FTAFP: A Feedthrough-Aware Floorplanner for Hierarchical Design of Large-Scale SoCs

Zirui Li*
lzr_official@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

Kanglin Tian*
tiankl@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

Jianwang Zhai†
zhaijw@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

Zixuan Li
gxydbc@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

Shixiong Kai
kaishixiong@huawei.com
Noah's Ark Lab, Huawei
Beijing, China

Siyuan Xu
xusiyuan520@huawei.com
Noah's Ark Lab, Huawei
Shenzhen, China

Bei Yu
byu@cse.cuhk.edu.hk
The Chinese University of Hong Kong
Hong Kong SAR

Kang Zhao
zhaokang@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

ABSTRACT

Floorplanning is a critical step in the physical design of digital integrated circuits (ICs). As circuit complexity grows, the hierarchical design paradigm of large-scale systems on chips (SoCs) is gradually emerging, introducing new optimization challenges, particularly with feedthrough. Feedthrough is a through-module connection, yet it would require additional buffers and ports inside the module for data transmission. Excessive feedthroughs will inevitably hinder the routability within reusable modules, causing congestion and timing problems. However, few works have addressed the challenges of feedthrough modeling and optimization.

In this work, we propose FATFP, a feedthrough-aware SoC floorplanner, to address the aforementioned issues. First, an estimation model is proposed to assess feedthroughs required in the floorplan. Then, we introduce a novel topological representation, SCB-Tree, which incorporates slack computation into the CB-Tree. We also develop a two-phase simulated annealing (SA) framework and an automatic optimization cost scheme to enhance performance. Experimental results demonstrate that our floorplanner achieves notable optimization in terms of common edge, feedthroughed modules, and feedthrough wirelength over previous work, with only minor trade-offs in total wirelength and runtime.

*Co-first authors with equal contribution.
†Corresponding author.

ASPDAC '25, January 20–23, 2025, Tokyo, Japan

ACM ISBN 979-8-4007-0635-6/25/01…$15.00
https://doi.org/10.1145/3658617.3697728

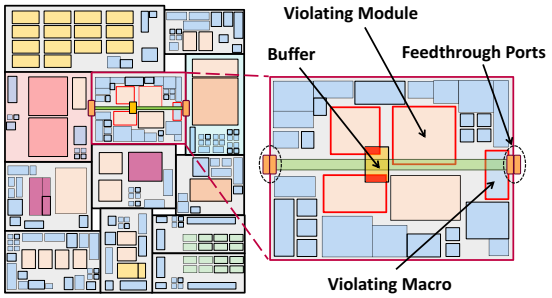


Figure 1: The violations caused by feedthrough insertion in hierarchical floorplanning of the large-scale SoC.

KEYWORDS

Electronic Design Automation, Floorplanning, Feedthrough

ACM Reference Format:
Zirui Li, Kanglin Tian, Jianwang Zhai, Zixuan Li, Shixiong Kai, Siyuan Xu, Bei Yu, Kang Zhao. 2025. FTAFP: A Feedthrough-Aware Floorplanner for Hierarchical Design of Large-Scale SoCs . In *30th Asia and South Pacific Design Automation Conference (ASPDAC '25), January 20–23, 2025, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3658617.3697728

1 INTRODUCTION

Floorplanning is the initial step in IC physical design and significantly impacts the quality of subsequent design stages, e.g., placement and routing. Given a module list and a netlist, the floorplanner determines the shape and position of the modules according to specific strategies to achieve comprehensive optimization, e.g., wirelength, area, timing, congestion, etc. Due to the increasing scale and complexity of SoCs, modern chip design commonly uses hierarchy and modularization concepts, which bring the floorplanning problem to the sub-chip level [1]. Hierarchical design decomposes

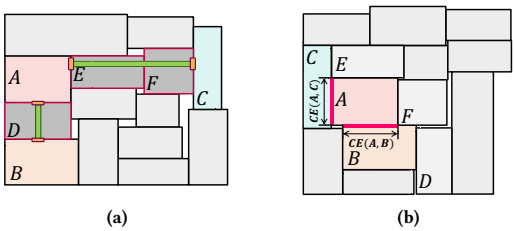
complex systems into multiple levels of subsystems that encapsulate specific functionalities and interact through interfaces. Concurrently, modular design facilitates the encapsulation and reuse of functionalities at each level. These integrated, bottom-up design methods significantly speed up the front-end chip design process. However, they also present new optimization challenges in physical design, particularly in hierarchical floorplanning for large-scale SoCs [2, 3], where feedthrough [4] is a key challenge.

In hierarchical SoC design, it is important to consider the completeness, reusability, and interaction of modules at each level. When higher-level modules are traversed by wires, feedthroughs are used to connect the nets. To meet connectivity and timing constraints, additional buffers, ports, registers, etc., often need to be inserted. As shown in Figure 1, directly inserting these elements may cause violations with the original components inside the feedthroughed module, necessitating the reevaluation and adjustment of design feasibility. Such adjustment may also present challenges to subsequent placement and routing, resulting in degradation of timing, area, and other metrics [2]. When there are too many feedthroughs, the modules may need to be redesigned, affecting the design and reuse of modules at each hierarchy. Therefore, it is necessary to minimize the generation of feedthrough.

The most direct way to minimize feedthrough demand is to reduce through-module connections and routing. Existing works, such as BOB-RSMT [5] aim to minimize over-block connection by optimizing RSMT construction during the pre-routing and global routing stages. Although optimization in the later physical design stages such as routing can avoid some feedthrough insertions, the optimization space is limited and still inevitably prolongs the design iterations. To improve design efficiency and optimize quality, feedthrough minimization should be implemented as early as possible in the design cycle. Therefore, it is necessary and urgent to consider feedthrough optimization in hierarchical floorplanning.

After decades of development, floorplanning techniques have advanced significantly and can be broadly categoried into three types: analytical, heuristic, and learning-based methods. Analytical-based methods [6, 7] generally adopt a two-stage framework of global distribution and legalization. Although known for their efficiency, quality, and robustness, analytical methods are limited by the need for differentiable constraint models [8, 9], making complex constraints and objectives challenging to address. Heuristic methods rely on topological representations, e.g., Sequence Pair (SP) [10, 11], Corner Block List (CBL) [12, 13], B*-Tree [14, 15] and their variants. These methods employ heuristic algorithms to optimize floorplans, leveraging geometric relations in different topologies to address complex constraints. For instance, CB-Tree [15] integrates corner stitching into a B*-Tree to refine neighboring relations. Recently, deep learning [16, 17] and reinforcement learning [18–20] frameworks have been used to formulate floorplanning problem, achieving promising results. However, these methods struggle to handle large cases and lack generalization across diverse circuit designs.

In hierarchical floorplanning for SoCs, complex objectives like feedthrough optimization should be considered, requiring modules within the same net to be placed as adjacent as possible to minimize feedthrough. However, overlaps in the global distribution stage of analytical methods complicate feedthrough handling, which relies on neighbor information. Learning-based methods, constrained in representation and generalization, are also unsuitable for feedthrough optimization. Considering both efficiency and generality, heuristic-based methods are promising approaches to address the feedthrough challenge. In this work, we propose a feedthrough-aware floorplanner to address feedthrough optimization at the sub-chip level. Our main contributions are as follows:



(a) (b)

Figure 2: Feedthrough example with two nets, i.e, $N_1 = \{A, B\}$ and $N_2 = \{A, C\}$. (a) Floorplan with feedthroughed module D, E, F; (b) Floorplan with no feedthrough in net N_1 and N_2.

- To achieve fast optimization of feedthrough, We introduce an accurate feedthrough model that estimates the length of feedthrough wires, the number of feedthroughed modules, and the length of the common edge.
- To address the fixed-outline constraint, we propose a new representation called SCB-Tree, which performs slack computation by corner stitching while packing modules.
- To better satisfy the fixed-outline constraints and enhance the search for optimal solutions, we propose a two-phase SA framework that leverages slack information.
- We design an automatic cost evaluation method that more effectively normalizes wire length and feedthrough-related metrics based on the input module list and netlist.

2 PRELIMINARIES

2.1 Fixed-outline Floorplanning

Let $B = \{b_i | 1 \le i \le n\}$ be a set of rectangle modules, each module b_i has width w_i and height h_i. Modules can be classified into three categories: hard, soft, and pre-placed. Hard modules have fixed dimensions, while soft modules can adjust their shapes within a fixed area; pre-placed modules are hard modules with preset coordinates. The connections among modules are described in netlist $N = \{N_i | 1 \le i \le m\}$, where each net N_i specifies a set of modules requiring connectivity.

The fixed-outline floorplanning aims to place all modules without overlapping in a rectangular outline R, with width W_0 and height H_0. Given the total modules' area A and a maximum whitespace ratio σ, the width W_0 and the height H_0 are calculated as:

$$W_0 = \sqrt{(1+\sigma)A\lambda},\ H_0 = \sqrt{(1+\sigma)A/\lambda}, \tag{1}$$

where λ represents the aspect ratio. On the premise of ensuring the above constraints, the floorplanner aims to optimize metrics (e.g. wirelength, feedthrough, etc.) to obtain the optimal floorplan.

2.2 Feedthrough Problem

Since our work focuses on the feedthrough optimization problem of hierarchical floorplanning for large-scale SoCs, the relevant terms are briefly explained below.

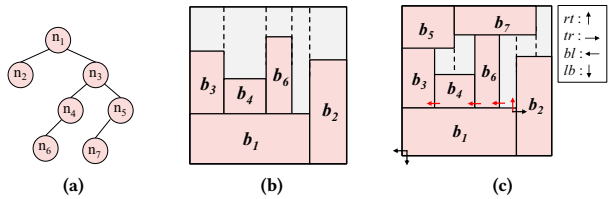


Figure 3: (a) A CB-Tree example. (b) The corresponding floorplan (tile plane) before packing subtree $n_5 \rightarrow n_7$. (c) The corresponding floorplan(tile plane). All top neighbors of module b_1 can be found by tracing through the red pointers.

Feedthrough. In hierarchical floorplanning, unlike traditional floorplanning, the through-module nets need to be connected by feedthroughs. As described in Section 1, module-scale feedthrough wires may cause problems such as timing and routing congestion. Moreover, the additional inserted registers and buffers may necessitate a redesign of the entire module. Therefore, modules within the same net should be adjacently placed to the fullest extent to reduce feedthrough demand. The estimation of feedthroughs involves two metrics: total feedthrough wirelength FTH_{wl} and the number of feedthroughed modules FTH_{num}. Let $fth_{wl}(N_i)$ and $fth_{num}(N_i)$ represent these metrics for a net N_i. For example, for the two nets shown in Figure 2(a), the feedthrough metrics $FTH_{wl} = w(E) + w(F) + h(D), FTH_{num} = 3$. The more detailed estimating methods will be described in Section 3.1.

Common Edge. The common edge is defined as the edge shared by two adjacent modules within the same net, as shown in Figure 2(b). The length of this edge dictates the capacity of a module's ports. A floorplan featuring more total common edge (CE_{len}) suggests a higher interactivity between modules, fulfilling more extensive connectivity requirements. Let $ce_{len}(A, B)$ denote the common edge length between module A and B. Assume that the coordinates of the bottom-left and top-right corner of the module are (x_{bl}, y_{bl}) and (x_{tr}, y_{tr}). Then, $ce_{len}(A, B)$ can be calculated as:

$$\begin{aligned} &\text{If } A, B \text{ adjacent horizontally ,then:} \\ &\quad ce_{len}(A, B) = \min(A.y_{tr}, B.y_{tr}) - \max(A.y_{bl}, B.y_{bl}); \\ &\text{If } A, B \text{ adjacent vertically ,then:} \\ &\quad ce_{len}(A, B) = \min(A.x_{tr}, B.x_{tr}) - \max(A.x_{bl}, B.x_{bl}). \end{aligned} \tag{2}$$

2.3 CB-Tree Representation

A CB-Tree is a B*-Tree integrated with corner stitching [15], a classical data structure for representing non-overlapping rectangular modules in the 2D plane (called tile plane). It can be traversed in depth-first search (DFS) order to locate the corresponding modules. Let n_i be the corresponding tree node of module b_i. It has coordinates (x_i, y_i) and width w_i and height h_i. The tree structure can directly compute the horizontal coordinate of each module. Let n_j and n_k be the left and right child of n_i, then $x_j = x_i + w_i$ and $x_k = x_i$. Each module should be placed in the lowest possible position to determine the vertical coordinate, and then the tile plane should be updated. Figure 3 gives a CB-Tree and the corresponding floorplan.

The corner stitching models modules and empty spaces as tiles and links all of them with four pointers. As shown in Figure 3(c), the rt, tr, bl, and lb pointers point to the tiles adjacent to the top, right, left, and bottom boundaries, respectively. Corner stitching provides many efficient operations to support handling geometric constraints. Neighbor finding is the most commonly used one in this work. Figure 3(c) provides an example of finding all neighbors of the target tile at the given direction.

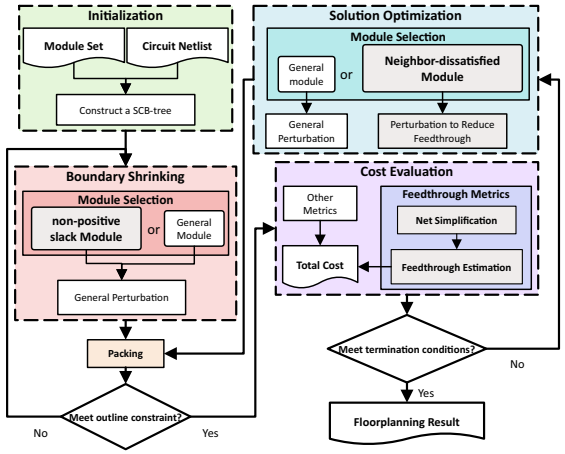


Figure 4: Flowchart of FTAFP.

2.4 Slack Computation

The slack of a module refers to the range within which it can move without overlapping with or pushing other modules. Slack computation is widely used in constraint graph-based placement legalization algorithms [6], based on the following observations:

- The x and y coordinates of modules are computed separately.
- In each dimension, the floorplan is constrained by one or more "critical paths" in corresponding constraint graphs.
- Any change in the location of a module on the critical path will produce overlaps or increase the span of the floorplan.

Take the horizontal constraint graph G_h as an example. Let module v_i be represented as vertex v_{h_i} in G_h. Its horizontal slack (x-slack) is calculated as Equation (3) [21] , where $R(\cdot)$ and $L(\cdot)$ denote the furthest right and left positions that $\cdot$ can reach:

$$slack(v_{h_i}) = R(v_{h_i}) - L(v_{h_i}). \tag{3}$$

3 FTAFP FRAMEWORK

We first give an overview of our proposed feedthrough-aware floorplanner (i.e., FTAFP), which includes: a feedthrough estimation model, slack computation with corner stitching, a two-phase SA framework, and an optimization cost scheme. The feedthrough estimation model initially simplifies the nets into sub-nets, constructs a Minimum Spanning Tree (MST) for each to estimate feedthrough wirelength, and employs a greedy algorithm to count feedthrough modules. The proposed SCB-Tree integrates slack computation with CB-Tree to better exploit the features of boundary and adjacency awareness. Figure 4 shows the overall flow of FTAFP, which adopts a novelly introduced two-phase SA framework. Each phase includes a distinctive module selection and perturbation process, aiming to shrink the boundary and optimize the solution quality, separately. Besides, a meticulously designed cost scheme is applied.

3.1 Feedthrough Estimation Model

In the actual design process, engineers often connect directly to adjacent modules within the net, using feedthroughs only when it is

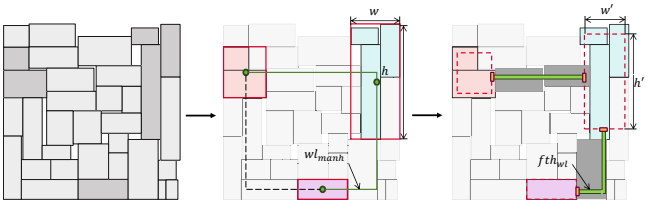


Figure 5: An example of net simplification. A net with 6 modules which are clustered into 3 sub-nets.

unavoidable to connect through modules outside the net. Therefore, our feedthrough model is based on the following rules:

- Modules that are directly adjacent are assumed to be connected and no feedthrough is required.
- There are no feedthrough requirements arising between modules separated by empty tiles (whitespace).
- Connections between adjacent modules are transitive within the same net.

Based on these rules, feedthroughs are estimated with the minimum wire length and the fewest number of modules traversed to ensure all modules within the net form a connected graph. Our feedthrough modeling method is divided into two parts: net simplification and feedthrough estimation.

Net simplification is to cluster adjacent modules into sub-nets, allowing feedthrough estimation to be performed within these sub-nets rather than between modules. At first, we transform nets into undirected graphs, where edges are only constructed between adjacent modules. The connected components within are defined as sub-nets. Algorithm 1 delineates the process described above and Figure 5 presents an example. We utilize the union-find algorithm to merge each pair of adjacent modules within, meanwhile, computing the length of the common edge between the pair using Equation (2). Through the *Find* operation,the net N_i is simplified into a set of sub-nets $N_{sub} = \{N_{sq} | 1 \le q \le n_i, N_{sq} \subset N_i\}$.

For the sub-net N_{sq}, the coordinates of its central node (x_q, y_q) are calculated as the average of the central coordinates of all its modules. Treating these centers as nodes, the shortest feedthrough connections between sub-nets can be found using an MST. Let E_i be the set of edges in the MST for net N_i. Consider two sub-nets, A and B, connected in the MST, with the edge weight $\omega(A, B)$, the weighted distance $D_w(A, B)$ between which can be calculated by:

$$\begin{aligned} \omega(A,B) &= \lceil n_{blk}(A) + n_{blk}(B) \rceil / 2, \\ D_w(A,B) &= wl_{manh}(A,B) \times \omega(A,B), \end{aligned} \tag{4}$$

where $wl_{manh}(A, B)$ represent the manhattan distance between the center of A and B, $n_{blk}(A)$ is the amount of modules in sub-net A.

For net N_i, regardless of the actual route shape, its feedthrough wirelength $fth_{wl}(N_i)$ can be estimated as the sum of the wirelength of each feedthrough edge within the MST:

$$\begin{aligned} fth_{wl}(N_i) &= \sum_{e(A,B) \in E_i} fth_{wl}(A,B) \times \omega(A,B), \\ fth_{wl}(A,B) &= \left[wl_{manh}(A,B) - wl'(A) - wl'(B) \right] \times \omega(A,B), \\ wl'(A) &= \frac{w(A)+h(A)}{2} \times \sqrt{\frac{area'(A)}{area(A)}}, \end{aligned} \tag{5}$$

Algorithm 1 Net Simplification Algorithm.

Input: A net cell list N_i and all modules $B \in N$
Output: Sub-net list $N_i.Subnets$

```
for B_j ∈ N, j = 1 → N_i.size() do
    for B_k ∈ N, k = j → N_i.size() do
        if B_j.neighbors.count(B_k) then
            CE_len+ = ce_len(B_j.B_k);
            Union( Find(B_j), Find(B_k) );
for B_j ∈ N_i, j = 1 → N_i.size() do
    for N_sq ∈ N_i.Subnets, q = 1 → N_i.Subnets.size() do
        if Find(B_j == N_sq.parent) then
            N_sq.cellspush_back(B_j);
            Update the Outline and coordinate of N_sq;
    if N_i.Subnets.size() == 0 then
        N_i.Subnets.push_back(New Subnet(B_j));
return N_i.Subnets.
```

Algorithm 2 Algorithm to Estimate fth_{num}.

Input: Two sub-nets A, B
Output: Feedthrough number between A and B fth_{num}

```
Let P equal to the tile contains A's center point;
reach_x, reach_y, fth_blk = 0;
while reach_x * reach_y == 0 do
    dist = 0;
    Neighbors_search = Connect( P.nbr(dir[0]), P.nbr(dir[1]) );
    for nbr_next ∈ Neighbors_search do
        pnt = nbr_next.point(dir[0], dir[1]);
        dist_x = (B.x_mid − pnt.x) * dir[0], reach_x = (dist_x <= 0);
        dist_y = (B.y_mid − pnt.y) * dir[1], reach_y = (dist_y <= 0);
        if dist_x*!reach_x + dist_y*!reach_y > dist then
            dist = dist_x*!reach_x + dist_y*!reach_y;
            P = nbr_next;
    if P.isSolid() then fth_num++;
return fth_num.
```

where $e(A, B)$ is the edge of the constructed MST, $w(A)$ and $h(A)$ are the side length of a sub-net, $area'(A)$ is the summary of modules within A, $area(A)$ is the area of the sub-net A's boundary.

In addition to wirelength, the number of feedthrough modules (fth_{num}) is a crucial factor in assessing floorplan quality—a better floorplan will have fewer feedthrough modules for the same wirelength. To estimate fth_{num}, we propose a greedy detection algorithm based on neighbor searching. The algorithm selects the tile with the largest horizontal or vertical span along the estimated route until it lies within the target sub-net boundary in both directions, as summarized in Algorithm 2. The search begins at the tile containing the center of sub-net A, using the array dir to indicate the relative 2D direction from A to B (where $dir[0]$ values -1 and 1 denote left and right, and $dir[1]$ values represent up and down). The algorithm finds the neighbor tile with the furthest-reaching endpoint, then checks if the target range is met. If not, it moves to this neighbor tile and repeats until the number of feedthrough modules along each edge e is counted, yielding the total feedthrough modules for net N_i.

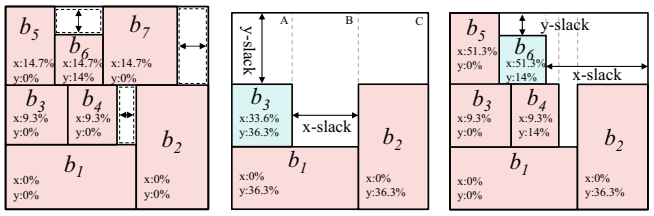


Figure 6: (a) An example tile plane contains seven packed modules. (b) Slacks of b_3 are initialized its slacks based on the associated tiles A and B. (c) When packing module b_6, we update module b_5's x-slack and module b_4's y-slack as b_6's.

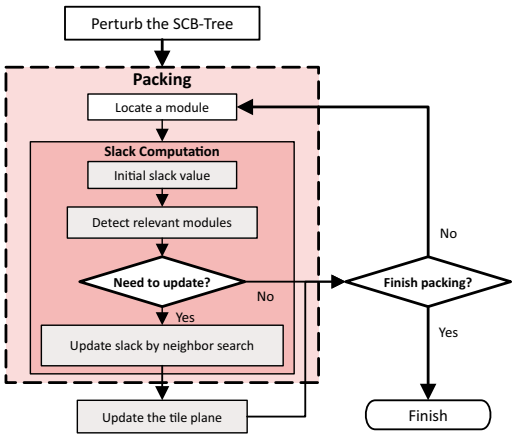


Figure 7: The slack computation flow by SCB-Tree.

3.2 Slack Computation by Corner Stitching

To enhance the outline awareness of the CB-Tree while accommodating the newly proposed optimization objective, we propose the SCB-Tree, which extends the slack computation to corner stitching. The corresponding floorplan of a given SCB-Tree is determined by traversing the tree in DFS order. Each module's coordinate is determined based on the principles outlined in Section 2.3. As each module is packed, the tile plane and the slack of each module associated with the current module are updated. After packing a module, the current tile plane appears as shown in Figure 6(a), where tiles with solid lines represent modules and tiles with dashed lines represent empty tiles.

We first set the module's initial slack to facilitate later updates. For an unplaced module b_i with height h_{b_i} and width w_{b_i}, the initial slack in its x-dimension is the sum of the widths of the empty tiles to its right whose height exceeds h_{b_i}. Similarly, the initial slack in its y-dimension is the height of the empty tile above it whose width exceeds w_{b_i}. Figure 6(b) gives an example of slack initialization.

Next, we implement the following update strategy to dynamically adjust the module's slack during packing. Once a module is packed, the slacks in the x and y dimensions are updated separately based on the corresponding modules to its left and bottom, as shown in Figure 6(c). Since b_1, b_3, b_5 are modules on a critical path and their y-slack is 0, the y-slack of b_1 and b_3 do not need updating. Only the y-slack of b_4 is updated to match that of b_6.

Figure 7 shows our slack computation flow. For a perturbed SCB-Tree, we first determine the positions of modules based on the packing strategy of the CB-Tree, while initializing the slack. After completing the module packing, we identify adjacent modules through neighbor finding in the corner stitching and update their x-slack and y-slack accordingly.

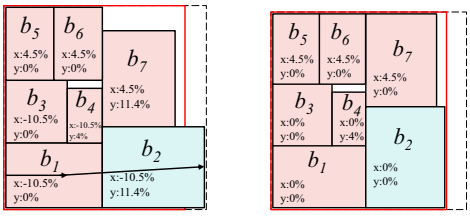


Figure 8: (a) A floorplan with 7 modules, the black arrow represents the critical path on x-dimension, and the x-slack on this path is non-positive. (b) After rotating module b_2 on the critical path, the x-span of the floorplan is reduced.

3.3 Two-phase SA Framework

In traditional SA processes, it is often unclear which modules influence the span of the floorplan, and random perturbation operations fail to effectively control the process of satisfying fixed-outline constraints. Especially when optimizing feedthroughs, frequent changes in neighboring modules often lead to violations of the fixed-outline constraints. In response to the above observations and facts, we propose a two-phase SA framework as shown in Figure 4. The two phases adopt different module-selecting strategies and perturbation operations. In the boundary shrinking phase, modules are classified into two categories: non-positive slack and normal modules by computing their slack. The annealing perturbation operation actively selects the non-positive slack modules to better satisfy the fixed-outline constraints. In the solution optimization phase, we propose a novel perturbation that operates on neighbor-dissatisfied modules to reduce feedthroughs.

A critical path is defined as the longest path in a floorplan, and modules on it are non-positive slack modules. There may be multiple critical paths in a floorplan, with a module possibly located on more than one critical path. The length of critical paths is equal to the span of the floorplan. Therefore, the perturbation on non-positive slack modules may crucially impact the outline span. Figure 8 illustrates how relocating modules on the critical path can reduce the floorplan span.

In the first phase of our framework, modules with non-positive slack in any dimension are more likely to be chosen for perturbation operations. The B*-Tree is commonly used for three main operations, which are explained in the following examples:

- Op1: Rotate a module. A module is an ideal candidate for this operation if it has a non-positive slack in one dimension but a large slack in another, as shown in Figure 8.
- Op2: Relocate a module. This operation tends to operate on modules with two dimensions of non-positive slack and relocate it to be a child of the module that has a larger slack than its shape. That is, we attempt to move it into a large whitespace.
- Op3: Swap two modules. This operation is ideal for combining two modules with two dimensions of non-positive slack and one dimension of non-positive slack.

Once the fixed-outline constraints are satisfied, the annealing process enters the solution optimization phase. It mainly targets modules that do not expand the floorplan span for perturbation operations. Area and wirelength optimization are achieved by the above perturbation operations, while for feedthrough optimization, a new perturbation operation is proposed:

- Op4: Change the module's neighbors. Select a module with minimal neighbor satisfaction and randomly swap it to its neighborhood-demanding child nodes.

In a floorplan, we analyze each module's neighbor satisfaction during feedthrough calculation and compare it to its neighbor requirement, which is determined by the netlist indicating ideal neighboring modules.

3.4 Cost Evaluation

The optimization objective comprises five metrics: area, wirelength, common edge length, feedthrough wirelength, and the number of feedthroughed modules. The cost of the floorplan is calculated by summing weighted metrics. However, the wide range in size and unclear relationship between measurements necessitate additional pre-floorplanning iterations to estimate appropriate weights. To streamline this process, we develop a normalizing method to predict the expectations of each metric.

It's straightforward to prove that the average Manhattan distance between every 2 modules in a $n \times n$ square grid is $\frac{2(n^2-1)}{3n-2}$. If we divide a chip consisting of n modules into a $\sqrt{n} \times \sqrt{n}$ uniform square grid, each grid's width and height is S. Therefore, under ideal even partitioning, the total half-perimeter wirelength (HPWL) expectation of m nets can be calculated as:

$$\overline{HPWL} = \frac{2m(n-1)}{3\sqrt{n}-2} \times S. \tag{6}$$

Similarly, considering the net N_i may have 0 to $(n_i - 1)$ feedthrough paths, the expectation number of feedthroughed modules $\overline{FTH_{num}}$ and the feedthrough wirelength $\overline{FTH_{wl}}$ can be computed as:

$$\begin{aligned} \overline{FTH_{num}} &= \frac{2(n-1)}{3\sqrt{n}-2} \times \sum_{i=1}^{m} \frac{n_i - 1}{2}, \\ \overline{FTH_{wl}} &= \overline{FTH_{num}} \times S. \end{aligned} \tag{7}$$

Assuming that N_i is evenly partitioned and that each module is closely spaced, the parameter r_i represents the largest perfect square root that is less than or equal to n_i. The expectation value of common edge length $\overline{CE_{len}}$ can be estimated using:

$$\overline{CE_{len}} = \frac{1}{2} \sum_{i=1}^{m} 2(n_i - r_i) - (n_i > r_i^2) - (n_i > r_i^2 + r_i). \tag{8}$$

Finally, we can obtain the total cost ϕ, defined as:

$$\phi = \alpha \frac{Area_{total}}{Area} + \beta \frac{\overline{HPWL}}{HPWL} - \gamma \frac{\overline{CE_{len}}}{CE_{len}} + \delta \left(\frac{\overline{FTH_{num}}}{FTH_{num}} + \frac{\overline{FTH_{wl}}}{FTH_{wl}} \right), \tag{9}$$

where $\alpha, \beta, \gamma, \delta$ are the weight of each metric, and $Area_{total}$ is the sum of all modules' area.

4 EVALUATION

FTAFP is implemented in C++ and operates in single-threaded mode on a Linux system, with Intel(R) Xeon(R) Silver 4214R @ 2.4GHz and 128GB memory. The FTAFP framework is compared with three competitive heuristic-based methods based on the topological representation: Corblivar [13], SP-FOFP [11], and CB-Tree [15]. Where CB-Tree is reproduced by ourselves, SP-FOFP is an open executable and Corblivar is open source. It should be noted that the goal of our work is feedthrough optimization, which has not been considered in previous works. Therefore, we chose the publicly available floorplaners mentioned above as our baselines. The test cases are derived from the GSRC [22] and MCNC [23] benchmarks. To improve the reliability of the results and reduce the impact of random errors from the heuristic algorithm, each benchmark is executed 10 times using our approach and baselines to ensure fair and precise measurements. The results are averaged in the same environment.

4.1 Results without Feedthrough Optimization

First, to verify the effectiveness of the SCB-Tree and optimization framework, we test the performance of the general fixed-outline floorplanning using FTAFP without feedthrough optimization. The evaluation metrics used in this experiment are the half-perimeter wirelength (HPWL), aspect ratio (AR), and CPU runtime (RT). Similar fixed-outline and AR constraints have been added to all baselines for a fair comparison. As shown inTable 1, FTAFP has reduced HPWL compared to all baselines under the fixed-outline constraint and strictly meets the aspect ratio constraint. Our FTATP results in average wirelength reductions of 23%, 12%, and 6% over [13], [11] and [15], respectively. This improvement can be attributed to the combination of slack computation and corner stitching. The proposed SCB-Tree makes full use of the whitespace, resulting in a more compact overall floorplan and thereby reducing wirelength. Due to the slack computation when packing the module, there is a slight increase in the runtime of around 13% compared to [15].

4.2 Results with Feedthrough Optimization

As the main optimization goal of this work, we test the performance of the proposed feedthrough optimization method. The constraints and settings are the same as in the previous subsection, except that the feedthrough optimization is added. The results are illustrated in Table 2. We developed an evaluator to analyze feedthrough-related metrics for the baselines by parsing their floorplan results and computing all metrics using the same model as our floorplanner.

Our method demonstrates significant improvements in feedthrough optimization compared to the three baselines. Compared to the CB-Tree, the FTNUM and FTWL metrics are reduced by 12% and 28% on average respectively, and CEL is also significantly improved by 25%. HPWL increased by only 4%, which is considered acceptable in the hierarchical design of large-scale SoC. Additionally, Table 2 demonstrates that our methods excel particularly in cases with more multi-module nets(e.g. ami33 and ami49) and a large number of modules(e.g. n100 and n200). For the former, the reason lies in our net simplification which encourages more adjacent arrangements of modules in the same net. Larger-scale designs tend to have more feedthrough wires with longer spans and traveling through more modules. Consequently, our feedthrough optimization proves to be more effective in these scenarios.

Table 1: Comparison of baselines with FTAFP, without feedthrough optimization.

Benchmarks			Corblivar [13]			SP-FOFP [11]			CB-Tree [15]			FTAFP		
Case	# Modules	# Nets	HPWL	AR	RT	HPWL	AR	RT	HPWL	AR	RT	HPWL	AR	RT
n10	10	118	47,899	1.02	**0.05**	43,071	**1.00**	0.1	42,007	0.99	3.21	**40,778**	1.01	3.25
n30	30	349	175,684	0.98	**0.43**	160,774	**1.00**	0.85	126,952	0.98	6.51	**111,877**	**1.00**	7.90
n50	50	485	208,356	0.74	**1.22**	193,478	**1.00**	2.69	165,783	0.99	8.12	**162,011**	**1.00**	10.25
n100	100	885	328,202	0.90	**6.13**	304,279	**1.00**	7.91	310,582	**1.00**	21.19	**293,497**	**1.00**	24.31
n200	200	1585	607,503	0.86	**33.04**	554,992	**1.00**	31.71	546,521	**1.00**	42.80	**524,989**	**1.00**	52.01
ami33	33	123	99,355	1.08	**0.48**	91,037	**1.00**	0.97	96,166	1.01	6.69	**90,503**	**1.00**	7.16
ami49	49	408	1,202,310	0.99	**0.96**	1,010,759	**1.00**	2.06	1,042,454	1.01	7.50	**1,007,618**	**1.00**	8.46
Ratio			1.23	0.94	**0.18**	1.12	**1.00**	0.24	1.06	0.99	0.87	**1.00**	**1.00**	1.00

Table 2: Comparison of HPWL, common edge length (CEL), feedthrough number (FTNUM) and feedthrough wirelength (FTWL).

Case	Corblivar [13]				SP-FOFP [11]				CB-Tree [15]				FTAFP			
	HPWL	CEL	FTNUM	FTWL	HPWL	CEL	FTNUM	FTWL	HPWL	CEL	FTNUM	FTWL	HPWL	CEL	FTNUM	FTWL
n10	47,899	4,369	149	15,440	43,701	4,489	146	14,098	**42,007**	4,619	141	11,427	43,625	**4,934**	**138**	**10,935**
n30	175,684	1,698	483	52,325	160,774	2,363	478	46,433	**126,952**	2,142	467	47,742	142,507	**3,143**	**454**	**43,181**
n50	208,356	2,436	798	110,124	193,478	1,645	807	102,863	**165,783**	2,186	776	109,105	182,524	**4,141**	**744**	**91,213**
n100	328,202	2,359	1,474	179,963	**304,279**	3,339	1,781	146,137	310,582	2,052	1,461	176,137	304,584	**6,545**	**1,358**	**134,075**
n200	607,503	1,345	2,971	409,287	554,992	4,012	3,442	306,281	**546,521**	4,263	3,672	386,471	549,286	**6,053**	**2,802**	**294,796**
ami33	99,355	45,822	172	45,810	**91,037**	40,376	194	62,531	96,166	58,730	186	51,495	93,674	**67,886**	**152**	**29,848**
ami49	1,202,310	39,634	824	2,149,060	**1,010,759**	105,112	722	1,180,280	1,042,454	139,426	662	1,089,270	1,047,228	**177,436**	**597**	**836,962**
Ratio	1.13	0.52	1.13	1.55	1.02	0.65	1.15	1.34	**0.96**	0.75	1.12	1.28	1.00	**1.00**	**1.00**	**1.00**

5 CONCLUSION

We propose a feedthrough-aware floorplanner named FTAFP to solve the feedthrough challenge faced by the hierarchical design of large-scale SoCs. To the best of our knowledge, we are the first to model and optimize the feedthrough problem in floorplanning. We introduce SCB-Tree to better satisfy the fixed-outline constraint and optimization objectives and propose a two-phase SA framework with targeted perturbation operations. Experimental results demonstrate that FTAFP can significantly optimize objectives such as feedthrough wirelength, quantity, and common edge resources, thereby better meeting actual design needs.

ACKNOWLEDGMENT

This work is supported by the National Key R&D Program of China (2022YFB2901100), the National Natural Science Foundation of China (No. 62404021), the Beijing Natural Science Foundation (No. 4244107, QY24216), the MIND project (MINDXZ202404), and AI Chip Center for Emerging Smart Systems (ACCESS), Hong Kong.

REFERENCES

[1] S. Garg and N. K. Shukla, "A study of floorplanning challenges and analysis of macro placement approaches in physical aware synthesis," *International Journal of Hybrid Information Technology*, vol. 9, no. 1, pp. 279–290, 2016.
[2] I.-L. Tseng, "Challenges in floorplanning and macro placement for modern SoCs," in *ACM International Symposium on Physical Design (ISPD)*, 2024, p. 71–72.
[3] J. Shin, J. Darringer, G. Luo, M. Aharoni, A. Lvov, G.-J. Nam, and M. Healy, "Floorplanning challenges in early chip planning," in *IEEE International System-on-Chip Conference (SOCC)*, 2011, pp. 388–393.
[4] Y. Hong, C. Huang, Y. Gao, and C. Li, "Channel based soc feedthrough insertion methodology," in *International Conference on Communications, Circuits and Systems (ICCCAS)*, 2022, pp. 125–130.
[5] Y. Zhang, A. Chakraborty, S. Chowdhury, and D. Z. Pan, "Reclaiming over-the-IP-block routing resources with buffering-aware rectilinear Steiner minimum tree construction," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2012, pp. 137–143.
[6] F. Huang, D. Liu, X. Li, B. Yu, and W. Zhu, "Handling orientation and aspect ratio of modules in electrostatics-based large scale fixed-outline floorplanning," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023, pp. 1–9.
[7] X. Li, K. Peng, F. Huang, and W. Zhu, "PeF: Poisson's equation-based large-scale fixed-outline floorplanning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 6, pp. 2002–2015, 2023.
[8] M. Kuwano and Y. Takashima, "Stable-lse based analytical placement with overlap removable length," in *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, 2010, pp. 115–120.
[9] A. B. Kahng and Q. Wang, "Implementation and extensibility of an analytic placer," in *ACM International Symposium on Physical Design (ISPD)*, 2004, pp. 18–25.
[10] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 15, no. 12, pp. 1518–1524, 1996.
[11] Q. Xu, S. Chen, and B. Li, "Combining the ant system algorithm and simulated annealing for 3D/2D fixed-outline floorplanning," *Applied Soft Computing*, vol. 40, no. C, p. 150–160, 2016.
[12] X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C.-K. Cheng, and J. Gu, "Corner block list: An effective and efficient topological representation of non-slicing floorplan," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2000, pp. 8–12.
[13] J. Knechtel, E. F. Y. Young, and J. Lienig, "Structural planning of 3D-IC interconnects by block alignment," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2014, pp. 53–60.
[14] Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu, "B*-Trees: a new representation for non-slicing floorplans," in *ACM/IEEE Design Automation Conference (DAC)*, 2000, p. 458–463.
[15] H.-F. Tsao, P.-Y. Chou, S.-L. Huang, Y.-W. Chang, M. P.-H. Lin, D.-P. Chen, and D. Liu, "A corner stitching compliant B*-tree representation and its applications to analog placement," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2011, pp. 507–511.
[16] Y. Liu, Z. Ju, Z. Li, M. Dong, H. Zhou, J. Wang, F. Yang, X. Zeng, and L. Shang, "GraphPlanner: Floorplanning with graph neural network," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 28, no. 2, 2022.
[17] Y. Liu, H. Zhou, J. Wang, F. Yang, X. Zeng, and L. Shang, "Hierarchical graph learning-based floorplanning with dirichlet boundary conditions," *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, vol. 32, no. 5, pp. 810–822, 2024.
[18] Z. He, Y. Ma, L. Zhang, P. Liao, N. Wong, B. Yu, and M. D. Wong, "Learn to floorplan through acquisition of effective local search heuristics," in *IEEE International Conference on Computer Design (ICCD)*, 2020, pp. 324–331.
[19] Q. Xu, H. Geng, S. Chen, B. Yuan, C. Zhuo, Y. Kang, and X. Wen, "GoodFloorplan: Graph convolutional network and reinforcement learning-based floorplanning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 41, no. 10, pp. 3492–3502, 2021.
[20] M. Amini, Z. Zhang, S. Penmetsa, Y. Zhang, J. Hao, and W. Liu, "Generalizable floorplanner through corner block list representation and hypergraph embedding," in *ACM International Conference on Knowledge Discovery and Data Mining (KDD)*, 2022, pp. 2692–2702.
[21] J. Cong and M. Xie, "A robust mixed-size legalization and detailed placement algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 8, pp. 1349–1362, 2008.
[22] W. Dai, L. Wu, and S. Zhang. (2000) GSRC benchmarks. [Online]. Available: http://vlsicad.eecs.umich.edu/BK/GSRCbench/
[23] M. C. of North Carolina (MCNC). (2000) MCNC benchmarks. [Online]. Available: http://vlsicad.eecs.umich.edu/BK/MCNCbench/