SuperFlow: A Fully-Customized RTL-to-GDS Design Automation Flow for Adiabatic Quantum-Flux-Parametron Superconducting Circuits

Yanyue Xie[1]*, Peiyan Dong[1]*, Geng Yuan[2], Zhengang Li[1], Masoud Zabihi[3], Chao Wu[1], Sung-En Chang[1], Xufeng Zhang[1], Xue Lin[1], Caiwen Ding[4], Nobuyuki Yoshikawa[5], Olivia Chen[6] and Yanzhi Wang[1]
[1]*Northeastern University,* [2]*University of Georgia,* [3]*IBM Research,*
[4]*University of Connecticut,* [5]*Yokohama National University,* [6]*Tokyo City University*
[1]{xie.yany, dong.pe, yanz.wang}@northeastern.edu, [6]olivia.chen@ieee.org

***Abstract*—Superconducting circuits, like Adiabatic Quantum-Flux-Parametron (AQFP), offer exceptional energy efficiency but face challenges in physical design due to sophisticated spacing and timing constraints. Current design tools often neglect the importance of constraint adherence throughout the entire design flow. In this paper, we propose SuperFlow, a fully-customized RTL-to-GDS design flow tailored for AQFP devices. SuperFlow leverages a synthesis tool based on CMOS technology to transform any input RTL netlist to an AQFP-based netlist. Subsequently, we devise a novel place-and-route procedure that simultaneously considers wirelength, timing, and routability for AQFP circuits. The process culminates in the generation of the AQFP circuit layout, followed by a Design Rule Check (DRC) to identify and rectify any layout violations. Our experimental results demonstrate that SuperFlow achieves 12.8% wirelength improvement on average and 12.1% better timing quality compared with previous state-of-the-art placers for AQFP circuits.**

I. Introduction

Superconducting logic circuits exhibit extremely high energy efficiency over their Complementary Metal–Oxide–Semiconductor (CMOS) counterparts [1]. Adiabatic Quantum-Flux-Parametron (AQFP) logic [2] is a representative energy-efficient superconducting logic that is designed to achieve a reduction in both static and dynamic power consumption by adopting adiabatic switching [3]. AQFP can potentially achieve $10^4 - 10^5$ energy efficiency gain compared with state-of-the-art CMOS technology with a clock frequency of several GHz [4]. AQFP differs from CMOS circuits in terms of active components, passive components, logic gates, data propagation, clocking scheme, fan-out requirements, and power consumption, as detailed in Table I. Therefore, the design automation tools developed for CMOS cannot be directly applied to the design of AQFP circuits.

There have been several existing works on customized design automation tools for AQFP circuits. [5]–[7] solve the buffer and splitter insertion problem during the logic synthesis stage for path balancing and fan-out branching imposed by AQFP technology. [8] and [9] consider spacing constraints and max-wirelength constraints of AQFP during the placement stage. [10] further improves the timing of AQFP designs. However, all of these works focus solely either on the logic synthesis stage or the placement stage, without addressing the entire design flow, i.e., a complete RTL-to-GDS (Register-Transfer Level to Graphic Design System) flow. While some studies offer complete design flows, they only focus on specific steps like logic synthesis or AQFP cell library design [11] [12], both leaving placement and routing stage to commercial tools and lacking flexibility.

*Equal contributions.

TABLE I
Comparison of AQFP with CMOS.

Circuits	AQFP	CMOS
Active component	Josephson junction (JJ)	Transistor
Passive component	Inductor	Capacitor
Logic gate	Majority-based gates	And, or, inverter gates
Data propagation	Current pulse	Voltage level
Clocking	Four-phase clocking	Synchronous
Fan-out	$= 1$(Splitter for fan-outs)	≥ 1
Power	Alternating Current (AC)	Direct Current (DC)

Transitioning from CMOS to the more energy-efficient AQFP circuits necessitates a comprehensive and dedicated AQFP design automation tool flow to optimize Power, Performance, and Area (PPA). The absence of such a comprehensive tool flow restricts design optimization, potentially leading to issues like congestion and post-routing timing violations. Standard electronic design automation (EDA) tools for CMOS logic are not applicable to AQFP logic, due to inherent differences such as clocking constraints and fan-out requirements. Relying on commercial EDA tools for certain or all stages would result in limited flexibility, particularly considering that AQFP is an emerging technology and the AQFP cell library is under active development [12]. In light of these challenges, we develop a fully-customized RTL-to-GDS design automation flow tailored for AQFP circuits, enabling designers to easily adjust the design objectives for AQFP and incorporate timely updates to the AQFP cell library.

Our major contributions can be summarized as follows:

- We present a fully-customized RTL-to-GDS design flow, SuperFlow, for AQFP circuits. To the best of our knowledge, this is the first non-commercial RTL-to-GDS design

979-8-3503-4859-0/DATE24/© 2024 EDAA

automation tool that targets AQFP devices.

- We optimize the placement quality by optimizing wirelength and timing simultaneously while respecting the clocking and the mixed-cell-size constraints at the detailed placement stage.
- We introduce a layer-wise routing strategy, capable of routing with space expansion, thereby addressing potential routability issues.
- Experimental results show that SuperFlow achieves 12.8% wirelength improvement on average with 12.1% better timing quality over previous state-of-the-art placers for AQFP circuits.

II. Background

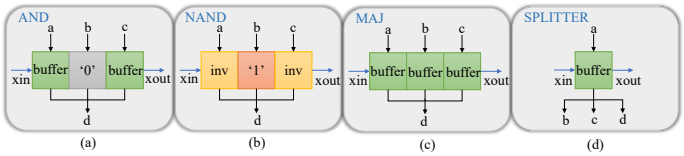

Fig. 1. **The symbols of AQFP logic gates.** (a) `and` gate; (b) `nand` gate; (c) `majority` gate; (d) `splitter` gate.

A. AQFP Superconducting Logic

The fundamental building block of AQFP logic gates is the AQFP `buffer`, which consists of a double-Josephson junction (JJ) superconducting quantum interference device (SQUID) [1]. Together with the inverter gate and constant gate, these basic gates constitute the AQFP cell library, including `and`, `or`, `not`, `majority`, `buffer`, and `splitter`, as shown in Fig. 1. The AQFP standard cell library is built via the minimalist design approach, utilizing a bottom-up method to construct more complex gates [6]. The design of the AQFP cell library follows the AIST standard process 2 (STP2) and the MIT Lincoln Laboratory (MIT-LL) SQF5ee fabrication process, both of which are niobium-based integrated-circuit technologies suitable for AQFP.

Different from traditional CMOS logic that utilizes and-or-inverter-based (AOI) representation, AQFP logic favors majority-based (MAJ) gates, due to the efficient utilization of JJ resources. Three-input majority gates in AQFP consume the same amount of JJ resources as two-input AOI gates [6]. Please note that, unlike CMOS gates which can connect to fan-out directly, all AQFP gates require the use of `splitter` cells for multiple fan-outs.

B. AQFP Clocking Architecture

The clocking architecture and timing requirements of AQFP differ from CMOS. In CMOS circuits, multi-level gates are expected to meet timing constraints as a group, i.e., multi-level gates for pipelining, while in AQFP circuits, each gate must satisfy the timing requirements, i.e., the gate-level pipelining. Specifically, four-phase AC bias currents serve as both power supply and clock signal [1], which triggers the data from one clock phase to the next, as illustrated in Fig. 2. AQFP utilizes this feature to mitigate the power consumption overhead of DC bias shown in other superconducting logic technologies, such as RSFQ [13]. While AC biasing contributes to exceptional energy efficiency, AQFP circuits also adopt a deep-pipelined architecture since each AQFP logic gate is connected to an AC clock signal and occupies one clock phase. This deep-pipelined architecture mandates all inputs for a logic gate to have the same delay (clock phases) from the primary inputs [6], necessitating rigorous path balancing. Furthermore, when it comes to a complete design flow, AQFP's gate-level pipelining provides more flexibility, allowing simultaneous optimization of spacing and timing constraints.

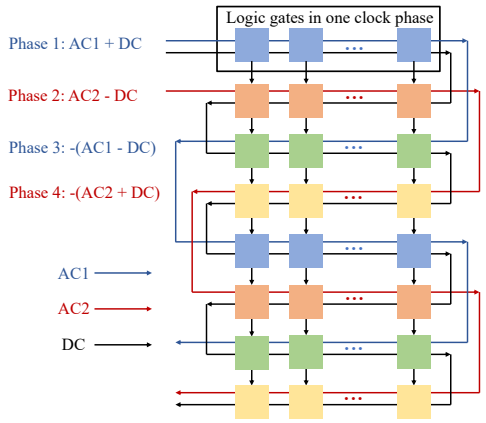

Fig. 2. **AQFP clocking architecture.** AQFP clocking scheme utilizes one DC and two AC signals to create a four-phase clocking scheme. The two AC signals have a phase difference of 90 degrees. The intricate zigzag clocking signals impose strict timing constraints on AQFP logic cells and formulate a deep-pipelined architecture.

C. Challenges of AQFP Physical Design

AQFP physical design is complicated and challenging due to various design constraints: (i) The AQFP placement problem involves fulfilling multiple spacing limitations, including cell spacing and zigzag spacing constraints [8]. Cell spacing requires that neighboring cells in a row either touch or maintain a minimum distance. Regarding zigzag spacing, when vias are utilized to change the wire directions, the wire zigzags must adhere to a predetermined minimum spacing (e.g., 10μm for the MIT-LL process). (ii) AQFP circuits should comply with maximum wirelength requirements, which specify that a single wire connection should not exceed W_{max} [9]. If a single connection exceeds the maximum wirelength restriction, it is necessary to insert an entire row of buffers between the two rows. (iii) Due to the different sizes of AQFP buffers relative to other majority-based cells, and the possibility of having various types of cells within a single clock phase, AQFP placement presents a mixed-cell-size problem. Within a single densely populated clock phase with multiple mixed-cell-size cells, perturbations or cell swapping can potentially cause significant intra-row violations. (iv) Strict timing constraints should be maintained since AQFP circuits run at a clock frequency of several GHz. The zigzag clocking scheme and the deep-pipelined architecture impose great challenges to AQFP placement algorithms as the cell positions in a fixed row have a significant impact on the timing closure. As a result, traditional placers [8], [9] that address wirelength alone may lead to placement results with

severe timing violations. (v) AQFP routing is limited to two metal layers between two adjacent clock phases, conforming to the zigzag clocking architecture and process requirements.

Hence, relying solely on separate synthesizers, placers, or routers for AQFP circuits is insufficient. A comprehensive design flow that encompasses and adheres to the various design constraints of AQFP circuits is imperative. For instance, following logic synthesis and the insertion of buffers/splitters for path balancing, each logic cell is assigned a specific clock phase or row index. The placement stage should preserve the assigned row index for each logic cell while optimizing their horizontal positions within the rows. Subsequently, during the routing stage, clock wires must be routed based on the placement results, ensuring no violations occur. Furthermore, both placement and routing must consider clocking constraints to achieve optimal timing and meet the desired clock frequency.

III. Methods

A. Overall AQFP Design Flow

Fig. 3 demonstrates the overall design flow of SuperFlow for AQFP circuits. The input files are the standard cell library of AQFP and the Register-Transfer Level (RTL) files describing circuit architecture. After logic synthesis and an AQFP interpreter, SuperFlow maps the netlist to an AQFP-based netlist, which serves as the input to the following place-and-route step. Then SuperFlow generates the circuit layout based on the physical information provided after the place-and-route step. Should any violations emerge after the Design Rule Check (DRC), SuperFlow automatically rectifies these violations and proceeds to finalize the layout files for AQFP circuits in Graphic Design System II (GDSII) format

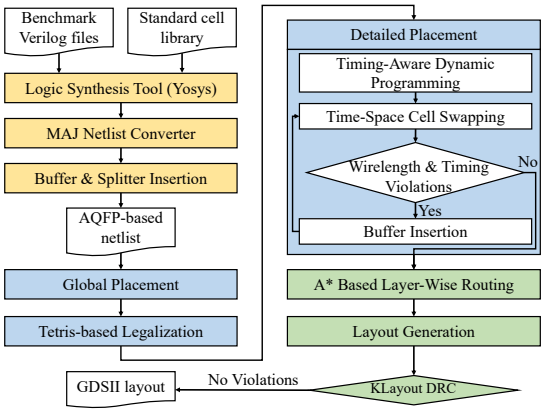

Fig. 3. **The overall design flow of SuperFlow.**

B. Majority-Based Logic Synthesis for AQFP

1) ***Majority Netlist Conversion****:* Fig. 3 illustrates the logic synthesis stage. AQFP prefers a majority-based netlist and we first leverage a CMOS-based synthesis tool, Yosys, to generate AOI netlists for input RTL files. The AOI netlist is then converted to an MAJ netlist by mapping all feasible three-input nets to their corresponding two-level majority-based implementation while maintaining the minimum overall resource consumption.

To elaborate, we view the netlist as a directed graph. In an AOI netlist, each node has two parent gates. While in a majority-based netlist, each node has two or three parents depending on its type of logic gate, e.g. an `and` gate or a `majority` gate. The majority netlist conversion involves three steps: (i) Identifying convertible three-input nets in AOI netlists. (ii) Applying a table-based method to map these nets to majority-based logic. (iii) Selecting the most resource-efficient mapping for each net.

We initiate by identifying feasible three-input nets in the AOI netlist using a depth-first search, starting from the netlist output. The search begins with a two-input net formed by the node and its two parent inputs, progressively adding parent nodes to expand the net. This iterative search continues until three independent parent nodes are found, ensuring no parent node is a descendant of another in this search. If more than three parent nodes are identified, the search is aborted, deeming the node unsuitable for a three-input net.

Feasible three-input nets identified earlier can be mapped to majority-based gates. Each net can be represented by up to two majority gates. To check for majority mapping suitability, we use a Karnaugh map checking method. This method compares the Karnaugh maps of all combinations of three parent nodes with the target net. If a match is found with primary gates, the net is mapped directly; otherwise, it is mapped to two-level majority-based logic with three majority gates at the first level and one at the second level, connecting to the outputs of the first three gates.

In optimizing the majority-based netlist, our goal is to minimize both Josephson junction count and clock delays. This requires a thorough search across the entire directed graph. We prioritize majority mappings that use more logic gates and clock phases, iterating through all mapping schemes for each three-input net to find the one that uses the least resources overall.

2) ***Buffer and Splitter Insertion****:* After converting the AOI netlist to a majority-based netlist, we insert `splitter` cells to the converted netlist to comply with the AQFP fan-out requirements. Gates with over two fan-outs require a `splitter` cell, selected based on the number of fan-outs. After addressing all fan-outs, we reset and recalculate net delays. Given the updated delay for each net, buffers are inserted in each data path to every gate accordingly to produce equal delay (clock phase), which is required by the AQFP technology [6]. Since the logic structure of the converted netlist is fixed, buffer insertion could be resolved in any order and will not change the total clock phases or the critical path. With all buffers and `splitter` cells in place, we finalize the majority-based AQFP netlist for subsequent placement and routing stages.

C. AQFP Placement

The placement in Fig. 3 can be divided into global placement, legalization, and detailed placement. In this section, we clarify the implementation details of these steps.

1) ***Problem Formulation****:* The AQFP placement problem can be formulated as a hypergraph $G = (V, E)$, where V and E denote the cells and nets, respectively. Let x_i and y_i be the x and y coordinates of the center of logic cell v_i. Given the AQFP

clocking architecture and a design netlist, our objective is to determine the positions of the cells that optimize the routed wirelength subject to the following constraints:

- Each logic cell should maintain consistent delay (clock phase), respecting the sequence from the netlist.
- Cells must not overlap with others, and two horizontally neighboring cells in a row can either be abutting or keeping a minimum spacing s_{min} (spacing constraint).
- All the wires must not exceed the maximum wirelength W_{max} (max-wirelength constraint).
- All timing requirements should be satisfied based on the clocking signal (timing constraint).
- Cells of different sizes should not cause perturbations that result in intra-row violations (mixed-cell-size constraint).

Therefore, we formulate our objective function for the row-wise AQFP placement problem as follows:

$$\begin{aligned} \min_{\mathbf{x}} \quad & \sum_{e_i \in E} W(e_i) + \lambda_t T(e_i), \\ s.t. \quad & \forall e_i \in E \quad W(e_i) \leq W_{max}, \\ & x_i + w_i \geq x_{i+1} - z_i B, \\ & x_i + w_i \leq x_{i+1} - z_i s_{min}, \end{aligned} \tag{1}$$

where the λ_t is the positive weight for timing cost, $W(e_i)$ is the wirelength cost function of net e_i, W_{max} is the maximum allowed wirelength, z_i is a binary value indicating whether two cells are abutted, B is a big positive constant, s_{min} is the minimum spacing, and w_i is the cell width. When $z_i = 0$, it means two cells are abutted, and the two equations correspond to $x_i + w_i = x_{i+1}$. When $z_i = 1$, only $x_i + w_i \leq x_{i+1} - s_{min}$ is required and two cells should maintain a minimal spacing horizontally. $T(e_i)$ is the four-phase timing model cost of net e_i, which depends on the located clock phase and is defined as follows:

$$T(e_i) = \begin{cases} (x_{end} - x_{start})^{\alpha}, & \text{if } phase \ \% \ 4 = 0 \\ (x_{end} + x_{start})^{\alpha}, & \text{if } phase \ \% \ 4 = 1 \\ (-x_{end} + x_{start})^{\alpha}, & \text{if } phase \ \% \ 4 = 2 \\ (2\hat{W} - x_{end} - x_{start})^{\alpha}, & \text{if } phase \ \% \ 4 = 3 \end{cases} \tag{2}$$

where x_{start} and x_{end} are the start and end x coordinates of net e_i, $\hat{W}$ is the layer width of the clock phase, and α is a parameter that modulating the relative importance of the connection across different clock phases, which we set to 2.

*2) **Global Placement**:* Global placement aims to determine the best possible cell locations that minimize the overall wirelength while respecting the max-wirelength and spacing constraints of the AQFP. Meanwhile, timing should be addressed during global placement to ensure that the generated placement results do not hold a negative slack. Thus, the objective function should also be timing-aware such that the placement results can work at the desired clock frequency.

A prevalent approach to this constrained minimization problem is to relax the constraints into the objective function and solve the unconstrained minimization problem:

$$\min_{\mathbf{x}} \quad \sum_{e_i \in E} W(e_i) + \lambda_t T(e_i) + \lambda_w (W(e_i) - W_{max}). \tag{3}$$

where λ_w is the weight for max-wirelength cost.

We utilize DREAMPlace [14] as our analytical global placement engine, incorporating a timing-aware objective function. To address the non-smooth, non-convex nature of the half-perimeter wirelength (HPWL) model, we use a weighted-average (WA) model for more effective wirelength cost optimization.

During global placement, we limit cell location optimization to the one-dimensional x axis, keeping the cell row (clock phase) fixed. Following this, we apply a Tetris-like legalization approach, aiming to preserve global placement results while minimizing cell overspreading. This legalization is iteratively performed to eliminate cell overlaps within each clock phase.

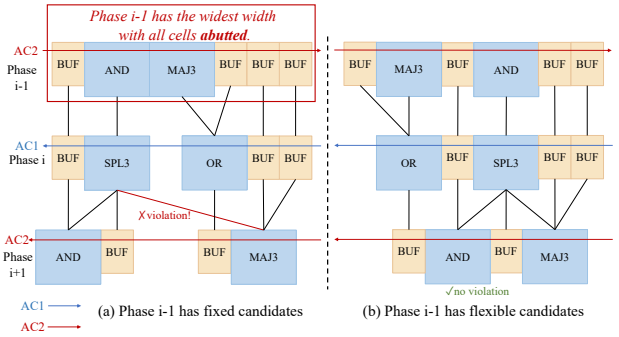

Fig. 4. **An illustration of the detailed placement stage where one clock phase has the widest width and all cells are abutted.** (a) When cell candidates are strictly matched to cells of identical sizes, it is possible to end up with a sub-optimal state, leaving some nets with timing violations (highlighted by the red line, SPL3 in phase i to MAJ3 in phase i+1); (b) By allowing flexibility in cell candidates and permitting cell swaps to avoid overlaps (MAJ3, AND, and BUF in phase i-1, in contrast to only MAJ3 and AND in (a)), the detailed placement result is better than (a), exhibiting no timing violations.

*3) **Detailed Placement**:* In the detailed placement stage, our goal is to optimally position cells in a non-overlapped circuit netlist $G = (V, E)$ with specified locations $L = (x, y)$, adhering to clocking and mixed-cell-size constraints while minimizing wirelength and timing costs. Since AQFP cells reside in dedicated layers to satisfy the path balancing requirement, a straightforward method is to transform detailed placement to the shortest path problem [15]. This task involves accommodating AQFP's four-phase clocking architecture and varying cell sizes, like 40μm by 30μm buffers and 60μm by 70μm majority gates. These cells, despite size differences, may coexist in the same clock phase, complicating cell swapping due to proximity issues. As shown in Fig. 4, if the phase i-1 has the widest width and all cells are abutted, then cell swapping only between candidates of identical sizes may result in sub-optimal solutions. Our framework overcomes this by allowing flexibility in choosing adjacent, non-overlapping cells of different sizes, effectively reducing timing violations in detailed placement.

D. Layer-Wise Fast A Routing with Space Expansion*

Given a netlist with all placed cell locations, the routing procedure aims to connect all nets with minimized wires and vias following AQFP wiring resources. There are two major differences between AQFP routing and CMOS routing: (i) AQFP uses `splitter` cells for fan-outs, so the pin connection

is one-to-one; (ii) the zigzag clocking architecture forces wires to only connect two adjacent layers (clock phases), not across whole chip area. Therefore, a layer-wire routing is sufficient for AQFP, instead of a separate global and detailed router.

As shown in Algorithm 1, we adopt the A* routing algorithm with a priority queue and estimated costs to find the shortest path, incorporating a dynamic step size to limit computation complexity. This means routing wires turn only after a set minimum spacing, for example, 10μm for the MIT-LL process. While this spacing is efficient, it may cause routability issues in large circuits. To resolve this, we iteratively increase layer distance by the minimum spacing and reroute until all constraints are satisfied. AQFP architecture ensures that this expansion only affects the relative positions of adjacent layers, keeping each layer's placement intact.

Algorithm 1: Layer-wise A* Routing with Space Expanding

```
Input  : Placement results with all cell locations and layers.
Output: Routing wires between each layer.
Function A_star(graph, start, goal):
    let the Queue and List equal an empty list of nodes;
    put start node on the Queue;
    while Queue is not empty do
        get current node from Queue;
        if current node is the goal then
            break;
        end
        update current cost;
        for adjacent nodes not in List do
            if cost is lower then
                update current cost;
                put adjacent node on the Queue;
                append current node to List;
            end
        end
    end
return List, cost
foreach layer do
    while routing space is not enough do
        expand spacing between two adjacent layers;
        A_star(graph, start, goal);
        update whole graph with routed wires;
    end
end
```

E. Layout Generation and DRC

The final step of SuperFlow is the layout generation and Design Rule Check (DRC). After placement and routing, we obtain the physical information (coordinates, rotation, wiring paths, track assignment, etc.) for all cells and wires. Using the AQFP standard cell library, these details are referenced in the layout file, compatible with most layout tools. We leverage KLayout for DRC, checking for rule compliance (e.g., metal layer density, via sizes, contact layer spacing). If the layout passes DRC, we generate the final AQFP GDS layout file; otherwise, we identify and address violation locations, adjusting placement and routing as needed. As an illustration, Fig. 5 displays the completed layout for the AQFP circuit apc128.

IV. Experimental Results

Settings: We implement SuperFlow in Python and conduct tests on a Linux machine that features an AMD Ryzen Threadripper 2920X processor with 12 cores and 64GB DDR4 memory.

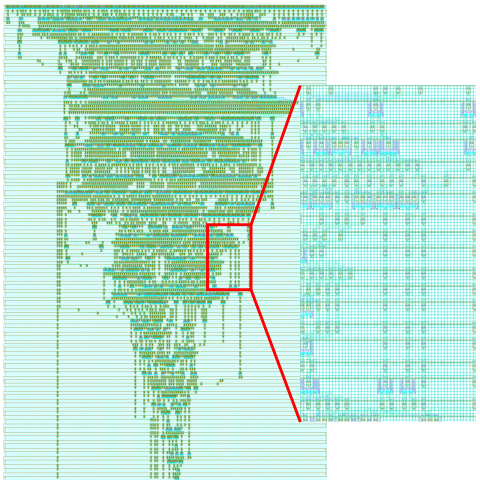

Fig. 5. **Layout for AQFP circuits apc128.**

TABLE II
MAJORITY-BASED LOGIC SYNTHESIS RESULTS

Circuits	#JJs	#Nets	#Delay
adder8	960	462	23
apc32	746	513	21
apc128	5,048	2,355	45
decoder	2,210	989	19
sorter32	3,788	1,474	30
c432	2,500	1,048	40
c499	4,946	2,202	31
c1355	4,996	2,236	31
c1908	4,716	2,182	34

Benchmark Circuits: We test our framework using classic benchmark circuits for AQFP testing [16], including 8-bit Kogge-Stone adder (adder8), 32-bit/128-bit approximate parallel counter (apc32/apc128), decoder, and 32-bit sorter (sorter32). We also validate it on the ISCAS'85 benchmark circuits to showcase the effectiveness of our framework [17]. Our comprehensive design flow is demonstrated from RTL netlists to final layouts, comparing our results with state-of-the-art AQFP placement tools.

A. Logic Synthesis Results of SuperFlow

Table II shows the statistic results of AQFP-based netlist after the logic synthesis step. We give the number of JJs, nets, and delay (clock phases) for each AQFP-based netlist. Note that the most basic structure of an AQFP cell (buffer) is a double-Josephson junction, and all other AQFP cells use more than 2 JJs. Therefore, the number of JJs is larger than the number of nets, and the results include all inserted buffers and splitters after the logic synthesis stage.

B. Placement Results of SuperFlow

Table III compares SuperFlow with the GORDIAN-based approach [8] and a timing-aware placer, TAAS [10] for AQFP circuits. We report the HPWL, inserted buffer lines, and the worst negative slack (WNS) using a timing analysis engine [10].

SuperFlow shows reduced HPWL and fewer buffers for small circuits like adder8 and apc32. For larger circuits (apc128,

TABLE III
THE COMPARISON ON THE DESIGN PERFORMANCE BETWEEN GORDIAN-BASED APPROACH [8], TAAS [10], AND SUPERFLOW.

Circuits	GORDIAN-Based [8]			TAAS [10]			SuperFlow			
	HPWL (μm)	Buffers	WNS (ps)	HPWL (μm)	Buffers	WNS (ps)	HPWL (μm)	Buffers	WNS (ps)	Runtime (s)
adder8	**10,948**	24	-	12,360	24	-	11,850	**16**	-	12.1
apc32	15,915	26	-	15,915	26	-	**15,530**	26	-	13.8
apc128	254,068	117	-40.7	245,416	110	-10.1	**177,620**	**67**	**-9.6**	374.8
decoder	**141,151**	34	-8.8	156,213	**33**	-1.4	153,030	43	**-1.0**	162.5
sorter32	168,208	29	-6.9	180,427	29	-3.3	**132,640**	29	**-2.3**	113.4
c432	51,009	46	-	52,208	45	-	**36,050**	**29**	-	50.1
c499	430,658	62	-29.9	431,108	62	-8.9	**385,845**	**59**	**-6.7**	517.5
c1355	422,556	58	-31.4	426,099	58	-9.1	**396,640**	**56**	**-8.9**	690.9
c1908	358,271	67	-25.5	361,071	**66**	-6.9	**357,570**	68	-6.9	353.3
Average	1.112	1.178	4.045	1.128	1.153	1.121	**1**	**1**	**1**	

Note: '-' means that the WNS is positive and no timing violations are found under the target clock frequency, which is set as 5GHz. For SuperFlow, the HPWL for all circuits are all integers of 10μm, because the AQFP standard cell library goes through an update, and now the cell height, width, pin location, are all integers of 10μm.

TABLE IV
ROUTING RESULTS OF SUPERFLOW.

Circuits	#JJs after Routing	#Nets	Routed WL (μm)
adder8	2,170	1,064	21,100
apc32	2,040	986	22,510
apc128	13,860	6,761	260,770
decoder	7,896	3,807	252,050
sorter32	8,768	3,938	218,210
c432	5,286	2,531	75,710
c499	19,050	9,329	816,240
c1355	21,004	10,315	932,960
c1908	15,408	7,574	617,350

c499, c1355, c1908), it achieves significant buffer reductions (up to 42.7% over GORDIAN-based, 39.1% over TAAS) and shorter wirelength. While GORDIAN-based method excels in wirelength, it faces timing issues. TAAS improves timing at a slight wirelength cost. This shows that SuperFlow optimization strategy is effective on large-scale circuits and cell swapping within different-sized cells could save inserted buffer lines. Overall, SuperFlow achieves 12.8% better wirelength with 15.3% fewer inserted buffer lines and 12.1% timing improvement over previous state-of-the-art placer, TAAS [10].

C. Routing Results of SuperFlow

Table IV demonstrates the final routing results for AQFP benchmarks including JJs, nets, and routed wirelength. We calculated all the cells and wires in the final layout, including buffers and splitters inserted during the logic synthesis and placement stage.

V. CONCLUSIONS

In this paper, we present SuperFlow, a fully-customized RTL-to-GDS design flow specifically tailored for AQFP superconducting circuits. SuperFlow simultaneously optimizes wirelength and timing while adhering to the clocking and mixed-cell-size constraints inherent in AQFP circuits throughout the design process. In the routing phase, SuperFlow employs a layer-wise strategy to support space expansion and address potential routability issues. Experimental results demonstrate that SuperFlow outperforms previous design tools in terms of wirelength and timing for AQFP circuits, setting a robust groundwork for future AQFP applications like RISC-V CPUs and neural network accelerators. [18].

VI. ACKNOWLEDGMENT

This work was supported by JST FOREST Program (Grant Number JPMJFR226W, Japan), and the NSF Expedition program CCF-2124453, NSF CCF-2008514.

REFERENCES

[1] N. Takeuchi *et al.*, "An adiabatic quantum flux parametron as an ultra-low-power logic device," *Supercond. Sci. Technol.*, 2013.
[2] Y. Harada *et al.*, "Basic operations of the quantum flux parametron," *IEEE Trans. Magn.*, 1987.
[3] N. Takeuchi *et al.*, "Energy efficiency of adiabatic superconductor logic," *Supercond. Sci. Technol.*, 2014.
[4] R. Cai *et al.*, "A stochastic-computing based deep learning framework using adiabatic quantum-flux-parametron superconducting technology," in *ISCA*, 2019.
[5] C.-Y. Huang *et al.*, "An optimal algorithm for splitter and buffer insertion in adiabatic quantum-flux-parametron circuits," in *ICCAD*, 2021.
[6] R. Cai *et al.*, "A majority logic synthesis framework for adiabatic quantum-flux-parametron superconducting circuits," in *GLSVLSI*, 2019.
[7] S.-Y. Lee *et al.*, "Beyond local optimality of buffer and splitter insertion for aqfp circuits," in *DAC*, 2022.
[8] H. Li *et al.*, "Towards aqfp-capable physical design automation," in *DATE*, 2021.
[9] Y.-C. Chang *et al.*, "Asap: An analytical strategy for aqfp placement," in *ICCAD*, 2020.
[10] P. Dong *et al.*, "Taas: a timing-aware analytical strategy for aqfp-capable placement automation," in *DAC*, 2022.
[11] G. Meuli *et al.*, "Majority-based design flow for aqfp superconducting family," in *DATE*, 2022.
[12] T. Tanaka *et al.*, "A full-custom design flow and a top-down rtl-to-gds flow for adiabatic quantum-flux-parametron logic using a commercial eda design suite," *IEEE Trans. Appl. Supercond.*, 2023.
[13] K. K. Likharev *et al.*, "Rsfq logic/memory family: A new josephson-junction technology for sub-terahertz-clock-frequency digital systems," *IEEE Trans. Appl. Supercond.*, 1991.
[14] Y. Lin *et al.*, "Dreamplace: Deep learning toolkit-enabled gpu acceleration for modern vlsi placement," in *DAC*, 2019.
[15] S. Dhar *et al.*, "An effective timing-driven detailed placement algorithm for fpgas," in *ISPD*, 2017.
[16] O. Chen *et al.*, "Adiabatic quantum-flux-parametron: Towards building extremely energy-efficient circuits and systems," *Scientific reports*, 2019.
[17] EPFL, "ISCAS'85 and Simple Arithmetic Benchmarks," https://github.com/lsils/SCE-benchmarks.
[18] Z. Li *et al.*, "Superbnn: Randomized binary neural network using adiabatic superconductor josephson devices," in *MICRO*, 2023.