Beyond Local Optimality of Buffer and Splitter Insertion for AQFP Circuits

Siang-Yun Lee
EPF Lausanne
Switzerland

Heinz Riener
Cadence Design Systems
Germany

Giovanni De Micheli
EPF Lausanne
Switzerland

Abstract

Adiabatic quantum-flux parametron (AQFP) is an energy-efficient superconducting technology. Buffer and splitter (B/S) cells must be inserted to an AQFP circuit to meet the technology-imposed constraints on path balancing and fanout branching. These cells account for a significant amount of the circuit's area and delay. In this paper, we identify that B/S insertion is a scheduling problem, and propose (a) a linear-time algorithm for locally optimal B/S insertion subject to a given schedule; (b) an SMT formulation to find the global optimum; and (c) an efficient heuristic for global B/S optimization. Experimental results show a reduction of 4% on the B/S cost and 124× speed-up compared to the state-of-the-art algorithm, and capability to scale to a magnitude larger benchmarks.

ACM Reference Format:
Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. 2022. Beyond Local Optimality of Buffer and Splitter Insertion for AQFP Circuits. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530661

1 Introduction

Superconducting electronics is an emerging domain arising from the demand for high performance computation. It gains increasing interests from both academia and industry because the high switching speed of Josephson junctions empowers superconductor-based circuits to push their clock frequencies beyond the limit of CMOS-based circuits. [15] Among various superconducting logic families, the *adiabatic quantum-flux parametron* (AQFP) [19] is a technology featuring, in addition to high computation speed, zero static energy consumption and very small switching energy dissipation. Two of the challenges in AQFP circuit design come from the *path-balancing* and *fanout-branching* requirements which are not needed in traditional CMOS logic circuits.

Path-balancing: The AQFP gates are AC-biased. Each AQFP gate receives an alternating excitation current to periodically release its output signal and reset its state. All AQFP clocking schemes [17, 18] require that the input signals of a logic gate be released at the previous clocking phase. In other words, all data paths must be of the same length. Whereas shortening longer paths is not always possible, *buffers* need to be inserted to delay shorter paths.

Fanout-branching: In the AQFP technology, logic 0 and 1 are represented with different current directions. As the output current of an AQFP gate is limited, it has to be amplified by a *splitter* before branching into multiple fanouts. AQFP splitters are also clocked.

As the research at the physical level rapidly develops and the fabrication capability grows for larger and more complex circuits, design automation tools specialized to consider buffer and splitter (B/S) costs are in need. In AQFP circuits, B/S often make up for about half of the area and delay costs, even after optimization [2, 5, 6, 8, 10, 22]. The path-balancing problem has also been researched for another superconducting technology, the *rapid single-flux quantum* (RSFQ), where efficient algorithms have been developed [12, 13]. However, the splitters in RSFQ are not clocked and not involved in path balancing, whereas splitters in AQFP are. Having to consider the path-balancing and fanout-branching constraints simultaneously makes the problem much harder in AQFP. Whereas some previous works try to consider B/S cost in logic optimization [10, 22], we focus on the problem of B/S insertion without logic transformation, such that (1) our approach can be applied after any existing logic synthesis algorithm; (2) the effect of logic synthesis and B/S insertion can be separated and the impact of the constraints can be studied; and (3) the problem is simplified and its optimality can be analyzed. While locally-optimal B/S insertion for a single net has been tackled in [8], a roadmap towards global optimality is still lacking.

In this paper, we observe that B/S insertion for AQFP is a *scheduling* [9] problem. A linear-time algorithm relating a given schedule (depth assignment to logic gates) to the minimum possible number of B/S needed to legalize the circuit is proposed. On the one hand, global B/S minimization is formulated as an optimization modulo linear integer arithmetic problem, which can be solved by a *satisfiability modulo theory* (SMT) [3] solver. On the other hand, an efficient heuristic based on moving chunks (groups of gates) is developed. Experimental results show that, when comparing to the state-of-the-art B/S insertion algorithm [8], our heuristic not only reduces the number of inserted B/S by 4% on average, but also provides a drastic 124× speed up. Moreover, by decoupling scheduling and B/S insertion, our approach scales a magnitude higher than [8] in circuit size, depth, and maximum fanout size.

2 Background

2.1 Adiabatic Quantum-Flux Parametron

The *adiabatic quantum-flux parametron* (AQFP) is an emerging superconducting technology shown to achieve promising energy efficiency. [19] The basic circuit components in AQFP are the buffer cell and the branch cell. A majority-3 logic gate can be constructed by combining three buffer cells with a 3-to-1 branch cell, from which other logic gates, such as the AND gate and the OR gate, can be built with constant cells (biased buffer cells). Input negation of logic gates is realized using a negative mutual inductance and is of no extra cost. [20] The commonly-used cost metric of AQFP circuits is the *Josephson junction* (JJ) count. A buffer costs 2 JJs, a branch cell is of no JJ cost, and a logic gate based on majority-3 costs 6 JJs. [20]

Logic gates in an AQFP circuit need to be activated and deactivated periodically by an excitation current. [17] In other words, every gate in an AQFP circuit is clocked, and all input signals have to arrive at the same clock cycle. To ensure this, shorter data paths need to be delayed with clocked buffers. Moreover, the output signal of AQFP logic gates cannot be directly branched to feed into

DAC '22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530661

multiple fanouts. Instead, splitters are placed at the output of multi-fanout gates to amplify the output current. A splitter cell is composed of a buffer cell and a 1-to-n branch cell (usually, $2 \leq n \leq 4$) and is also clocked. As the cost of splitters comes mostly from the buffer cells, in the remainder of this paper, we do not distinguish buffers from splitters and will model them with the same abstract data structure.

2.2 Terminology

A *(logic) network* is a directed acyclic graph defined by a pair (V, E) of a set V of nodes and a set E of directed edges. The node set $V = I \cup O \cup G$ is disjointly composed of a set I of *primary inputs* (PIs), a set O of *primary outputs* (POs), and a set G of *(logic) gates* chosen from a library. In this paper, we assume that an AQFP-compatible gate library (e.g., composed of AND2, OR2, MAJ3 with optional input negation) is used. Each PI has in-degree 0 and unbounded out-degree, whereas each PO has in-degree 1 and out-degree 0. The out-degree of each gate is unbounded and the in-degree is a fixed number depending on the type of the gate. For any gate $g \in G$, the *fanins* of g, denoted as $\mathrm{FI}(g)$, is the set of gates and PIs connected to g with an incoming edge. Similarly, the *fanouts* of g, denoted as $\mathrm{FO}(g)$, is the set of gates and POs connected to g with an outgoing edge. Fanouts are similarly defined for PIs.

A *mapped network* N' is a network whose node set V' is extended with a set B of *buffers*. A buffer is a node with in-degree 1. In a mapped network, the definition of the fanouts of a gate is modified by ignoring any intermediate buffers, i.e., a path from a gate g to one of its fanouts $g_o \in \mathrm{FO}(g) \subset (G \cup O)$ may include any number of buffers in B, but never another gate. The definition of fanins is modified similarly. The *fanout tree* of a gate g, denoted by $\mathrm{FOT}(g)$, is the set of buffers between g and any gate or PO in $\mathrm{FO}(g)$. Fanout trees are also defined for PIs.

A *schedule* of a network is a function $d : V \longrightarrow \mathbb{Z}_{\geq 0}$ that assigns a non-negative integer $d(n)$ to each node $n \in V$, called the *depth* of n. The depth of a network $N = (V = I \cup O \cup G, E)$ is defined as $d(N) = \max_{o \in O} d(o)$. The *relative depth* between a PI or a gate $n \in (I \cup G)$ and one of its fanouts $n_o \in \mathrm{FO}(n) \subset (G \cup O)$ is denoted and defined as $rd(n, n_o) = d(n_o) - d(n)$.

2.3 Technology Assumptions

To fulfill the needs in the AQFP technology for fanout-branching and path-balancing, we define the following two properties subject to the *splitting capacities* $s_i = 1, s_g = 1$ and $s_b \geq 1$ of PIs, gates and buffers, respectively. Given a mapped network $N' = (V' = I \cup O \cup G \cup B, E')$ and a schedule d,

1. N' is *path-balanced* if
$$\forall n_1, n_2 \in V' : (n_1, n_2) \in E' \Rightarrow d(n_1) = d(n_2) - 1, \quad (1)$$
$$\forall i \in I : d(i) = 0, \text{ and} \quad (2)$$
$$\forall o \in O : d(o) = d(N'). \quad (3)$$
2. N' is *properly-branched* if every PI has an out-degree no larger than $s_i = 1$, every gate has an out-degree no larger than $s_g = 1$, and every buffer has an out-degree no larger than s_b.

A schedule d for an (unmapped) network N is said to be *legal* if a path-balanced and properly-branched mapped network N' can be extended from N and d.

Logic networks defined in Section 2.2 model the combinational parts of digital circuits. In practice, PIs of a network are usually provided by the register outputs of the previous sequential stage and POs are connected to the register inputs of the next stage. As different implementations of AQFP registers are still rapidly being developed [14], different assumptions on whether PIs and POs need to be balanced or branched may arise.

Path-balancing of PIs and POs. It is possible to design registers that can hold and output their values at every clock cycle [14]. In this case, the PI nodes in our model can be placed at any depth, i.e., condition 2 is removed. Similarly, if the PI values are always available and stable until the next register update, shorter paths stabilize to the same result in the later cycles when longer paths are still computing. In this case, shorter paths do not have to be aligned with the longest path (the critical path) [14]. In other words, the POs in our model are no longer limited to be placed at the same depth, i.e., condition 3 is removed.

Branching of PIs. When a register drives multiple outputs, we may or may not need to insert splitters to ensure a large-enough current, depending on the physical implementation of the register. If the registers are capable of producing large current, s_i can be set to infinity. Otherwise, it is also possible to duplicate the frequently-used PIs in the register file to avoid deep splitter trees, or to design special large-capacity buffers having a higher s_b value and use them for PIs with many fanouts.

Branching and inversion of POs. If a gate output feeds into multiple registers, then splitters are always needed. If the negated output of a majority gate is required by the next sequential stage, we can push the output inversion to the gate's inputs because the majority function is self-dual [11] and input negation is for free in AQFP. However, if a gate output is needed by the next stage in both negated and non-negated forms, then we not only need a splitter, but also an additional NOT gate made of an input-negated buffer.

2.4 Problem Formulation

In this paper, we focus on the problem of AQFP B/S insertion after logic synthesis without changing the structure of the original network, formulated as follows:

Given a network $N = (V = I \cup O \cup G, E)$ and the value of the parameter s_b, find a mapped network $N' = (V' = I \cup O \cup G \cup B, E')$, such that:

1. N' is path-balanced and properly-branched.
2. For all gates $g \in G$, $\mathrm{FO}(g)$ and $\mathrm{FI}(g)$ remain the same in N' as in N.
3. $|V'|$ is minimized. Since $V' = V \cup B$, it is equivalent to $|B|$ being minimized.

We call such N' a *minimum* mapped network for N.

3 Locally-Optimal Buffer Insertion

In this section, we will explain how the problem formulated in Section 2.4 can be approached, starting from the following observation.

Claim 1. *Given an unmapped network, finding a minimum mapped network is equivalent to finding an optimum, legal schedule.*

To show why Claim 1 is true, we will first introduce the notion of *irredundant* mapped network in Definition 2 and formulate Proposition 3 to show that the buffer set in an irredundant mapped network can be decomposed into fanout trees of each gate. Then, we will present Algorithm 1 to show how the irredundant fanout tree of a gate g can be constructed given the relative depths of its fanouts. Thus, once a schedule is given, the locally optimal fanout trees are determined, i.e., the size of the mapped network (Claim 1).

Definition 2. *A mapped network is said to be* irredundant *if the following two conditions hold.*

1. *There is no dangling buffer, i.e., every buffer has at least one outgoing edge.*

2. *There does not exist any pair of two buffers whose incoming edges are connected from the same node and both of them have out-degrees smaller than s_b.*

Proposition 3. *In any irredundant mapped network with PI set I, gate set G, and buffer set B,*

$$B = \bigcup_{g \in G} FOT(g) \cup \bigcup_{i \in I} FOT(i).$$

Proof. By definition, a buffer has exactly one incoming edge. The adjacent node connected to a buffer with its incoming edge is either another buffer in B, a gate in G, or an PI in I because POs have no outgoing edge. Going from a buffer b in the opposite direction of the edges and continue tracing until a gate g or a PI i is met, we have $b \in \mathrm{FOT}(g)$ (or $b \in \mathrm{FOT}(i)$) because there is no dangling buffer tree (rule 1 for irredundant networks). Hence, for each buffer $b \in B$, there is either a gate $g \in G$ such that $b \in \mathrm{FOT}(g)$, or there is a PI $i \in I$ such that $b \in \mathrm{FOT}(i)$. Moreover, this gate or PI is unique for each b. For each gate $g \in G$ and for each PI $i \in I$, $\mathrm{FOT}(g) \subseteq B$ and $\mathrm{FOT}(i) \subseteq B$ by definition. Thus, the set of non-empty fanout trees is a partition of B. ■

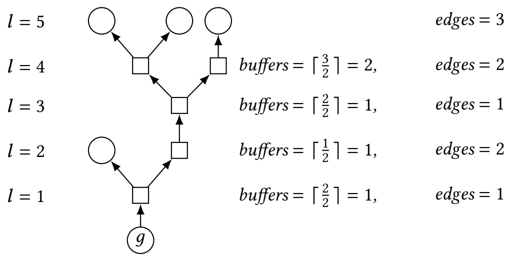
```
**Input:** A gate $g$ and a schedule $d$
**Output:** The size $|\mathrm{FOT}(g)|$ of the fanout tree of $g$
1 $l_{\max} \leftarrow \max_{g_o \in FO(g)} rd(g, g_o)$
2 $count \leftarrow 0$
3 $edges \leftarrow |\{g_o \in FO(g) : rd(g, g_o) = l_{\max}\}|$
4 for $l = l_{\max} - 1$ downto $1$ do
5     $buffers \leftarrow \lceil \frac{edges}{s_b} \rceil$
6     $count \leftarrow count + buffers$
7     $edges \leftarrow buffers + |\{g_o \in FO(g) : rd(g, g_o) = l\}|$
8 assert $edges = 1$
9 return $count$
```

Algorithm 1: Irredundant fanout tree construction given relative depths of fanouts.

For any gate g, given relative depths $rd(g, g_o)$ of its fanouts $g_o \in \mathrm{FO}(g)$, the size of its fanout tree $|FOT(g)|$ can be computed with Algorithm 1. The algorithm iterates over all possible values of relative depth (variable l) and counts the number of buffers (variable *buffers*) needed at each l. The total number of buffers is accumulated in variable *count* (line 6). Variable *edges* keeps the number of edges ending in some node of relative depth l, which is the number of buffers and fanouts at l (line 7). Then, the number of buffers needed at $l - 1$ is computed from the number of edges starting at $l - 1$ (i.e., the number of edges ending at l), divided by the splitting capacity s_b and rounded up (line 5). This algorithm works also for constructing the fanout tree of a given PI. Figure 1 illustrates an example execution of Algorithm 1, where circles are gates and squares are buffers, and $s_b = 2$. The concerned gate g has one fanout of relative depth 2 and three fanouts of relative depth 5. The total number of buffers in the fanout tree is 5.

Algorithm 1 runs in linear time with respect to $|\mathrm{FO}(g)|$. The constructed fanout tree is irredundant because only the minimum number of buffers is inserted at each relative depth l based on the number of outgoing edges needed. The retiming optimization proposed in [5], which pushes buffers from the outputs of a splitter to its input, is subsumed in the construction of irredundant fanout trees.

Algorithm 1 also verifies whether it is possible to build a properly-branched network with the given schedule. In line 8, the assertion makes sure that the gate g has only one outgoing edge. Running the algorithm for all PIs and gates in a depth-assigned network, by Proposition 3, an irredundant mapped network is derived. The mapped network is properly-branched if the assertion in line 8 never fails. It is also path-balanced as each node is connected to a node at relative depth 1. An irredundant network is locally optimal with respect to the given schedule. Thus, we consider only irredundant networks in the remainder of this paper.



$l = 5$
$edges = 3$
$l = 4$
$buffers = \lceil \frac{3}{2} \rceil = 2,$
$edges = 2$
$l = 3$
$buffers = \lceil \frac{2}{2} \rceil = 1,$
$edges = 1$
$l = 2$
$buffers = \lceil \frac{1}{2} \rceil = 1,$
$edges = 2$
$l = 1$
$buffers = \lceil \frac{2}{2} \rceil = 1,$
$edges = 1$
g

Figure 1. Example sub-network to illustrate Algorithm 1.

4 Solving for Global Optimum

In this section, we formulate the global B/S insertion problem as a *satisfiability modulo theory* (SMT) [3] problem using linear integer arithmetic as the underlying theory. The primary variables of the instance are integers corresponding to the depth of each gate. Auxiliary variables are used to compute the total number of B/S using Algorithm 1 and Proposition 3. Four types of constraints are encoded:

1. *Bounds:* An upper bound on the network depth is assumed. Lower and upper bounds on the possible depths of each gate can be obtained using *as-soon-as-possible scheduling* (ASAP) and *as-late-as-possible scheduling* (ALAP), respectively.
2. *Sequencing:* The directed edges are encoded by asserting $\forall g \in G, \forall g_o \in \mathrm{FO}(g) : d(g) < d(g_o)$.
3. *B/S counting:* The number of buffers at the fanout of each gate can be counted by unrolling the for-loop in Algorithm 1. The maximum possible relative depth is used as $l_{\max}$, and lines 5 and 7 are encoded $l_{\max} - 1$ times using $2(l_{\max} - 1)$ auxiliary variables. Line 5 is encoded by
$$buffers = \lceil \frac{edges}{s_b} \rceil \iff s_b(buffers - 1) < edges \le s_b \cdot buffers,$$
where s_b is a constant. Line 7 is encoded with the help of the *if-then-else* (ITE) operator. Finally, all the *buffers* variables are summed up.
4. *Legality:* The legality of d is ensured by assuming the assertion in line 8 of Algorithm 1.

To find the global minimum, the satisfiability problem is extended to an optimization problem, either by using an optimization modulo theory solver [4], or by imposing an upper bound on the B/S count and iteratively decreasing the bound until the problem becomes UNSAT. The problem has an exponential search space and optimization modulo theory is NP-hard, thus this formulation may be only practical for small networks. Nevertheless, it provides the possibility to understand how good existing and future-developed heuristics are, and can possibly be used to generate databases of small optimum circuits.

5 Heuristic Global Optimization

Following Claim 1, in this section, we attempt to find a good schedule to minimize $|B|$ heuristically. During the entire process, we always ensure that the network is legal and count the buffers irredundantly using Algorithm 1.

First, an initial schedule can be obtained by ASAP. To ensure that the network can be path-balanced and properly-branched after mapping, enough depths for a balanced fanout tree are reserved at the output of each multi-fanout gate, which is calculated as $\lceil \log(|\text{FO}(g)|)/\log(s_b) \rceil$. Then, optionally, ALAP can be applied similarly using an upper bound $d(N)$ obtained by ASAP. Depending on the technology assumptions made, ASAP and ALAP can lead to big differences in buffer count. For example, when PIs need to be balanced but POs need not, ASAP usually lead to better results, whereas in the opposite case ALAP often perform better.

Next, we try to move gates up or down to reduce the total number of buffers. *Moving* a gate g up (down) by l levels means that $d(g)$ is increased (resp. decreased) by l while the depths of the other gates remain the same. A movement is *legal* if the network remains legal after the movement. For example, if a gate g has a fanout g_o of relative depth $rd(g, g_o) = 1$, then moving g up alone is not legal. Similarly, if a gate g has more than one fanout, then moving any of its fanouts down to $d(g) + 1$ is not legal because there must be a splitter occupying the only outgoing edge of g at $d(g) + 1$.

We observe that sometimes it is impossible to legally move a single gate, or that moving it alone does not reduce the total buffer count. However, rearranging some neighboring gates altogether might eventually lead to further reduction. Thus, we propose to identify groups of connected gates and move them together as *chunks*.

A pair of gates $(g, g_o) : g_o \in \text{FO}(g)$ are said to be *close* if either one of the following conditions holds:

1. $rd(g, g_o) = 1$, implying that g_o is the only fanout of g.
2. $|\text{FO}(g)| > 1$ and $rd(g, g_o) = 2$.

If a gate g and its fanout g_o are not close, then there is *flexibility* at the output of g and at the input of g_o. A *chunk* is a set C of closely-connected gates. Seen as a group altogether, it has multiple incoming and outgoing edges, called the *input interfaces* (IIs) and *output interfaces* (OIs), respectively. An interface is a pair (g_c, g_e) of a gate in the chunk ($g_c \in C$) and an external gate ($g_e \notin C$), where either $g_e \in \text{FI}(g_c)$ (for an II) or $g_e \in \text{FO}(g_c)$ (for an OI).

```
   Input: An initial gate $g_0$
   Output: A chunk $C$ and its interfaces $T$
1  $C \leftarrow \{g_0\}$
2  $F \leftarrow \{(g_0, g) : g \in \text{FI}(g_0) \cup \text{FO}(g_0)\}$
3  $T \leftarrow \emptyset$
4  while $F \neq \emptyset$ do
5      $(g_c, g_e) \leftarrow \text{pop}(F)$
6      if $g_e \in C$ then continue
7      if $g_c$ and $g_e$ are close then
8          $C \leftarrow C \cup g_e$
9          $F \leftarrow F \cup \{(g_e, g) : g \in \text{FI}(g_e) \cup \text{FO}(g_e)\}$
10     else
11         $T \leftarrow T \cup \{(g_c, g_e)\}$
12 return $C, T$
```

Algorithm 2: Chunk construction.

Algorithm 2 illustrates how a chunk can be constructed. Starting from an initial gate g_0, a chunk is formed by exploring towards its fanins and fanouts and adding gates into the chunk if they are close (line 8), or recording an input or output interface if there is flexibility (line 11). When a new gate is added into the chunk, its fanins and fanouts are also explored (line 9).

A chunk constructed with Algorithm 2 has flexibilities at all of its interfaces. Thus, even though individual gates in the chunk cannot be moved legally, a chunk may be moved as a whole. Figure 2 shows an example chunk. Starting from the initial gate g_0, closely-connected gates g_1, g_2, g_3, g_4 are added into the chunk in the respective order. The gate g_1, for example, cannot be moved up nor down legally without moving other gates at the same time. Also, although the gate g_0 can be legally moved down, moving it alone increases the total number of buffers.

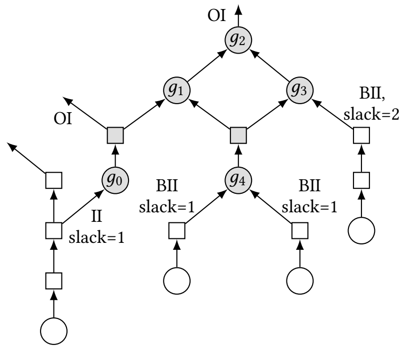


Figure 2. A chunk to be moved down.

To see how many levels a chunk can be moved and whether the movement reduces the total number of buffers, we define the *slack* of a chunk and *beneficial* interfaces as follows.

Moving down: When trying to move a chunk downwards, a *slack* is computed at each input interface (g_c, g_e) by

$$\text{slack}(g_c, g_e) = \begin{cases} rd(g_e, g_c) - 1, \text{ if } |\text{FO}(g_e)| = 1 \\ rd(g_e, g_c) - 2, \text{ otherwise} \end{cases} \tag{4}$$

Taking the minimum slacks at all IIs, the slack of the chunk is the maximum number of levels by which we can move the chunk down.

Moreover, (g_c, g_e) is said to be a *beneficial input interface* (BII) if

$$\forall g_o \in \text{FO}(g_e), g_o \neq g_c : rd(g_e, g_o) < rd(g_e, g_c). \tag{5}$$

If a chunk has x BIIs and y OIs with distinct g_c, moving the chunk down by l levels eliminates $l \cdot (x - y)$ buffers in total.

Moving up: Similarly but conversely, when trying to move a chunk upwards, a *slack* is computed at each output interface (g_c, g_e) by

$$\text{slack}(g_c, g_e) = \begin{cases} rd(g_c, g_e) - 1, \text{ if } |\text{FO}(g_c)| = 1 \\ rd(g_c, g_e) - 2, \text{ otherwise} \end{cases} \tag{6}$$

The slack of the chunk is the minimum slack at all of its OIs. Output interfaces are always beneficial. If a chunk has x OIs with distinct g_c and y IIs, moving the chunk up by l levels eliminates $l \cdot (x - y)$ buffers in total.

Overall, our heuristic algorithm first chooses a better initial schedule among ASAP and ALAP. Then, in each iteration, it constructs a chunk using Algorithm 2 for each node which is not yet in a chunk and tries to move the chunk up or down, applying the movement only when it is legal and beneficial. Depending on the desired optimization effort, no iteration at all, one iteration, or many iterations until no more beneficial movement is found, of chunked movement is executed. Finally, B/S are inserted using Algorithm 1 and the resulting mapped circuit is written out and verified.

6 Experimental Results

The proposed algorithms are implemented in C++ as part of the EPFL logic synthesis library *mockturtle*[1] [16]. Experiments were conducted on a 1.6 GHz dual-core Intel i5 CPU with 8 GB RAM. We have verified that our mapping results fulfill the path-balancing and

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

Table 1. Comparison against state of the art.

	Initial circuit			State of the art [8]				Our heuristic				Global optimum		
Bench.	#gates	max |FO|	$d(N)$	#B/S	#JJs	$d(N')$	Time (s)	#B/S	#JJs	$d(N')$	Time (s)	#B/S	#JJs	$d(N')$
adder1	7	2	4	16	74	8	0.13	16	74	8	0.00	16	74	8
adder8	77	3	17	371	1204	33	0.16	371	1204	33	0.00	371	1204	33
mult8	439	9	35	1833	6300	70	0.47	1869	6372	71	0.00	-	-	-
counter16	29	4	9	82	338	17	0.18	65	304	17	0.00	65	304	17
counter32	82	4	13	189	912	23	0.18	155	802	23	0.00	154	800	23
counter64	195	4	17	419	2134	30	0.23	352	1874	30	0.00	347	1864	30
counter128	428	4	22	895	4652	38	0.56	760	4088	38	0.00	747	4062	38
c17	6	2	3	12	60	5	0.15	12	60	5	0.00	12	60	5
c432	121	10	26	837	2406	37	0.34	874	2474	38	0.00	-	-	-
c499	387	8	18	1251	4858	30	0.38	1275	4872	31	0.00	-	-	-
c880	306	9	27	1723	5296	40	0.45	1703	5242	41	0.01	-	-	-
c1355	389	9	18	1216	4784	29	0.40	1290	4914	31	0.00	-	-	-
c1908	289	14	21	1505	4810	35	0.35	1298	4330	35	0.01	-	-	-
c2670	368	32	21	2055	7392	27	0.71	2132	6472	30	0.02	-	-	-
c3540	794	38	32	2395	9610	53	1.15	2266	9296	55	0.10	-	-	-
c5315	1302	41	26	6447	20854	41	4.00	6026	19864	42	0.12	-	-	-
c6288	1870	17	89	9297	29814	179	5.70	9893	31006	180	0.12	-	-	-
c7552	1394	170	33	8342	25140	59	85.27	8759	25882	66	0.12	-	-	-
sorter32	480	2	15	480	3840	30	0.35	480	3840	30	0.00	480	3840	30
sorter48	880	3	20	880	7040	35	0.52	880	7040	35	0.00	880	7040	35
alu32	1513	128	100	17178	43574	170	64.68	14655	38388	171	0.84	-	-	-
Total				57423	185092		166.36	55131	178398		1.34			

Table 2. Evaluation on larger EPFL benchmarks.

	Initial circuit			ASAP/ALAP		One iteration				Until convergence				
Bench.	#gates	max |FO|	$d(N)$	#B/S	Time (s)	#B/S	Δ (%)	Time (s)	#chunks	#B/S	Δ (%)	Time (s)	#chunks	$d(N')$
div	53225	425	2467	2214406	1.43	2094310	5.42	173.24	14492	2084772	5.85	271.71	8313	4918
hyp	136299	377	8911	8335403	158.61	-	-	> 300	-	-	-	> 300	-	17910
log2	24419	343	204	157429	0.32	102027	35.19	146.11	8559	98047	37.72	194.92	1619	414
multiplier	19355	194	142	109153	0.20	79658	27.02	6.16	428	79651	27.03	13.21	157	286
sin	4274	81	126	21818	0.03	18098	17.05	2.42	1463	17470	19.93	5.67	515	225
sqrt	21042	186	4933	1751842	5.71	1751745	0.01	5.27	11	1751742	0.01	5.64	4	8191
square	12184	81	126	68029	0.16	61061	10.24	20.18	2433	60552	10.99	42.71	363	256
arbiter	6997	43	59	33398	0.02	31282	6.34	3.70	830	31011	7.15	5.80	125	65
mem_ctrl	42627	763	95	311821	0.09	305714	1.96	57.47	1968	305689	1.97	87.86	786	182
voter	7538	9	54	21431	0.03	18129	15.41	4.71	2105	18044	15.80	5.43	842	99

fanout-branching constraints subject to the technology assumptions made and have published the best results of our mapped circuits for public validation[2].

6.1 Comparison Against State of the Art

Considering the state-of-the-art B/S insertion algorithms [5, 8], as [8] has shown to outperform [5], we only compare our proposed method against [8]. We use the same benchmarks[3] and the same technology assumptions, i.e., both PIs and POs are branched and balanced, and the splitting capacity of buffers $s_b = 4$. Our heuristic applies chunked movement repeatedly until convergence. Table 1 shows the number of gates after structural hashing (#gates), maximum fanout size (max |FO|), depth of the unmapped networks ($d(N)$), the number of inserted B/S (#B/S)[4], the total number of JJs including those from B/S and those from logic gates (#JJs), the mapped network depths ($d(N')$), and the runtime in seconds (Time)[5]. The global optimum results are listed for some smaller benchmarks[6]. For five benchmarks (marked in blue), both [8] and our heuristic obtained the optimum. For the counters, the B/S counts given by our heuristic are within 2% difference to the optimum, whereas the B/S counts of [8] are about 20% more. On average, we insert 4% less B/S and reduce JJ count by 3.6%. Moreover, the runtime of our algorithm is significantly lower than state of the art, giving an 124× speed-up.

6.2 Evaluation on Larger Benchmarks

As many of the benchmarks in Table 1 are small, we use the EPFL benchmark suite [1] to demonstrate the scalability of our heuristic[7]. As the intrinsic logic gate in AQFP is the majority-3 gate, the benchmarks are mapped into *majority-inverter graphs* (MIGs) using a graph mapper [21]. The assumptions are made the same as in the previous section (both PIs and POs are branched and balanced, $s_b = 4$). Table 2 lists B/S count (#B/S) and total runtime of scheduling, optimization and B/S insertion (Time) with zero (ASAP/ALAP), one (One iteration), or many (Until convergence) iterations of chunked movement. Columns Δ show the reduction

[2] Available: https://github.com/lsils/SCE-benchmarks

[3] We obtained the benchmarks from the authors of [8]. However, we noticed that some of the circuits are not structurally hashed, and some redundant inverter cells are included which should have been integrated as input negations. Thus, there are still slight differences in our starting points.

[4] This information is not provided in [8] and is inferred by subtracting #JJs of the (unstrashed) initial circuit from #JJs of their mapped circuit and dividing by 2.

[5] Values smaller than 0.005 are written as 0.00

[6] We use the Z3 SMT solver extended with optimization objectives [4, 7]. Where no result is listed in the table, the instances are still being solved on a server at the time of submission.

[7] Due to space limitation, only the 10 largest benchmarks are shown.

Table 3. Comparison of different technology assumptions.

Branch PIs?		Yes				No	
Balance PIs?		Yes		No		No	
Balance POs?		Yes	No	Yes	No	Yes	No
$s_b = 3$	$\sum$ #B/S	58135	41363	47213	35059	31544	22482
	$\sum d(N')$	1045	1045	1027	1040	1003	1004
$s_b = 4$	$\sum$ #B/S	55516	39043	45279	33392	29974	21518
	$\sum d(N')$	1012	1012	996	1005	977	977
$s_b = 8$	$\sum$ #B/S	53047	37120	44160	32635	29023	21237
	$\sum d(N')$	988	988	982	985	963	964

percentage of B/S count comparing to ASAP/ALAP, and columns #chunks show the number of chunks constructed in the last iteration. For the largest benchmark (hyp), although chunked movement times out after 300 seconds, a legal B/S insertion can still be obtained using ASAP/ALAP. On average, one iteration of chunked movement eliminates 13% of B/S. Our heuristic is capable of optimizing (within 5 minutes) large circuits with tens of thousands of gates, of over 100 in depth, and having maximum fanout sizes of up to several hundreds, which are a magnitude higher in all metrics than the largest ones [8] can deal with.

6.3 Impact of Technology Assumptions

As the physical constraints on the technology are ever-changing and assumptions on the requirements are often unclear, in this section, we study the impact of different technology assumptions discussed in Section 2.3. Table 3 lists the B/S insertion results obtained by applying our heuristic on the benchmarks used in Section 6.1. Due to space limitation, only the total number of B/S ($\sum$ # B/S) and the total depth ($\sum d(N')$), summed over all benchmarks, are shown. The three header rows list all the possible combinations about branching and balancing of PIs and POs. It can be observed that branching of PIs increase B/S count by about 50% because PIs often have a large fanout size. Thus, designing high driving-capacity registers will largely reduce the B/S cost in AQFP. Because the majority-3 gate uses a 3-to-1 branch cell, which has the same dimension as the one in an 1-to-3 splitter, there may be benefits in cell placement to use 1-to-3 splitters instead of 1-to-4. Decreasing s_b from 4 to 3 increases B/S count for about 5%, whereas the impact of having high-capacity splitters with $s_b = 8$ is smaller. These results motivate future research on the design of AQFP registers.

7 Conclusion and Discussion

Based on the observation that B/S insertion for AQFP is a scheduling problem (Claim 1), in this paper, we (a) proposed a linear-time locally-optimal B/S insertion algorithm subject to a given schedule (Algorithm 1); (b) formulated the search of the global optimum as an SMT problem; and (c) designed an efficient heuristic for global B/S insertion. While previous works focus on local optimality, this is the first work providing a global view on AQFP B/S insertion.

Although some related works integrate B/S cost in logic optimization, this paper focuses on B/S insertion and can be interleaved with, or applied after, logic synthesis. The state-of-the-art B/S insertion [8] focuses on an optimal algorithm for inserting B/S for a single net subject to a complex cost composed of maximum and total additional delay and number of B/S. Their local insertion allows increasing the depths of fanout gates ("additional delay"), but tries to minimize it. As a result, the time complexity of their local insertion is $O(n^3 \log n)$, where n is the fanout size, and cannot guarantee global optimality due to the interplay between gates. In contrast, our local B/S insertion is optimal in terms of B/S count subject to a given schedule and is of $O(n)$ complexity. An initial schedule can be found in $O(m)$ time, where m is the number of gates. Our global heuristic focuses on refining the schedule by chunked movement, whose effort is adjustable according to user demands. Thus, our approach provides a drastic 124× speed-up and is scalable to circuits of at least one magnitude larger.

Acknowledgments

This work was supported in part by the SNF grant "Supercool: Design methods and tools for superconducting electronics", 200021_1920981.

References

[1] Luca Amarú, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL combinational benchmark suite. In *Proceedings of IWLS*.

[2] 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).

[3] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (Eds.). 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, Vol. 185. IOS Press.

[4] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. *νZ* - An Optimizing SMT Solver. In *Proceedings of TACAS*, Vol. 9035. Springer, 194–199.

[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 *Proceedings of ICCD*. 429–436.

[6] Ruizhe Cai, Xiaolong Ma, Olivia Chen, Ao Ren, Ning Liu, Nobuyuki Yoshikawa, and Yanzhi Wang. 2019. IDE Development, Logic Synthesis and Buffer/Splitter Insertion Framework for Adiabatic Quantum-Flux-Parametron Superconducting Circuits. In *Proceedings of ISVLSI*. IEEE, 187–192.

[7] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of TACAS*, Vol. 4963. Springer, 337–340.

[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 *Proceedings of ICCAD*.

[9] Cheng-Tsung Hwang, Jiahn-Hurng Lee, and Yu-Chin Hsu. 1991. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. on CAD* 10, 4 (1991), 464–475.

[10] Dewmini Sudara Marakkalage, Heinz Riener, and Giovanni De Micheli. 2021. Optimizing Adiabatic Quantum-Flux-Parametron (AQFP) Circuits using Exact Methods. In *Proceedings of IWLS*.

[11] Saburo Muroga, Iwao Toda, and Satoru Takasu. 1961. Theory of majority decision elements. *Journal of the Franklin Institute* 271, 5 (1961), 376–418.

[12] Ghasem Pasandi and Massoud Pedram. 2018. PBMap: A path balancing technology mapping algorithm for single flux quantum logic circuits. *IEEE Trans. Appl. Supercond.* 29, 4 (2018), 1–14.

[13] Ghasem Pasandi and Massoud Pedram. 2019. Balanced factorization and rewriting algorithms for synthesizing single flux quantum logic circuits. In *Proceedings of GLSVLSI*. 183–188.

[14] Ro Saito, Christopher L Ayala, Olivia Chen, Tomoyuki Tanaka, Tomohiro Tamura, and Nobuyuki Yoshikawa. 2021. Logic synthesis of sequential logic circuits for adiabatic quantum-flux-parametron logic. *IEEE Trans. Appl. Supercond* 31, 5 (2021), 1–5.

[15] Ryo Sato, Yuki Hatanaka, Yuki Ando, Masamitsu Tanaka, Akira Fujimaki, Kazuyoshi Takagi, and Naofumi Takagi. 2016. High-speed operation of random-access-memory-embedded microprocessor with minimal instruction set architecture based on rapid single-flux-quantum logic. *IEEE Trans. Appl. Supercond.* 27, 4 (2016), 1–5.

[16] Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, and Giovanni De Micheli. 2019. The EPFL logic synthesis libraries. *arXiv preprint arXiv:1805.05121v2* (2019).

[17] Naoki Takeuchi, Shuichi Nagasawa, Fumihiro China, Takumi Ando, Mutsuo Hidaka, Yuki Yamanashi, and Nobuyuki Yoshikawa. 2017. Adiabatic quantum-flux-parametron cell library designed using a 10 kA cm^{-2} niobium fabrication process. *Superconductor Science and Technology* 30, 3 (2017), 035002.

[18] 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), 072601.

[19] 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), 035010.

[20] Naoki Takeuchi, Yuki Yamanashi, and Nobuyuki Yoshikawa. 2015. Adiabatic quantum-flux-parametron cell library adopting minimalist design. *Journal of Applied Physics* 117, 17 (2015), 173912.

[21] Alessandro Tempia Calvino, Heinz Riener, Shubham Rai, Akash Kumar, and Giovanni De Micheli. 2021. A Versatile Mapping Approach for Technology Mapping and Graph Optimization. In *Proceedings of IWLS*.

[22] Eleonora Testa, Siang-Yun Lee, Heinz Riener, and Giovanni De Micheli. 2021. Algebraic and Boolean optimization methods for AQFP superconducting circuits. In *Proceedings of ASP-DAC*. IEEE, 779–785.