Irredundant Buffer and Splitter Insertion and Scheduling-Based Optimization for AQFP Circuits

Siang-Yun Lee
EPFL, Switzerland

Heinz Riener
EPFL, Switzerland

Giovanni De Micheli
EPFL, Switzerland

Abstract

The adiabatic quantum-flux parametron (AQFP) is a promising energy-efficient superconducting technology. Before technology mapping, additional buffer and splitter cells need to be inserted into AQFP circuits to fulfill two special constraints: (1) Input signals to a logic gate need to arrive at the same time, thus shorter paths need to be delayed with buffers. (2) The output signal of a logic gate has to be actively branched with splitters if it drives multiple fanouts. Buffers and splitters largely increase the area and delay in AQFP circuits. Naïve buffer and splitter insertion and light-weight optimization using retiming techniques have been used in related works, and it is not clear how much space there is for further optimization. In this paper, we develop (a) a linear-time algorithm to insert buffers and splitters irredundantly, and (b) optimization methods by scheduling and by moving groups of gates, called chunks, together. Experimental results show a reduction of up to 39% on buffer and splitter cost. Moreover, as the technology is still developing and assumptions on the physical constraints are not clear yet, we also discuss the impacts of different assumptions with experimental results to motivate future research on AQFP register design.

Keywords: AQFP, superconducting electronics, path balancing, combinational circuit, scheduling

1 Introduction

Superconducting electronics is an emerging domain arising from the demand for ultra-low power consumption. Among various superconducting logic families, the *adiabatic quantum-flux parametron* (AQFP) [11] is a technology featuring 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 [9, 10] 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, logical 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 for AQFP are in need. Pioneering works attempt to adapt existing tools to fulfill the path-balancing and fanout-branching constraints with post-synthesis modifications and optimization. In [2] and [3], after classical logic synthesis, path-balancing buffers and fanout-branching splitters are inserted separately and then retiming-like algorithms are applied to reduce the buffer and splitter cost locally. A majority-based logic synthesis flow considering AQFP buffer and splitter costs is proposed in [13], which emphasizes on reducing circuit depth and restricting the increase of fanout count. In [4], consideration of the balancing and branching constraints is integrated in exact-synthesis-based rewriting. However, in the results of these works, buffers and splitters still make up for over 50%, and sometimes up to 80%, of the total cost. Moreover, as the technology is still being developed, the physical constraints are ever-changing and assumptions on the requirements vary across different works and are often unclear, which makes them difficult to compare with. For example, whether primary inputs and primary outputs need to be path-balanced and fanout-branched depends on the design of interfacing registers, which is still under development [7].

While the path-balancing constraint also exists for the *rapid single-flux-quantum* (RSFQ) technology and optimization methods have been researched [6], an important distinction is that RSFQ splitters are not clocked but AQFP splitters are. For this reason, fanout-branching has to be considered together with path-balancing in AQFP, which makes the problem more complicated. If only path balancing needs to be done, as in RSFQ, the optimal way of inserting buffers without changing the logic structure can be found in linear time. However, buffer and splitter insertion in AQFP is non-trivial even without logic optimization. Thus, we limit our investigation to the problem of AQFP buffer and splitter insertion without logic transformation.

In this paper, observations about the complexity of the defined problem and systematic methods to deal with it are presented, and the impact of the technology assumptions are experimented and discussed. In Section 4, a linear-time algorithm to count irredundant buffers is presented, which subsumes the retiming and optimization techniques proposed in [3]. Different than the previous work, we consider buffers and splitters together and count them in an irredundant way such that the "optimizations" in [3] are considered without extra effort. Also, we observe that the irredundant construction is not optimal yet. On top of the locally irredundant buffer and splitter insertion, efforts have to be made in finding a suitable depth assignment to logic gates to achieve the global optimum. In Section 5, methods to obtain an initial depth assignment and to adjust it incrementally are presented. To escape from local minimum, we propose to move groups of gates together as a chunk. Experimental results show that obtaining better depth assignments using the proposed methods reduces the number of buffers by up to 39%. Moreover, various possible technology assumptions are first discussed in Section 3 and then experimented in Section 6. The results suggest that branching and balancing of primary inputs have greater impacts on the buffer count of about 50% and 30%, respectively.

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. [11] The basic circuit components in this technology are the buffer cell and the branch cell. The 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. [12] The commonly-used cost metric of AQFP circuits is the *Josephson junction* (JJ) count. A buffer costs 2 JJs and a majority-3 gate costs 6 JJs.

Logic gates in an AQFP circuit need to be activated and deactivated periodically by an excitation current. [9] 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 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 remaining of this paper, we do not distinguish buffers and 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*. 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 also 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 in $G - \{g, g_o\}$. 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.

For each node n in a network, the *depth* of n, denoted by $d(n)$, is a non-negative integer assigned to n. The depth of a network $N = (V = I \cup O \cup G, E)$ is defined as

$$d(N) = \max_{o \in O} d(o). \tag{1}$$

Moreover, the *relative depth* between a PI or a gate $n \in (I \cup G)$ and one of its fanout $n_o \in \mathrm{FO}(n) \subset (G \cup O)$, is denoted and defined as

$$rd(n, n_o) = d(n_o) - d(n). \tag{2}$$

Note that relative depth is only defined among PIs, gates, and POs.

3 Technology Assumptions

To fulfill the needs in the AQFP technology for fanout-branching and path-balancing, we define the following two properties for a mapped network $N' = (V' = I \cup O \cup G \cup B, E')$ with a depth assignment. Given the *splitting capacities* s_i, s_g, s_b of each type of node,

1. N' is *path-balanced* if

$$\forall n_1, n_2 \in V' : (n_1, n_2) \in E' \Rightarrow d(n_1) = d(n_2) - 1, \tag{3}$$

$$\forall i \in I : d(i) = 0, \text{ and} \tag{4}$$

$$\forall o \in O : d(o) = d(N'). \tag{5}$$

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.

An (unmapped) network N with a depth assignment is said to be *legal* if a path-balanced and properly-branched mapped network N' can be extended from N.

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. Depending on how the registers are implemented, different assumptions on whether PIs and POs need to be path-balanced or branched may arise.

3.1 Path-Balancing of PIs

It is possible to design registers that can hold and output its value at every clock cycle. In this case, the PI nodes in our model can be placed at any depth, i.e., condition 4 is removed.

3.2 Path-Balancing of POs

In most related works, it is assumed that all PO signals must arrive at the register inputs at the same clock cycle. That is, POs are path-balanced to ensure robust operations. If the PI values are always available and stable until the next register update, shorter paths from PI to PO simply compute the same result repeatedly 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). In other words, the PO nodes in our model are no longer limited to be placed at the same depth, i.e., condition 5 is removed. However, there may still be constraints on the PO depths depending on the clocking scheme used. For example, a 4-phase clocking scheme [9] may require that the depths of PO nodes must be a multiple of 4 because the registers can only take inputs in one of the four clock phases.

3.3 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.

3.4 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 [5] and input negation is for free in AQFP. However, if a gate output is needed by the next stage once in the regular form and once in the negated form, then we not only need a splitter, but also an additional NOT gate made of an input-negated buffer.

3.5 Problem Formulation

In this paper, we focus on the problem of AQFP buffer 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.

4 Irredundant Buffer Insertion

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.

We consider only irredundant networks in the remaining of this paper. In this section, we will explain how the problem formulated in Section 3.5 can be approached, starting from the following observation.

Claim 1. *Given a network $N = (V, E)$, finding a minimum mapped network for N is essentially finding a depth assignment to every node in V.*

To show why Claim 1 is true, we will first formulate Lemma 2 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 depth assignment is given, the total size of fanout trees is decided, so as the size of the mapped network.

Lemma 2. *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 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 partitioning of B. ■

```
Input: A gate $g$
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.

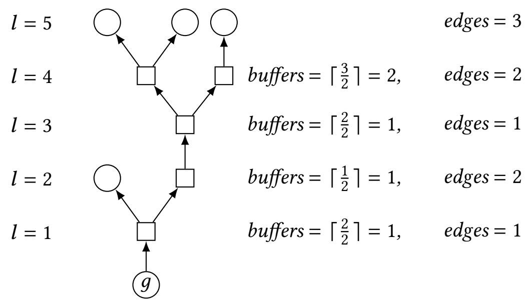

$l = 5$
$l = 4$
$l = 3$
$l = 2$
$l = 1$
g
$buffers = \lceil \frac{3}{2} \rceil = 2,$
$buffers = \lceil \frac{2}{2} \rceil = 1,$
$buffers = \lceil \frac{1}{2} \rceil = 1,$
$buffers = \lceil \frac{2}{2} \rceil = 1,$
$edges = 3$
$edges = 2$
$edges = 1$
$edges = 2$
$edges = 1$

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

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 levels l from the relative depth of the highest fanout down to 1, and counts the number of buffers (variable *buffers*) needed at each level. The total number of buffers is accumulated in variable *count* (line 6). At each level l, variable *edges* keeps the number of edges ending in some node of relative depth l, which is simply the number of buffers and fanouts at this level (line 7). Then, the number of buffers needed at the lower level $l - 1$ is computed by 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. If the fanout information is stored in a data structure that the size $|\{g_o \in \mathrm{FO}(g) : rd(g, g_o) = l\}|$ for any given value l can be queried in constant time, then the algorithm runs in linear time with respect to $|\mathrm{FO}(g)|$.

Figure 1 illustrates an example execution of Algorithm 1, where circles are gates and squares are buffers, and $s_b = 2$ is assumed. 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.

The constructed fanout tree is guaranteed to be irredundant because only the minimum number of buffers is inserted at each level based on the number of outgoing edges needed. Note that the retiming optimization proposed in [3], which pushes buffers from the outputs of a splitter to its input, is already considered during construction of irredundant fanout trees.

Moreover, Algorithm 1 also verifies whether it is possible to build a properly-branched network with the given depth assignment. 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 Lemma 2, a mapped network is derived. The mapped network is guaranteed to be properly-branched if the assertion in line 8 never fails. It is also path-balanced because every node is connected to a node at exactly one level lower. As Algorithm 1 is deterministic, we conclude that the number of irredundant buffers for a given depth assignment is unique.

5 Optimization on Depth Assignment

Following Claim 1, in this section, we attempt to find a good depth assignment to minimize the total number of buffers in the mapped network. In Section 5.1, we first obtain an initial depth assignment using scheduling algorithms. Then, in Section 5.2, we try to move gates up or down to reduce the total number of buffers. *Moving* a gate g *up* by l levels means that $d(g)$ is increased by l while the depths of the other gates remain the same. Similarly, moving g *down* means $d(g)$ is decreased. During the entire process, we always ensure that the network is legal.

5.1 Obtaining an Initial Depth Assignment

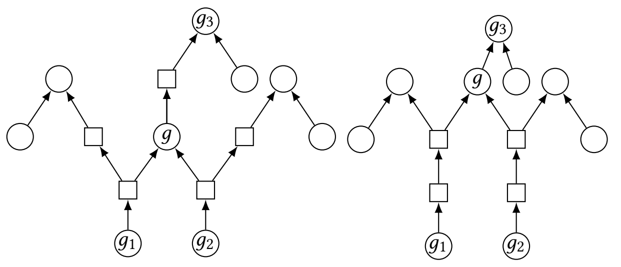
An initial depth assignment can be obtained using an *as-soon-as-possible scheduling* (ASAP) algorithm which assigns the smallest possible depth to each gate. To ensure that the network can be path-balanced and properly-branched after mapping, enough depths for a balanced fanout tree are reserved at the output of every multi-fanout gates, which is calculated as

$$\left\lceil \frac{\log(|\mathrm{FO}(g)|)}{\log(s_b)} \right\rceil. \tag{6}$$

Then, an *as-late-as-possible scheduling* (ALAP) can be applied using an upper bound $d(N)$ obtained by ASAP.

However, neither ASAP nor ALAP achieves the global optimum. Figure 2 (a) shows an example sub-network after ASAP, where circles are gates and squares are buffers. The gate g is not the highest fanout of either of its fanins, thus moving g up does not increase sizes of the fanout trees of g_1 and g_2. Moreover, the fanout g_3 is lower-bounded by its other fanin. Thus, by moving up g, as shown in Figure 2 (b), a buffer is eliminated in its fanout tree.

The reason why this problem is not trivial is because a movement of a gate affects both its own fanout tree and its fanins' fanout trees. Moreover, in some cases, it is impossible to legally move a single gate and reduce the buffer count, but rearranging some gates altogether eventually leads to



(a) Original sub-network. **(b)** Optimized sub-network.

Figure 2. Example sub-network where ASAP does not lead to the optimum.

further reduction. Thus, in the following sections, groups of gates are identified and moved together as *chunks*.

5.2 Chunked Movement

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 fanouts, then moving any of its fanouts to $d(g) + 1$ is not legal because there must be a buffer occupying the only outgoing edge of g at $d(g) + 1$.

A pair of gates $(g, g_o) : g_o \in \mathrm{FO}(g)$ are *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. $|\mathrm{FO}(g)| > 1$ 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* C is a set of closely-connected gates and can be seen as a super-node having multiple incoming and outgoing edges, called the *input interfaces* (IIs) and *output interfaces* (OIs), respectively. An interface is a pair (g_c, g_f) of gates, where $g_c \in C$, $g_f \notin C$, and either $g_f \in \mathrm{FI}(g_c)$ (II) or $g_f \in \mathrm{FO}(g_c)$ (OI).

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

Algorithm 2: Chunk construction.

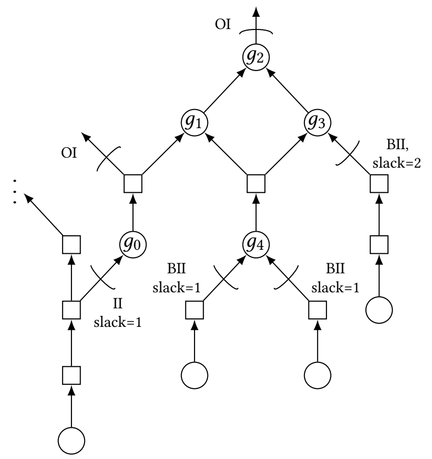


Figure 3. A chunk to be moved down.

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 the individual gates in the chunk cannot be moved legally, a chunk may be moved as a whole. Figure 3 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 without moving other gates at the same time. In contrast, the gate g_0 can be legally moved down, but moving it alone only increases the total number of buffers.

To see how many levels a chunk can be moved and whether the movement reduces the total number of buffers, we define some more properties for chunk interfaces.

Moving down: When a chunk is intended to be moved down, a *slack* is computed at each input interface (g_c, g_f) by

$$\mathrm{slack}(g_c, g_f) = \begin{cases} rd(g_f, g_c) - 1, & \text{if } |\mathrm{FO}(g_f)| = 1 \\ rd(g_f, g_c) - 2, & \text{otherwise} \end{cases} \quad (7)$$

The slack of the chunk is the maximum number of levels by which we can move the chunk down, and it is calculated as the minimum slack of all of its input interfaces. Moreover, (g_c, g_f) is said to be a *beneficial input interface* (BII) if

$$\forall g_o \in \mathrm{FO}(g_f), g_o \neq g_c : rd(g_f, g_o) < rd(g_f, g_c). \quad (8)$$

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 and conversely, when a chunk is intended to be moved up, a *slack* is computed at each output interface (g_c, g_f) by

$$\text{slack}(g_c, g_f) = \begin{cases} rd(g_c, g_f) - 1, \text{ if } |\text{FO}(g_c)| = 1 \\ rd(g_c, g_f) - 2, \text{ otherwise} \end{cases} \tag{9}$$

The slack of the chunk is the minimum slack of all of its output interfaces. When moving up, 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.

6 Experimental Results

In this section, we present experimental results using different combinations of technology assumptions discussed in Section 3. The irredundant buffer insertion and chunked movement algorithms are implemented in C++-17 as part of the EPFL logic synthesis library *mockturtle*[1] [8]. As discussed in Section 2.1, the intrinsic logic gate in the AQFP technology is the majority-3 gate, *majority-inverter graphs* (MIGs) [1] are used as the data structure for (unmapped) networks in our experiments. We use the same initial MIGs as in [13] from the MCNC benchmark suite [14].

6.1 Balancing of PIs and POs

In this section, we use the assumptions that PIs need to be branched ($s_i = 1$) and $s_b = 3$. Table 1 shows the four possible combinations of the assumptions on whether PIs and POs need to be path-balanced. When the POs are not balanced, we do not impose the requirement of modulo-4 path lengths either. The block **Unmapped** lists the benchmark names (Bench.), numbers of majority gates (#gates), network depths before buffer insertion (Depth), and numbers of PIs (#PIs) and POs (#POs). There are five columns in each block, listing the numbers of irredundant buffers after the initial ASAP scheduling (ASAP) and after ALAP (ALAP), the final number of buffers after optimization with chunked movement (Opt.), the depth of the mapped networks (Depth), and the number of chunks (#chunks). The row Improv. lists the percentage improvements of ALAP and Opt. comparing to ASAP. The row Ratio lists the ratios of the initial ASAP (**bold**), ALAP (*italic*), and optimized (underlined) buffer counts across different experiments using **Balance PIs + Balance POs** as the baseline.

The scheduling method (ASAP or ALAP) that leads to fewer buffers is used to obtain the initial depth assignment for chunked movement, decided independently for each benchmark and for each experiment. When PIs need to be balanced (the upper half of Table 1), ALAP do not lead to much improvement, but chunked movement is able to optimize away 17% of the buffers. On the other hand, when PIs do not need to be balanced (the lower half of Table 1), ALAP usually leads to much better results, reducing about 20-30% of buffers, and chunked movement further eliminates another 6-8%. Observing the bold ratios, we see that the path-balancing buffers for POs constitute about 14% of the total when ASAP is applied; observing the italic ratios, we see that the path-balancing buffers for PIs constitute about 30% of the total when ALAP is applied. With the chunked-movement-based optimization, the two extremes are balanced, and we see a larger impact of the PI-balancing assumption.

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

6.2 Branching of PIs and Splitting Capacity

In this section, we use the assumptions that neither PIs nor POs need to be balanced (i.e., the last case in Table 1), and we study the impact of branching PIs and the value of buffer's splitting capacity s_b. Table 2 shows the number of buffers (Opt.) and the circuit depth (Depth) after optimization using different assumptions on PI branching and s_b value. The two columns under $|\text{FO}(i)|$ show, respectively, the maximum (Max.) and the average (Avg.) fanout size of PIs in each benchmark. Row Ratio lists the ratios of the buffer counts in each experiment comparing to **Branch PIs**, $s_b = 3$.

If PIs do not need to be branched, the number of buffers needed is halved. In other words, about half of the buffers are used to branch high-fanout PIs. This phenomenon is even more obvious when the splitting capacity is smaller. Indeed, PIs with high fanout counts are common in many benchmarks, and PI-branching splitters can hardly be eliminated with any optimization.

When not branching PIs, the impact of splitting capacity is relatively minor, with less than 5% difference between $s_b = 3$ and $s_b = 4$. Thus, except for branching PIs, design of high-capacity splitters is not particularly necessary.

7 Conclusion and Future Work

In conclusion, this paper provides a different viewpoint to the problem of AQFP buffer and splitter insertion. With the linear-time irredundant buffer insertion algorithm presented in Section 4 and simple scheduling algorithms discussed in Section 5.1, a good starting point can be obtained efficiently. Then, the chunked movement method illustrated in Section 5.2 provides possibility to further minimize the cost and escape from local minima. In Section 6, experimental results show that the proposed optimization flow with scheduling and chunked movement reduces about 17-39% of buffers, depending on the technology assumptions imposed. Moreover, experiments on different assumptions suggest that PI balancing has a greater impact than PO balancing, and that PI branching accounts for half of the inserted buffers. These results motivate future research on the design of AQFP registers and splitters. For future work, we hope to develop an exact algorithm to find the global optimal, which will allow us to evaluate how good the existing and future-developed heuristics are. We also plan to integrate the proposed buffer optimization with logic synthesis algorithms considering AQFP constraints such as [4, 13].

Table 1. Impact of PI and/or PO balancing and quality of chunked movement.

Unmapped			**Balance PIs**									
			Balance POs					**Not balance POs**				
Bench.	#gates	Depth	ASAP	ALAP	Opt.	Depth	#chunks	ASAP	ALAP	Opt.	Depth	#chunks
c1908	381	38	3011	3296	2820	64	56	2605	3296	2413	64	56
c432	174	44	2471	2647	2220	68	25	2423	2635	2198	70	30
c5315	1270	33	9936	11844	9457	60	200	6409	11402	5986	59	205
c880	300	28	2577	2911	2159	42	44	1854	2884	1501	42	45
chkn	421	28	1607	1280	1241	38	8	1536	1280	1232	38	8
count	119	18	816	1004	766	29	31	639	1004	585	29	31
dist	535	16	1086	814	809	28	3	1066	814	808	28	3
in5	443	19	1413	1056	1042	30	18	1278	1056	1020	30	18
in6	370	17	1184	938	884	23	22	1002	938	811	23	22
k2	1955	25	5177	4570	4171	43	53	4512	4528	3722	43	139
m3	411	13	833	636	620	22	14	761	634	615	22	14
max512	713	17	1399	1093	1078	28	3	1361	1093	1070	28	3
misex3	1532	24	4181	3004	2879	38	16	4113	3004	2883	38	15
mlp4	462	16	915	668	653	26	9	839	668	647	26	9
prom2	3477	22	6855	5442	5300	33	59	6777	5442	5298	33	59
sqr6	138	13	381	246	246	20	8	287	241	229	20	6
x1dn	152	14	479	561	428	22	16	453	561	399	22	16
Total			44321	42010	36773			37915	41480	31417		
Improv.				5.2%	17.0%				-9.4%	17.1%		
Ratio			**(1.00)**	*(1.00)*	(1.00)			**0.86**	*0.99*	0.85		

			Not balance PIs									
			Balance POs					**Not balance POs**				
Bench.	#PIs	#POs	ASAP	ALAP	Opt.	Depth	#chunks	ASAP	ALAP	Opt.	Depth	#chunks
c1908	33	25	3011	2910	2549	62	24	2605	2910	2202	62	44
c432	36	7	2471	1903	1689	65	6	2423	1891	1673	65	6
c5315	178	123	9936	4520	3934	56	64	6409	4197	3574	56	64
c880	60	26	2577	1475	1306	40	22	1854	1448	1238	40	22
chkn	29	7	1607	785	720	34	8	1536	785	715	34	8
count	35	16	816	343	287	24	15	639	343	286	24	15
dist	8	5	1086	791	762	24	2	1066	791	761	24	2
in5	24	14	1413	814	762	27	14	1278	814	746	27	13
in6	33	23	1184	674	627	23	21	1002	674	621	23	19
k2	45	45	5177	3854	3375	37	59	4512	3812	3249	37	56
m3	8	16	833	613	576	19	13	761	611	567	19	12
max512	9	6	1399	1081	1036	26	3	1361	1081	1028	26	3
misex3	14	14	4181	2983	2815	34	19	4113	2983	2811	34	19
mlp4	8	8	915	645	609	23	8	839	645	603	23	8
prom2	9	21	6855	5435	5261	33	57	6777	5435	5259	33	57
sqr6	6	12	381	230	217	17	8	287	225	200	17	6
x1dn	27	6	479	399	362	19	5	453	399	362	19	5
Total			44321	29455	26887			37915	29044	25895		
Improv.				33.5%	39.3%				23.4%	31.7%		
Ratio			**1.00**	*0.70*	0.73			**0.86**	*0.69*	0.70		

Table 2. Impact of PI branching and splitting capacity.

	$\|FO(i)\|$		**Branch PIs**						**Not branch PIs**					
			$s_b = 2$		$s_b = 3$		$s_b = 4$		$s_b = 2$		$s_b = 3$		$s_b = 4$	
Bench.	Max.	Avg.	Opt.	Depth	Opt.	Depth	Opt.	Depth	Opt.	Depth	Opt.	Depth	Opt.	Depth
c1908	22	0.67	2624	69	2202	62	2073	58	1245	66	952	61	870	59
c432	6	0.17	1932	75	1673	65	1512	58	497	74	456	67	423	57
c5315	84	0.47	4818	65	3574	56	3087	51	2668	60	2043	51	1796	50
c880	11	0.18	1568	50	1238	40	1192	40	528	45	437	42	428	42
chkn	42	1.45	1001	40	715	34	602	34	264	34	235	33	229	33
count	32	0.91	373	26	286	24	273	24	72	25	57	23	57	23
dist	96	12.00	1093	27	761	24	659	23	392	22	376	22	376	22
in5	52	2.17	1019	29	746	27	670	27	351	26	307	26	306	26
in6	46	1.39	835	25	621	23	544	21	216	20	206	20	205	20
k2	152	3.38	4554	39	3249	37	2915	36	3019	38	2349	35	2207	35
m3	77	9.62	861	25	567	19	481	19	306	18	278	18	267	18
max512	126	14.00	1475	30	1028	26	894	24	563	23	541	23	539	23
misex3	144	10.29	3769	43	2811	34	2558	34	2029	33	1864	33	1841	34
mlp4	79	9.88	888	25	603	23	514	23	281	22	263	22	263	22
prom2	451	50.11	7369	37	5259	33	4568	31	3813	26	3405	26	3346	26
sqr6	33	5.50	292	17	200	17	179	17	92	16	90	16	90	16
x1dn	15	0.56	415	21	362	19	331	19	139	20	124	18	123	18
Total			34886		25895		23052		16475		13983		13366	
Ratio			1.35		(1.00)		0.89		0.64		0.54		0.52	

Acknowledgments

This work was supported in part by the EPFL Open Science Fund and by the SNF grant "Supercool: Design methods and tools for superconducting electronics", 200021_1920981. The authors thank Prof. Nobuyuki Yoshikawa and Prof. Christopher L. Ayala from Yokohama National University for their discussion on the AQFP technology.

References

[1] Luca Amaru, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. Majority-inverter graph: A new paradigm for logic optimization. *IEEE Transactions on CAD* 35, 5 (2015), 806–819.

[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), 054006.

[3] 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*. IEEE, 429–436.

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

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

[6] Ghasem Pasandi and Massoud Pedram. 2018. PBMap: A path balancing technology mapping algorithm for single flux quantum logic circuits. *IEEE Transactions on Applied Superconductivity* 29, 4 (2018), 1–14.

[7] 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 Transactions on Applied Superconductivity* 31, 5 (2021), 1–5.

[8] 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).

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Saeyang Yang. 1991. *Logic synthesis and optimization benchmarks user guide: version 3.0.* Microelectronics Center of North Carolina (MCNC).

adding up the complexity of the nodes:

$$group\_complexity = \sum_{h=0}^{2^l-1} C(h, 2^l) \quad (35)$$

The computational complexity of a whole row is obtained with respects to the complexity of the groups and the number of groups in a row:

$$row\_complexity = 2^{(\log_2 \frac{n}{2} - l)} \times \sum_{h=0}^{2^l-1} C(h, 2^l) \quad (36)$$

The computational complexity of the PC block is obtained by adding up the complexity of all rows:

$$complexity_{[PC]} = \sum_{l=0}^{\log_2(n)-1} \left(2^{(\log_2 \frac{n}{2} - l)} \times \sum_{h=0}^{2^l-1} C(h, 2^l) \right)$$
$$= \frac{9}{14}n^4 + \frac{23}{4}n^3 + \frac{93}{4}n^2 + \frac{15}{2}n\log_2 n - \frac{415}{14}n \quad (37)$$

Finally, the overall computational complexity of a Ladner-Fischer adder is calculated by adding up the complexity of the three stages in Eq. (14), Eq. (18), and Eq. (37). Based on the calculated complexity, we can observe that the order of the verification complexity is $O(n^4)$. Therefore, proving correctness of a Ladner-Fischer adder using BDDs has quartic time complexity.

E. Verification Complexity of a Kogge-Stone adder

The Kogge-Stone adder is another parallel prefix adder with a parallel tree of prefix operators (see Fig. 4). The computational complexity of each prefix operator is shown in Fig. 8 as a C function. Note that if the inputs of a prefix operator are $(G_{[i:k]}, P_{[i:k]})$ and $(G_{[k-1:j]}, P_{[k-1:j]})$, the complexity can be calculated by $C(i-k, k-j)$.

For an n-bit Kogge-Stone adder, the depth (i.e., number of rows) and the number of nodes in each row are:

$$depth = \log_2(n),$$
$$nodes\_in\_row = n - 2^l \quad (38)$$

where l is the row number. Please note that the equations are exact for all word lengths being a power of 2 (i.e., $n = 2^m$) [2].

We divide the nodes in each row into two groups based on the input values of the C functions. In the first group (green boxes in Fig. 8), the input values of the C functions are identical, i.e., $C(2^l-1, 2^l)$. In the second group (red boxes in Fig. 8), the first input values are exactly the same and equal $2^l - 1$. However, the second value is equal to $h+1$ for the h^{th} node in the group.

The number of nodes in the first group ($group1$) and second group ($group2$) are as follows:

$$nodes\_in\_group1 = n - 2^{l+1} + 1,$$
$$nodes\_in\_group2 = 2^l - 1 \quad (39)$$

The computational complexity of each group is obtained by adding up the complexity of the inside nodes:

$$group1\_complexity = (n - 2^{l+1} + 1) \times C(2^l - 1, 2^l),$$
$$group2\_complexity = \sum_{h=0}^{2^l-2} C(2^l - 1, h+1) \quad (40)$$

TABLE I
RUN-TIME OF VERIFYING ADDERS (SECONDS)

Size	Benchmarks		
	serial prefix	Ladner-Fischer	Kogge-Stone
1024	1.28	1.64	1.84
2048	6.37	7.56	8.37
3072	15.24	17.94	21.60
4096	27.21	33.59	39.01
5120	43.05	49.85	69.89
6144	67.87	78.07	104.47
7168	97.36	114.06	142.42
8192	129.78	153.67	177.43
9216	164.53	184.33	234.78
10240	200.45	241.49	315.52

We can add the complexity of the first and second group to get the computational complexity of a row. The computational complexity of the PC block is obtained by adding up the complexity of all rows:

$$complexity_{[PC]} =$$
$$\sum_{l=0}^{\log_2(n)-1} \left((n - 2^{l+1} + 1) \times C(2^l - 1, 2^l) + \sum_{h=0}^{2^l-2} C(2^l - 1, h+1) \right) =$$
$$\frac{81}{70}n^4 + \frac{111}{14}n^3 + 22n^2 + 6n\log_2 n - \frac{321}{7}n + \frac{517}{35} \quad (41)$$

By adding up the complexity of the three stages in Eq. (14), Eq. (18), and Eq. (41), the overall complexity is obtained. After calculating the computational complexity, we can conclude that the order of the BDD-based verification complexity is $O(n^4)$. Therefore, proving correctness of a Kogge-Stone adder has quartic time complexity.

IV. EXPERIMENTAL RESULTS

We have implemented the BDD-based verifier in C++. The tool takes advantage of the symbolic simulation to obtain the BDDs for the primary outputs. Then, the BDDs are evaluated to see whether they match the BDDs for an adder. In order to handle the BDD operations, we used the CUDD library [20]. The benchmarks for the three prefix adders are generated using GenMul [21]. All experiments are performed on an Intel(R) Core(TM) i7-8565U with 1.80 GHz and 24 GByte of main memory.

Table I reports the verification times for adders. The first column **Size** denotes the size of the adder based on the inputs' bit-width. The run-time (in seconds) of the BDD-based verification method is reported in the second column **Benchmarks** for the three prefix adders.

It is evident in Table I that the BDD-based verification reports very good results for prefix adders. A Kogge-Stone adder with 10240 bits per input, which consists of more than 400K gates, can be verified in less than 6 minutes. Thus, the experimental results for the three prefix adders confirm the scalability of the BDD-based verification method.

In order to check the correctness of the complexity bounds obtained in Section III, we first show the results of Table I as three graphs in Fig. 9. Then, we fit a curve to the points