster-base κ can be used in lower-level packings to ensure quality, and a small cluster-base can be used in higher-level packings for efficiency.
- CompaSS uses shape-curves to represent a cluster. The sizes of the shape-curves grow as more

Table 3: BloBB runtimes.

	optimal non-slicing			optimal slicing			hierarchical
	random	3 block-types	2 block-types	random	3 block-types	2 block-types	random
# blks	dead space % / runtime (s)						
6	4.12% / 0.24s	2.72% / 0.043s	1.88% / 0.014s	5.51% / 0.015s	3.63% / 0.009s	2.48% / 0.002s	5.51% / 0.013s
7	3.52% / 2.25s	2.16% / 0.19s	1.20% / 0.030s	4.85% / 0.057s	2.55% / 0.014s	1.32% / 0.009s	4.85% / 0.059s
8	3.07% / 38.4s	3.02% / 1.35s	1.10% / 0.20s	4.49% / 0.29s	3.30% / 0.068s	1.30% / 0.026s	4.49% / 0.29s
9	2.48% / 664s	1.89% / 8.06s	1.68% / 1.19s	3.81% / 1.54s	2.05% / 0.16s	1.91% / 0.15s	3.85% / 0.24s
10	—	1.96% / 46.9s	1.74% / 4.20s	3.90% / 28.0s	2.20% / 0.88s	1.99% / 0.45s	5.04% / 0.46s
11	—	—	0.91% / 19.3s	3.52% / 96.2s	1.68% / 6.49s	1.08% / 1.09s	5.35% / 0.44s
12	—	—	0.96% / 83.7s	3.16% / 545s	2.22% / 12.9s	1.08% / 2.85s	
13	—	—	—	—	2.13% / 30.9s	1.52% / 17.9s	
14	—	—	—	—	1.94% / 131s	2.39% / 46.4s	
15	—	—	—	—	1.87% / 617s	0.94% / 63.0s	
16	—	—	—	—	—	1.29% / 309s	
50	—	—	—	—	—	—	10.21% / 13.2s
100	—	—	—	—	—	—	9.41% / 44.2s
300	—	—	—	—	—	—	10.72% / 38.0s
500	—	—	—	—	—	—	11.80% / 211.3s

Average performance of BloBB on 10 randomly-generated test cases. The dimensions are distributed uniformly in the range 1..200. All blocks in random test cases are distinct and the number of blocks in k-block-type test cases are as close to each other as possible. 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: Gap in Optimal Slicing and Non-slicing Packings.

(a) Random test cases. (b) 3-type test cases. (c) 2-type test cases.

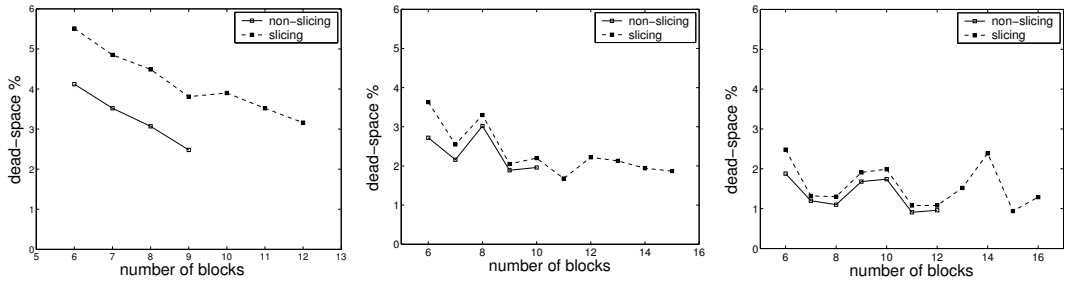


Table 5: Results for MCNC benchmarks by BloBB.

Test case	Block area	Optimal non-slicing			Optimal slicing		
		area / dead-space / runtime			area / dead-space / runtime		
apte	46.56	**46.93**	**0.78%**	2.38s	**46.93**	**0.78%**	0.23s
xerox	19.35	**19.80**	**2.30%**	9812s	**20.02**	**3.45%**	12.8s
hp	8.831	**8.947**	**1.32%**	891s	**9.032**	**2.28%**	0.74s

Table 6: Results for soft version of the MCNC benchmarks by CompaSS.

Test case	# of blocks	Optimal slicing			Hierarchical slicing		
		area / dead-space / runtime			area / dead-space / runtime		
apte	9	**46.91**	**0.75%**	0.96s	**46.91**	**0.75%**	11.79s
xerox	10	19.35	0.00%	13.79s	19.35	0.00%	1.47s
hp	11	8.831	0.00%	6.94s	8.831	0.00%	3.58s
ami33	33	—	—	—	**1.156**	**0.00%**	2.76s
ami49	49	—	—	—	**35.45**	**0.00%**	4.49s

All soft blocks have aspect ratio within [0.5, 2.0]. Boldfaced results have never been claimed before. The dead-space % are accurate up to 0.01%. Parameters in each case can be found in [20].

Table 7: CompaSS and BloBB versus Parquet, TCG-S and B*-Tree v1.0.

Test case	CompaSS		BloBB		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)		area (mm^2) / runtime (s)	
apte	46.92	0.020	47.30	0.035	51.81	0.016	49.74	0.25	48.06	8.26
xerox	20.11	0.41	20.31	0.078	22.09	0.020	20.31	0.24	20.46	0.037
hp	9.03	1.27	9.26	0.027	9.59	0.022	9.38	0.34	11.60	25.7
ami33	1.21	9.74	1.25	1.73	1.25	0.16	1.22	4.48	1.21	14.2
ami49	37.17	9.96	38.18	3.01	38.89	0.34	38.17	18.3	36.96	15.1
n100	192647	6.61	192234	5.62	200328	1.49	199290	143	186686	125
n100b	171633	7.19	175263	34.7	178880	1.49	175497	144	166110	126
n200	187074	17.39	191040	7.09	197769	6.81	198739	1286	185931	522
n200b	186570	21.87	187824	13.34	197904	6.79	249473	847	186313	494
n300	291796	11.63	297018	11.04	310213	16.8	324996	4889	300132	1007
n600	671818	151	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, and parameters of CompaSS in each test case can be found in [20]. Default parameters are used in Parquet, TCG-S and B*-Tree v1.0. We run each of them 10 times, except that TCG-S is run once on each of GSRC benchmarks, CompaSS and BloBB once on all benchmarks. For the annealers, minimum areas and average runtimes are reported.

Table 8: CompaSS, BloBB versus MB*-Tree.

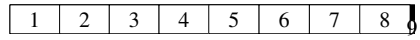
Test case	# of blocks	CompaSS	BloBB	MB*-Tree
		area (mm^2) / dead-space % / runtime (s)		
ami49_40	1960	1464 / 3.25% / 185s	1551 / 9.38% / 22s	1473 / 3.87% / 1488s
ami49_100	4900	3652 / 3.04% / 110s	3844 / 8.45% / 9.70s	3671 / 3.57% / 3096s
ami49_200	9800	7298 / 2.95% / 128s	7628 / 7.60% / 10.3s	7341 / 3.56% / 15372s
ami49_400	19600	14578 / 2.54% / 414s	15633 / 10.26% / 29.1s	—
ami49_800	39200	29251 / 3.15% / 216s	32256 / 13.75% / 42.8s	—
ami49_1600	78400	58678 / 3.47% / 330s	64712 / 14.11% / 108s	—
ami49_3200	156800	113429 / 3.34% / 682s	128930 / 13.64% / 371s	—

In test case *ami49_X*, there are 49X blocks. For example, *ami49_800* consists of 39200 blocks. CompaSS and BloBB are evaluated on a 1.2GHz Linux Athlon workstation while results of MB*-Tree are taken from [8], where it is evaluated on a 450MHz SUN Ultra 60 workstation.

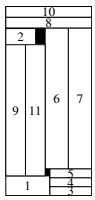
Table 9: CompaSS on soft versions of the MCNC and large GSRC benchmarks.

Aspect ratio	[0.5,2.0]	[0.55, 1.8]	[0.59, 1.7]	[0.63, 1.6]	[0.67, 1.5]
Test case	dead-space % / runtime (s)				
apte	0.76% / 0.07s	0.88% / 0.09s	0.97% / 0.08s	1.07% / 0.09s	1.18% / 0.10s
xerox	0.00% / 0.10s	0.00% / 0.10s	0.00% / 0.10s	0.00% / 0.11s	**0.00% / 0.10s**
hp	0.00% / 0.09s	0.00% / 0.05s	0.00% / 0.04s	0.00% / 0.05s	**0.00% / 0.05s**
ami33	0.00% / 2.38s	0.00% / 2.73s	0.00% / 2.79s	0.00% / 2.93s	**0.00% / 2.85s**
ami49	0.00% / 4.64s	**0.00% / 3.46s**	0.01% / 2.53s	0.04% / 2.91s	0.20% / 4.53s
n100	0.00% / 17.6s	0.00% / 20.76s	0.00% / 20.7s	0.01% / 24.0s	**0.00% / 31.7s**
n100b	0.00% / 13.2s	0.00% / 15.7s	0.00% / 20.2s	0.00% / 22.6s	**0.00% / 28.3s**
n200	0.00% / 43.4s	0.00% / 43.3s	0.00% / 50.4s	**0.00% / 65.1s**	0.02% / 95.9s
n200b	0.00% / 41.8s	0.00% / 40.2s	0.00% / 43.2s	**0.00% / 54.7s**	0.05% / 48.3s
n300	0.00% / 104s	0.00% / 146s	**0.00% / 137s**	0.01% / 137s	0.03% / 158s

In each test case, except for *apte*, boldfaced column corresponds to the strictest aspect ratio bound under which CompaSS is able to pack with 0.00% dead-space. Parameters in each case can be found in [20].

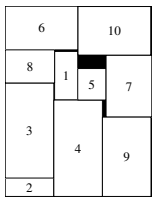


(a) *apte*: slicing and non-slicing

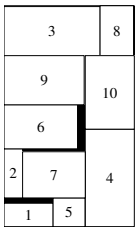


(b) *hp*: non-slicing

(c) *hp*: slicing

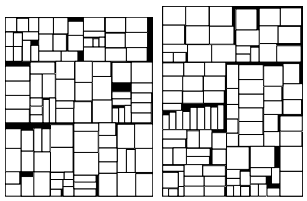


(d) *xerox*: non-slicing

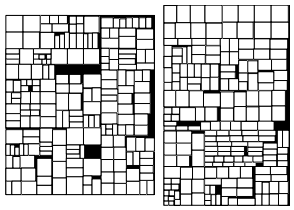


(e) *xerox*: slicing

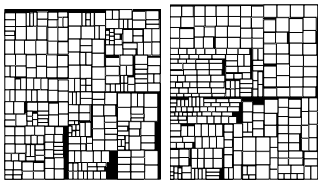
Figure 13: Optimal packings for *apte*, *xerox* and *hp* produced by BloBB.



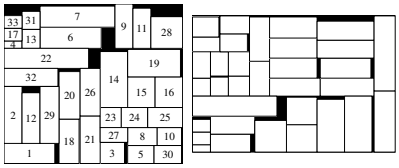
(a) *n100*: BloBB, CompaSS.



(b) *n200*: BloBB, CompaSS.

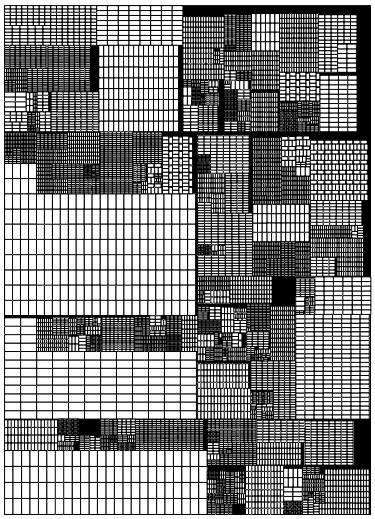


(c) *n300*: BloBB, CompaSS.

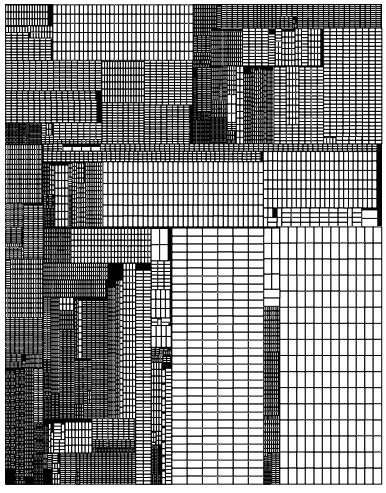


(d) *ami33*: BloBB, CompaSS.

(e) *ami49*: BloBB, CompaSS.



(f) *ami49_200*: BloBB



(g) *ami49_200*: CompaSS

Figure 14: Sample packings produced by BloBB and CompaSS.

and more blocks are present. This size increases dominates the growth of runtime of CompaSS. In fact, when there are many points on a shape-curve, many of them can be ignored since its neighboring points are close approximations. Dropping unnecessary points of shape-curves keep CompaSS efficient when the number of blocks increases.

- Among different levels, the lowest levels and the highest two levels of packing dominate runtime. For small instances (up to 1000), the highest levels take about 70% of the total runtime on average, while for large instances (more than 20K blocks), the lowest levels take more than half of the runtime.
- More post-processing techniques, such as packing compaction, can be developed. We observe that NPE can be easily converted to a B*-Tree that compacts the packing horizontally.
- CompaSS groups lower-level blocks/supermodules into clusters considering only their areas. The shape-curve may facilitate more sophisticated notions of similarity among blocks/supermodules (such as the correlations between pairs of curves). These notions may lead to easier instances that can be packed tighter and more quickly.
- In Fig.14, 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].

Besides chip design, the rectangle packing problem has applications in factory layout, container shipment optimization, scheduling and other areas. In particular, it is closely related to the 2D bin-packing problem, which also has a wide range of applications such as multi-project reticle floorplanning [10]. 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 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.

A Proofs

In this section, we prove the soundness of our minimum min-edge estimation, redundant packing detection and the dominance-breaking mechanism in Section 2. We will consider a partial packing (t, σ, δ) and an arbitrary extended packing (T, π, θ).

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.15).

Figure 15: Minimum Min-Edge Estimation.
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.

Proposition A.1. *Let $\{d_k\}$ be the minimum squares added according to t, and $\{D_k\}$ be the blocks occupying the same position as d_k in T. The x-coordinate of minimum square d_k is at most that of D_k.*

Proof. For each minimum square d_k, we consider a chain of minimum squares $d_{l_1} \ldots d_{l_k}$ where d_{l_1} is adjacent to a block in σ, d_{l_s} is adjacent to the left of $d_{l_{s+1}}$, and $d_{l_p} = d_k$. For example d_1, d_2 and d_3 in Fig.15. Similarly, we consider $D_{l_1} \ldots D_{l_p}$ where D_{l_s} and d_{l_s} occupies the same position in t. For example D_1, D_2 and D_3 in Fig.15. Note that the x-coordinate of d_{l_1} equals that of D_{l_1}, and call it x. Then since $d_k = x + \sum_{i=1}^{p-1} width(d_{l_i})$ and $D_k = x + \sum_{i=1}^{p-1} width(D_{l_i})$, the x-coordinate of d_k is at most that of D_k, as $width(d_{l_i}) \leq width(D_{l_i})$ by the definition of minimum squares. □

Proposition A.2. *There exists some D_l such that the y-coordinate of D_l is at least that of d_k.*

Proof. From the proof of Proposition A.1, the interval from the x-coordinate of d_k to right-end of d_k, is fully contained in the interval from the x-coordinate of D_{l_1}, and the right-end of D_k. By the O-Tree realization algorithm in [5], there exists some D_{l_s} which has y-coordinate at least that of d_k. □

From Propositions A.1, the width of the partial packing with d_i is at most that of any of its extended packings, since d_i's are the minimum squares. Similarly, the partial packing with d_i never overestimates the height of each of its extended packings.

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.

To show the soundness of this criterion, it suffices to show the following proposition.

Proposition A.3. *If a block B does not have an overlapping y-span with its parent, then every extended packing is either non L-compact or identified by an alternative O-Tree in which the block does.*

Proof. Consider an extended packing P. If B has an overlapping y-span with another block B'in P, then P is specified by another O-Tree, in which B has parent B'. Otherwise, P is not L-compact. □

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.

Given an arbitrary complete packing P, we can flip it across the diagonal, preserving the lower-left block, to make it satisfy constraint (1). We call this flipping α-*transformation*. If the packing P already satisfies constraint (1), then its α-transformation leaves it unchanged. Similarly, we can flip it horizontal, vertically or across the diagonal to make it satisfy constraint (2). We call this the β-*transformation*. Again, if the packing P already satisfies constraint (2), its β-transformation leaves it unchanged.

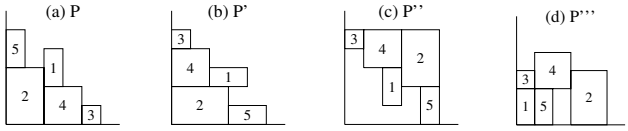


Figure 16: Dominance-Breaking: α and β-transformations.
The packing P satisfies (2) but not (1). When we apply an α-transformation to get P', P' does not satisfy (2) anymore. Thus we apply a β-transformation to get P''' by flipping P' to P'' and then compacting to P''', which satisfies both constraints.

It is obvious that both α and β-transformations do not increase the area of the packing. We now show that given a packing P, we can apply a finite number of α and β-transformations, horizontal and vertical compactions, to make it satisfy both constraints (1) and (2). We define a *cycle of transformations* on the packing P to be:

- its α-transformation.
- horizontal and/or vertical compactions until P is LB-compact.
- its β-transformation.
- horizontal and/or vertical compactions until P is LB-compact.

Starting with P, we apply a cycle of transformations to get $P^{(1)}$, and then apply another cycle to get $P^{(2)}$. The process must stop after at most n rounds, where P contains n blocks. In other words, $P^{(i)}$ are identical for all $i \geq n$. It is because the index of the lower-left block decreases or remains unchanged when a cycle of transformations is applied. Moreover, the resultant packing is not greater than P in area. Therefore, given an arbitrary packing P, we can transform it in at most n rounds of transformations described above. As a result, we can safely ignore the dominated packings.

Shape-Curve Arithmetic

In Section 5.1, we claim that if C_A and C_B are the shape-curves of block/supermodules A and B respectively, then C_{A+B}, the shape-curve of $A+B$ equals the vertical sum of C_A and C_B. We prove this relation by considering the sets of points above the shape-curves. The case for $A*B$ is omitted by symmetry. Let S_{A+B} be the set of points above the shape-curve of C_{A+B}, and T_{A+B} be the set of points above D_{A+B}, the vertical sum of C_A and C_B. It suffices to show that $S_{A+B} = T_{A+B}$.

Proof. First, we observe that a shape-curve C has the property that (x,y) lies above it implies $(x+\varepsilon_1, y+\varepsilon_2)$ lies above it for all $\varepsilon_1, \varepsilon_2 \geq 0$. The horizontal and vertical sums of shape-curves also have this property.
To show $S_{A+B} \subseteq T_{A+B}$**.** Let (x,y) be a point above C_{A+B}. By definition of shape-curve, a bounding box of width x and height y contains $A+B$, and hence $y \geq h_A + h_B$ where (x,h_A) is a bounding box of A, and (x,h_B) is a bounding box of B. Hence, (x,h_A+h_B) lies above D_{A+B}, and hence $(x,y) \in T_{A+B}$.
To show $T_{A+B} \subseteq S_{A+B}$**.** Let (x,y) be a point above D_{A+B}. Then, there is a point (x,h_A) on C_A and a point (x,h_B) on C_B such that $y \geq h_A + h_B$. Since (x,h_A) and (x,h_B) are bounding boxes of A and B respectively, (x,h_A+h_B) is a bounding box of $A+B$. Therefore, (x,h_A+h_B) lies above C_{A+B} and hence $(x,y) \in S_{A+B}$. □

B Realizing a Slicing Packing

To realize a slicing packing specified by a Polish expression P, we construct its slicing tree T from P, compute the dimensions or shape-curves of all block/supermodules by a post-order traversal and then assign the locations of the blocks (which appear in the leaves) by a pre-order traversal.

Constructing Slicing Trees from Polish Expressions

We recall that a Polish expression P is the post-order traversal of a (unique) slicing tree T. To reconstruct the slicing tree T back from the Polish expression P, we only have to scan through P once from back to front. Since P is the post-order traversal of T, the sequence of nodes from the back to the front of P, results from a "pre-order traversal"of T, except that the right subtrees are explored before the left subtrees. When we scan a symbol of P starting from the back, we add a node to the T as follows. We add the last symbol of P as the root of tree. It must be an operator, if the packing has more than one block. We set the *next-empty-spot* to be its right child. If the

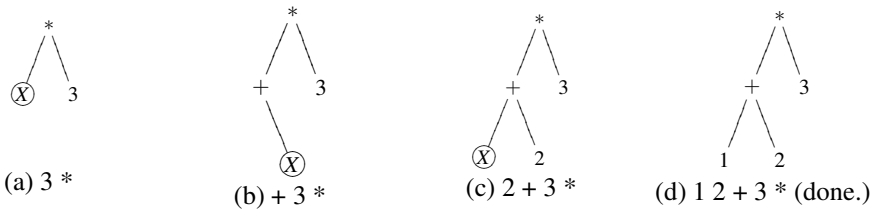


Figure 17: Converting a Polish Expression to its Slicing Tree.
Constructing the slicing tree for the Polish expression $12+3*$. X shows the next-empty-spot of each step. We start by adding * as the root of the tree, and set its right child as the root. Then we add 3 as its right child, traverse up and find $*$ has an empty left child. The next-empty-spot X is set as shown in (a). Then we add $+$ in that position. Since $+$ is an operator, we set the next-empty-spot X to be its right child, as shown in (b). We then add 2 in that position, traverse up and find that $+$ has an empty left child, where we set the next-empty-spot to be (c). In (d), we add 1 in that position, traverse up, and fail to find any ancestor with empty left child, and hence we are done.

next symbol is an operator, we add it to the next-empty-spot and update the next-empty-spot to be its right child. Otherwise, if the next symbol is an operand, we add it to the next-empty-spot, and traverse up until we find a node without left child. We set the next-empty-spot to be the left child of that node. Fig.17 illustrates an example. In BloBB and CompaSS, we use this slicing tree to realize the final packing.

BloBB — with Hard Blocks only

Suppose we are given a packing of n blocks that is specified a slicing tree T. Firstly, we compute the dimension for each block/supermodule. The dimensions of the blocks are given, while the dimensions of the supermodules can be deduced from the sign ($+$ or $*$) and the dimensions of its children. The dimensions of all n blocks/supermodules can be deduced in $O(n)$ time by a post-order traversal (Fig.18). Since the packing is slicing, we can treat it as a packing of 2 supermodules, and easily find the locations of them by pre-order traversal. The root has location $(0,0)$, a left child has the same location as its parent and the location of a right child can be easily deduced from the location and dimensions of its sibling, and the sign of the parent (either $+$ or $*$). This recurrence relation naturally allows an efficient recursive algorithm for realizing the packing. Fig.19 illustrates an example.

CompaSS — with Shape-Curves

CompaSS evaluates the packing in a similar way, except that it uses shape-curves, instead of dimensions, to describe a block/supermodule. Suppose we are given a packing of n blocks, specified by a slicing tree T. For simplicity, we only consider solutions with $+$ and $*$'s only, but not $\sqcup$ for now.

Firstly, we compute the shape-curves for each block/supermodule in the slicing tree T, in a bottom-up manner. For each leave, the shape-curve is given as input, and for each internal node, its shape-curve is formed by adding those of its children vertically or horizontally (Fig.21). Once we have the shape-curves for all nodes of the tree, we search for the min-box (or other shapes of the packing that meets the user-defined constraints such as aspect ratios or outline) of the overall shape-curve. Then, we compute the location and dimensions of each supermodule by a pre-order

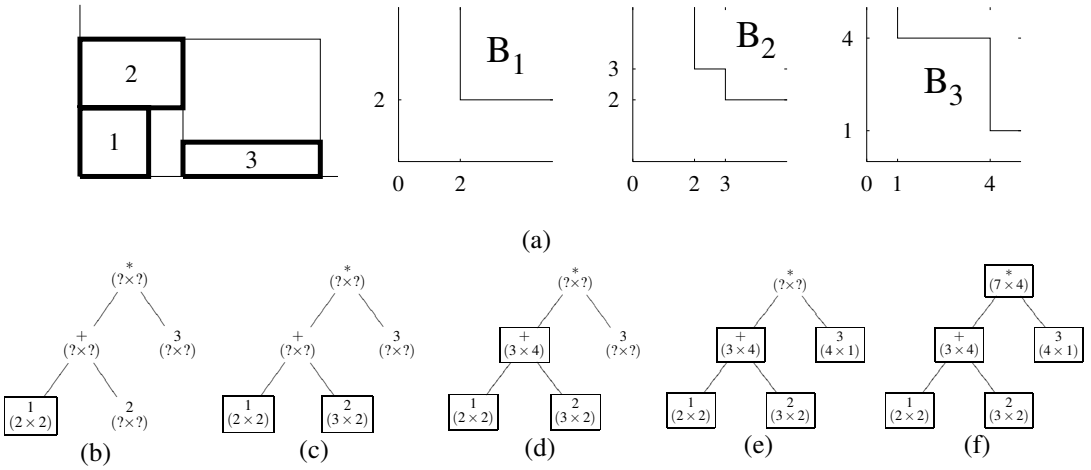


Figure 18: Determining the Dimensions of the Supermodules.
(a) shows a packing $12+3*$, and for blocks $B_1 : (2 \times 2)$, $B_2 : (3 \times 2)$ and $B_3 : (4 \times 1)$, and their shape-curves respectively. We first construct the slicing tree for $12+3*$ as in Fig.17. We traverse the tree to solve for the dimensions of all blocks/supermodules. (b) We start from $*$ to $+$ and then arrive at 1, whose dimensions are given; (c) we arrive 2 next, whose dimensions are also given; (d) we then arrive $+$, which has width $\max\{2,3\} = 3$ and height $2+2=4$, since it is formed by merging blocks B_1 and B_2 vertically; (e) we arrive at B_3 next, whose dimensions are given; (f) we arrive at the root, which has width $3+4=7$ and height $\max\{4,1\} = 4$.

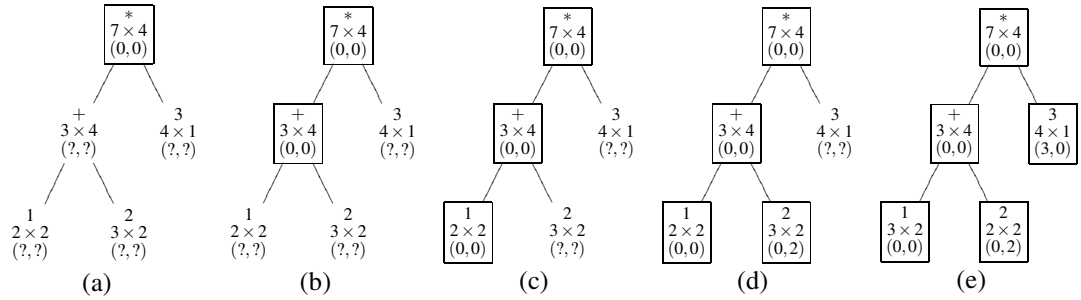


Figure 19: Determining the Locations of all Block/Supermodules from Fig.18.
Once the dimensions of all blocks/supermodules are computed, we can compute the locations of all blocks/supermodules. We follow the example from Fig.18. (a) The root always has location $(0,0)$ since it represents the whole packing; (b) next, we arrive at $+$, which has the same location $(0,0)$ as its parent since it is a left child; (c) similarly, the left child of $+$, B_1 has location $(0,0)$; (d) we arrive at B_2 next; it has location $(0,2)$ since it is placed on top of B_1, and B_1 has height 2; (e) we arrive at B_3 next, which has location $(3,0)$ since it is placed on the right of the supermodule $12+$, that has width 3.

traversal. Similar to BloBB, the location of a left child is the same as its parent. If the parent of node α is a $+$ $(*)$, we search for a point on the shape-curve of α whose x-coordinate (y-coordinate) is the width (height) of its parent, and the shape the α is determined. For a right child β, its dimensions and location depend on those of the dimensions of its sibling and the operator of its parents, similar to BloBB. Fig.22 illustrates an example.

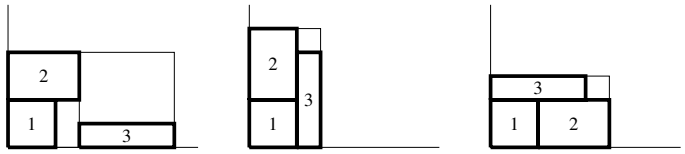
The realization algorithm can be generalized to optimize area by introducing the operator $\sqcup$. Consider a Polish expression $12+3*$, we first replace the operators $+$ and $*$ by $\sqcup$, and then determine which operator ($+$ or $*$) should go to each $\sqcup$. Theoretically, this algorithm may need exponential time, but is very efficient in practice. It takes no more than $10s$ to optimize the operators for *ami49_800*, that consists of 39200 blocks. Similar to the evaluation algorithm, we compute the shape-curves for each block/supermodule in T through a post-order traversal, except that the shape-curve of each internal node is formed by $C_A \sqcup C_B$, where C_A and C_B are the shape-curves of its children (Fig.23). After determining the shape-curves, we compute the following for each block/supermodule:

- its location (x, y),
- its width and height,
- if it is a supermodule (an internal node), we determine whether it is a $+$ or $*$.

These can be determined during the pre-order traversal, we determine whether we should which operator to substitute for $\sqcup$ by trying both ways, and pick the one that works (the resultant module completely lies inside its bounding box). Fig.24 illustrates an example.

Consider shape-curves C_A and C_B that has d_A and d_B points respectively. Then, the shape-curves $C_A + C_B$ and $C_A * C_B$ can have $(d_A + d_B)$ points, but $C_A \sqcup C_B$ can have $2(d_A + d_B)$ points. For a Polish expression with only $+$ and $*$ as operators, the shape-curve of the packing can have as many as $O(nd)$ points, where d is the maximum number of line segments among all shape-curves of the n blocks. On the other hand, for a Polish expression of n blocks where all operators are $\sqcup$, the shape-curve of the resultant packing can have $O(4^n d)$ points. However, in practice, there are much fewer points in the final shape-curve since substantial overlap often (almost always) occurs when we compute the curve $C_A \sqcup C_B$ from C_A and C_B.

Fig.20 compares the packing realized by the three algorithms above. BloBB realizes the packing without any optimization (Fig.20a). Without operator-optimization, CompaSS optimizes only the aspect ratios of the blocks (Fig.20b). In cases when the blocks are hard, it essentially optimizes the orientations of the hard blocks. Coupled with operator-optimization, CompaSS allows the final packing to have a very different structure (Fig.20c). The operators and the aspect ratios (or orientations) of the blocks can all be different. The effect of orientation and operator-optimization becomes more and more significant when there are more and more blocks.



(a) $12+3*$ (b) $12+3*$ (c) $12 \sqcup 3 \sqcup$

Figure 20: The Packings Produced by Different Algorithms.
The packing for (a) $12+3*$ realized by BloBB in Fig.18-19, (b) $12+3*$ realized by CompaSS in Fig.21-22, and (c) $12 \sqcup 3 \sqcup$ realized by CompaSS with operator-optimization in Fig.23-24.

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

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

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

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

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

[8] 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.

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

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

[11] 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.

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

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

[14] S. Prestwitch, "Supersymmetric Modeling for Local Search", *SymCon '02*, September 2002; `http://user.it.uu.se/~pierref/astra/SymCon02/`.

[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/

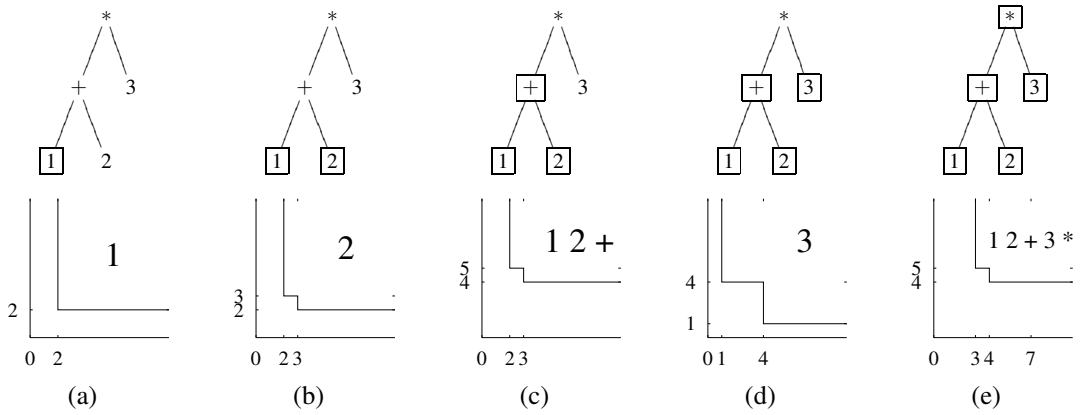


Figure 21: Determining the Shape-Curves of all Supermodules.
Given the Polish expression $12+3*$, we first construct the slicing tree for $12+3*$ as in Fig.17; We perform a post-order traversal of the tree to solve for the shape-curves of all blocks/supermodules; (a) we first arrive at B_1, whose shape-curve is given; (b) we arrive at B_2 next, whose shape-curve is also given; (c) we arrive at $+$ next,whose shape-curve is given by adding the shapes curves of B_1 and B_2 vertically; (d)-(e) then we arrive at B_3 and back to the root, whose shape-curve is given by adding the shape curves of $12+$ and B_3 horizontally. The shape-curve of the root is the shape-curve of the whole packing.

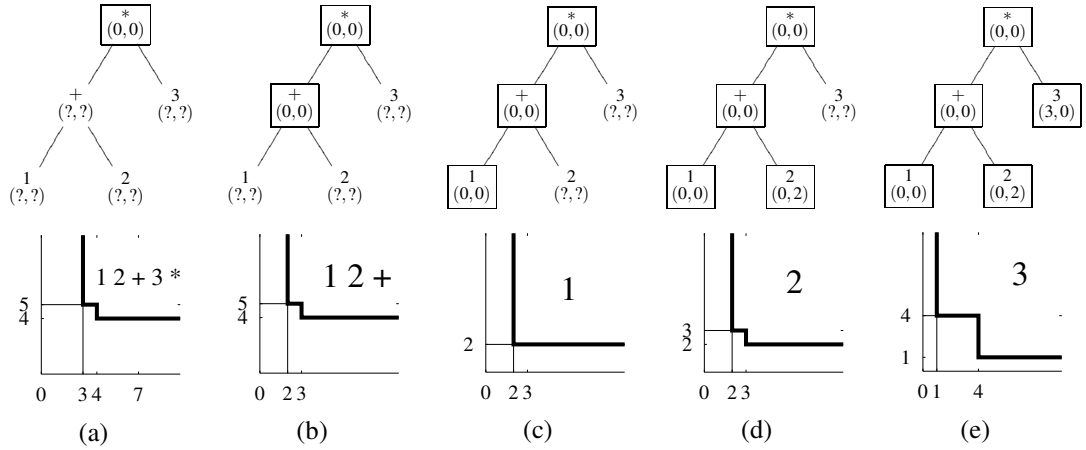


Figure 22: Determining the Locations of all Blocks/Supermodules from Fig.21.
Once the shape-curves of all blocks/supermodules are computed, we can compute the dimensions and locations of all blocks/supermodules. We follow the example from Fig.21. (a) The root always has location $(0,0)$ since it represents the whole packing; moreover, we choose the packing to have width 3 and height 5, since it is the smallest possible; (b) next, we arrive at $+$, which has the same location $(0,0)$ as its parent since it is a left child; it must have height at most 5 since it merges with B_3 horizontally to give the root; its dimensions are determined to be 2×5; (c) similarly, the left child of $+$, B_1 has location $(0,0)$; its width is at most 2 since it merges with B_2 to give its parent; B_1 is determined to have dimensions 2×2; (d) we arrive at B_2 next; it has location $(0,2)$ since it is placed on top of B_1, and B_1 has height 2; its width is at most 2 since it merges with B_1 to form its parent; it is determined to be 2×3; (e) we arrive at B_3 next, which has location $(3,0)$ since it is placed on the right of the supermodule $12+$, that has width 3; its height is at most 5 and it is determined to be 1×4.

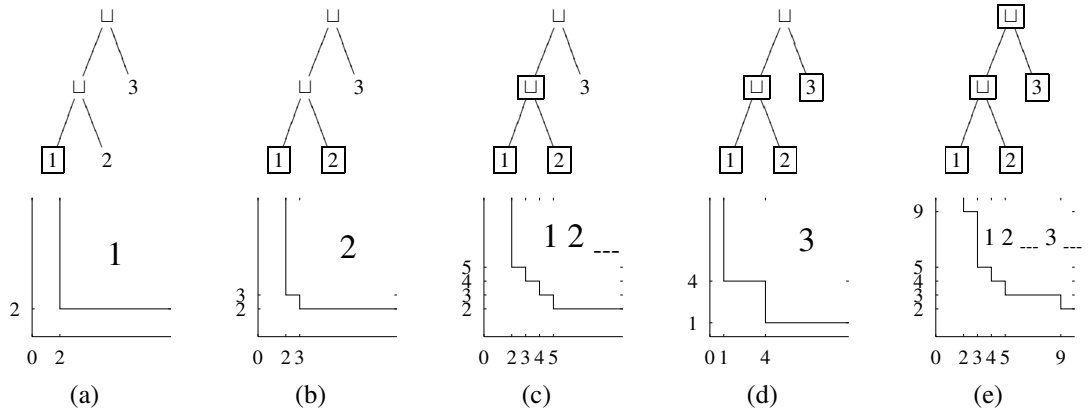


Figure 23: Determining the Shape-Curves of all Supermodules when Optimizing Operators.
(a)-(c) Similar to Fig.21, we compute the shape-curves for each block/supermodules by a post-order traversal. From the shape-curves of B_1, B_2, we compute the shape-curve of the supermodule $B_1 \sqcup B_3$ by first computing those of $B_1 + B_2$ and $B_1 * B_2$, and then that of $(B_1 + B_2) \vee (B_1 * B_2)$; (d)-(e) we proceed similarly to get the shape-curves of B_3 and $(B_1 \sqcup B_2) \sqcup B_3$ (or in post-fix notation $B_1 B_2 \sqcup B_3 \sqcup$).

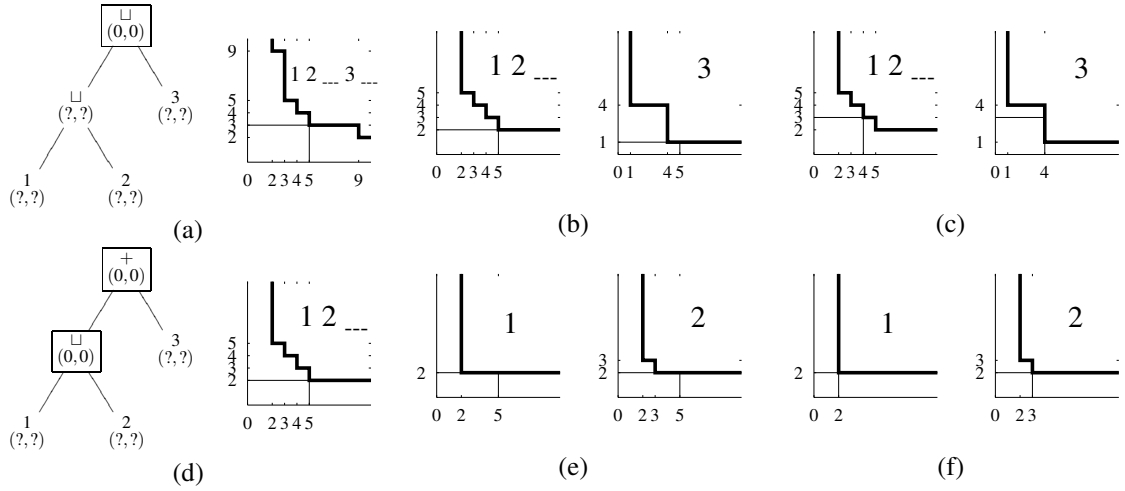


Figure 24: Determining the Locations of all Blocks/Supermodules from Fig.23.
We follow the example from Fig.23. (a) We determine the whole packing to have width 3 and height 5, and set the root to locate at $(0,0)$; next we have to determine whether we should replace $\sqcup$ by $+$ or $*$ by trying both cases; (b) suppose we replace $\sqcup$ in the root by $+$, then both children must have width at most 5; therefore the supermodule $B_1 \sqcup B_2$ has height 2 and B_3 has height 1; (c) on the other hand, if the operator is $*$, then $B_1 \sqcup B_2$ and B_3 has height at most 3, but then $B_1 \sqcup B_2$ needs to have width 4 and B_3 needs to have width 4 as well; hence the bounding box needs to have width at least 8, which is impossible; therefore the operator in the root must be $+$; (d)-(f) we first compute the dimensions and location of $B_1 \sqcup B_2$ as usual and then determine its operator similarly and it has to be $*$. Since B_1, B_2 and B_3 are leaves, we can use the ordinary shape-curve realization method to determine their locations and dimensions as in Fig.22. We find that B_1 locates at $(0,0)$, B_2 at $(2,0)$ and B_3 at $(0,2)$.