RTL-MP: Toward Practical, Human-Quality Chip Planning and Macro Placement

Andrew B. Kahng
University of California San Diego
La Jolla, CA, USA
abk@ucsd.edu

Ravi Varadarajan
University of California San Diego
La Jolla, CA, USA
rvaradarajan@ucsd.edu

Zhiang Wang
University of California San Diego
La Jolla, CA, USA
zhw033@ucsd.edu

ABSTRACT

In a typical RTL-to-GDSII flow, floorplanning plays an essential role in achieving decent quality of results (QoR). A good floorplan typically requires interaction between the frontend designer, who is responsible for the functionality of the RTL, and the backend physical design engineer. The increasing complexity of macro-dominated designs (especially machine learning accelerators with autogenerated RTL) has made the floorplanning task even more challenging and time-consuming. In this paper, we propose *RTL-MP*, a novel macro placer which utilizes RTL information and tries to "mimic" the interaction between the frontend RTL designer and the backend physical design engineer to produce human-quality floorplans. By exploiting the logical hierarchy and processing logical modules based on connection signatures, *RTL-MP* can capture the dataflow inherent in the RTL and use the dataflow information to guide macro placement. We also apply autotuning [37] to optimize hyperparameter settings based on input designs. We have built *RTL-MP* based on OpenROAD infrastructure [25, 49] and applied *RTL-MP* to a set of industrial designs. *RTL-MP* outperforms state-of-the-art commercial macro placers and achieves QoR similar to that of handcrafted floorplans.

CCS CONCEPTS

• **Hardware → Electronic design automation**; **Physical design (EDA)**; **Partitioning and floorplanning**.

KEYWORDS

Macro placement, RTL-driven, dataflow

ACM Reference Format:
Andrew B. Kahng, Ravi Varadarajan, and Zhiang Wang. 2022. RTL-MP: Toward Practical, Human-Quality Chip Planning and Macro Placement. In *Proceedings of the 2022 International Symposium on Physical Design (ISPD '22), March 27–30, 2022, Virtual Event, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3505170.3506731

1 INTRODUCTION

Modern SoCs can have blocks that are very complex and arduous to comprehend by humans, with multiple millions of standard cells and hundreds or even thousands of macros. This has made fast prototyping of blocks a necessity for efficient design space exploration of the SoC. Prototyping of a block entails taking a footprint from the top level which includes the fixed outline, pin locations and constraints for the block, and then determining the feasibility of implementation. A key task in this process is floorplan generation, i.e., determining the placement of the macros in the block. Once the macro placement is fixed, it is rarely changed during the subsequent stages of the standard cell place and route *(P&R)* flow. Therefore, the macro placement can have a significant impact on final design QoR, and a "bad" floorplan may leave performance on the table or lead to design convergence issues. In today's flows floorplanning is done manually by a backend designer who must capture the structure of the design and dataflow through frequent interactions with the frontend designer. This becomes particularly challenging with RTL designs produced by automatic RTL generators: such designs can have complex RTL structures with very long autogenerated module names, with no human frontend designer available.

In this work, we propose a novel macro placer, *RTL-MP*, which utilizes RTL information and tries to "mimic" the behavior of human experts. By exploiting logical hierarchy and processing logical modules based on connection signatures, *RTL-MP* can fully capture the dataflow defined by RTL designers and use the dataflow information to guide macro placement. Our main contributions are:

- We propose a novel macro placer called *RTL-MP* which converts the structural netlist representation of the design into a "clustered netlist", by analyzing logical hierarchy, dataflow, connectivity between macros and *input-output (IO)* pins, and critical timing paths. Then, *RTL-MP* operates on the clustered netlist model and generates a legal macro placement. This is very similar to the way human experts create manual floorplans.
- We present a powerful clustering method, which enables the abstraction of the block into a clustered netlist representation. Our clustering method fully exploits the logical hierarchy of the original RTL structure and the regularity and connectivity of macros. In contrast to all existing works, we merge "small" clusters based on "connection signature" to further reduce the complexity of the clustered netlist without sacrificing the functional interactions between the logical components. Macros are also grouped into array structures based on regularity and the connection signatures. Buffer "transparency" is used to effectively capture the dataflow of the RTL structure in the clustered netlist. Implicit timing information such as the number of flop stages (or *hops*) between clusters is also captured in the clustered netlist model.
- Our macro placer takes care of pin access, notch region avoidance and the common practice of pushing macros to peripheries, which are key factors that backend experts usually consider when they place macros manually.
- We experimently confirm that *RTL-MP* can generate stable and predictable macro placements based on users' specifications. This matches how backend designers rely on previous revisions of the design to do prototyping, and how macro locations are left unchanged when RTL changes are small. *RTL-MP* allows users to

ISPD '22, March 27–30, 2022, Virtual Event, Canada.

ACM ISBN 978-1-4503-9210-5/22/03…$15.00
https://doi.org/10.1145/3505170.3506731

specify preferred locations for macros, and/or macro placement blockages, thus enabling stable and predictable macro placements for small changes in the RTL or constraints.

- We apply autotuning to optimize the weights of different objectives for a given input design. A macro placer usually must consider many objectives, such as wirelength, timing, overlap and so on. Since each design can have different utilization, macro types, critical timing paths and logical hierarchy, a set of "perfectly" tuned weights for one design may fail to achieve a decent macro placement for another design. Backend design engineers usually launch multiple runs to sweep these weights and pick the best candidate as the starting point. Such a "grid" sweep is usually inefficient and quite time-consuming. In this work, we combine our macro placer and a scalable hyperparameter tuning tool – *Tune* [37] – to tune weights for different objectives automatically. Our experiment results confirm the effectiveness of autotuning.

The remaining sections are organized as follows. Section 2 reviews related works. Sections 3-6 discuss our approach. Section 7 shows experiment results, and Section 8 concludes the paper.

2 RELATED WORK

There are many published works on macro placement. We classify these into three categories: analytical methods, packing-based methods and ML-based methods. Analytical methods model the objectives to be optimized (wirelength, routability, etc.) as terms in the objective function or constraints, then solve the constrained optimization problem mathematically. For example, [18] models the nonuniformity of the module distribution as a penalty term and solves the constrained optimization problem to realize soft-module floorplanning. Packing-based methods typically combine floorplan representations with heuristics such as Simulated Annealing (SA) or Particle Swarm Optimization (PSO) to solve the floorplanning problem. Researchers have proposed a number of efficient representations, such as Sequence Pair [15], Corner Stitching [31], B*-tree [4, 7], MP-tree [40], CP-tree [42] and MDP-tree [41]. ML-based methods apply machine learning techniques such as expert systems [3, 16] or Reinforcement Learning [12, 22, 23] to perform macro placement.

Most existing macro placers focus on legalizing the placement of macros and optimizing wirelength and/or routability without considering design features such as design hierarchy, macro regularity, dataflow, macro guidance, pin access and notch area avoidance. On the other hand, chip experts do pay attention to these design features to produce high-quality macro placements. To automatically generate a competitive, closer to human-quality macro placement, some recent works have begun to consider these features.

[27]-[30] and [34, 35, 39, 40, 46, 47] utilize design hierarchy to guide macro placement. [20]-[24] and [30, 34, 35] exploit dataflow and/or timing information to improve the quality of macro placement. [28]-[30], [34, 35] and [41]-[44] reduce macro displacement to honor the macro guidance given by placement prototyping, and [32, 36, 40] as geometrical constraints directly. [28]-[30] and [40]-[44] can handle macro blockages and/or preplaced macros. [44, 46, 47] try to avoid notches during macro placement to improve routability and low-density regions during standard-cell placement. [29, 44] pay special attention to the effect of regular placement of macros. However, none of these previous works provides all of the above features. Table 1 summarizes the main differences between our *RTL-MP* method and previous works.

Methods	Design Hier	Shape Reg	Dataflow Timing	Macro Guide	Pin Access	Macro Blockage	Notch Align
[2]-[18]							
[27, 39]	✓						
[20]-[24]			✓				
[32, 36]				✓			
[41]-[43]						✓	
[28]	✓					✓	
[46, 47]	✓						✓
[34, 35]	✓		✓	✓			
[40]	✓			✓		✓	
[44]		✓				✓	✓
[29]	✓	✓		✓		✓	
[30]	✓		✓	✓		✓	
RTL-MP	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison with previous methods. Columns 2-8 respectively indicate use of design hierarchy; use of regular placement of macros; use of dataflow and/or timing; handling of macro guidance (preferred location or regions); handling of pin access; handling of macro blockages and/or preplaced macros; and handling of notches.

3 OUR APPROACH

A central aspect of *RTL-MP* is its conversion of the structural netlist representation of the design into a *clustered netlist* abstraction model. The clustered netlist not only reduces the problem size, but also helps to break down the overall design closure problem into (i) a *global* design closure problem that operates on the clustered netlist, and (ii) a *detailed* design closure problem that operates on the detailed gate-level netlist. The clustered netlist is generated based on analyzing logical hierarchy, dataflow, the connection between macros and IO pins, and timing constraints. *RTL-MP* is built on open-source OpenROAD infrastructure [25, 49]. As shown in Figure 1, it consists of four major components.

- The **Autoclustering Engine** converts the structural netlist representation of the RTL design into a clustered netlist. In the clustered netlist abstraction model, nodes are clusters and nets are bundled connections between clusters.
- The **Shape Engine** determines possible shapes and aspect ratio constraints for all the clusters. This considers the contents of the clusters, as clusters containing macros can only have discrete shape choices based on the tiling of the macros in the cluster.
- The **Macro Placement Engine** places all the clusters and finalizes the shape of each cluster.
- The **Pin Alignment Engine** finalizes the location and orientation for each macro.

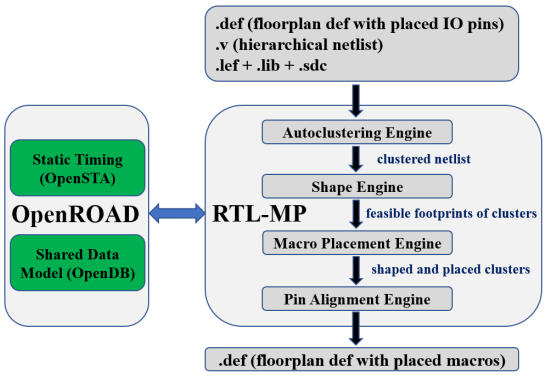

Figure 1: *RTL-MP* flow.

4 AUTOCLUSTERING ENGINE

Clustering is an essential preprocessing step for macro placement. In this step the structural netlist representation of the design is converted to a clustered netlist in which the nodes are clusters and nets are bundled connections between clusters. A cluster [45] refers to a group of densely-connected instances such that the number of the interconnections among elements inside the group is much larger than the number of connections spanning different groups. Based on types of instances within a cluster, we classify a cluster into one of the following three types:

- **Standard-Cell Cluster** containing only standard cells;
- **Macro Cluster** containing only macros;
- **Mixed Cluster** containing both macros and standard cells.

The clustering step is typically done by users interactively and in a top-down manner, by analyzing logical hierarchy, dataflow, connections between macros and IO pins, and critical timing paths [48]. Such analysis helps the user to understand the structure of the design and the dataflow, which provides insights into the "ideal" locations of the various clusters and macros.

While it is useful for users to perform clustering manually and understand physical implications of the design, it is also important to have an autoclustering engine that can fully and automatically generate meaningful clusters. This is especially true for designs produced by automatic RTL generators for ML applications; these can have complex RTL structure with long, inscrutable autogenerated module names. We therefore propose autoclustering based on logical hierarchy of the design, connection signature of clusters, and timing hops or indirect connections between macros and IO pins.

The detailed algorithm is shown in Algorithm 1. The entire autoclustering algorithm can be divided into the following steps.

- **Step 1:** [Lines 3-4] We create bundled pins by dividing each boundary edge evenly into segments, and assigning each bundled pin to the center of its corresponding segment. (In the experiments below, we set the number of bundled pins on each boundary edge to 3). Each bundled pin is treated as a cluster without physical area. A design may have hundreds of IO pins. The bundled pins can reduce connection complexity significantly and improve the robustness of our macro placer.
- **Step 2:** [Lines 5-34] We traverse the logical hierarchy of the netlist in a top-down manner. During this step, we break down each large (number of instances > *max_num_inst*) hierarchical logical module into submodules according to the logical hierarchy, and merge the small submodules based on connection signatures, i.e., connection topology or adjacency between clusters. Only small (number of instances < *min_num_inst*) clusters with the same connection signature will be merged. Salient details include: (i) sameness of connection signature is based on connection topology (i.e., adjacency) between clusters, rather than the exact number of connections between clusters; (ii) all multiple-pin nets are decomposed using a directed star model; (iii) to remove the "fake" difference caused by common global nets such as scan or reset signals, a parameter -*net_threshold* is used to determine whether two clusters are connected; and (iv) because the existence of buffers in a hierarchical netlist can make connection topology between logical modules ambiguous, we make related connections "transparent" and ignore the area of buffers.
- **Step 3:** [Line 35] We break large clusters (number of instances > *max_num_inst*) that have no child modules into smaller clusters, using the open-source **MLPart** hypergraph bipartitioner [38].

Algorithm 1: Autoclustering Engine

Input : $G \leftarrow$ synthesized hierarchical netlist
Output : $G_{cluster} \leftarrow$ clustered representation of G

```
1  initialize an empty queue break_cluster_queue
2  initialize empty lists cluster_list, merge_cluster_list
3  create bundled pins by dividing each boundary into bundled
     segments
4  model each bundled pin as a cluster and add it to cluster_list
5  create cluster top_cluster for top module
6  cluster_list.push_back(top_cluster)
7  if top_cluster.GetNumInst() > max_num_inst or
     top_cluster.GetNumMacro() > max_num_macro then
8  |  break_cluster_queue.push_back(top_cluster)
9  end
10 while !break_cluster_queue.empty() do
11 |  c ← break_cluster_queue.front()
12 |  break_cluster_queue.pop()
13 |  if c has child modules then
14 |  |  for each child module child_module in c do
15 |  |  |  create a cluster child for child_module
16 |  |  |  if child.GetNumMacro() > max_num_macro or
   |  |  |    child.GetNumInst() > max_num_inst then
17 |  |  |  |  break_cluster_queue.push_back(child)
18 |  |  |  |  cluster_list.push_back(child)
19 |  |  |  else if child.GetNumMacro() > min_num_macro
   |  |  |    or child.GetNumInst() > min_num_inst then
20 |  |  |  |  cluster_list.push_back(child)
21 |  |  |  else
22 |  |  |  |  merge_cluster_list.push_back(child)
23 |  |  |  end
24 |  |  end
25 |  |  create a cluster glue_logic for child leaf instances of c
26 |  |  if glue_logic.GetNumInst() < min_num_inst or
   |  |    glue_logic.GetNumMacro() < min_num_macro then
27 |  |  |  merge_cluster_list.push_back(glue_logic)
28 |  |  else
29 |  |  |  cluster_list.push_back(glue_logic)
30 |  |  end
31 |  end
32 |  merge clusters in merge_cluster_list based on connection
   |    signature
33 |  cluster_list.remove(c)
34 end
35 call MLPart to break down clusters with number of instances larger
     than max_num_inst
36 partition mixed clusters into standard-cell clusters and macro clusters
37 For each macro cluster, mark its macros as single-macro macro
     clusters and group them based on connection signature. Then, for
     each newly formed macro cluster with macros of different sizes,
     break it down and group its macros based on their sizes
38 add virtual connections between macro clusters and its
     corresponding standard-cell clusters
39 add virtual connections between clusters based on information flow
     and number of hops
40 return clustered netlist G_cluster
```

- **Step 4:** [Lines 36, 38] We partition mixed clusters into standard-cell clusters and macro clusters. Separating macro clusters and standard-cell clusters makes it easier to determine possible footprints for all the clusters. We add a virtual weighted connection

between each macro cluster and its corresponding standard-cell cluster to induce the macro placer to place them together.

- **Step 5:** [Line 37] For each macro cluster, we mark each of its macros as a *single-macro macro cluster* (that is, a macro cluster consisting of only one macro) and group these single-macro macro clusters based on connection signature. If a newly-formed macro cluster has macros of different sizes, we break it down and group macros within it based on the footprint of macros. Grouping macros based on identical footprint enables decent tilings of macros in a given macro cluster, thereby achieving regular placement of macros.
- **Step 6:** [Line 39] We add virtual connections between clusters to take care of critical timing paths. The physical distances between components on critical timing paths should be minimized to improve performance. One approach is to determine all the critical timing paths (e.g., having negative slack) and overlay them on clusters. This is time-consuming and unnecessary since slack calculation is not accurate at the floorplan stage. In this work, we adopt a similar idea as [34, 35] and define virtual connections as

$$Virtual\_Connections(A, B) = \frac{Information\_Flow(A,B)}{2^{Num\_Hops}} \quad (1)$$

as shown in Figure 2. Here, *Information_Flow* corresponds to connection bitwidth and *Num_Hops* is the length of a shortest path of registers between clusters. However, unlike the *Dataflow_Affinity* defined in [34, 35], we use a more aggressive decaying factor as in [22, 23] to capture the most important "indirect" connections. If the register distance (*Num_Hops*) between clusters is greater than 4, then no virtual connection is added. Since the positions of standard cells will be optimized later by the placement engine, we consider only macro clusters so as to reduce runtime without compromising final QoR. Further, we treat each bundled pin as a macro cluster without physical area, so that indirect connections between macro clusters and IO pins can also be accounted for.
- We use six hyperparameters: *max_num_inst* and *min_num_inst*, the maximum and minimum number of standard cell instances in a cluster; *max_num_macro* and *min_num_macro*, the maximum and minimum number of macros in a cluster; *virtual_weight*, the virtual weight between each macro cluster and its corresponding standard-cell cluster; and *net_threshold*, the minimum number of connections between two clusters needed for the clusters to be considered as connected. These hyperparameters are determined empirically.

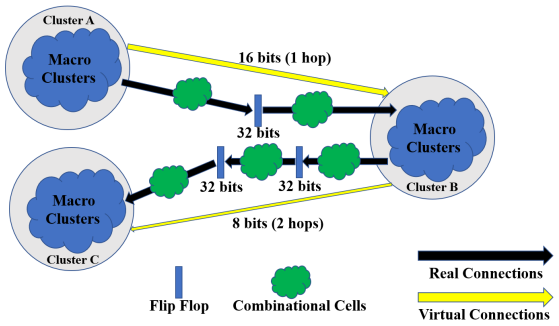

Figure 2: Virtual connections between macro clusters. Black arrows represent real connections and yellow arrows represent virtual connections. The virtual connections represented by yellow arrows can help capture critical timing paths between Clusters A, B and C.

5 TOP LEVEL MACRO PLACER

After autoclustering, we have a clustered netlist in which the nodes are clusters and the nets are bundled connections between clusters. Since we have partitioned all mixed clusters into standard-cell clusters and macro clusters, we will not have mixed clusters in the clustered netlist. In this step, we model each standard-cell cluster as a *soft block*, i.e., having fixed area, with upper- and lower-bounded continuously variable aspect ratios [19]. We also model each macro cluster as a *semi-soft block*, i.e., having fixed area, with discrete allowable aspect ratio choices, usually corresponding to alternative tiling realizations of the group of macros in the cluster [19]. We then call the shape engine to calculate possible aspect ratios for each block. After that, we call the macro placement engine to place all the blocks in the clustered netlist.

5.1 Shape Engine

The shape engine is used to determine possible aspect ratios for all the blocks (in our work, all the blocks are rectangles). For a soft block, the aspect ratio is specified by the upper bound (*max_ar*) and the lower bound (*min_ar*) given by users. For a semi-soft block, since we have grouped macros based on size in the autoclustering step, all the macros in the same macro cluster have the same size. We thus simply enumerate all the possible macro tilings and keep the macro tilings with minimum area. An allowed macro tiling must fit into the fixed floorplan.

5.2 Macro Placement Engine

After calling the shape engine, we have aspect ratio choices for all the clusters. The function of the macro placement engine is then to determine position and shape for each of the clusters. In this phase, we assume that the bundled pin of each block is in the center of the block. We use Sequence Pair [15] to represent blocks in the netlist and Simulated Annealing [26] to optimize the cost function. Further, we adopt a "go-with-the-winners" [1] scheme to further improve the performance of Simulated Annealing and we set the number of threads to 10 in our experiments.[1] As applied in this phase, the Sequence Pair-based annealing supports four solution perturbation (move) operators with respective probabilities 0.3, 0.3, 0.3 and 0.1:

- **Op1:** Swap two blocks in first sequence;
- **Op2:** Swap two blocks in second sequence;
- **Op3:** Swap two blocks in both sequences; and
- **Op4:** Resize a block. For soft blocks (standard-cell clusters), we use the same soft-block resizing algorithm as in [7]; for semi-soft blocks (macro clusters), we change the aspect ratio randomly.

To generate a decent, human-quality floorplan, we have enhanced the macro placement engine to handle the following constraints:

- **fixed outline:** All blocks should be placed within the fixed outline specified by users.
- **macro peripheral bias:** All macros should be pushed to peripheries as much as possible.
- **pin access:** All macros should be kept from blocking access of IO pins.
- **macro blockage:** All macros should not overlap with macro (placement) blockages. Preplaced macros can be treated as macro blockages, hence our macro placer can also handle preplaced macros.

[1] This number of threads is easy to accommodate on any standard server.

- **macro guidance:** All blocks should be placed near specified regions if users provide such constraints.
- **notch avoidance:** A decent floorplan should avoid "dead space" which cannot be used effectively by P&R tools.

We refer to the example of Figure 3 when describing these enhancements, in the following.

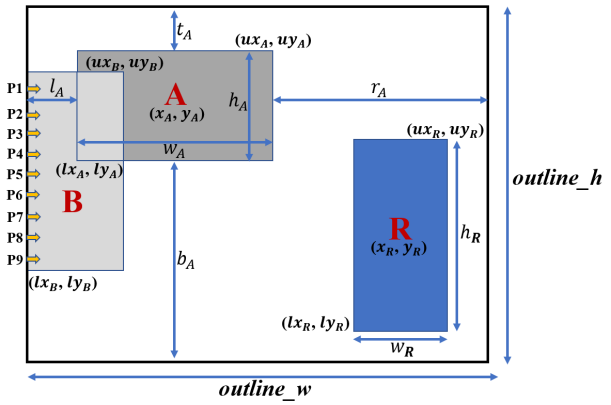

Figure 3: Example to illustrate macro placer enhancements. *outline_w* and *outline_h* denote the width and height of the fixed outline. *A* is a semi-soft block (macro cluster) with width w_A, height h_A and center (x_A, y_A). (lx_A, ly_A) and (ux_A, uy_A) are the lower-left and upper-right coordinates of *A*. *R* is the preferred region for *A*. *B* is a special blockage for IO pins. P1, P2, ..., P9 are IO pins.

5.2.1 Fixed Outline. In reality, the use of floorplanning during the chip design process almost always comes after the die size and package have been chosen. Moreover, when the top-level SoC is split into implementation blocks, the top-level constraints for the blocks usually include the DEF file that defines the fixed outline of the block along with IO pin positions, as well as timing constraints. Thus, floorplanning is properly cast as a fixed-die problem [19]. Floorplans with rectilinear outlines can be modeled as a rectangular outline with "keep-out" blockages. We accomodate the fixed-die constraint by adding an outline violation penalty $p_{outline}$ into the macro placement cost function:

$$\begin{aligned} floorplan\_area &= max(outline\_w, w) \times max(outline\_h, h) \\ outline\_area &= outline\_w \times outline\_h \\ p_{outline} &= floorplan\_area - outline\_area \end{aligned} \tag{2}$$

where $outline\_w$ and $outline\_h$ are the width and height of the fixed outline; w and h are the width and height of the current floorplan; and $p_{outline}$ is the penalty for the fixed-outline violation.

5.2.2 Macro Peripheral Bias. When expert engineers create a floorplan manually, they prefer to place macros at the peripheries, to avoid creating a standard-cell region that is difficult for the P&R tool to handle. We mimic this behavior by adding a *peripheral bias* term into the cost function. For the semi-soft block (macro cluster) A shown in Figure 3, we define a bias penalty p_{bias_A} for A as

$$\begin{aligned} l_A &= lx_A \\ b_A &= ly_A \\ r_A &= \|outline\_w - lx_A - w_A\| \\ t_A &= \|outline\_h - ly_A - h_A\| \\ p_{bias\_A} &= min(l_A, b_A, r_A, t_A)^2 \times A.GetNumMacro() \end{aligned} \tag{3}$$

5.2.3 Pin Access and Macro Blockage. When engineers create floorplans manually, they will leave some empty space between macros and IO pins to improve pin accessibility. We mimic this behavior by creating special blockages around IO pins. This kind of special blockages only affects semi-soft blocks (macro clusters); it will not influence a soft block consisting of only standard cells. In the example of Figure 3, for a semi-soft block (macro cluster) A and a special blockage B, we define a pin access penalty $p_{pin\_A\_B}$ for A and B:

$$\begin{aligned} width &= max(0.0, min(ux_A, ux_B) - max(lx_A, lx_B)) \\ height &= max(0.0, min(uy_A, uy_B) - max(ly_A, ly_B)) \\ p_{pin\_A\_B} &= width \times height \times A.GetNumMacro() \end{aligned} \tag{4}$$

Aside from these special blockages around IO pins, which are generated by *RTL-MP* automatically, our macro placer also allows users to specify other macro blockages. We handle these additional macro blockages in the same manner as special blockages for IO pins.

5.2.4 Macro Guidance and Incremental Use Support. During prototyping and floorplan design exploration, backend engineers often draw from experience with previous revisions of the design. *RTL-MP* allows users to specify preferred locations for some or all the macros by adding a macro guidance penalty term into the cost function. In contrast to the classical fence constraint, *RTL-MP* tries to place macros around the preferred locations instead of forcing the macros into fences. Figure 3 shows how a semi-soft block A may have a specified preferred region R (which will not influence other blocks). In this example, a macro guidance penalty term $p_{guidance\_A}$ is applied for A:

$$\begin{aligned} width &= w_R + w_A \\ height &= h_R + h_A \\ x\_dist &= \|x_R - x_A\| - width \\ y\_dist &= \|y_R - y_A\| - height \\ p_{guidance\_A} &= max(0.0, x\_dist) + max(0.0, y\_dist) \end{aligned} \tag{5}$$

5.2.5 Notch avoidance. Notches are small P&R regions in the floorplan that cannot be effectively used for standard-cell placement during ensuing optimization stages of physical implementation. These usually appear between macro edges and the design boundary, or between adjacent macro clusters. As final standard-cell utilizations in notch regions are typically very low, the presence of notches increases effective core utilization and hence can cause routability issues. With this in mind, our macro placer adds a notch region penalty term into the cost function. The method of notch penalty calculation is shown in Algorithm 2.

In summary, the final cost function of our macro placer is

$$\begin{aligned} cost = \alpha \times Area + \beta \times WL + \gamma \times p_{outline} + \zeta \times p_{bias} \\ + \eta \times p_{blockage} + \theta \times p_{guidance} + \lambda \times p_{notch} \end{aligned} \tag{6}$$

where $Area$ is the area of the current floorplan, WL is the wirelength (HPWL), $p_{outline}$ is the penalty for violating the fixed outline constraint, p_{bias} is the penalty to promote macro peripheral bias, $p_{blockage}$ is the penalty for pin access and macro blockage, $p_{guidance}$ is the penalty for macro guidance, p_{notch} is the penalty for notch regions, and α, β, γ, ζ, η, θ, λ are the corresponding weights. $Area$, WL, $p_{outline}$, p_{bias}, $p_{blockage}$, $p_{guidance}$ and p_{notch} are each normalized to the corresponding initial value.

The reader will note that determining "optimal" weights can be nontrivial. On the one hand, results for a given input design can change significantly with different weight values. On the other hand, a set of "perfectly" tuned weights for one design may fail to achieve

Algorithm 2: Notch Penalty Calculation

$p_{notch} \leftarrow 0.0$
$notch_w \leftarrow \frac{outline\_w}{10.0}$
$notch_h \leftarrow \frac{outline\_h}{10.0}$
$w \leftarrow$ width of current floorplan
$h \leftarrow$ height of current floorplan
if $w > outline\_w \,||\, h > outline\_h$ **then**
  $area \leftarrow max(w, outline\_w) \times max(h, outline\_h)$
  $p_{notch} \leftarrow sqrt(\frac{area}{outline\_w \times outline\_h})$
  return p_{notch}
end
move macro clusters near to boundaries to corresponding boundary
use macro clusters along the boundaries as seeds to align other macro clusters
divide the entire floorplan into grids using the coordinates of macro clusters
for *each grid g* **do**
  if *g is surrounded by boundaries or macro clusters on at least three sides* **then**
    $w_g \leftarrow$ width of g
    $h_g \leftarrow$ height of g
    if $(w_g <= notch_w)$ || $(h_g <= notch_h)$ **then**
      $p_{notch} \leftarrow p_{notch} + sqrt(\frac{w_g \times h_g}{outline\_w \times outline\_h})$
    end
  end
end
return p_{notch}

a decent macro placement for another design due to different utilization, macro types, critical timing paths and logical hierarchy of input designs. Previous works (e.g., [34, 35]) usually launch multiple runs to sweep these weights, and pick the best result over all weight combinations. However, such a "grid" sweep can be inefficient and time-consuming. In our work, we use a scalable hyperparameter tuning tool – *Tune* [37] – to tune these weights automatically. We have found that when given a user-specified loss function, *Tune* can search the weight parameter space efficiently and find high-quality combinations of weight values. Details of our autotuning approach are presented in Section 7.

6 PIN ALIGNMENT ENGINE

After calling the top-level macro placer, we know the position and shape for each cluster. We next determine the location and orientation of macros in each macro cluster, one macro cluster at a time. For a given macro cluster A, we first create a bundled pin for each macro in A. The bundled pin of a macro is located at the geometric center of the bounding box of the macro's pin locations. Second, we extract the connections between macros in A and other clusters. Here, other clusters behave like fixed terminals. Any connections between macros in A are also considered. As in Step 2 of Algorithm 1, we "remove" all the buffers when we calculate connections. As with the Macro Placement Engine (Subsection 5.2), we use Sequence Pair [15] to represent macro placement in A and Simulated Annealing [26] to optimize the cost function, again with "go-with-the-winners" [1] to further improve the performance of Simulated Annealing, and 10 threads in all of our experiments. We use four solution perturbation (move) operators in the annealing, with respective probabilities 0.3, 0.3, 0.3 and 0.1:

- **Op1:** Swap two blocks in first sequence;
- **Op2:** Swap two blocks in second sequence;
- **Op3:** Swap two blocks in both sequences; and
- **Op4:** Flip all the macros.

The cost function used in this step is

$$cost = \alpha \times Area + \beta \times WL + \gamma \times p_{outline} \quad (7)$$

where $Area$ is the area of the current macro packing, WL is the wirelength (HPWL), $p_{outline}$ is the penalty for violating the fixed-outline constraint, and α, β and γ are corresponding weights.

7 EXPERIMENT RESULTS

Our macro placer (*RTL-MP*) is implemented with approximately 6K lines of C++ on top of OpenROAD [25, 49] infrastructure. We have validated our macro placer using five industrial designs and one machine learning accelerator generated by an automatic RTL generator. All studies use a commercial foundry 12nm technology (13 metal layers) with cell library and memory generators from a leading IP provider. Table 2 shows information about the designs. To show the effectiveness of our macro placer, the following five scenarios are evaluated and compared.[2]

- *Comm*: Macro placement is done by a 2020 release of a state-of-the-art commercial P&R tool using high-effort settings.
- *Manual*: All macros are manually placed by expert backend engineers.
- *TMP*: Macro placement is generated by TritonMP, which is the default macro placer in the OpenROAD project [25].
- *HiDaP*: Macros are placed by a recent state-of-the-art academic tool, HiDaP [34].
- *RTL-MP*: Results are obtained using our macro placer.[3]

Designs	Std Cells	Macros	IOs	Nets	Macro Blockage	Macro Guidance
swerv_wrapper [50]	78K	28	1416	94K		
ariane [52]	114K	37	495	131K		
simd	207K	46	4210	236K	✓	
coyote [51]	208K	15	784	327K		
bp_single [53]	323K	49	135	508K		
ca53	445K	25	1352	483K		✓

Table 2: Benchmarks. *simd* is the machine learning accelerator generated by an automatic RTL generator. User-specified macro blockages or macro guidance can be present.

Our experiments use the following flow. (1) We first synthesize a design using a state-of-the-art commercial synthesis tool. (2) Next, we determine the core size of the testcase and place all the IO pins using a manually-developed script. (3) Then, the macros are placed using different methods (*Comm*, *Manual*, *TMP*, *HiDaP* and *RTL-MP*).[4] (4) Finally, the placement of standard cells and routing are completed by the state-of-the-art commercial P&R tool. All metrics are collected after post-routing optimization.

As noted in Subsection 5.2, hyperparameter settings can significantly impact solution quality. We therefore apply the hyperparameter tuning tool *Tune* [37] to automatically tune weights in the cost

[2] [30] has recently reported an excellent dataflow-driven macro placer. Unfortunately, no testcases or executables can be released by their group. We also tried to compare against the mixed-size placer in OpenROAD, but were not able to generate legal floorplans for many of our designs.

[3] *RTL-MP*$_B$ denotes *RTL-MP* with user-specified macro blockages. *RTL-MP*$_L$ denotes *RTL-MP* with user-specified "loose" macro guidance. *RTL-MP*$_T$ denotes *RTL-MP* with user-specified "tight" macro guidance. These modes of operation are explored in the experiments below.

[4] All steps or methods referred to as "manual" are performed by an expert layout engineer who has over 30 years of industry experience in SoC and physical floorplanning.

function (Eq. 6) for different designs. An appropriate loss function is needed to guide the search process [54]. In our use of *Tune*, we define the loss function as

$$loss = 0.1 \times WL + 1.0 \times p_{outline} + 1.0 \times p_{bias} + 1.0 \times p_{blockage} + 1.0 \times p_{guidance} + 1.0 \times p_{notch} \quad (8)$$

Compared with Eq. 6, we remove the area term ($Area$) because the area is not important as long as the macro placement is valid.

The number of trials allowed to *Tune* affects QoR: more trials achieve better QoR at the cost of longer tuning time. In our experiments, we set the number of trials for each design to 20 and use 5 threads to obtain an acceptable tuning walltime equal to 4 times that of a single *RTL-MP* run, without any undue CPU needs. Besides the weights in the cost function (Eq. 6), we also add the hyperparameter *-min_ar* mentioned in Subsection 5.1 to the list of tuning parameters. We define values for remaining hyperparameters based on empirical experience: (i) we set *max_num_macro* = 12 and *min_num_macro* = 4; (ii) we set *virtual_weight* = 500 and *net_threshold* = 5. And, (iii) to accommodate designs of different sizes, we calculate *min_num_inst* and *max_num_inst* as

$$min\_num\_inst = max(1K, floor(\frac{total\_num\_inst}{50 \times 5K}) \times 5K)$$
$$max\_num\_inst = min(50K, 5 \times min\_num\_inst) \quad (9)$$

where *total_num_inst* is the total number of instances of the design.[5] The *HiDaP* tool also has hyperparameters that require tuning. We launch 20 runs[6] for each design to sweep hyperparameters and extract the best result in terms of minimum number of DRCs and better wirelength, because we cannot extract corresponding metrics (for purposes of loss function evaluation) from the executable we were provided. We report the metrics of *TMP* in the same manner.

Table 3 shows the experiment results after completion of post-routing optimization. Rows represent circuits and macro placement flows, and columns give information on number of standard cells, wirelength (in meters), number of DRCs, timing information (worst negative slack and total negative slack, in ns), power (in mW) and turnaround time (in minutes) for a single run.[7] Figure 4 gives example post-routing layouts. We see clearly that *RTL-MP* achieves the smallest routed wirelength for all testcases. The degradation in wirelength for other approaches is because the standard cell P&R tool has to insert more buffers/inverters and use more complicated routing patterns to achieve similar timing closure. To validate the effectiveness of the autotuning method, we compare the results of autotuning and grid sweep (denoted as *RTL-MP*$_G$) using design *ca53*. We can observe that the autotuning outperforms grid sweep in terms of all metrics. For *HiDaP*, we show the post-routing layout of design *swerv_wrapper* in Figure 4(a). We were unable to find parameter settings for *HiDaP* that yield competitive results for other designs, and therefore do not show any other comparison to *HiDaP* here.

Predictability of results under user-specified constraints such as macro blockages and macro guidance is one of the key features for a stable prototyping tool. In stark contrast to the chaotic behavior of commercial P&R tools [33], *RTL-MP* can handle these informative constraints quite well. As an example, we provide macro blockages for *simd* and macro guidance for *ca53* (see Table 2). The post-routing layouts are presented in Figure 4 and the corresponding metrics are shown in Table 3. For *simd*, the user-specified macro blockage (purple rectangle in Figure 4(d)) is created based on the macro placement given by chip experts (see Figure 4(c)). In this case, the commercial P&R tool cannot generate a valid macro placement and we use NaN to indicate this in Table 3; *TMP* is unable to comprehend any user-specified macro blockage, so we do not show a result for *TMP*. Similarly, we specify macro guidance for *ca53* based on the manual floorplan in Figure 4(f). In Figure 4(i), we provide "loose" macro guidance by specifying the bounding box of each macro cluster in the manual floorplan as the macro cluster's preferred region. And, in Figure 4(j), we provide "tight" macro guidance by specifying the center region (10 *um* × 10 *um*) of each macro cluster in the manual floorplan as the cluster's preferred region. Our experiment results suggest very strong stability and predictability of our macro placer.

Design	Flow	Std Cells	WL (m)	DRC	WNS (ns)	TNS (ns)	Power (mW)	TAT (min)
swerv_wrapper	*Comm*	101K	1.48	0	-0.010	-0.352	116.75	1.00
	Manual	101K	1.38	0	-0.020	-0.497	115.75	weeks
	TMP	102K	1.46	1	-0.013	-0.200	118.66	1.53
	HiDaP	111K	1.39	0	-0.009	-0.511	115.55	1.68
	RTL-MP	100K	1.37	0	-0.018	-0.319	115.99	59.18
ariane	*Comm*	134K	2.27	0	-0.065	-12.269	174.52	1.50
	Manual	132K	1.89	0	-0.012	-0.678	168.98	weeks
	TMP	132K	1.86	0	-0.006	-0.314	169.44	2.18
	RTL-MP	132K	1.83	0	-0.012	-0.108	169.11	49.68
simd	*Comm*	NaN	NaN	NaN	NaN	NaN	NaN	NaN
	Manual	256K	4.14	0	-0.026	-1.195	288.70	weeks
	RTL-MP$_B$	255K	3.99	0	-0.032	-3.003	299.26	87.07
coyote	*Comm*	325K	3.22	0	-0.025	-0.094	110.14	2.00
	Manual	322K	3.17	0	-0.017	-0.129	109.84	weeks
	TMP	323K	3.27	0	-0.023	-0.222	110.17	2.92
	RTL-MP	322K	3.16	0	-0.035	-0.090	109.77	38.83
bp_single	*Comm*	523K	5.53	136	-0.105	-111.976	424.86	3.00
	Manual	512K	5.05	35	-0.086	-3.148	414.83	weeks
	TMP	507K	5.61	0	-0.086	-2.905	423.04	5.87
	RTL-MP	507K	4.95	0	-0.116	-3.202	414.09	22.08
ca53	*Comm*	508K	7.95	1	-0.047	-3.177	651.30	4.50
	Manual	505K	7.66	0	-0.014	-1.894	636.55	weeks
	TMP	507K	7.73	0	-0.021	-2.842	640.01	8.80
	RTL-MP$_G$	504K	7.47	1	-0.017	-2.887	630.35	71.55
	RTL-MP	503K	7.41	0	-0.015	-2.639	627.26	58.00
	RTL-MP$_L$	505K	7.50	0	-0.018	-2.985	636.06	57.15
	RTL-MP$_T$	504K	7.50	0	-0.021	-1.645	631.80	55.28

Table 3: Summary of experiment results.

From Table 3, we can see that the turnaround time of *RTL-MP* is relatively long compared to other macro placers. Improving this is an obvious direction for future work. Profiling results for our macro placer on the *ca53* testcase are shown in Figure 5. Nearly 90% of the runtime is spent on solution evaluation ($p_{guidance}$ (29.2%), p_{bias} (29.0%) and WL calculation (28.9%)). This said, across all of our studies *RTL-MP* still performs at least 1921 solution evaluations per second. The autoclustering engine is also a significant consumer of runtime. The shape engine and pin alignment engine take only small portions of runtime. From Table 3, we also observe that the turnaround time of *RTL-MP* does not necessarily increase linearly with the size of input designs, indicating that *RTL-MP* has good scalability. This is because the autoclustering engine will adjust the size of clusters according to the input design, such that the number of clusters will not increase too much as the size of input designs increases.

8 CONCLUSION

We have proposed a novel macro placer called *RTL-MP*, which utilizes RTL information and tries to "mimic" the behavior of human experts in creating macro placement. By exploiting logical hierarchy and processing logical modules based on connection signatures,

[5] *min_num_inst* and *max_num_inst* are tunable hyperparameters. Here, we simply show how we calculate these based on our testcases.

[6] Based on our experiment results, 20 runs can already generate decent results. More runs (and larger walltime) can be applied if desired.

[7] A single run means running the corresponding macro placer once, without any parameter sweeping or autotuning.

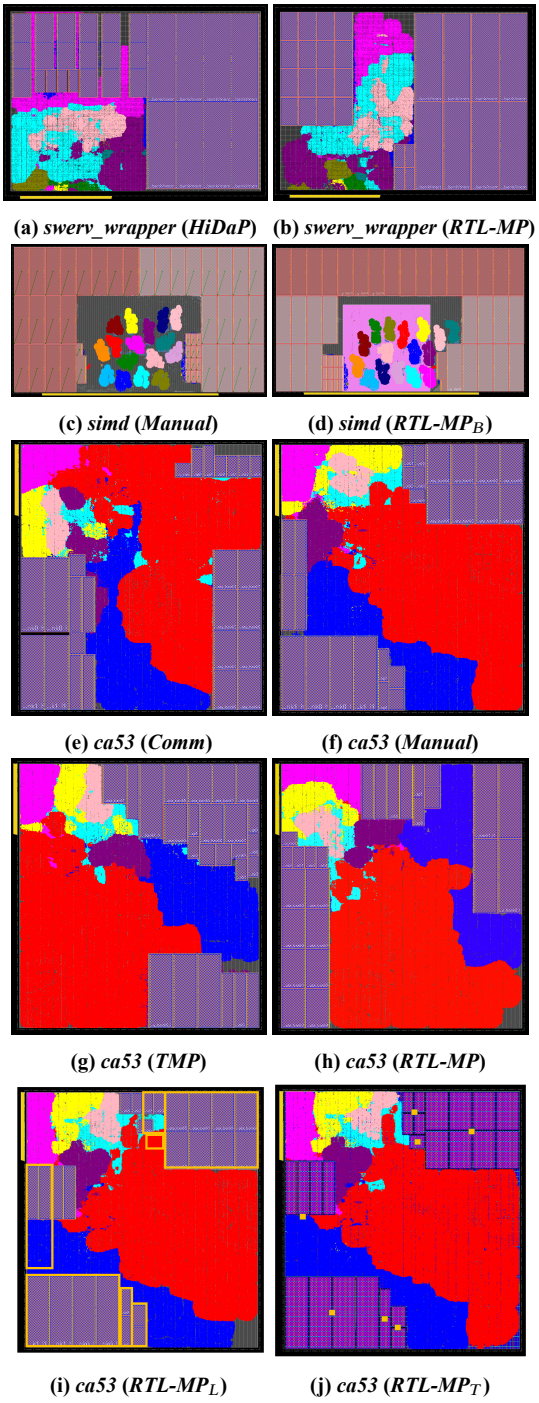

Figure 4: Post-route layouts of *swerv_wrapper*, *simd* and *ca53* designs with different flows under different user-specified constraints. Design/Flow: (a) *swerv_wrapper* / *HiDaP*; (b) *swerv_wrapper* / *RTL-MP*; (c) *simd* / *Manual*; (d) *simd* / *RTL-MP* with user-specified macro blockages (purple rectangle); (e) *ca53* / *Comm*; (f) *ca53* / *Manual*; (g) *ca53* / *TMP*; (h) *ca53* / *RTL-MP*; (i) *ca53* / *RTL-MP* with user-specified "loose" macro guidance (orange rectangles); (j) *ca53* / *RTL-MP* with user-specified "tight" macro guidance (orange rectangles).

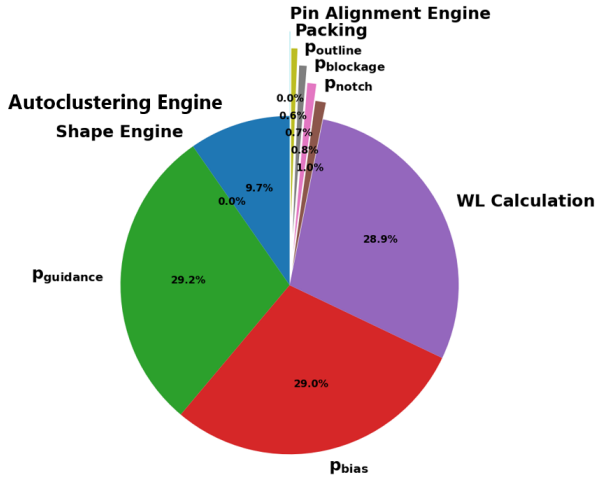

Figure 5: Runtime profiling of *RTL-MP* on the *ca53* testcase. Nearly 90% of the runtime is spent on solution evaluation ($p_{guidance}$, p_{bias} and WL calculation).

RTL-MP can fully capture the dataflow defined by RTL designers and use the dataflow information to guide macro placement. To further improve macro placement QoR, *RTL-MP* takes care of pin access, notch area avoidance and the common practice of pushing macros to peripheries. Furthermore, *RTL-MP* generates stable and predictable macro placements when given typical forms of user guidance and constraints. We also apply autotuning to optimize weights of objective function terms, on a per-design basis. Compared to a state-of-the-art commercial macro placer and floorplans generated by a human expert engineer, *RTL-MP* outperforms the commercial tool and achieves similar QoR as handcrafted floorplans (evaluated after standard-cell P&R in the commercial tool). This suggests that *RTL-MP* is already a very promising tool for quick prototyping.

A key area of future work is to extend the autoclustering engine to support hierarchical clusters. This is essential for very large blocks with hundreds or thousands of macros where a single level of macro clusters would lead to suboptimal floorplans, as noted in [8]. In the hierarchical clustering approach, we would generate a top-level parent cluster corresponding to a logical module in the hierarchy and also create child clusters underneath it. At the top level of the floorplan, the parent clusters are first placed and shaped, and for each parent cluster its child clusters – which could be a mix of standard-cell and macro clusters – would be placed inside the outline of the parent cluster. For this approach to produce high-quality floorplans, "global" route planning would be introduced after the top-level parent clusters are placed; this would determine pin access regions for the parent clusters that prevent the macro placement inside the parent clusters from blocking routes and causing congestion.

ACKNOWLEDGMENTS

We thank Dr. Alex Vidal-Obiols for sharing the *HiDaP* [34] executable and providing support on how to run *HiDaP*. We also thank Dr. Jiajia Li for helpful discussions. Research at UCSD is partially supported by DARPA IDEA HR0011-18-2-0032 and RTML FA8650-20-2-7009.

REFERENCES

[1] D. Aldous and U. Vazirani, "'Go with the winners' algorithms", *Proc. IEEE Symp. on FOCS*, 1994, pp. 492-501.

[2] S. N. Adya and I. L. Markov, "Fixed-outline floorplanning: enabling hierarchical design", *IEEE Trans. VLSI* 11(6) (2003), pp. 1120-1135.

[3] R. Bruck, K.-H. Temme and H. Wronn, "FLAIR-a knowledge-based approach to integrated circuit floorplanning", *Proc. Intl. Workshop on Artificial Intelligence for Industrial Applications*, 1988, pp. 194-199.

[4] Y.-C. Chang, Y.-W. Chang, G.-M. Wu and S.-W. Wu, "B*-trees: a new representation for non-slicing floorplans", *Proc. DAC*, 2000, pp. 458-463.

[5] W. Choi and K. Bazargan, "Hierarchical global floorplacement using simulated annealing and network flow area migration", *Proc. DATE*, 2003, pp. 1104-1105.

[6] J. Cong, M. Romesis and J. R. Shinnerl, "Fast floorplanning by look-ahead enabled recursive bipartitioning", *IEEE Trans. CAD* 25(9) (2006), pp. 1719-1732.

[7] T.-C. Chen and Y.-W. Chang, "Modern floorplanning based on B^*-tree and fast simulated annealing", *IEEE Trans. CAD* 25(4) (2006), pp. 637-650.

[8] T. Chen, Y. Chang and S. Lin, "A new multilevel framework for large-scale interconnect-driven floorplanning", *IEEE Trans. CAD* 27(2) (2008), pp. 286-294.

[9] G. Chen, W. Guo, H. Cheng, X. Fen and X. Fang, "VLSI floorplanning based on particle swarm optimization", *Proc. Intl. Conf on Intelligent System and Knowledge Engineering*, 2008, pp. 1020-1025.

[10] C.-C. Hu, D.-S. Chen and Y.-W. Wang, "Fast multilevel floorplanning for large scale modules", *Proc. ISCAS*, 2004, pp. 205-208.

[11] B. H. Gwee and M. H. Lim, "A GA with heuristic-based decoder for IC floorplanning", *Integration* 28(2) (1999), pp. 157-172.

[12] Z. He, Y. Ma, L. Zhang, P. Liao, N. Wong, B. Yu and M. D.-F. Wong, "Learn to floorplan through acquisition of effective local search heuristics", *Proc. ICCD*, 2020, pp. 324-331.

[13] J. Lu, H. Zhuang, P. Chen, H. Chang, C.-C. Chang, Y.-C. Wong et al., "ePlace-MS: electrostatics-based placement for mixed-size circuits", *IEEE Trans. CAD* 24(5) (2015), pp. 685-698.

[14] M.-C. Kim, N. Viswanathan, C. J. Alpert, I. L. Markov and S. Ramji, "MAPLE: multilevel adaptive placement for mixed-size designs", *Proc. ISPD*, 2012, pp. 193–200.

[15] H. Murata, K. Fujiyoshi, S. Nakatake and Y. Kajitani, "VLSI module placement based on rectangle-packing by the sequence-pair", *IEEE Trans. CAD* 15(12) (1996), pp. 1518-1524.

[16] K. -H. Temme and R. Bruck, "Chip-architecture planning based on expert knowledge", *Proc. Intl. Workshop on Artificial Intelligence for Industrial Applications*, 1998, pp. 188-193.

[17] J. Z. Yan and C. Chu, "DeFer: Deferred decision making enabled fixed-outline floorplanner", *Proc. DAC*, 2008, pp. 161-166.

[18] Y. Zhan, Y. Feng and S. S. Sapatnekar, "A fixed-die floorplanning algorithm using an analytical approach", *Proc. ASP-DAC*, 2006.

[19] A. B. Kahng, "Classical floorplanning harmful?", *Proc. ISPD*, 2000, pp. 207-213.

[20] D. H. Kim and S. K. Lim, "Bus-aware microarchitectural floorplanning", *Proc. ASP-DAC*, 2008, pp. 204-208.

[21] M. Ekpanyapong, J. R. Minz, T. Watewai, H. S. Lee and S. K. Lim, "Profile-guided microarchitectural floorplanning for deep submicron processor design", *IEEE Trans. CAD* 25(7) (2006), pp. 1289-1300.

[22] A. Mirhoseini, A. Goldie, M. Yazgan, J. Jiang, E. Songhori, S. Wang et al., "Chip placement with deep reinforcement learning", *arXiv 2004.10746*, 2020.

[23] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang et al., "A graph placement methodology for fast chip design", *Nature* 594 (2021), pp. 207-212. https://github.com/google-research/circuit_training

[24] V. Nookala, Y. Chen, D. J. Lilja and S. S. Sapatnekar, "Microarchitecture-aware floorplanning using a statistical design of experiments approach", *Proc. DAC*, 2005, pp. 579-584.

[25] A. B. Kahng and T. Spyrou, "The OpenROAD project: unleashing hardware innovation", *Proc. GOMACTech*, 2021.

[26] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "Optimization by simulated annealing", *Science* 220(4598) (1983), pp. 671-680.

[27] J. Lin, S. Li and Y. Wang, "Routability-driven mixed-size placement prototyping approach considering design hierarchy and indirect connectivity between macros", *Proc. DAC*, 2019, pp. 1-6.

[28] J.-M. Lin, Y.-L. Deng, Y.-C. Yang, J.-J. Chen and Y.-C. Chen, "A novel macro placement approach based on simulated evolution algorithm", *Proc. ICCAD*, 2019, pp. 1-7.

[29] J. Lin, Y. Deng, S. Li, B. Yu, L. Chang and T. Peng, "Regularity-aware routability-driven macro placement methodology for mixed-size circuits with obstacles", *IEEE Trans. VLSI* 27(1) (2019), pp. 57-68.

[30] J.-M. Lin, Y.-L. Deng, Y.-C. Yang, J.-J. Chen and P.-C. Lu, "Dataflow-aware macro placement based on simulated evolution algorithm for mixed-size designs", *IEEE Trans. VLSI* 29(5) (2021), pp. 973-984.

[31] J. K. Ousterhout, "Corner stitching: a data-structuring technique for vlsi layout tools", *IEEE Trans. CAD* 3(1) (1984), pp. 87-100.

[32] X. Tang and D. F. Wong, "FAST-SP: a fast algorithm for block placement based on sequence pair", *Proc. ASP-DAC*, 2001, pp. 521-526.

[33] T.-B. Chan, A. B. Kahng and M. Woo, "Revisiting inherent noise floors for interconnect prediction", *Proc. SLIP*, 2020, pp. 1-7.

[34] A. Vidal-Obiols, J. Cortadella, J. Petit, M. Galceran-Oms and F. Martorell, "RTL-aware dataflow-driven macro placement", *Proc. DATE*, 2019, pp. 186-191.

[35] A. Vidal-Obiols, J. Cortadella, J. Petit, M. Galceran-Oms and F. Martorell, "Multi-level dataflow-driven macro placement guided by RTL structure and analytical methods", *IEEE Trans. CAD* 40(12) (2020), pp. 2542-2555.

[36] J. Z. Yan, N. Viswanathan and C. Chu, "An effective floorplan-guided placement algorithm for large-scale mixed-size design", *ACM TODAES* 19(3) (2014), pp. 1-25.

[37] Tune. https://docs.ray.io/en/latest/tune/index.html.

[38] A. Caldwell, A. B. Kahng and I. Markov, "MLPart: high-performance min-cut bisector", 2006. https://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/Partitioning/MLPart/

[39] Y. Chuang, G. Nam, C. J. Alpert, Y. Chang, J. Roy and N. Viswanathan, "Design-hierarchy aware mixed-size placement for routability optimization", *Proc. IC-CAD*, 2010, pp. 663-668.

[40] T.-C. Chen, P.-H. Yuh, Y.-W. Chang, F.-J. Huang and T.-Y. Liu, "MP-trees: a packing-based macro placement algorithm for modern mixed-size designs", *IEEE Trans. CAD* 27(9) (2008), pp. 1621-1634.

[41] Y.-C. Liu, T.-C. Chen, Y.-W. Chang and S.-Y. Kuo, "MDP-trees: multi-domain macro placement for ultra large-scale mixed-size designs", *Proc. ASP-DAC*, 2019.

[42] Y. Chen, C. Huang, C. Chiou, Y. Chang and C. Wang, "Routability-driven blockage-aware macro placement", *Proc. DAC*, 2014, pp. 1-6.

[43] C.-H. Chiou, C.-H. Chang, S.-T. Chen and Y.-W. Chang, "Circular-contour-based obstacle-aware macro placement", *Proc. ASP-DAC*, 2016, pp. 172-177.

[44] C.-H. Chang, Y.-W. Chang and T.-C. Chen, "A novel damped-wave framework for macro placement", *Proc. ICCAD*, 2017, pp. 504-511.

[45] M. Fogaça, A. B. Kahng, E. Monteiro, R. Reis, L. Wang and M. Woo, "On the superiority of modularity-based clustering for determining placement-relevant clusters", *Integration* (74) (2020), pp. 32-44.

[46] M. Hsu, Y. Chen, C. Huang, T. Chen and Y. Chang, "Routability-driven placement for hierarchical mixed-size circuit designs", *Proc. DAC*, 2013, pp. 1-6.

[47] M.-K. Hsu, Y.-F. Chen, C.-C. Huang, S. Chou, T.-H. Lin, T.-C. Chen and Y.-W. Chang, "NTUplace4h: A novel routability-driven placement algorithm for hierarchical mixed-size circuit designs", *IEEE Trans. CAD* 33(12) (2014), pp. 1914-1927.

[48] Team VLSI, "Floorplan strategies for macro dominating blocks", 2014. https://www.teamvlsi.com/2021/02/floorplan-strategies-for-macro.html

[49] The OpenROAD Project. https:github.com/The-OpenROAD-Project/OpenROAD

[50] The SweRV CoreTM version 1.1 design RTL. https://github.com/westerndigitalcorporation/swerv_eh1

[51] The Coyote RISC-V Rocketcore. http://opencelerity.org/

[52] The CVA6 RISC-V CPU. https://github.com/openhwgroup/cva6

[53] The BlackParrot. https://github.com/black-parrot/black-parrot

[54] R. Liaw, E. Liang, R. Nishihara, P. Moritz, J. E. Gonzalez and I. Stoica, "Tune: A Research Platform for Distributed Model Selection and Training", *arXiv:1807.05118*, 2018. https://arxiv.org/abs/1807.05118