Algebraic and Boolean Methods for SFQ Superconducting Circuits

Alessandro Tempia Calvino, Giovanni De Micheli
Integrated Systems Laboratory, EPFL, Lausanne, Switzerland

***Abstract*—Rapid single-flux quantum (RSFQ) is one of the most advanced and promising superconducting logic families, offering exceptional energy efficiency and speed. RSFQ technology requires delay registers (DFFs) and splitter cells to satisfy the path-balancing and driving-capacity constraints. In this paper, we present a comprehensive exploration of methods for synthesizing and optimizing SFQ circuits. Our approach includes algebraic and Boolean optimization techniques that work on the *xor-and graph* (XAG) representation of combinational logic. Additionally, we introduce a technology mapping method to satisfy the path-balancing and fanout constraints while minimizing the area. Finally, we propose a synthesis flow for SFQ circuits. In the experimental results, we show an average reduction in the area and delay of 43% and 34%, respectively, compared to the state-of-the-art.**

I. Introduction

Superconducting electronics (SCE) stands out as one of the most promising post-CMOS technology, proposing high-speed and power-efficient solutions. Superconducting circuits are based on *Josephson junctions* (JJs) and operate at a few degrees Kelvin (typically 4K) where resistive effects are negligible. The switching speed of Josephson junctions supports the realization of circuits clocked up to several tens of Gigahertz [1] and considerably lower power consumption compared to CMOS, even considering the refrigeration power [2].

Rapid Single-Flux Quantum (RSFQ) is the most mature superconducting logic family [3]. Multiple variants of RSFQ, such as the eSFQ [4], are commonly grouped under the term SFQ. Unlike CMOS, SFQ circuits encode the logic "true" in a small voltage pulse and the logic "false" in a pulse absence. Consequently, most SFQ logic gates are clocked to discern between these two states. These gates function as latches, with a clock input and one or more data inputs. When a pulse arrives at a data input, it alters the internal state of the gate. Subsequently, a clock pulse resets the gate to its initial state and may generate an output pulse based on the internal state. As SFQ circuits rely on the clock signal, they necessitate pipelining at the gate level. To ensure correct data propagation, i.e., data at each gate must be present at specific time-frames for correct computations, SFQ circuits require delay registers (DFFs) in the combinational paths so that every path from primary inputs to logic gates traverses the same number of clocked gates. This constraint is referred to as *path balancing*. Furthermore, due to the quantized nature of SFQ pulses, most RSFQ primitives have a maximum driving capacity of one gate. Consequently, a special cell called *splitter* is necessary to drive multiple fanouts.

Due to path-balancing DFFs and splitters, the area of SFQ circuits can grow prohibitively large. In the literature, several approaches have been proposed to tackle this problem. In [5], a few logic synthesis algorithms have been enhanced by integrating an approximate path-balancing cost. A similar extension has been proposed in [6] for technology mapping. Post-mapping optimization of path-balancing DFFs has been also proposed in [7] using retiming [8]. Other work proposes new SFQ gates [9], [10] and different clocking schemes [11], [12] to decrease the design cost.

In this paper, we present an innovative synthesis flow to carry out the optimization and mapping for SFQ technology. In particular, we focus on delay optimization which is key to synthesizing efficient SFQ circuits. Technology-independent optimization is carried over the *xor-and graph* (XAG) since it closely abstracts the logic primitives of SFQ. In fact, both 2-input XOR and 2-input AND (OR) gates have similar delay and area costs. Moreover, XAGs are more compact than the commonly used *and-inverter graph* (AIGs) offering better opportunities to restructure logic through additional rewriting rules and Boolean methods. We present several techniques namely, mapping, re-mapping, algebraic rewriting, *exclusive sums-of-products* (ESOP) balancing, Boolean rewriting, and resubstitution. Technology mapping is performed directly on the XAG without previous decomposition into an AIG. We describe post-mapping methods to satisfy the path-balancing and fanout constraints. Finally, we use minimum-area retiming to optimally minimize the number of inserted balancing DFFs.

In the experiments, we show that our synthesis flow efficiently reduces the delay without causing an area explosion. We compare against the state-of-the-art showing 43% and 34% reduction on average in area and delay, respectively.

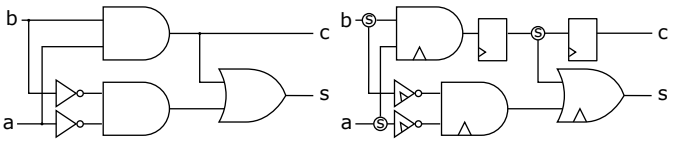
II. Background

In this section, we introduce the basic notations and the necessary background on RSFQ ciruits.

A. Rapid Single-Flux Quantum

Rapid Single-Flux Quantum (RSFQ) is a fast and energy-efficient superconducting logic family [3]. The particularity of the RSFQ technology is that it is based on pulsing logic utilizing *Josephson junctions* (JJs) as the primary switching elements. Different variants have been proposed in the literature to improve the energy efficiency of RSFQ, such as the *energy-efficient single-flux quantum* (eSFQ, [4]). In this paper, we refer in general to all the variants as *SFQ*.

Logic cells in SFQ technology are implemented as clocked elements. The behavior of these cells can be described as a state machine, where data input pulses modify the cell's state, and clock pulses generate the output signal based on



(a) CMOS circuit (b) Equivalent RSFQ circuit

Fig. 1: RSFQ mapping with clocked gates, path-balancing DFFs, splitters, and balanced POs

that state while resetting it. Hence, a gate can be viewed as a combination of combinational logic coupled with a DFF. Consequently, SFQ circuits may implement gate-level pipelining. SFQ circuits require *delay registers* (DFFs) in the combinational paths to ensure correct data propagation. These DFFs ensure that each path from the primary inputs to logic gates traverses the same number of clocked gates, a condition known as *path balancing*. Additionally, to enable gate-level pipelining, i.e., new data can be provided every clock cycle, also POs must be balanced. Balanced POs are a constraint in sequential systems where the register-level clock is a multiple of the gate-level clock. Due to the quantized nature of SFQ pulses, most RSFQ primitives have a maximum driving capacity of just one gate. A special cell called *splitter* is employed to drive multiple fanouts, commonly up to 2 gates.

Figure 1 shows the difference between a CMOS circuit and an equivalent RSFQ circuit. Notably, in Figure 1b gates are clocked and DFFs are inserted to satisfy the path-balancing constraint. Specifically, a DFF is placed before the OR gate, and another one before the output C. Last, splitter cells are inserted to drive multiple fanouts.

Cell libraries in SFQ technology typically comprise a set of basic combinational blocks, such as DFF, INV, AND2, OR2, and XOR2 [6], [13]. The evaluation of the area in SFQ technology, prior to place and route, is commonly based on counting the number of Josephson junctions (JJs) utilized. Similarly, the delay is typically expressed as the maximum number of cycles required for the circuit to complete its computation. This value corresponds to the length of the path that traverses the highest number of clocked cells in the combinational logic. The clock frequency is typically neglected before place and route as hard to characterize [6].

B. Notations and Definitions

A *Boolean network* is modeled as a directed acyclic graph (DAG) with nodes represented by Boolean functions. The sources of the graph are the *primary inputs* (PIs) of the network, the sinks are the *primary outputs* (POs). For any node n, the *fanins* of n is a set of nodes that have an outgoing edge towards n. Similarly, the *fanouts* of n is a set of nodes that have an incoming edge from n.

A *cut* C is a pair (n, $\mathcal{K}$), where n is a node called *root*, and $\mathcal{K}$ is a set of nodes, called *leaves*, such that 1) every path from any PI to node n passes through at least one leaf and 2) for each leaf $v \in \mathcal{K}$, there is at least one path from a PI to n passing through v and not through any other leaf. The *size* of a cut is defined as the number of leaves. A cut *covers* all the nodes encountered on the paths between the leaves and the root, including the root and excluding the leaves.

A *cover* of a Boolean network is a set of cuts such that 1) each node in the network is covered by at least one cut and 2) the root of each cut in the cover is either a PO of the network or a leaf of one or more cuts in the cover. A cover can be extracted in reverse topological order by selecting cuts rooting in the POs and recurring on the leaves.

III. Optimization for SFQ circuits

Technology libraries for SFQ are typically simple and implement basic functions. Notably, logic cells within these libraries are all clocked and have similar areas. Interestingly, XOR cells demonstrate a similar level of efficiency as AND cells. Based on this observation, we center our technology-independent synthesis flow for SFQ on *xor-and graphs* (XAGs). XAGs have multiple advantages over the commonly used *and-inverter graphs* (AIGs). XAGs are more compact since they contain one additional primitive, which is implemented using 3 2-input ANDs in AIGs. Consequently, circuits represented by XAGs tend to be smaller and shallower. Moreover, XAGs offer more opportunities to restructure logic through additional rewriting rules and Boolean methods.

Minimizing the logic depth is essential for a fast and compact SFQ circuit. First, the logic depth directly relates to the circuit latency. Latency is generally dominated by the number of clock cycles needed to realize a function. Since each SFQ gate is clocked, the latency can be approximated as the logic depth of the circuit. Second, delay optimization helps to minimize the number of necessary balancing DFFs. Intuitively, longer critical paths require more DFF elements due to longer paths to balance.

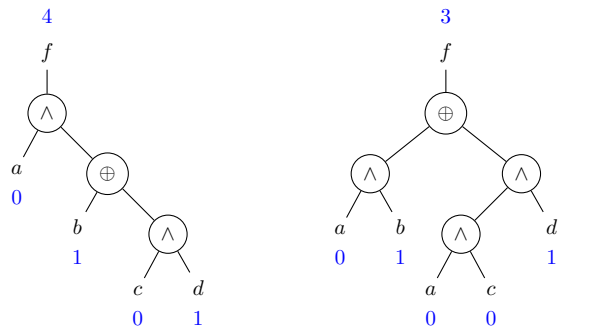
In this section, we present several techniques aimed at optimizing delay and area over the XAGs, thereby enhancing the overall efficiency and performance of SFQ circuits. First, we present a method to obtain an optimized XAG representation. Then, we present techniques that target delay optimization. Finally, we describe two methods to reduce the area.

A. Obtain the initial XAG

Given a generic Boolean network, the first problem to address is how to obtain a good initial *xor-and graph* (XAG) representation. To achieve it, we employ the state-of-the-art mapping strategy based on [14]. It consists of a delay-oriented mapping algorithm that leverages a database of size-optimum XAG structures as a library. The XAG structures are derived using SAT-based exact synthesis over the 4-input NPN classes [15]. Multiple minimum structures with different pin-to-pin delays are stored for each class. The utilization of this method yields significantly improved results compared to merely identifying XOR gates, as it incorporates Boolean optimization and rewriting by utilizing locally optimum structures.

B. Depth Optimization

We propose three efficient algorithms, one algebraic and two Boolean, to optimize depth over the primitives AND and XOR.



(a) Before rewrite (b) After rewrite

Fig. 2: Rewriting with AND-XOR distributivity in Ψ

Algorithm 1: XAG depth algebraic rewriting

```
Input : XAG N
Output: Optimized XAG N
foreach edge p ∈ PO(N) do
    if in_critical_path(N, p) then
        foreach node n ∈ critical_path(N, p) do
            Ω.A_AND(N, n);
            Ω.A_XOR(N, n);
            Ψ.D_AND−OR(N, n);
            Ψ.D_AND−XOR(N, n);
        end
    end
end
return N;
```

1) XAG algebraic rewriting: Boolean algebra defines primitive transformations and properties, referred to as a set Ω. In the context of delay optimization, a subset of these fundamental properties plays an important role. In particular, for XAGs, we can identify the following algebraic rules contained in Ω:

$$\Omega.A_{AND}: \; a \wedge (b \wedge c) = (a \wedge b) \wedge c \tag{1}$$

$$\Omega.A_{XOR}: \; a \oplus (b \oplus c) = (a \oplus b) \oplus c \tag{2}$$

$$\Omega.D_{AND-OR}: \; a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c) \tag{3}$$

where (1) is the associative property of AND, (2) is the associative property of XOR, and (3) is the distributive property of AND over OR. Note that the associative property of OR is contained in (1) and that (3) is valid also for AND-XOR. In XAGs, transformations (1) and (2) can push critical signals forward towards the POs, when the associative property holds. Transformation (3), on the other hand, is less directly applicable. Experimental results have shown that the transformation is rarely useful, especially in optimized circuits. Nevertheless, two powerful transformations can be derived from (1), (2), and (3) when considering logic cones of 3 operators and the interaction between AND-OR and AND-XOR. We refer to the extended set as Ψ:

$$\Psi.D_{AND-OR}: \quad a \wedge (b \vee (c \wedge d)) = (a \wedge b) \vee ((a \wedge c) \wedge d) \tag{4}$$

$$\Psi.D_{AND-XOR}: \quad a \wedge (b \oplus (c \wedge d)) = (a \wedge b) \oplus ((a \wedge c) \wedge d) \tag{5}$$

These two rules define the distribution of AND over OR (1) and AND over XOR (2). Note that in an XAG, the OR is represented as an AND with inverted inputs and output according to De Morgan's laws.

Figure 2 shows an example of how the derived AND-XOR distributive rule can reduce the delay of a circuit. The transformation is applied to the XAG in Figure 2a. The critical signal d binds the arrival time at the output. The transformation pushes signal d forward by distributing a through the XOR operation and applying AND-associativity to d. The result in Figure 2b shows a delay reduction by one, from 4 to 3, at the cost of one additional AND operation.

The algebraic rules are graph-based. Hence, they are extracted structurally from an XAG. We employ rules to detect properties and apply transformations based on input arrival times, inverted edges, and gate type. Node duplication is also enabled in the case of nodes with multiple fanouts.

Algorithm 1 shows how the algebraic rewriting rules are carried out on an XAG. The algorithm applies rewriting rules on critical paths reachable from one PO at a time. The critical paths are updated after every successful move. The algorithm first tries to apply moves based on associativity, which incur less area increase compared to distributive rules. When associativity fails, distributive rules are attempted.

2) ESOP balancing: *Exclusive-Sums-of-Products* (ESOPs) are a two-level representation of Boolean functions composed of an exclusive OR of product terms. Differently from *Sums-of-Products* (SOPs), which are based on the primitives AND and OR, ESOPs utilize AND and XORs. In the SFQ context, ESOPs are interesting since XORs are efficient gates. Moreover, for many Boolean functions, the number of cubes in minimal ESOPs is lower than the number of cubes in minimal SOPs [16].

SOP-balancing [17] is a scalable technique to optimize for delay. It consists of extracting small cones of logic, generally up to ten variables, converting them into SOPs, and applying AND balancing to each term and the sum. In practice, each term is considered as a multi-input AND and it is decomposed into 2-input ANDs while minimizing the arrival time of the term root. The sum is decomposed similarly into 2-input ORs.

In this work, we employ ESOP-balancing to achieve delay optimization over the primitives AND and XOR. Algorithm 2 presents a high-level view of ESOP balancing. Given a large XAG, *cut enumeration* [18] is used to break the circuit into multiple logic cones. For each cone, an initial ESOP cover is extracted using the algorithm described in [19] that computes an exact minimum Pseudo-Kronecker expression (PKRMs). PKRMs are a specific subset of ESOP expressions that can be generated using only positive or negative Davio expansion and Shannon expansion. In our implementation, we extract PKRMs directly from truth tables without involving BDDs as in the original formulation. Then, the initial cover is minimized to obtain a compact ESOP using the *EXORCISM* family of heuristics [20]. To reduce the run time, ESOPs are cached for later reuse using a hash table. Next, AND- and XOR-balancing is performed to generate a decomposition tree that minimizes the arrival time at the root n. The arrival time of a leaf is defined as the arrival time of the best cut computed at the

Algorithm 2: ESOP balancing

```
1  Input  : XAG N
2  Output: Optimized XAG M
3  C ← ∅;
4  foreach node n ∈ N in topological order do
5      C(n) ← compute_cuts(N, C, n);
6      foreach cut c ∈ C(n) do
7          tt ← compute_truth_table(N, c);
8          esop ← compute_exact_pkrm(tt);
9          exorcism(esop);
10         compute_balancing_cost(N, C, n, esop);
11         if better delay than best cut then
12             set_best_cut(C(n), c);
13         end
14     end
15 end
16 area_recovery(N, C);
17 M ← extract_cover(N, C);
18 return M;
```

leaf. Algorithm 2 works similarly to a technology mapper [14], [17]. In the first section, from line 4 to line 15, the algorithm computes the cuts and selects one having the best arrival time for each node. At line 16, the area is recovered by selecting lower-cost decompositions in paths where the slack is positive. Area recovery works similarly to [14]. This step is crucial to minimize the area increase derived from balancing due to logic duplication. Finally, at line 17, a cover is extracted and converted into a balanced and delay-optimized XAG.

3) Remapping: : XAG-remapping [14] is a rewriting technique that maps an XAG network to obtain a new XAG implementation. A library of pre-computed XAG structures is used to optimize the logic. This method is equivalent to the one in Section III-A but with the difference that the input and output data structures are the same. In the synthesis flow, remapping minimizes the delay first and then recovers the area constrained by the found delay.

C. Area recovery

In area recovery, *rewriting* and *resubstitution* have been extended to work for XAG optimization.

DAG-aware rewriting [21] is a fast greedy algorithm that aims at minimizing the size of a logic network by iteratively replacing sub-graphs rooted in a node with smaller pre-computed structures while preserving the functionality at the root node. A database of pre-computed structures covers all the 4-variable functions classified into the NPN equivalence classes for compactness [15]. In our implementation, pre-computed structures are the same used for previous mapping tasks. Differently from remapping, this algorithm is DAG-aware, i.e., it can reuse nodes that are already present in the network. Hence, it is more suited for area optimization. Our implementation constraints transformations to not increase the circuit depth. Thus, required times are used to filter transformations.

Resubstitution (re)expresses the function of a node using other nodes, called *divisors*, that are already present in the network. The transformation is accepted if the new implementation of a node is better in size compared to the current implementation of the node in terms of its immediate fanins. This approach generalizes to *k-resubstitution*, which adds k new nodes and removes at least $k + 1$ nodes. In XAG-resub [22], added gates are 2-input ANDs and XORs with optional inverters at the inputs/outputs.

IV. Technology mapping

After technology-independent optimization, technology mapping translates the optimized XAG in terms of the connection of cells from an SFQ cell library. This process involves 3 steps: mapping, balancing DFF insertion, and splitter insertion.

A. Mapping

In our approach, we adopt a direct mapping strategy that starts from *xor-and graphs* (XAGs) as the subject graph, enabling us to efficiently map into the SFQ cell library. To achieve reduced latency and area, the mapper is configured for minimal delay, focusing on optimizing performance. To further enhance the quality of the mapping and minimize delays, we introduce a pre-computed library of *supergates* [23]. These supergates are single-output networks constructed from a few library cells, treated as a single complex cell. The use of supergates provides the distinct advantage of mitigating the structural bias of the mapping algorithm, which often heavily relies on the initial subject graph structure. The supergates are generated recursively in multiple rounds using an enumeration process, ensuring a thorough exploration of possibilities.

Previous work in SFQ mapping introduced a technology mapper called PBMap [6], which incorporates an approximate path-balancing cost into the area cost function. The approach defines this cost based on dynamic programming, ensuring local-optimal balancing for logic trees. However, when dealing with optimized logic in a DAG format, where nodes may have multiple fanouts, using path-balancing cost in technology mapping presents three key disadvantages. First, PBMap overlooks area costs and relies solely on path-balancing costs that only work for trees of logic. In contrast, DAG-mapping approaches and heuristics have been demonstrated to be superior in comparison to minimizing delay and area [24]. Second, the heuristic used in PBMap treats the path-balancing cost for each cell as independent, disregarding the potential sharing of DFFs among cells connected to the same node. Consequently, PBMap overestimates the penalty of imbalanced solutions. Additionally, the exact DFF sharing information remains unknown until the entire network is mapped, making it challenging for their dynamic programming approach to efficiently capture this complexity. Moreover, the number of padding DFFs can be further optimized in a post-processing phase by moving DFFs upwards or backward through logic. Last, the simplicity of SFQ libraries enables technology mappers to already map logic trees with optimal solutions by prioritizing local delay first and area second, without the need to incorporate additional balancing costs. It is worth noting that this proposition may not hold for libraries containing multi-input cells (more than 2 inputs). However, these complex cells lack efficient implementations in SFQ. Experimental results confirmed these claims, indicating that delay-oriented mapping yields better area results on average compared to the cost

function in PBMap. Furthermore, attempts to use the balancing cost as a tie-breaker during cell selection heuristics have not demonstrated any significant advantage.

Similar considerations apply to the integration of the path-balancing cost in technology-independent algorithms [5] where also real critical paths are not known. For instance, since inverters are not represented, their contribution to the delay is not considered during the optimization leading to incorrect balancing.

B. Path balancing

After technology mapping, our approach inserts padding DFFs to fulfill the balancing constraint of the circuit for internal nodes and POs. The DFFs are inserted utilizing ASAP scheduling, ensuring that the arrival times at each cell's input are synchronized (balancing constraint), while maintaining a constant delay at their outputs (ASAP balancing policy). To optimize the area, our method shares DFFs among nodes connected to the same input node at the same clock level. This algorithm guarantees an optimal DFF insertion for the ASAP schedule, and it operates linearly with respect to the number of gates in the circuit.

After initial insertion, balancing DFFs are minimized using the minimum-area retiming algorithm in [25], [26]. This algorithm iteratively pushes DFFs backward toward the PIs in order to globally minimize their number.

C. Splitter insertion

As the final stage of technology mapping, we introduce splitter cells to address the fanout constraint. Splitters are inserted as balanced trees, considering that logic is inherently balanced. Given that splitters in SFQ have a driving capacity of 2 gates, their number for any arbitrary gate n is equivalent to $|FO(n)|-1$, where $FO(n)$ represents the fanouts of gate n.

V. Experiments

The methods and the synthesis flow have been implemented in C++17 in the open-source logic synthesis framework *Mockturtle* [27]. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified to be functionally equivalent and to fulfill the path-balancing and fanout constraints.

A. Synthesis flow

Algorithm 3 shows the synthesis flow used in our experiments. The initial point is a Boolean network which is simply an *and-inverter graph* (AIG) in our experiments. First, some fast and light area optimizations are performed using the area-recovery algorithms in section III-C. The objective is to get a compact starting point by removing logic redundancy. It is crucial to not over-optimize at this step, and transformations are accepted only if there is a significant improvement and zero delay increase. Then, the core optimization starts using the algorithms in Section III. An iteration alternates delay optimization with area recovery. On the one hand, more the iterations better the delay that can be obtained in the final SFQ circuits. On the other hand, over-optimization leads to area increase. In our experiments, we perform one iteration. After, technology-independent optimization, the SFQ circuit is obtained and optimized using the methods in section IV. To maintain tractability and efficiency, we impose constraints on the supergates. Specifically, we generate 6294 supergates limiting the number of inputs to 5 and the number of cell levels to 3.

Algorithm 3: Synthesis flow

```
Input : Boolean network N, Iterations i, Library L
Output: SFQ circuit M
xag ← map_to_xag(N);
fast_area_opt(xag);
for i iterations do
    xag_depth_algebraic_rw(xag);
    xag_resub(xag);
    xag ← map_to_xag(xag);
    xag ← esop_balancing(xag);
    xag_boolean_rw(xag);
end
M ← map(xag, L);
path_balance(M);
min_area_retime(M);
insert_splitters(M);
return M;
```

The most computationally-intensive algorithm of the synthesis flow is the minimization of path-balancing DFFs using retiming. The complexity of retiming depends on the delay of the circuit and the number of path-balancing DFFs. Nevertheless, problems of a million of DFFs can be solved in a few seconds. All the other algorithms are linear w.r.t the number of nodes in the circuit.

B. Comparing agaist the state-of-the-art

In our experiments, we conducted a comparison against the current state-of-the-art RSFQ results in PBMap [6]. To perform the evaluation, we utilized the Suny RSFQ cell library [28]. This library shares the same cells as the one employed in [6], which is not openly available, although the JJ count per cell might vary slightly (delay and DFF count remain unaffected). We report the results over the ISCAS [29] and EPFL [30] benchmarks. The baseline consists of unoptimized designs in the AIG format. We evaluate the quality of the design in terms of JJ count for area and logic depth for latency, as clock frequency cannot be truly characterized before place and route. This is in line with prior work. Differently from PBmap, we map the benchmarks to enable gate-level pipelining. Hence, in our method, also POs are balanced.

Table I shows the results of the comparison. For our approach, we show the size and depth of the optimized XAG, and the area (number of JJs), number of path-balancing DFFs, and delay (as number of cycles) of the obtained RSFQ circuit. The time shows the total synthesis and mapping run time. For PBMap, we show the area, number of DFFs, and delay. Our synthesis algorithms considerably reduce the average size and depth of the baseline by 21.72% and 46.30%, respectively. After technology mapping, area, DFFs, and delay are reduced by 43.00%, 24.20%, and 34.44% compared to PBMap. Our approach obtains higher area and DFF count only on benchmark *s38417* due to additional DFFs for PO balancing.

TABLE I: Evaluation of our method against the state of the art

Benchmark	Baseline		PBMap [6]			Our method					
	Size	Depth	Area (JJs)	DFFs	Delay	Size	Depth	Area (JJs)	DFFs	Delay	Time (s)
c499	398	19	7758	476	13	217	9	4157	276	9	0.74
c880	325	25	12909	774	22	311	13	7187	482	14	0.85
c1908	341	27	12013	696	20	163	11	3634	287	11	0.74
c3540	1024	41	28300	1159	31	782	21	16278	798	22	2.03
c5315	1776	37	52033	2908	23	1096	17	23849	1735	16	2.08
c7552	1469	26	48482	2429	19	1010	17	22582	1824	16	2.97
s1196	477	19	15332	746	18	454	12	9701	506	12	1.35
s1238	532	23	17617	864	19	499	12	10060	572	13	1.41
s38417	9219	30	208289	8405	21	7492	16	209775	24125	17	5.64
sin	5416	225	215318	13666	182	5254	85	110254	5550	82	13.10
cavlc	693	16	16339	522	17	644	12	11888	381	12	1.70
dec	304	3	5469	8	4	304	3	5096	8	4	0.48
int2float	260	16	6432	270	16	210	10	3891	149	10	0.75
priority	978	250	102085	9064	127	490	13	10099	572	14	4.53
Improvement						**21.72%**	**46.30%**	**43.00%**	**24.20%**	**34.44%**	

VI. Conclusion

In this work, we proposed a logic synthesis flow targeting *single-flux quantum* (SFQ) superconducting circuits. Logic optimization is performed on the *xor-and graph* (XAG) for two reasons: i) SFQ primitive gates, namely the 2-input AND and 2-input XOR, exhibit similar area and delay costs; ii) XAGs are more compact than *and-inverter graphs* (AIGs) and offer more opportunities to restructure logic. First, we presented algebraic and Boolean methods that primarily target delay optimization. Then, we presented a technology mapping technique that involves *supergates*. Path-balancing DFFs are inserted and then optimized using minimum-area retiming. Finally, balanced splitter trees are inserted to satisfy the fanout constraints. In the experiments, we showed an improvement of 43%, 24%, and 34% for area, DFF count, and delay respectively compared to the state-of-the-art.

Acknowledgments

This research was supported by the SNF grant "Supercool: Design methods and tools for superconducting electronics", 200021_1920981, and Synopsys Inc.

References

[1] T. Kawaguchi, M. Tanaka, K. Takagi, and N. Takagi, "Demonstration of an 8-bit sfq carry look-ahead adder using clockless logic cells," in *International Superconductive Electronics Conference*, July 2015.

[2] D. S. Holmes, A. L. Ripple, and M. A. Manheimer, "Energy-efficient superconducting computing—power budgets and requirements," *IEEE Trans. on Applied Superconductivity*, 2013.

[3] K. Likharev and V. Semenov, "RSFQ logic/memory family: a new josephson-junction technology for sub-terahertz-clock-frequency digital systems," *IEEE Trans. on Applied Superconductivity*, 1991.

[4] O. A. Mukhanov, "Energy-efficient single flux quantum technology," *IEEE Transactions on Applied Superconductivity*, pp. 760–769, 2011.

[5] G. Pasandi and M. Pedram, "Balanced factorization and rewriting algorithms for synthesizing single flux quantum logic circuits," in *Proc. GLSVLSI*, 2019.

[6] G. Pasandi and M. Pedram, "PBMap: A path balancing technology mapping algorithm for single flux quantum logic circuits," *Trans. on Applied Superconductivity*, 2019.

[7] N. K. Katam and M. Pedram, "Logic optimization, complex cell design, and retiming of single flux quantum circuits," *IEEE Trans. on Applied Superconductivity*, 2018.

[8] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, p. 5–35, jun 1991.

[9] R. Bairamkulov and G. De Micheli, "Compound logic gates for pipeline depth minimization in single flux quantum integrated systems," in *Proc. GLVLSI*, 2023.

[10] R. Bairamkulov, A. T. Calvino, and G. De Micheli, "Synthesis of SFQ circuits with compound gates," in *Proc. VLSI-SoC*, 2023.

[11] K. Gaj, E. G. Friedman, and M. J. Feldman, *Timing of Multi-Gigahertz Rapid Single Flux Quantum Digital Circuits*, p. 135–164. Springer US, 1997.

[12] G. Pasandi and M. Pedram, "An efficient pipelined architecture for superconducting single flux quantum logic circuits utilizing dual clocks," *IEEE Trans. on Applied Superconductivity*, 2020.

[13] S. Yorozu, Y. Kameda, H. Terai, A. Fujimaki, T. Yamada, and S. Tahara, "A single flux quantum standard logic cell library," *Physica C: Superconductivity*, vol. 378-381, pp. 1471–1474, 2002.

[14] A. T. Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *ASP-DAC*, 2022.

[15] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Trans. CAD*, 2020.

[16] T. Sasao and M. Fujita, "Representation of discrete functions," Springer, 1996.

[17] A. Mishchenko, R. Brayton, S. Jang, and V. Kravets, "Delay optimization using SOP balancing," in *Proc. ICCAD*, pp. 375–382, 2011.

[18] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. FPGA*, 1999.

[19] R. Drechsler, "Pseudo-kronecker expressions for symmetric functions," *IEEE Transactions on Computers*, vol. 48, no. 9, pp. 987–990, 1999.

[20] A. Mishchenko and M. Perkowski, "Fast heuristic minimization of exclusive-sums-of-products," *Intern. Reed-Muller Workshop*, 2001.

[21] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *Proc. DAC*, 2006.

[22] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, J. Olson, R. Brayton, and G. De Micheli, "Improvements to boolean resynthesis," in *Proc. DATE*, pp. 755–760, 2018.

[23] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping," in *Proc. ICCAD*, 2005.

[24] Y. Kukimoto, R. Brayton, and P. Sawkar, "Delay-optimal technology mapping by DAG covering," in *Proc. DAC*, pp. 348–351, 1998.

[25] A. P. Hurst, A. Mishchenko, and R. K. Brayton, "Fast minimum-register retiming via binary maximum-flow," in *Proc. FMCAD*, 2007.

[26] A. T. Calvino and G. De Micheli, "Depth-optimal buffer and splitter insertion and optimization in AQFP circuits," in *Proc. ASP-DAC*, 2023.

[27] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, S.-Y. Lee, A. T. Calvino, D. S. Marakkalage, and G. D. Micheli, "The EPFL logic synthesis libraries," *CoRR*, vol. arXiv:1805.05121v3, 2022.

[28] "Suny RSFQ cell library." http://www.physics.sunysb.edu/Physics/RSFQ/Lib/contents.html.

[29] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Des. Test. Comput.*, 1999.

[30] L. Amarù, P.-E. Gaillardon, and G. D. Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, 2015.