Depth-optimal Buffer and Splitter Insertion and Optimization in AQFP Circuits

Alessandro Tempia Calvino
EPFL
Lausanne, Switzerland

Giovanni De Micheli
EPFL
Lausanne, Switzerland

ABSTRACT

The Adiabatic Quantum-Flux Parametron (AQFP) is an energy-efficient superconducting logic family. AQFP technology requires buffer and splitting elements (B/S) to be inserted to satisfy path-balancing and fanout-branching constraints. B/S insertion policies and optimization strategies have been recently proposed to minimize the number of buffers and splitters needed in an AQFP circuit. In this work, we study the B/S insertion and optimization methods. In particular, the paper proposes: i) an algorithm for B/S insertion that guarantees global depth optimality; ii) a new approach for B/S optimization based on minimum register retiming; iii) a B/S optimization flow based on (i), (ii), and existing work. We show that our approach reduces the number of B/S up to 20% while guaranteeing optimal depth and providing a 55× speed-up in run time compared to the state-of-the-art.

ACM Reference Format:
Alessandro Tempia Calvino and Giovanni De Micheli. 2023. Depth-optimal Buffer and Splitter Insertion and Optimization in AQFP Circuits. In *28th Asia and South Pacific Design Automation Conference (ASPDAC '23), January 16–19, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3566097.3567895

1 INTRODUCTION

In recent years, superconducting electronics (SCE) gained increasing interest 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 at several tens of Gigahertz and a considerably lower power consumption compared to CMOS. The potential of SCE is well supported by academic and industrial projects which address the electronic design automation (EDA) challenges of digital SCE design [7, 14, 20].

The *adiabatic quantum-flux parametron* (AQFP) is a superconducting logic family that targets low-energy consumption. In this technology, adiabatic switching operations drastically reduce the dynamic and static power consumption compared to other superconducting logic families [17]. AQFP circuits operate at frequencies up 10 Gigahertz with a power dissipation of two orders of magnitude lower compared to CMOS when accounting also for the cryo-cooling energy [7].

ASPDAC '23, January 16–19, 2023, Tokyo, Japan

ACM ISBN 978-1-4503-9783-4/23/01.
https://doi.org/10.1145/3566097.3567895

In AQFP circuits, due to a different encoding of the information compared to CMOS, each logic gate needs an alternating excitation current that periodically releases the computation. The excitation current is delivered as a clock [16]. Thus, data at each gate must be present at specific time frames for correct functionality. This may require the insertion of clocked *buffers* such that all data paths at each gate's fanin have the same length. This design constraint is called *path-balancing*. Logic gates also have limited driving capabilities. Branching elements called *splitters* are necessary for multiple fanouts. Splitters need a clock to operate and typically support up to 4 fanouts. This second design constraint is called *fanout-branching*.

The path-balancing and fanout-branching requirements complicate the design process and significantly affect area and delay. In some applications, buffers and splitters (B/S) arrived to occupy half of the total area even after optimization [3–5, 11, 13]. Hence, developing EDA tools able to minimize the number of buffers and splitters is of primary importance. Existing work considered AQFP constraints during logic optimization to reduce imbalances and high-fanouts by modifying the logic [4, 13, 19]. Other previous work developed techniques to insert and minimize the number of buffers and splitters needed in an AQFP circuit after logic synthesis [5, 8, 11].

In this paper, we tackle the B/S insertion problem for area and delay minimization given a logic network. Similarly to [11], we formulate the B/S insertion problem as a *scheduling* problem. First, we propose a linear-time algorithm to insert B/S elements such that the resulting AQFP circuit is delay-optimal. Minimizing the delay is beneficial for the area since it reduces the number of buffers needed for path-balancing. None of the previous approaches guarantee global delay optimality [5, 8, 11]. Then, we present a novel B/S optimization method based on minimum-register retiming [12] to reduce the number of buffers and splitters in an AQFP circuit. Finally, we propose an AQFP B/S insertion and optimization flow based on depth-optimal B/S insertion, retiming, and the chunk movement algorithm presented in [11].

In the experiments, we show that our approach reduces the number of buffers and splitters up to 20% compared to the state-of-the-art algorithm [8] while providing a 55× speed-up in run time. Finally, we show that our approach scales up to large benchmarks.

2 BACKGROUND

2.1 Adiabatic Quantum-Flux Parametron

The *adiabatic quantum-flux parametron* (AQFP) is an energy-efficient superconducting technology. The main elements of AQFP circuits are the buffer cell and the branch cell. The buffer cell is realized using *Josephson junctions* (JJs) and superconductive inductors to form a two-junction SQUID [17]. The basic functional block is the *majority-of-3* gate (MAJ3) that can be realized using three buffers

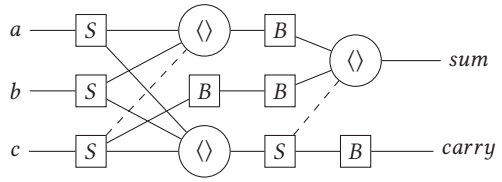


Figure 1: Full-adder in AQFP technology

and a 3-to-1 branch cell at their outputs. By modifying a buffer cell of the MAJ3 to produce a constant *zero* or *one* output, the MAJ3 implements the AND2 or the OR2 cell. Inverters can be implemented in a buffer without additional cost by using a negative mutual inductance instead of a positive one [18]. Since the AQFP logic is inherently majority-based, the functionality of an AQFP circuit can be represented by a *majority-inverter graph* (MIG) [2].

AQFP gates need an AC excitation current to operate. This signal also serves as a clock used to synchronize the computation at the output of the gates. Consequently, data at the inputs of a gate must be available in the same clock cycle to perform the correct operation. Hence, fast signals need to be delayed by inserting clocked buffers as delay elements. This problem is called *path-balancing*. AQFP circuits also have limited driving capabilities. Branching elements composed of a buffer cell and a branch cell are necessary whenever a gate needs to drive more than one output. These elements are called *splitters*. This constraint is called *fanout-branching*. Typically, splitters have a maximum driving capacity of 3 or 4 and are clocked cells. Hence, splitters affect path-balancing. Figure 1 shows a full adder realized using 3-input majority gates that fulfil these constraints. Splitters (S squares) are correctly inserted to drive multiple gates. Buffers (B squares) are used to balance the paths such that each path input to output traverses the same number of gates.

Different technology assumptions have been proposed to approach the path-balancing and fanout-branching problems [11]. In particular, these assumptions consider the cases in which *primary inputs* (PIs) and *primary outputs* (POs) need to be branched and balanced or not. For instance, if primary inputs are available for multiple cycles, they do not need to be balanced. In this paper, we assume that PIs and POs need to be balanced and branched. Nevertheless, our work supports all the other assumptions.

The area and delay in an AQFP technology are commonly evaluated in terms of Josephson junctions (JJs). The area cost of a buffer or splitter cell is 2 JJs, while the area cost of a MAJ3, AND2, and OR2 gates is 6 JJs. Each cell has a JJ depth of one. Hence, we describe the delay of an AQFP circuit in terms of JJ depth.

2.2 Notation

A Boolean network N is modeled as a direct acyclic graph defined by the pair (V, E) where V represents the set of nodes and E represents the set of directed edges. In this paper, we use Boolean network and circuit interchangeably.

For any node v, the *fanins* of v, denoted as $FI(v)$, is a set of nodes driving node v, i.e., nodes that have an outgoing edge towards v. Similarly, the *fanouts* of v, denoted as $FO(v)$, is a set of nodes which are driven by node v, i.e., nodes that have an incoming edge from v. The set of primary inputs (PIs) I is a subset of nodes without fanin in the network. The set of primary outputs (POs) O is a subset of nodes without fanout in the network. The set of logic gates G is a subset of nodes from a predefined gate library. Each node in V is either in I, O, or G. In this paper, each gate in an AQFP-compatible network is either an AND2, OR2, or MAJ3 with optional input negations.

A mapped network N' extends the Boolean network N with a gate element *splitter* and a gate element *buffer*. A splitter is a gate with a fanin size of 1 and fanout size of s_b where s_b is the splitting capacity. A buffer is a special case of splitter with a fanout size of 1. A *splitter tree* of a node v, denoted by $FOT(v)$, is a set of splitters and buffers reachable from v to any other node in $FO(v)$. Figure 2 shows a splitter tree originated from a node n. A splitter tree is said to be *irredundant* if each B/S element has at least one fanout and there is not a pair $(s_1, s_2) \in FOT(v)$ such that their incoming edges are connected to the same node and they both have fanout size smaller than s_b [11].

A *schedule* of a network $d : V \to \mathbb{N}_0$ annotates a level as a non-negative integer for each node in the network. The depth of a network $d(N)$ is defined as $d(N) = \max_{o \in O} d(o)$. A schedule of the network is *valid* if and only if a mapping function $f : (N, d) \to N'$ exists such that buffers and splitters can be inserted respecting the path-balancing and fanout-branching constraints. A possible mapping function that runs in linear time has been presented in [11].

2.3 Minimum Register Retiming

Minimum register retiming is the problem of relocating the registers in a circuit in order to minimize their number while preserving the functionality [12]. The repositioning is captured by the integer-valued *retiming lag* function $r(v) : V \to \mathbb{Z}$ that describes the number of registers moved backward over node v, from its fanout to the fanin. Given a circuit where $w(e)$ is the initial number of registers on an edge e, the minimum register retiming problem can be formulated as a linear problem as follows:

$$\min \sum_{\forall e=(u,v) \in E} r(v) - r(u) \qquad s.t. \qquad (1)$$

$$r(u) - r(v) \leq w(e) \qquad \forall e = (u, v) \qquad (2)$$

This linear problem is dual to the minimum-cost flow problem [12] for which many algorithms exist.

3 DEPTH-OPTIMAL BUFFER AND SPLITTER INSERTION

In this section, we present a depth-optimal B/S insertion policy formulating the AQFP mapping problem as a scheduling problem. The depth-optimal B/S insertion problem for a Boolean network N consists of finding splitter tree configurations such that the resulting circuit depth is minimal. Not well composed splitter trees could produce longer critical paths that increase the delay and also the area. In fact, the area of an AQFP circuit is related to its depth because of path-balancing. Intuitively, longer critical paths (considering B/S elements) would require to insert more buffers due to longer paths to balance. Thus, finding an optimal-depth AQFP circuit is crucial to improve power, performance, and area (PPA).

To introduce our idea, we first define our B/S insertion policy to find a level assignment $d(n)$ for a node n given a partial schedule d.

A method based on scheduling has been first proposed by Lee et al. [11] presenting an algorithm for irredundant B/S insertion on a single node given the relative depths of the fanout. Their approach applied to a pre-scheduled network finds a valid AQFP circuit. In this paper, we also propose an algorithm based on scheduling but to generate a minimum-height irredundant splitter tree for a node given the level assignment of its fanout. The algorithm is shown in Algorithm 1.

Algorithm 1 fits a minimum-height splitter tree for a node n given the level assignment of its fanout in a partial schedule d and assigns node n to a level in the schedule. First, the pairs composed of nodes and scheduled levels in the fanout of n are saved in L (line 2). Then, the highest level in L is stored in l_{last} (line 3). In the main loop (lines 5 to 9), for each node-level pair in L in descending order of level, *edges* is updated. Variable *edges* counts the number of nodes including B/S elements needed to be connected at level l (line 7). Every time a node at a lower level is encountered, the number of edges is reduced by fitting a balanced splitter tree of depth $l_{last} - l$ (line 6) and l_{last} is updated. Once that every fanout of n has been processed, the algorithm finds the highest level where n can be scheduled (lines 11 to 14). This level corresponds to the first position where *edges* is equal to one. Figure 2 illustrates an example for a node n with four fanout nodes to better understand Algorithm 1. Nodes are represented by circles with an id. The four fanout nodes are assigned to levels 8 (nodes 1, 2, 3) and 7 (node 4) in the partial schedule. The splitting capacity is $s_b = 2$. Variable $edges_{(v,l)}$ indicates the value of *edges* when node v (id) at level l is considered in the main loop (lines 5 to 9). First, $edges_{(1,8)} = 1$, $edges_{(2,8)} = 2$ (not in the figure) and $edges_{(3,8)} = 3$ are computed. The algorithm is counting the nodes at level $l = 8$. When a node at a lower level is considered, in this case node $(4, 7)$, the number of needed B/S elements at level 7 to drive the nodes at level 8 is computed by $\lceil 3/2^{8-7} \rceil$ and *edges* is updated. Finally, from the second loop (lines 11 to 14), *edges* is updated two times before it reaches value 1 at level 5. Gate n is then inserted as soon as $edges = 1$ at level 4. Figure 2 also shows the computed minimum-height splitter tree where B/S elements are squares. It follows that:

Lemma 1. *Algorithm 1 finds an irredundant minimum-height splitter tree for a node given a schedule of the fanout.*

Proof. Algorithm 1 has two parts. In the first one, it counts the number of nodes at level $l = l_{last}$ when the level is stable.

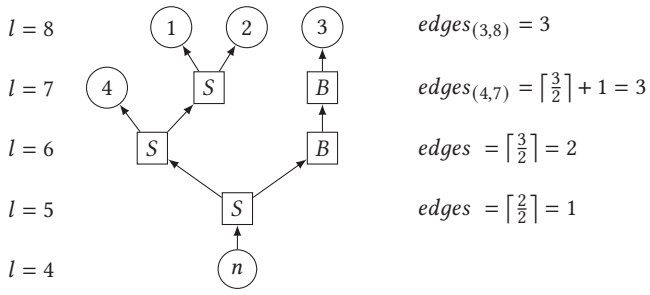


Figure 2: Example to illustrate Algorithm 1

Algorithm 1: Single node scheduling

```
1  Input : node n, partial scheduling d
2  L ← { (v, d(v)) | v ∈ FO(n) };
3  l_last ← max_{(v,l) ∈ L} l;
4  edges ← 0;
5  foreach (v, l) ∈ L in descending order of l do
6  |   splitters ← ⌈ edges / s_b^(l_last − l) ⌉;
7  |   edges ← splitters + 1;
8  |   l_last ← l;
9  end
10 l_last ← l_last − 1;
11 while edges ≠ 1 do
12 |   edges ← ⌈ edges / s_b ⌉;
13 |   l_last ← l_last − 1;
14 end
15 d(n) ← l_last;
```

When a new node at a lower level is considered, the minimum number of B/S elements needed at the lower level is computed at line 6. This number is minimum since $s_b^{(l_{last}-l)}$ computes the maximum number of nodes a single splitter tree can drive from level l to level l_{last}. The ceiling of the ratio between the number of nodes at level l_{last} and the maximum splitter tree capacity tell us the minimum number of splitters needed at level l'. In the second part, once all the fanout has been processed, the algorithm fits the minimum number of B/S elements level by level until $edges = 1$. Since Algorithm 1 inserts the minimum number of B/S elements per level, it also generates an irredundant and minimum-height splitter tree. ■

Since Algorithm 1 finds a level assignment that satisfies the path-balancing and fanout-branching constraints for a node based on its fanout, it can be used as a scheduling function in an ALAP schedule. An ALAP scheduling algorithm schedules all the POs of a network to a bound λ and applies a scheduling function for each node in reverse topological order.

Proposition 2. *Let λ be a sufficiently large bound to obtain a legal ALAP schedule. Let $l_{min} = \min_{i \in I} d(i)$ be the lowest level of a scheduled node in the network. Then, the depth of the AQFP circuit is $\lambda - l_{min}$.*

Proof. By definition, the ALAP scheduling bound λ is relaxed such that $\lambda \geq d(N)$. Since there are not scheduled nodes from level zero to $l_{min} - 1$, we could reduce λ by l_{min} to removed unassigned levels. ■

From Proposition 2, it follows that the scheduling objective for minimizing the depth is to maximize the level of PIs in the ALAP schedule since $d(N) = \lambda - l_{min}$ where λ is constant.

Theorem 3. *The global optimal-depth buffer and splitter insertion problem is solvable using dynamic programming by inserting minimum-height splitter trees in reverse topological order.*

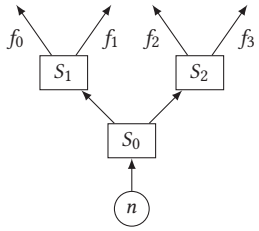


Figure 3: Example of splitter selection due to fanout limitations.

Proof. An ALAP scheduling algorithm assigns all the POs to the bound λ that is the maximum possible level. Given a node n and its already scheduled fanout, a minimum-height splitter tree can be inserted using Algorithm 1. Thus, n is scheduled to its maximum level. By induction, the algorithm maximizes the level of each node and thus the resulting AQFP circuit is depth-optimal. ■

Using Lemma 1, Proposition 2, and Theorem 3, we proved that Algorithm 1 in a ALAP scheduling obtains depth-optimality.

Our AQFP ALAP mapping method firsts computes a higher bound λ for the ALAP scheduling, for instance by assuming each node would need a balanced splitter tree to drive its fanout. Then, every PO is scheduled to the bound. Next, each node is scheduled in reverse topological order using Algorithm 1. The reverse topological ordering guarantees that each fanout has been already visited. Finally, the PIs are scheduled to level zero and the rest of the nodes are lowered by the minimum assigned level. The depth-optimal ALAP scheduling algorithm runs in linear time to the number of nodes in the network. The presented method can be used to generate a scheduling assignment that maps into a valid AQFP circuit. Then, a mapping function can be used to extract the mapped circuit in linear time [11].

Our approach can support different technology assumptions such as balanced or not-balanced POs and PIs. In case of not branched PIs, PIs are scheduled without running Algorithm 1 by placing them just below the internal node with the lowest level. For other assumptions, the ALAP scheduling method remains unchanged. The mapping function, instead, does not insert buffers or splitters for PIs or POs according to the specifications.

4 RETIMING-BASED BUFFER AND SPLITTER OPTIMIZATION

In this section, we present an algorithm based on minimum register retiming to globally minimize the numbers of buffers and splitters in an AQFP network. Previous work applied a retiming-like optimization to AQFP logic [3, 5]. However, their approach does not perform global retiming but moves buffers locally from the output of splitters to the input. This optimization is already included in our depth-optimal algorithm since our approach creates irredundant splitter trees.

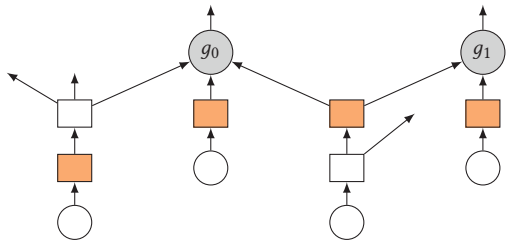
Buffers and splitters are used in AQFP circuits to meet the circuit constraints of path-balancing and fanout-branching. Minimizing the number of B/S elements consists of maximizing the sharing of B/S elements. Without accounting for fanout-branching, e.g., assuming that buffers have an infinite driving capability, the minimum number of buffers is achievable in polynomial time using a minimum register retiming algorithm considering each buffer as a register. Retiming preserves the path-balancing constraint since each path traverses the same number of registers before and after retiming. Existing work applied this idea to *Rapid Single-Flux Quantum* (RSFQ) superconducting logic [10]. However, when considering fanout limitations, splitters cannot be relocated freely since their movement is conditional on respecting the fanout constraints. Hence, retiming can be only used as a heuristic for B/S optimization.

Algorithm 2: B/S retiming

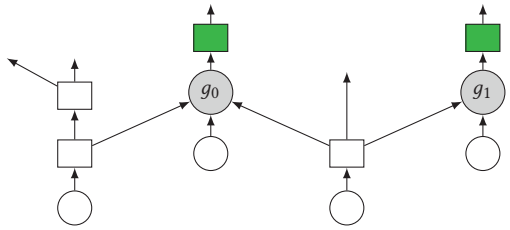
```
1  Input : AQFP circuit M, technology assumptions ps
2  while improvement do
3      select_retimeable_elements(M, ps);
4      set up retiming direction to forward;
5      maximum_flow(M);
6      get_minimum_cut(M);
7      move_retimed_elements(M);
8  end
9  while improvement do
10     select_retimeable_elements(M, ps);
11     set up retiming direction to backwards;
12     maximum_flow(M);
13     get_minimum_cut(M);
14     move_retimed_elements(M);
15 end
16 reconstruct_splitter_trees(M, ps);
17 return;
```

Figure 3 shows a splitter tree where a gate is represented by a circle and a B/S element is represented by a rectangle. Let us suppose that the maximum splitting capacity is $s_b = 3$. In this example, splitter S_0 cannot be selected for retiming since its movement would increase the fanout of n to 2 not satisfying the fanout constraint of a gate. Splitters S_1 and S_2 are only mutually selectable for retiming since the movement of both of them would increase the fanout of S_0 to 4 not satisfying the fanout constraint s_b. Note that the latter case may happen only in redundant splitter trees, as in Figure 3. Moreover, retiming optimization of the splitters S_1 and S_2 may depend on different fanout groupings such as $FO(S_1) = \{f_0, f_2\},\ FO(S_2) = \{f_1, f_3\}$ instead of the current $FO(S_1) = \{f_0, f_1\},\ FO(S_2) = \{f_2, f_3\}$. Some groupings may unlock a B/S minimization that is not achievable in others.

The B/S retiming algorithm is shown in Algorithm 2. The algorithm receives a valid AQFP circuit and the technologies assumptions as input. Since the retiming problem is solved as a maximum flow problem similarly to [9], the flow computation is separated in the forward and backward directions. The algorithm performs two optimization loops in both directions. A loop starts by selecting the elements to be retimed (line 3 or 11). Those are the splitters and buffers that can be relocated without exceeding the driving capacity of the fanin gate or the B/S element. In the case of mutually exclusive selections (e.g., two splitters cannot be retimed at the same time) one is picked using a deterministic random function. Each selected B/S element is a source and a sink of a unitary flow. Next,



(a) Initial configuration



(b) Configuration after retiming

Figure 4: Example of forward retiming of buffers and splitters.

Algorithm 3: Splitters and buffers insertion method

```
1  Input  : MIG network N_mig, technology assumptions ps
2  Output: AQFP circuit M
3  M ← empty network;
4  d_ALAP ← ALAP(N_mig, ps);
5  d_ASAP ← ASAP(N_mig, ps, d_ALAP);
6  if num_bs(d_ALAP) < num_bs(d_ASAP) then
7  |  M ← dump_circuit(N_mig, d_ALAP, ps);
8  else
9  |  M ← dump_circuit(N_mig, d_ASAP, ps);
10 end
11 bs_retiming(M, ps);
12 while improvement do
13 |  chunk_movement(M, ps);
14 |  bs_retiming(M, ps);
15 |  det_randomize(M);
16 end
17 return M;
```

the algorithm proceeds by selecting the retiming direction, computing the maximum flow, getting the minimum cut, and moving the selected elements to the new position if there is an improvement. Since retiming movements could create redundant splitter trees, the algorithm terminates by reconstructing each splitter tree to be irredundant (line 17).

An example of a forward retiming iteration is depicted in Figure 4. Figure 4a shows an initial configuration of a logic section in an AQFP circuit where circles represent gates, rectangles represent buffers and splitters, and arrows represent connections. In this example, the splitter capacity is $s_b = 3$. The algorithm selects the B/S elements in orange to perform retiming as they satisfy all the conditions (buffers and retimeable splitters). Figure 4b shows the configuration after retiming. Two new buffers are inserted (in green). The B/S elements have been reduced from 6 to 5 while maintaining the same paths lengths.

5 OPTIMIZATION FLOW

In this section, we present a flow for AQFP mapping consisting of B/S insertion followed by B/S optimization. The optimization flow is shown in Algorithm 3.

The mapping algorithm takes a MIG as an input expressing the AQFP circuit functionality. Then, the circuit is scheduled using the depth optimal algorithm in Section 3 (line 4). Then, an alternative scheduling approach based on ASAP is attempted (line 5). This approach takes an ALAP schedule and tries to move the gates as close as possible to the PIs by pushing fanin buffers forward to the output (similarly to forward retiming) and constraining the nodes to be scheduled within the ALAP schedule. In this way, also the ASAP schedule is depth-optimal. Then, an AQFP circuit is generated from the schedule. This step is motivated by different technology assumptions [11], e.g., an ASAP configuration is generally beneficial if POs do not need balancing (B/S elements connected to the POs can be removed). Nevertheless, the initial B/S configuration could affect the area results even after optimization in generic designs. Hence, it is worth to consider multiple starting configurations. After the two schedules are computed, the circuit with fewer B/S elements is selected for generating the AQFP circuit. This choice reduces the run time of the optimization flow since the starting AQFP circuit would contain less B/S elements. Then, the algorithm starts the optimization phase. It first performs an initial B/S retiming. Next, an optimization loop applies one pass of the chunk movement algorithm in [11], B/S retiming, and deterministic randomization. The chunk movement algorithm is used to locally reschedule some nodes up or down to reduce the number of B/S elements. This algorithm differs from B/S retiming since the latter works by globally maximising the sharing of existing buffers and splitters. The deterministic randomization picks a different topological ordering to escape local minima and find different fanout groupings when constructing splitter trees. The loop is iterated until no further improvement or iteration limit is reached. Finally, the mapped and optimized AQFP circuit is returned. This optimization flow preserves the depth optimality. Alternatively to this algorithm, we developed a portfolio approach which consists of carrying the optimization with both the ALAP and ASAP schedules separately and picking the best final result. This latter method offers better quality at the expense of more run time.

6 EXPERIMENTS

The optimization framework has been implemented in C++ 17 in the logic synthesis framework *Mockturtle*[1] [15]. The experiments have been conducted on an Intel i5 quad-core 2GHz on MacOS. All the results were verified to fulfill the path-balancing and fanout-branching assumptions.

6.1 Comparison Against the State of the Art

In this experiment, we compare our optimization flow in Algorithm 3 to the state-of-the-art [8]. The state of the art method

[1]Available at: https://github.com/lsils/mockturtle

consists of a B/S insertion followed by a heuristic optimization that pushes gates up and down to reduce the number of buffers. That approach has local optimality for inserting splitter trees on a single wire, like Algorithm 1, but no global optimality. Moreover, their buffer and splitter insertion approach has a higher computational complexity than the one we propose. To compare the two methods, the benchmarks have been obtained from the authors of the paper[2]. We use the same settings by balancing and branching PIs and POs using a splitting capacity of $s_b = 4$.

Table 1 shows the results of the comparison. For our approach, we include the results right after B/S insertion, indicated with the term "Ins.", and after the optimization flow. Our approach (Algorithm 3) performs significantly better in almost every benchmark reducing the number of B/S elements up to 20% with a 55× speed-up in run time compared to the state-of-the-art approach[3]. Our results improve even more using a portfolio approach[4]. The portfolio approach reduces the total number of B/S elements to 50002 while maintaining a 19× speed-up in run time. Our B/S retiming algorithm is responsible of the 84% of the total area reduction on average. The higher JJ depth of our method in benchmark *c2670* is due to the structural hashing pre-processing. Most of the benchmarks in Table 1 are relatively small. We could expect more significant improvements on larger benchmarks like the second half of the table suggests.

6.2 Results on the EPFL Benchmark suite

In this experiment, we applied our B/S insertion and optimization method (Algorithm 3) to the 10 largest designs in the EPFL benchmark suite[5] [1] to demonstrate its scalability. The baseline has been obtained by mapping the benchmarks into MIGs using the graph mapper in [6] in the delay mode with default settings. In this experiment, PIs and POs are balanced and branched, and the splitting capacity $s_b = 4$. To decrease the run time, we limited the retiming iterations to 250, the size of the chunks to 100, and we executed the main optimization loop only once. Moreover, we limited the execution run time budget to 300 seconds.

Table 2 shows the experimental results after depth-optimal B/S insertion and after carrying the optimization in Algorithm 3. The B/S insertion algorithm scales very well with limited run time for all the benchmarks. The B/S optimization reduces the number of splitters up to 37.83% and the total area up to 24.04%. However, our B/S optimization could have issues at finding a solution in the time budget for large benchmarks with more than 1 million of B/S elements such as *div* or *hyp*. Design partitioning could help at improving the scalability even more.

7 CONCLUSION

In this work, we studied the buffer and splitter insertion problem in AQFP circuits. While existing work focused only on area reduction [5, 8, 11], in this paper we stated the importance of delay reduction to minimize the path-balancing costs and consequently benefit the area. Moreover, previous work constructed the splitter trees separately for each node. Without a global policy, their approach cannot offer global optimality due to the interplay among gates. In contrast, we demonstrated that there exists a linear time algorithm based on ALAP scheduling to insert buffers and splitters such that the resulting AQFP circuit is globally depth-optimal. Next, we presented a novel algorithm to minimize buffers and splitters based on minimum register retiming. Finally, we proposed a B/S insertion and optimization flow based on the algorithms in this paper and the chunk movement algorithm in [11]. Our approach reduces the number of B/S elements up to 20% while guaranteeing optimal depth and providing a speed-up of 55× in run time compared to the state-of-the-art method. Additionally, we showed that our method scales to large benchmarks.

[2]The benchmarks contained redundant gates and inverter cells. Since the Mockturtle framework automatically applies structural hashing to the networks, the starting points of the experiments are slightly different.
[3]Since the tool in [8] is not openly available, it is not possible to run it on larger benchmarks and the run time has been taken from the paper.
[4]Results are not included in the table for space reasons.
[5]Available at: https://github.com/lsils/benchmarks

ACKNOWLEDGMENTS

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

REFERENCES

[1] L. Amarù, Pierre-Emmanuel Gaillardon, and G. D. Micheli. 2015. The EPFL Combinational Benchmark Suite. In *Proc. IWLS*.
[2] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2016. Majority-Inverter Graph: A New Paradigm for Logic Optimization. *IEEE Trans. CAD* 35, 5 (2016).
[3] Christopher L Ayala, Ro Saito, Tomoyuki Tanaka, Olivia Chen, Naoki Takeuchi, Yuxing He, and Nobuyuki Yoshikawa. 2020. A semi-custom design methodology and environment for implementing superconductor adiabatic quantum-flux-parametron microprocessors. *Superconductor Science and Technology* 33, 5 (2020).
[4] Ruizhe Cai, Olivia Chen, Ao Ren, Ning Liu, Caiwen Ding, Nobuyuki Yoshikawa, and Yanzhi Wang. 2019. A Majority Logic Synthesis Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. ACM.
[5] Ruizhe Cai, Olivia Chen, Ao Ren, Ning Liu, Nobuyuki Yoshikawa, and Yanzhi Wang. 2019. A Buffer and Splitter Insertion Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits. In *ICCD*.
[6] Alessandro Tempia Calvino, Heinz Riener, Shubham Rai, Akash Kumar, and Giovanni De Micheli. 2022. A Versatile Mapping Approach for Technology Mapping and Graph Optimization. In *ASP-DAC*.
[7] Olivia Chen, Ruizhe Cai, Yetang Wang, Fei Ke, Taiki Yamae, Ro Saito, Naoki Takeuchi, and Nobuyuki Yoshikawa. 2019. Adiabatic Quantum-Flux-Parametron: Towards Building Extremely Energy-Efficient Circuits and Systems. *Scientific Reports* 9 (2019).
[8] Chao-Yuan Huang, Yi-Chen Chang, Ming-Jer Tsai, and Tsung-Yi Ho. 2021. An Optimal Algorithm for Splitter and Buffer Insertion in Adiabatic Quantum-Flux-Parametron Circuits. In *ICCAD*.
[9] Aaron P. Hurst, Alan Mishchenko, and Robert K. Brayton. 2007. Fast Minimum-Register Retiming via Binary Maximum-Flow. In *Formal Methods in Computer Aided Design (FMCAD'07)*.
[10] Naveen Katam, Alireza Shafaei, and Massoud Pedram. 2017. Design of Complex Rapid Single-Flux-Quantum Cells with Application to Logic Synthesis. In *ISEC*.
[11] Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. 2021. Irredundant Buffer and Splitter Insertion and Scheduling-Based Optimization for AQFP Circuits. In *in Proc. IWLS*. arXiv:2109.00291
[12] Charles E. Leiserson and James B. Saxe. 1991. Retiming Synchronous Circuitry. *Algorithmica* 6, 1–6 (jun 1991), 5–35.
[13] Dewmini Sudara Marakkalage, Heinz Riener, and Giovanni De Micheli. 2021. Optimizing Adiabatic Quantum-Flux-Parametron (AQFP) Circuits using an Exact Database. In *NANOARCH*.
[14] Giulia Meuli, Vinicius Possani, Rajinder Singh, Siang-Yun Lee, Alessandro Tempia Calvino, Dewmini Sudara Marakkalage, Patrick Vuillod, Luca Amaru, Scott Chase, Jamil Kawa, and Giovanni De Micheli. 2022. Majority-based Design Flow for AQFP Superconducting Family. *DATE* (2022), 6.
[15] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, Siang-Yun Lee, Alessandro Tempia Calvino, Dewmini Sudara Marakkalage, and Giovanni De Micheli. 2022. The EPFL Logic Synthesis Libraries. (2022). arXiv:1805.05121v3

Table 1: Evaluation of our approach against the state of the art

Benchmark	Initial circuit		State of the art [8]				Our approach					
	Size	Depth	#B/S	#JJs	JJ Depth	Time (s)	#B/S Ins.	JJ Depth	Time Ins. (s)	#B/S	#JJs	Total Time (s)
adder1	7	4	16	74	8	0.13	16	8	0.00	16	74	0.00
adder8	77	17	371	1204	33	0.16	374	33	0.00	372	1206	0.01
mult8	439	35	1833	6300	70	0.47	1824	70	0.00	1688	6010	0.06
counter16	29	9	82	338	17	0.18	70	17	0.00	65	304	0.00
counter32	82	13	189	912	23	0.18	170	23	0.00	154	800	0.00
counter64	195	17	419	2134	30	0.23	377	30	0.00	347	1864	0.01
counter128	428	22	895	4652	38	0.56	807	38	0.00	747	4062	0.02
c17	6	3	12	60	5	0.15	12	5	0.00	12	60	0.00
c432	121	26	837	2406	37	0.34	862	37	0.00	839	2404	0.02
c499	387	18	1251	4858	30	0.38	1198	29	0.00	1173	4668	0.02
c880	306	27	1723	5296	40	0.45	1691	40	0.00	1511	4858	0.10
c1355	389	18	1216	4784	29	0.40	1206	29	0.00	1184	4702	0.03
c1908	289	21	1505	4810	35	0.35	1318	34	0.00	1236	4206	0.04
c2670	368	21	2055	7392	27	0.71	2080	28	0.00	1932	6072	0.09
c3540	794	32	2395	9610	53	1.15	2678	52	0.00	1972	8708	0.16
c5315	1302	26	6447	20854	41	4.00	7430	40	0.01	5646	19104	0.45
c6288	1870	89	9297	29814	179	5.70	14119	179	0.01	9009	29238	0.23
c7552	1394	33	8342	25140	59	85.27	10149	56	0.01	7505	23374	1.01
sorter32	480	15	480	3840	30	0.35	480	30	0.00	480	3840	0.01
sorter48	880	20	880	7040	35	0.52	960	35	0.00	880	7040	0.02
alu32	1513	100	17178	43574	170	64.68	15207	169	0.01	13837	36752	0.82
Total			57423	185092	989	166.36	61796	982	0.04	50605	169346	3.08

Table 2: Experimental results for AQFP mapping on the EPFL benchmark suite

Benchmark	Baseline		Depth-optimal B/S insertion				B/S optimization				
	Size	Depth	#B/S	#JJs	JJ depth	Time (s)	#B/S	ΔB/S (%)	#JJs	ΔJJs (%)	Time (s)
div	57300	2217	1881255	4106310	4371	0.87	-	-	-	-	>300
hyp	136109	8762	9035578	18887810	17246	2.78	-	-	-	-	>300
log2	24456	200	129547	405830	379	0.10	86705	33.07	320146	21.11	64.18
multiplier	19710	133	102005	322270	264	0.08	63414	37.83	245088	23.95	43.50
sin	4303	110	18905	63628	188	0.01	14886	21.26	55590	12.63	4.12
sqrt	23238	3366	1791005	3721438	6628	0.49	1343705	24.97	2826838	24.04	284.10
square	12180	126	89516	252112	251	0.03	63630	28.92	200340	20.54	18.30
arbiter	7000	59	27566	97132	63	0.01	25721	6.69	93442	3.80	1.28
mem_ctrl	42758	73	216927	690402	114	0.27	215202	0.80	686952	0.50	10.55
voter	7860	47	19263	85686	86	0.01	15736	18.31	78632	8.23	0.92

[16] Naoki Takeuchi, Mai Nozoe, Yuxing He, and Nobuyuki Yoshikawa. 2019. Low-latency adiabatic superconductor logic using delay-line clocking. *Applied Physics Letters* 115, 7 (2019).

[17] Naoki Takeuchi, Dan Ozawa, Yuki Yamanashi, and Nobuyuki Yoshikawa. 2013. An adiabatic quantum flux parametron as an ultra-low-power logic device. *Superconductor Science and Technology* 26, 3 (2013).

[18] Naoki Takeuchi, Yuki Yamanashi, and Nobuyuki Yoshikawa. 2015. Adiabatic quantum-flux-parametron cell library adopting minimalist design. *Journal of Applied Physics* 117, 17 (2015).

[19] Eleonora Testa, Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. 2021. Algebraic and Boolean Optimization Methods for AQFP Superconducting Circuits. In *Proc. ASP-DAC*.

[20] Stephen R. Whiteley and Jamil Kawa. 2019. Progress Toward VLSI-Capable EDA Tools for Superconductive Digital Electronics. In *2019 IEEE International Superconductive Electronics Conference (ISEC)*. 1–3.