RTL-Aware Dataflow-Driven Macro Placement

Alex Vidal-Obiols[1], Jordi Cortadella[1], Jordi Petit[1], Marc Galceran-Oms[2], Ferran Martorell[2]
[1]Department of Computer Science, Universitat Politècnica de Catalunya, [2]eSilicon EMEA, Barcelona, Spain

***Abstract*—When RTL designers define the hierarchy of a system, they exploit their knowledge about the conceptual abstractions devised during the design and the functional interactions between the logical components. This valuable information is often lost during physical synthesis. This paper proposes a novel multi-level approach for the macro placement problem of complex designs dominated by macro blocks, typically memories. By taking advantage of the hierarchy tree, the netlist is divided into blocks containing macros and standard cells, and their dataflow affinity is inferred considering the latency and flow width of their interaction. The layout is represented using slicing structures and generated with a top-down algorithm capable of handling blocks with both hard and soft components, aimed at wirelength minimization. These techniques have been applied to a set of large industrial circuits and compared against both a commercial floorplanner and handcrafted floorplans by expert back-end engineers. The proposed approach outperforms the commercial tool and produces solutions with similar quality to the best handcrafted floorplans. Therefore, the generated floorplans provide an excellent starting point for the physical design iterations and contribute to reduce turn-around time significantly.**

I. Introduction

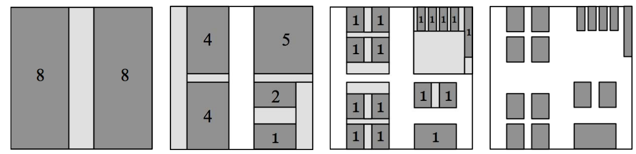
Current industrial EDA floorplanning solutions are fast but often do not produce good enough macro placements. Additional iterations by physical designers, which take significant effort, are needed. For circuits with more than 200 macros, it usually takes 2 to 4 weeks for the floorplan to reach the desired quality of results. As decisions taken at the floorplan stage have a critical impact on timing and power, considering structural properties during macro placement helps reduce design turn-around time.

Exploiting RTL-stage information, such as hierarchy and arrays, becomes an inestimable ally to physical design. Many algorithms devote high computational effort to infer structural properties that were known in previous stages of the design and have been destroyed or ignored. The hierarchical abstraction used by humans provides a logic vision of the relations between circuit components. Identifying array components (registers, ports...) provides a better view of the flow of data in the circuit, key to understanding the relations between macro components (and the standard cells) and providing floorplans that are more attuned to the structure of the system.

Previous work

Hierarchy exploitation has been used to do macro clustering in floorplanning [5] and other mixed-size placement flows [9] [10]. It has also been combined with multi-level optimization to reduce problem size in macro placement [8]. However, array information for dataflow analysis has not been sufficiently explored. Most mixed-size placers use the netlist to model connectivity, whereas some previous work on macro floorplan use pseudonets for macros that are hierarchically close [5].

*This work has been partially supported by a grant from eSilicon Corporation and funds from the Spanish Ministry for Economy and Competitiveness and the European Union (FEDER funds) under grant TIN2017-86727-C2-1-R and FPI 2015, and the Generalitat de Catalunya (2017 SGR 786).



(a) (b) (c) (d)

Fig. 1: Evolution of the multi-level block floorplan of a 16-macro design. Dark gray blocks contain macros, count inside. Light blocks contain only standard cells.

Whereas some authors propose simultaneous flows for macro and standard cell placement [11], our method advocates for performing macro placement prior to cell placement. One of the main challenges of such approaches is the modeling of standard cell logic. Some consider cell area implicitly by having macros close to circuit walls, e.g., initially in [5] and also in the recent works based on circular contours [3], [6], [7]. This is *de facto* the chosen approach for some industrial floorplanning tools. There are proposals to extend the model by considering macro and standard cell interaction [4]. Others have preferred to model cells and have two kinds of blocks, hard for macros with a fixed shape, and soft for standard cell groups with a flexible shape, e.g., using sequence-pairs [2] or slicing trees with shapes curves [14].

Our contribution

This paper introduces a multi-level floorplanner to find placement and orientation of macros by exploiting RTL-stage structure information. The hierarhical process is illustrated in Fig. 1 by defining the location of the 16 macros of a design. As shown in Fig. 1a, the first partition identifies three key components, two containing 8 macros each, and a third with cells connecting them. The process is repeated for the blocks at left and right, and each is again partitioned into blocks with macros and cells and blocks with only cells (Fig. 1b). After a final recursive call (Fig. 1c), coordinates for each one of the 16 macros are fixed and space has been left for their related standard cells (Fig. 1d).

After examining the circuit hierarchy, our tool creates a partition into blocks (modeling macros and standard cells) and analyses its array information to estimate dataflow affinity. The layout of blocks is decided using slicing structures and

simulated annealing, with a novel top-down layout generation algorithm designed to handle *hybrid* blocks with both hard and soft properties, not only hard as in [13].

The described algorithms have been implemented in our HiDaP tool (for Hierarchical Dataflow Placement). It has been tested on a suite of large industrial designs and compared against an state-of-the-art industrial floorplanner and hand-crafted macro placements. HiDaP improves on the industrial floorplan tool, and the obtained results are close to handcrafted designs with a fraction of the effort involved, providing a more promising start for the physical design iteration process. Our key contributions include:

1) Multi-level macro placement based on recursive block floorplanning with hierarchy-aware declustering.
2) Multiple graph-based data structures to model the circuit and estimate dataflow affinity between blocks, considering latency and array widths.
3) Block area modeling considering macros and standard cells, with a top-down area-budgeting strategy for hybrid shape blocks aimed at wirelength minimization.

II. Preliminaries

A. Problem Definition and Overview

The input of macro placement is a netlist N with hierarchy and array information derived from the RTL stage. The output is the coordinates and orientation of each macro targeting wirelength minimization. HiDaP uses hierarchical information to enable a multi-level approach, and array information to model the dataflow of the circuit. This dataflow is obtained by dividing the circuit in blocks, which consist of hybrid groups of macros and standard cells, and analyzing their connections.

B. Dataflow Affinity

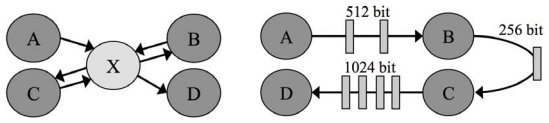
For every pair of blocks A and B, the *dataflow affinity* metric captures the "information closeness" between them. This concept is combined with physical distances to provide a layout quality metric. Dataflow affinity follows the relation

$$Dataflow\_Affinity(A,B) \propto \frac{Information\_Flow(A,B)}{Latency(A,B)}.$$

Blocks with a high information flow should be close in the floorplan, but a large latency reduces such need. Information flow is represented by array bitwidth, whereas latency is represented by the sequential element count in the shortest paths between blocks.

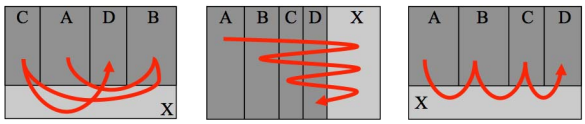
Two kinds of dataflow between blocks are proposed to better model current complex designs. *Block flow* considers the dataflow affinity between blocks and *macro flow* considers the dataflow affinity between macros inside blocks. Whereas the first models more accurately the physical connections present in the netlist, the second provides a more global insight of the data flow between blocks.

Consider a system with 4 blocks with macros (A to D) communicating through a standard cell block X. A block flow analysis of the netlist would reveal a connection pattern such as Fig. 2a, whereas inferring macro flow as in Fig. 2b would generate an alternative view on the circuit.



(a) (b)

Fig. 2: Block connection graphs for a small system.



(a) (b) (c)

Fig. 3: Possible layouts for the system in Fig. 2.

Three layouts for this system, with their communications marked in red, are shown in Fig. 3. With only block flow analysis, blocks A to D are close to X (as seen in Fig. 3a) without considering their relative positions. Fig. 3b shows a layout where only macro flow analysis is used. Its blocks are placed following the macro dataflow, but since X has no macros, it can end up anywhere in the circuit (Fig. 3b). Using a combination of both flows generates Fig. 3c.

C. Circuit Abstractions

Three graphs and a tree are used to model circuit connectivity (Table I). HT and G_{net} are derived from the input hierarchical netlist N, G_{seq} and G_{df} are derived from G_{net}.

In $HT = (V_{ht}, E_{ht})$, every node represents a level in the hierarchy, and edges represent subhierarchy relations. A hierarchy cut HC with regard to a node $n \in V_{ht}$ is defined as a set of nodes in its subtree such that each path between n and the leaves of its subtree crosses exactly one node in HC.

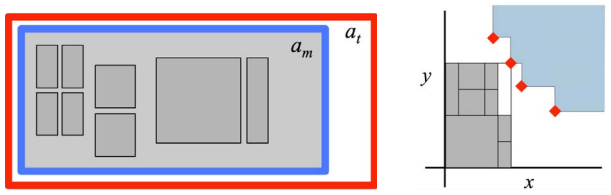
The graph $G_{net} = (V_{net}, E_{net})$ represents the original netlist, with $V_{net} = M \cup P \cup F \cup C$, respectively being macros, ports, sequential cells (flops) and combinational cells.

The sequential graph $G_{seq} = (V_{seq}, E_{seq})$ is a directed graph with weighted nodes. Its edges capture communication latency and array width between sequential components and ports. V_{seq} represents macros, multi-bit registers and ports.

The dataflow graph $G_{df} = (V_{df}, E_{df})$ is a directed graph with weighted nodes and edges which used to derive dataflow affinity. The node set is $V_{df} = B \cup P_{mb}$. Every node in B represents a block and is formed by a subset of nodes in V_{seq}. The edges in E_{df} hold information on the width and latency of the paths between these subsets of nodes in V_{seq}.

TABLE I: Data structures for different circuit abstractions.

Graph	Size	Type of vertices	Description
HT	–	Hierarchical blocks.	Circuit hierarchy representation.
G_{net}	$\sim 10^7$	Macros, ports, sequential and combinational cells.	Bit-level complete netlist connectivity.
G_{seq}	$\sim 10^5$	Macros, multi-bit ports and registers.	Multi-bit sequential connectivity.
G_{df}	$\sim 10^2$	Blocks and multi-bit ports.	Dataflow affinity among blocks and ports.



(a) Minimum and target areas. (b) Shape curve (Γ).

Fig. 4: Area model for a block.

D. Block Representation

Each block represents the cells and macros under a node of the hierarchy tree, and has properties of both kinds of components. It is characterized by the triple $\langle \Gamma, a_m, a_t \rangle$. Γ is a shape curve that contains a set of pairs (*width*, *height*) representing the smallest bounding boxes that can hold a placement of the macros in the block (standard cells are ignored for Γ). a_m (minimum area) is the sum of the area of all macros and standard cells under the hierarchy level. a_t is the *target area* for the block, i.e., a_m plus some extra area associated to the block (see Sect. IV-C).

Fig. 4 provides a visual reference of the block representation. Fig. 4a shows an abstraction of a block. The darker boxes represent macros. The blue rectangle (a_m) represents the minimum area of the block, whereas the red rectangle (a_t) represents the target area. Fig. 4b shows its shape curve, defined by a set of Pareto points (in red). The blue area represents all bounding boxes that could hold a placement of the eight macros of the block.

III. Algorithmic Overview

HiDaP follows a $\wedge$-style multi-level flow by considering the hierarchy as a clustering of the design and using a decluster-and-floorplan scheme. Algorithm 1 provides a high-level view of the top flow of the tool. The input is the hierarchical netlist N and the dimensions of the circuit. The first step is the generation of the hierarchical tree from the netlist. *shape_curve_generation* constructs the set S_Γ, used to ensure that a block layout fixed at a given hierarchical level can accommodate its macros under slicing constraints. It contains, for each $n_h \in V_{ht}$, a shape curve with the minimal sizes such that its macros can be placed. The next steps are finding macro position by calling *recursive_block_floorplan* and the orientation with *memory_flipping*, where a flipping post-process for wirelength reduction using macro side dataflow.

Algorithm 1 Top tool flow

```
1: Input: hierarchical netlist (N), width (w), height (h)
2: Output: location and orientation of the macro cells
3: HT ← obtain_hierarchy_tree(N)
4: S_Γ ← shape_curves_generation(HT)                    {Sect. IV-A}
5: macro_loc ← recursive_block_floorplan(N, w, h, root(HT), S_Γ)
6: macro_orient ← macro_flipping(N, macro_loc)
```

The recursive floorplanning function (Algorithm 2) takes as an input the netlist N, a hierarchical node $n_h \in V_{ht}$, whose subtree must be placed in a given space (w, h), and the shape curves for all hierarchical levels in S_Γ. The aim is to place

Algorithm 2 Recursive block floorplan

```
1: Input: hier. netlist (N), width (w), height (h), n_h ∈ V_ht, S_Γ
2: Output: location of the macros below n_h
3: B_{Γ,a_m} ← hierarchical_declustering(n_h, S_Γ)              {Sect. IV-B}
4: B_{Γ,a_m,a_t} ← target_area_assignment(B_{Γ,a_m}, N)         {Sect. IV-C}
5: M_aff ← dataflow_inference(B_{Γ,a_m,a_t}, N)                 {Sect. IV-D}
6: coords ← layout_generation(w, h, B_{Γ,a_m,a_t}, M_aff)       {Sect. IV-E}
7: for all b ∈ B_{Γ,a_m,a_t} do
8:     if macro_count(b) > 1 then
9:         w_b, h_b ← block_size(coords, b)
10:        recursive_block_floorplan(N, w_b, h_b, b, S_Γ)       {Recursion}
11:    else fix_position(b, w_b, h_b)
```

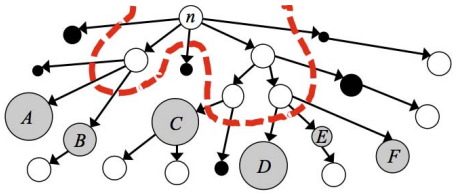


Fig. 5: Hierarchical declustering to find HC_B and HC_G.

the macros under n_h considering dataflow between themselves, standard cell logic under n_h, macros outside n_h and ports.

The first step is *hierarchical_declustering*, which identifies the blocks to be considered for floorplanning and partially characterizes them in the set B_{Γ,a_m}, with the minimum area a_m and shape curve Γ of each block. In *target_area_assignment*, the target area a_t of the blocks is stored in B_{Γ,a_m,a_t}, a set containing the full characterization $\langle \Gamma, a_m, a_t \rangle$ of each block.

In *dataflow_inference*, the dataflow affinity between each pair of blocks is derived and stored in the affinity matrix M_{aff}. During *layout_generation*, slicing structures are used to model the floorplan and, given the affinity metric and the mean distance between blocks in the layout, an optimization process is used to minimize their product to obtain the best possible layout. After the coordinates of each block are fixed in *coords*, the process is recursively called if the block has more than a single macro. Otherwise, its macro position is fixed in the corner of the available area that minimizes wirelength.

IV. Algorithmic Details

A. Shape Curves Generation

The input of this step is the hierarchical tree HT, and the output is a set of shapes curves S_Γ where each Γ is associated to a node in V_{ht} describing possible sizes for the floorplan of the macros under its subtree. These curves are used during layout generation to make sure that proposed sizes for blocks have at least one slicing macro layout that fits into the alloted area. The subtree shape curves are calculated in a bottom-up fashion up to the root of the hierarchical tree, only once at the beginning of the flow. At the leaf nodes of HT, the associated Γ contains the possible shapes of its macro. At each intermediate node of the hierarchy, it is not possible to compose the shapes of their children (the tree is not slicing). Area-optimization floorplanning using simulated annealing generates a set of shape combinations with small area which are valid for the node.

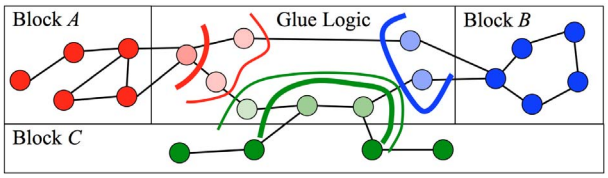
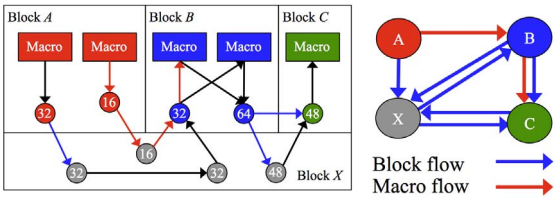


Fig. 6: Assigning HC_G area to HC_B blocks.



(a) G_{seq} graph (b) G_{df} graph

Fig. 7: Dataflow inference example.

B. Hierarchical declustering

The purpose of hierarchical declustering is to find the set of blocks to be considered during the floorplan of a given hierarchy level n_h, and to characterize their shape curve Γ and minimum area a_m. The idea is represented in Fig. 5, which shows the hierarchy tree under a node n. A possible hierarchical cut HC with respect to that node (defined in Sec. II-C), marked by a red line, generates block candidates for floorplanning. The nodes are divided in two sets, depending on the number of macros and area in their subtrees, and those with big area or with macros are selected (grey).

Formally, consider functions *area(n)* and *macro_count(n)*, returning the sum of the area and macro count of the subtree rooted at $n \in V_{ht}$. Given a parameter defining a minimum area, the nodes in any hierarchy tree cut HC can be divided in two sets, HC_B and HC_G:

$$HC_B = \{n \in HC | area(n) > min\_area \vee macro\_count(n) > 0\}$$
$$HC_G = HC \setminus HC_B \qquad \text{(small nodes with glue logic)}$$

Nodes in HC_B represent hierarchy levels with relatively big area or containing macros, whereas all others (HC_G) are small nodes with glue logic. Each node in HC_B is modeled as a block during layout generation. Area of nodes in HC_G is integrated with nodes in HC_B during later steps, as it represents small unstructured components.

Algorithm 3 Hierarchical declustering

```
1:  Input: n_h ∈ V_ht, open_area, min_area ∈ ℝ
2:  Output: HC_B, HC_G
3:  HC_B ← ∅; HC_G ← ∅
4:  queue.insert(n_h)
5:  while not queue.empty() do
6:      m ← queue.pop_front()
7:      if area(m) > open_area and macro_count(m) = 0 then
8:          for each child c ∈ m.children() do
9:              queue.insert(c)
10:     else if area(m) > min_area or macro_count(m) > 0 then
11:         HC_B ← HC_B ∪ {m}
12:     else
13:         HC_G ← HC_G ∪ {m}
```

The strategy used to find sets HC_B and HC_G for a given node n is shown in Algorithm 3. The parameter *open_area* is an area amount that controls how deep the tree will be explored. Both *min_area* and *open_area* are a fraction of $area(n_h)$, 40% and 1% in the experiments.

C. Target Area Assignment

Since blocks in HC_G are not directly considered for floorplanning, the goal of target area assignment is to incorporate their area to the target area a_t of blocks in HC_B. In G_{net}, a multi-source breadth-first search [12] finds the shortest paths from all nodes in HC_B to any cell in HC_G. Fig. 6 shows three blocks in HC_B with fully painted nodes, connected by glue logic components (elements in HC_G blocks) which are incorporated to their closest blocks as they are reached during the search. After this process, the sum of the area of HC_B blocks represents the whole area of the floorplanning instance and the triplet $\langle \Gamma, a_m, a_t \rangle$ for each block is characterized.

D. Dataflow Inference

Dataflow inference generates an affinity matrix M_{aff} estimating the affinity between each pair of blocks, ports and other macros in the circuit. M_{aff} is derived from the adjacency matrix of the dataflow graph G_{df}, which is obtained from G_{seq} and G_{net}. First G_{net} is transformed into G_{seq} in the following steps:

1) C nodes representing combinational cells are removed by connecting their predecessors to their successors
2) Nodes in P and F are clustered using component names to find array structures (`name[n]`, `name_n`).
3) Edges between sequential components are inferred by analyzing their transitive fanin/fanout in G_{net} and discovering their paths.
4) To reduce graph size but keep relatively big components, elements with fewer bits than a threshold are discarded.

The next step is constructing the dataflow graph G_{df} from G_{seq}. Each node in V_{df} represents a block, which is associated to a set of nodes from V_{seq}. Each edge summarizes two auxiliar edges, one which represents block flow E^b_{df} and one which represents macro flow E^m_{df}. A graph G_{seq} is shown in Fig 7a, where round nodes represent multi-bit registers with their bitwidth. Painted edges represent some paths that generate edges in G_{df}, shown in Fig 7b: blue represents block flow E^b_{df} edges, and red represents macro flow E^m_{df} edges.

Connectivity information of edge $e_{i,j}$ in E^b_{df} or E^m_{df} takes the form of a histogram, where bins represent latency and their height represents number of bits. In the case of block flow edges E^b_{df}, a breadth-first search at G_{seq} starts simultaneously from all components of block i traversing only outgoing edges through glue logic. When a component of block j is reached, the bitwidth of its predecessor in the path is added to the bin corresponding to the path length (blue paths in Fig 7a). To obtain macro flow edges in E^m_{df}, a similar process finds paths between macros in blocks, allowing the search to cross all nodes in V_{seq} (except macros) instead of only glue logic (red paths in Fig 7a).

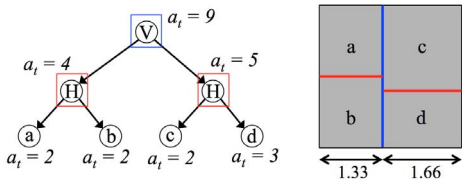


Fig. 8: Recursive layout generation.

In order to condense the dataflow information of an histogram, the weight of an edge is computed using $score(h, k)$ for an integer k according to the following formula, where i indexes bins in the histogram:

$$\sum_i \#bits_i \ / \ latency_i^k$$

The formula is based in the relation presented in Sec. II-B. Parameter k controls the exponential decay impact of latency.

Given the edge sets E_{df}^b and E_{df}^m and scoring their histograms, the score of dataflow affinity edges in E_{df} is obtained using a parametric formula defined as $\lambda \times score(e_{i,j}^b) + (1-\lambda) \times score(e_{i,j}^m)$. The value of λ balances block and macro flow and allows the generation of layouts with different emphasis on macro connectivity.

E. Layout Generation

The goal of layout generation is to find the coordinates for a set of blocks in a given area. Aiming to optimize wirelength, the function to minimize is:

$$penalty \cdot \left(\sum_{n_i,n_j}^{V_{df}} distance(n_i,n_j) \cdot M[i][j] \right)$$

Minimizing the sum of the products of dataflow affinity and distance ensures that blocks that have large array connections and relatively small latency are close in the layout, thus reducing wirelength. The position of ports and macros outside the subtree are considered a fixed point. The penalty multiplier forbids layouts with macro overlaps and adds solution flexibility by slightly punishing violations in the a_m and a_t of blocks, allowing them if the benefit in the cost function outweighs the cost of the penalty. It also allows the exploration to pass through illegal solutions.

Layout Representation and Search

The layout is represented as a slicing tree with a node per block. This structure allows to work with shape curve compositions and naturally use a top-down algorithm for layout generation. The solution space is explored using simulated annealing. The structure is perturbed with equal probability with one of three operations: operand swap, operator inversion or operand-operator swap (similar to [13]).

Since the shapes of the components are not fixed *a priori* as in other bottom-up layout generation approaches, the layout dimensions are not considered a constraint, but a budget: the layout always takes exactly the area it has been assigned. At the beginning, the entire area is available and the root of the slicing tree must be processed. At each node, the area is partitioned vertically or horizontally (depending on the node operator), according to the target area a_t sum of its subtrees. The process continues until the leaves are reached, when the block is assigned a rectangle in the layout. Fig. 8 shows a slicing tree where each leaf has target area a_t, and its layout considering a budget of 3x3 area units.

This technique guarantees all block a_t demands are met, but since it allows blocks to take any possible aspect ratio, some layouts are illegal when considering macros. It would be the case if node a contained a macro with $w = 2$ and $h = 1$, since such macro would not fit in the alloted space in the example. Let Γ^n, a_m^n and a_t^n for a given slicing tree node n characterize its subtree as a block, computed at the beginning of the layout generation from the blocks at the leaves by composing shape curves and adding areas up to the root. This characterization is used to solve illegalities by moving area from one child to the other and increase the cost function penalty depending on the kind of area that was yielded (a_t, a_m or macro area, from least to most severe).

TABLE II: Average WL, WNS and effort for the three flows.

	WL	WNS	Effort
IndEDA	1.143	-39.1%	10-30 mins (CPU)
HiDaP	1.013	-24.6%	0.5-2 hours (CPU)
handFP	1.000	-17.9%	2-4 weeks (engineers + CPU)

V. Experimental Evaluation

Array information for dataflow analysis is essential for the quality of results in our method. Unfortunately, this information is not available in open benchmarks available to academia such as the ICCAD'12 benchmarks [1]. For this reason, a set of 8 real industrial examples of challenging circuits were used to validate our approach. The final handcrafted layouts obtained by expert backend engineers were available for comparison. The following three floorplan flows are compared:

Industrial EDA (IndEDA) Floorplan obtained with a state-of-the-art industrial tool using high effort settings.

Hierarchical Dataflow Placement (HiDaP) Floorplan by HiDaP, best WL of three ($\lambda = 0.2, 0.5, 0.8$).

Handcrafted floorplan (handFP) Floorplan manually obtained by expert backend engineers at the company.

Metrics are taken after placement of standard cells using the same tool as **IndEDA**. Wirelength averages are shown using the geometric mean to reduce sensitivity to extreme values.

Table II shows summarized flow results, with average WL relative to **handFP**, WNS in clock cycle percentage and the effort to create the solutions. While **IndEDA** execution takes from 10 to 30 minutes, our approach **HiDaP** takes from 30 minutes to 2 hours and obtains results approaching those of the **handFP** flow, where floorplans have taken weeks of iterations by the physical design engineers.

Table III shows the detailed metrics. Rows represent a circuit and floorplan flow, and columns give information on wirelength (in meters, and normalized with respect to **handFP**), congestion (global routing overflow percentage) and timing information (worst negative slack, in percentage of the clock period, and total negative slack).

TABLE III: Metrics after placement using the three flows.

		Wirelength		Cong.	Timing	
	Flow	WL	norm.	GRC%	WNS%	TNS
c1	**IndEDA**	13.19	1.029	6.51	0.0	0
520k cells	**HiDaP**	13.40	1.046	7.83	0.3	0
32 macros	**handFP**	12.81	1.000	7.36	-0.2	0
c2	**IndEDA**	46.01	1.180	12.99	-44.5	-931
3.95M cells	**HiDaP**	40.72	1.045	13.00	-19.0	-329
100 macros	**handFP**	38.97	1.000	9.33	-11.2	-213
c3	**IndEDA**	44.83	1.175	10.09	-75.5	-553
3.78M cells	**HiDaP**	35.02	0.918	8.29	-17.5	-260
94 macros	**handFP**	38.16	1.000	9.15	-17.8	-317
c4	**IndEDA**	45.03	1.174	7.24	-54.4	-2167
4.81M cells	**HiDaP**	40.43	1.054	4.94	-31.2	-2686
122 macros	**handFP**	38.35	1.000	3.33	-22.8	-1736
c5	**IndEDA**	44.25	1.162	2.02	-30.8	-1940
1.39M cells	**HiDaP**	39.51	1.038	4.72	-25.1	-1149
133 macros	**handFP**	38.06	1.000	3.42	-39.8	-1017
c6	**IndEDA**	96.42	1.288	9.95	-70.0	-15341
2.87M cells	**HiDaP**	79.20	1.058	2.22	-37.0	-5051
90 macros	**handFP**	74.87	1.000	1.63	-27.3	-3688
c7	**IndEDA**	41.44	1.174	38.56	-34.9	-1060
1.67M cells	**HiDaP**	35.52	1.007	6.47	-29.9	-1059
108 macros	**handFP**	35.29	1.000	4.61	-20.4	-774
c8	**IndEDA**	24.85	0.987	1.02	-3.4	-44
2.20M cells	**HiDaP**	23.75	0.944	1.37	0.0	0
37 macros	**handFP**	25.17	1.000	0.93	-3.9	-24

When comparing the **HiDaP** and **IndEDA** flows, wirelength is smaller using HiDaP in all but one case. Congestion is similar, with two cases with noticeably less (*c6, c7*). Our flow always obtains lower WNS, and also lower TNS in most cases.

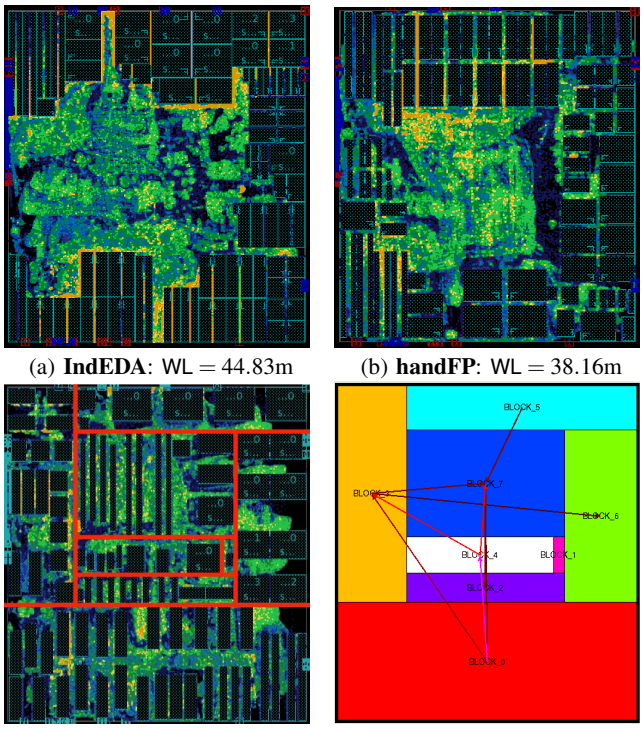
The mean wirelength in **IndEDA** is 14.3% higher than **handFP** while obtaining generally worse congestion and timing. When comparing **HiDaP** and **handFP**, the mean WL increase is only 1.3%, and the manual flow obtains generally better results in all metrics with 2 exceptions. In *c3*, our flow obtains 8.2% less wirelength with reduced congestion and timing, and in *c8*, HiDaP beats the other flows in wirelength and closes timing while maintaining similar congestion.

Density maps for the placed circuit *c3* using the three flows can be seen in Fig. 9. Whereas **IndEDA** and **handFP** placed macros on the block walls, **HiDaP** found more distributed locations given it considers standard cell distribution during macro placement. Our approach shows the smallest peak cell density near the macros in circuit walls.

Additionally, an interactive graphic tool has been developed to model and visualize the dataflow of complex designs. Fig. 9d shows a block floorplan of the topmost hierarchical level for circuit *c3* (corresponding to the stage of Fig. 1a in our initial example). Each colored box corresponds to a node in G_{df}, and the arrows are the edges representing their main dataflow relations: brighter colors mean more affinity. These diagrams generate additional feedback for the backend engineers to rapidly converge into better solutions.

VI. Conclusions

This paper proposes to exploit RTL information for macro floorplanning using a hierarchical multi-level flow. The effectiveness of the approach has been tested with an industrial suite of benchmarks. The results are generally better than using previously existing methods, with quality close to handcrafted floorplans, providing a better starting point for engineers to find a final layout in reduced turnaround time.



(a) **IndEDA**: WL = 44.83m (b) **handFP**: WL = 38.16m

(c) **HiDaP**: WL = 35.02m (d) Initial floorplan of G_{df}

Fig. 9: Density maps of standard cells for the placement of *c3* using the three flows, and top block floorplan from **HiDaP**.

References

[1] ICCAD 2012 Hierarchy-Aware Placement Contest.

[2] S. N. Adya and I. L. Markov. Combinatorial Techniques for Mixed-size Placement. *ACM Trans. Des. Auto. Electr. Syst.*, 10(1):58–90, 2005.

[3] C. H. Chang, Y. W. Chang, and T. C. Chen. A novel damped-wave framework for macro placement. In *ICCAD*, pages 504–511, 2017.

[4] S. T. Chen, Y. W. Chang, and T. C. Chen. An integrated-spreading-based macro-refining algorithm for large-scale mixed-size circuit designs. In *ICCAD*, pages 496–503, 2017.

[5] T. C. Chen et al. MP-Trees: A Packing-Based Macro Placement Algorithm for Modern Mixed-Size Designs. *IEEE Trans. on CAD*, 27(9):1621–1634, 2008.

[6] Y. F. Chen et al. Routability-driven blockage-aware macro placement. In *51st ACM/EDAC/IEEE DAC*, pages 1–6, 2014.

[7] C. H. Chiou et al. Circular-contour-based obstacle-aware macro placement. In *21st ASP-DAC*, pages 172–177, 2016.

[8] W. Choi and K. Bazargan. Hierarchical global floorplacement using simulated annealing and network flow area migration. In *DATE*, pages 1104–1105. IEEE, 2003.

[9] Y. L. Chuang et al. Design-hierarchy aware mixed-size placement for routability optimization. In *ICCAD*, pages 663–668, 2010.

[10] M. K. Hsu et al. Routability-driven placement for hierarchical mixed-size circuit designs. In *DAC*, pages 1–6, 2013.

[11] I. L. Markov, J. Hu, and M.-C. Kim. Progress and Challenges in VLSI Placement Research. *Procs. of the IEEE*, 103(11):1985–2003, 2015.

[12] M. Then et al. The more the merrier: efficient multi-source graph traversal. *Procs. of the VLDB Endowment*, 8(4):449–460, 2014.

[13] D. F. Wong and C. L. Liu. A New Algorithm for Floorplan Design. In *DAC*, pages 101–107, 1986.

[14] J. Z. Yan, N. Viswanathan, and C. Chu. Handling complexities in modern large-scale mixed-size placement. In *DAC*, pages 436–441, 2009.