# Technology Legalization and Optimization for Adiabatic Quantum-Flux Parametron

Siang-Yun Lee, Alessandro Tempia Calvino, *Graduate Student Member, IEEE*, Heinz Riener, and Giovanni De Micheli, *Life Fellow, IEEE*

***Abstract*—Adiabatic quantum-flux parametron (AQFP) is an energy-efficient superconducting technology. Before physical design can be performed, AQFP technology mapping involves not only mapping logic into supported gate types but also legalizing the circuit to fulfill the technology-imposed constraints on path balancing and fanout branching by inserting buffer and splitter cells. These cells account for a significant amount of the circuit's area, delay, as well as for increasing energy consumption. In this article, we 1) identify that the AQFP legalization problem is a scheduling problem; 2) propose linear-time depth-optimal scheduling and irredundant buffer insertion algorithms; 3) present heuristic optimization algorithms to further reduce buffer count; and 4) suggest an unsupervised design space exploration approach for AQFP technology mapping, mixing, and interleaving logic optimization and technology legalization. Experimental results show that our design space exploration, utilizing the proposed technology legalization and optimization flow, achieves 44% improvement on the energy–delay product compared to the state of the art.**

***Index Terms*—Adiabatic quantum-flux parametron (AQFP), logic synthesis, superconducting electronics, technology mapping.**

## I. Introduction

HIGH-PERFORMANCE computing of data centers and computing clusters contributes to a noticeable percentage of the world's energy consumption, demanding more energy-efficient computation paradigms. The adiabatic quantum-flux parametron (AQFP) is an emerging superconducting technology shown to achieve promising energy efficiency [1] and has attracted increasing attention in the past decade. While the technology is rapidly evolving [2], [3], [4], [5], [6] and larger-scale systems are being developed [7], [8], design automation for AQFP is also an extensively researched topic [9], [10], [11].

One major challenge in AQFP design automation is the legalization of the logic circuit to fulfill two unconventional technology constraints, *path balancing* and *fanout branching*, before physical design. Due to its gate-level clocking property, AQFP gates require all input signals to arrive at the same time, thus buffers have to be inserted on shorter data paths to balance with the longer paths. Moreover, splitters are needed at the output of AQFP gates driving multiple signals, and these splitters are also clocked. Thus, logic circuits generated by technology-independent logic synthesis must be *legalized* for the AQFP technology by inserting buffers and splitters (B/Ss). Legalization of AQFP circuits is essential to unlock its potential of pipelined computation while maintaining correct functionality.

In a legalized AQFP circuit, B/Ss often contribute to over 50% of the Josephson junction (JJ) count, which is the commonly used cost metric related to area as well as energy consumption. Thus, optimized algorithms for AQFP legalization are in need to reduce the overhead and increase scalability of AQFP circuits. In this article, we summarize a scalable and flexible framework for AQFP technology legalization and optimization, based on two previous papers [12], [13]. First, we show that the AQFP B/S insertion problem is a scheduling problem by formalizing an irredundant B/S insertion algorithm, which is optimal subject to a given schedule. Then, we propose depth-optimal scheduling algorithms, forming the basis for obtaining an initial legalized circuit. We then present two orthogonal heuristic optimization algorithms to further optimize the B/S count. Our AQFP legalization and optimization flow consists of obtaining two depth-optimal schedules, iteratively optimizing them separately, and then choosing the better one. Furthermore, we present an unsupervised design space exploration approach that interleaves logic synthesis and technology legalization. Finally, we discuss how logic as well as technology constraints can be verified after AQFP synthesis.

Our experiments demonstrate promising results in three possible scenarios of application.

1) When a runtime-efficient synthesis flow is of concern, it is better to separate logic synthesis and AQFP legalization. For the latter, we present a heuristic legalization and optimization flow which obtains similar, near-optimal quality as the state-of-the-art (SoTA) integer linear programming (ILP)-based algorithm within very little runtime.
2) When dealing with larger-scale designs, scalability is important. Our approach is flexible in runtime budget as the optimization part can be skipped and our scheduling-based legalization is fast and scalable. We demonstrate legalization results on benchmarks 10× to

Manuscript received 5 December 2023; revised 2 May 2024 and 15 July 2024; accepted 18 July 2024. Date of publication 26 July 2024; date of current version 26 December 2024. This work was supported by the SNF Grant "Supercool: Design Methods and Tools for Superconducting Electronics," under Grant 200021_1920981. This article was recommended by Associate Editor W. Qian. *(Corresponding author: Siang-Yun Lee.)*

Siang-Yun Lee and Heinz Riener are with Cadence Design Systems GmbH, 85622 Feldkirchen, Germany (e-mail: siang.yun.sonia.lee@gmail.com).

Alessandro Tempia Calvino and Giovanni De Micheli are with the Integrated Systems Laboratory, Swiss Federal Institute of Technology Lausanne, 1015 Lausanne, Switzerland.

Digital Object Identifier 10.1109/TCAD.2024.3434385

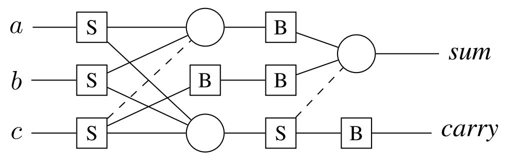

Fig. 1. AQFP-legalized full adder circuit.

100× larger than what any other related works could handle.

3) When circuits are small enough or the runtime budget is sufficient, logic restructuring and technology legalization can be interleaved to achieve better results. We propose an AQFP technology mapping approach combining existing logic optimization techniques with our legalization flow for design space exploration. Experimental results show a significant 44% improvement in the energy–delay product (EDP) compared to the best-known AQFP synthesis results.

## II. Preliminaries

### A. Adiabatic Quantum-Flux Parametron

AQFP is a superconducting electronics technology. In an AQFP circuit, JJs, instead of transistors, are the active components. By operating in the superconductive region, AQFP circuits achieve zero static energy dissipation [14]; by operating in the adiabatic mode, AQFP circuits achieve very small dynamic energy consumption [1]. 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 [2]. The commonly used cost metric of AQFP circuits is the 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 [2].

Logic gates in an AQFP circuit need to be activated and deactivated periodically by an excitation current [3]. 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 by 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 multifanout 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 article, we do not distinguish buffers from splitters and we will model them with the same abstraction. Also, in all figures, we use circles to represent MAJ gates and squares to represent buffers/splitters. To illustrate the AQFP technology constraints, Fig. 1 shows a full adder as a legalized AQFP circuit. Splitters ($S$ squares) are inserted to drive multiple gates and buffers ($B$ squares) are used to balance paths at the inputs of all gates and over all outputs.

### B. 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 article, we assume that an AQFP-compatible gate library (e.g., composed of AND2, OR2, and 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 $g$ on an incoming edge. Similarly, the *fanouts* of a gate (or a PI) $g$, denoted as $\mathrm{FO}(g)$, is the set of gates and POs connected to $g$ on an outgoing edge.

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 (or a PI) $n$, denoted by $\mathrm{FOT}(n)$, is the set of buffers between $n$ and any gate or PO in $\mathrm{FO}(n)$.

A *schedule* of a network is a function $\mathcal{S} : V \to \mathbb{Z}_{\geq 0}$ that assigns a non-negative integer $\mathcal{S}(n)$ to each node $n \in V$, called the *level* of $n$. The depth of a network $N = (V = I \cup O \cup G, E)$ with a schedule $\mathcal{S}$ is defined as $d(N) = \max_{o \in O} \mathcal{S}(o)$. If the schedule is omitted, then the depth of a network is the length of the longest path from any PI to any PO.

### C. Problem Formulation

To fulfill the needs in the AQFP technology for fanout-branching and path-balancing, we define the following properties subject to the *splitting capacities* $s_i = 1, s_g = 1$, and $s_b \geq 1$ of PIs, gates, and buffers, respectively.

*Definition 1:* Given a mapped network $N' = (V' = I \cup O \cup G \cup B, E')$.

1) $N'$ is *path-balanced* if there exists a schedule $\mathcal{S}$ of $N'$ such that

$$\forall n_1, n_2 \in V' : (n_1, n_2) \in E' \Rightarrow \mathcal{S}(n_1) = \mathcal{S}(n_2) - 1 \quad (1)$$
$$\forall i \in I : \mathcal{S}(i) = 0, \text{ and} \quad (2)$$
$$\forall o \in O : \mathcal{S}(o) = d(N'). \quad (3)$$

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 $s_b$.

3) $N'$ is *legal* if it is both path-balanced and properly branched.

In an AQFP design automation flow, the logic synthesis stage after RTL synthesis and before physical design converts an input specification netlist [represented as, e.g., an AND-inverter graph (AIG) or a majority-inverter graph (MIG)] into a legal mapped network whose gates are all AQFP-compatible. The problem to be solved is formulated as follows.

*Problem 1 (AQFP Technology Mapping):* Given a network $N = (V = I \cup O \cup G, E)$ with unconstrained gate types in $G$, find a mapped network $N' = (V' = I \cup O \cup G' \cup B, E')$ such that:

1) $N$ and $N'$ are logically equivalent;
2) all gates in $G'$ are of an AQFP-compatible type (i.e., AND2, OR2, or MAJ3 with optional input negation);
3) $N'$ is legal (i.e., path-balanced and properly branched).

Problem 1 may be solved as one problem, or it may be divided into two problems to be solved independently.

*Problem 2 (Majority-Based Logic Restructuring):* Given a network $N = (V = I \cup O \cup G, E)$ with unconstrained gate types in $G$, find a network $N^* = (V^* = I \cup O \cup G^*, E^*)$, such that:

1) $N$ and $N^*$ are logically equivalent;
2) all gates in $G^*$ are of an AQFP-compatible type (i.e., AND2, OR2, or MAJ3 with optional input negation).

*Problem 3 (AQFP Technology Legalization):* Given a network $N^* = (V^* = I \cup O \cup G^*, E^*)$ and the value of $s_b$, find a mapped network $N' = (V' = I \cup O \cup G' \cup B, E')$, such that:

1) $N'$ is legal (i.e., path-balanced and properly branched);
2) $G' = G^*$, and for all gates $g \in G^*$, $\mathrm{FO}(g)$ and $\mathrm{FI}(g)$ remain the same in $N'$ as in $N^*$.

Moreover, for all of the three problems, in addition to finding a network fulfilling the requirements, we also optimize for some common metrics. For the main problem to solve, Problem 1, common optimization objectives are minimizing JJ count ($\#\text{JJs} = 6 \cdot |G'| + 2 \cdot |B|$) and minimizing JJ depth $d(N')$.

Problem 2 is equivalent to mapping into and optimizing an MIG [15], which is a logic network where all gates are MAJ3 and edges may contain inverters, because AND2 and OR2 gates are equivalent to MAJ3 with a constant (0 and 1, respectively) input. Graph mapping [16] and MIG optimization [15], [17], [18] are well-researched problems with existing algorithms to use. These algorithms usually optimize for MIG size ($|G^*|$) or depth ($d(N^*)$).

In this article, we focus on solving Problem 3. Because $G' = G^*$, this problem is often referred to as the AQFP buffer (and splitter) insertion problem. Minimizing JJ count in Problem 1 is equivalent to minimizing $|B|$ in Problem 3.

## III. Related Works

In this section, we introduce existing works solving the three problems formulated in Section II-C. Section III-A corresponds to Problem 2, Section III-B corresponds to Problem 3, and Section III-C corresponds to Problem 1.

### A. Majority-Inverter Graph Optimization

MIG was proposed as an alternative technology-independent logic representation with an advantage in depth optimization especially in arithmetic circuits [15]. Due to the special properties of some emerging technologies including AQFP, MIG also becomes a good logic synthesis data structure for these technologies [19]. Various logic synthesis and optimization algorithms have been proposed and tailored for MIGs. To convert an AIG into an MIG, the simplest way is to translate each AND2 gate into an MAJ3 gate with a constant 0 input. Alternatively, a versatile graph mapping algorithm can also map from AIGs (or other types of networks) to MIGs while optimizing for depth and/or size in the process [16]. Prominent examples of tailored MIG optimization algorithms include algebraic rewriting, which applies special Boolean algebraic rules to reduce MIG depth [15], and resubstitution, which resynthesizes a small part of the network using majority gates to reduce MIG size [18], [20].

### B. Buffer and Splitter Insertion and Optimization

*(Rapid) single-flux quantum* (RSFQ or SFQ) [21], a sibling superconducting technology, shares similar path-balancing and fanout-branching constraints as AQFP and also requires buffer and splitter insertion [22], [23]. However, a key difference between the two technologies makes the problem computationally distinct for them: SFQ splitters are not clocked and thus not considered in path balancing, allowing fanout branching and path balancing to be handled separately, unlike AQFP, where clocked splitters require joint consideration of these constraints.

The earliest AQFP design automation tools legalized the circuit by inserting splitters at the output of multifanout gates, followed by buffers on imbalanced paths [9]. This naive approach guaranteed correct AQFP operation, but left out possible optimizations and often resulted in excessive B/Ss. Thus, a local optimization technique called retiming [10] or buffer merging [24] was proposed. This technique involves moving buffers across multifanin gates ([24, Fig. 8]) or multifanout splitters (in [10, Fig. 5]), reducing buffer counts by sharing buffers or delaying splitting. This idea was elaborated in [25] as a B/S insertion algorithm with the notion of virtual splitters.

Subsequent improvements in B/S optimization involved more complex algorithms. A quadratic-complexity algorithm focused on single-wire optimization, which is locally optimal subject to a complex cost function, was proposed in [26]. In [27], a schedule for the mapped network was first solved as an ILP problem, then a cubic-complexity locally optimal splitter-tree insertion algorithm was applied.

Exact methods solving for the global size-optimal B/S insertion were also researched. In [12], the B/S optimization problem was first formulated as a scheduling problem, and then encoded and solved as an optimization modulo linear integer arithmetic problem. The global minimum B/S insertion results were obtained for some small benchmarks. An ILP encoding was proposed in [28] which led to some improvement in efficiency, and optimal results for some more benchmarks were reported. Though size-optimality is still intractable, depth-optimal B/S insertion has been shown to be solvable in linear time [13].

### C. AQFP Logic Synthesis

Existing AQFP logic synthesis flows can be categorized into two approaches: solving Problems 2 and 3 separately, or considering Problems 2 and 3 together. The earliest works took the first approach to adapt available CMOS-based design automation tools for AQFP [9], [10]. Problem 2 was addressed by AND-based technology-independent logic synthesis followed by technology mapping into an AQFP-compatible library, and Problem 3 was solved separately in an additional buffer insertion stage before physical design. Later, to better leverage the intrinsic MAJ function in AQFP circuits, MAJ-based logic synthesis was adopted [19], [24]. At this time, Problem 3 was still solved separately using the naive insertion approach introduced in Section III-B.

Although solving the two problems separately is easier, it is hard to predict the impact of legalization in the logic restructuring stage. The smallest MIG in size may not be still the smallest after legalization. Thus, Marakkalage et al. [29] proposed to consider the two problems together and optimize directly for the final cost function. A database of optimal AQFP subcircuits is used in restructuring, and legalization is done during the process. This algorithm was used in a flow consisting of graph mapping, AQFP resynthesis, and post-synthesis buffer optimization [11].

The latest work on AQFP synthesis, presenting currently the best results, took the first approach (separating the two problems) and used Bayesian optimization to find the best MIG restructuring script with respect to the actual AQFP cost after legalization [30].

## IV. Buffer and Splitter Insertion

In this section, we will explain how Problem 3 shall be efficiently approached. First, in Section IV-A, we identify that the AQFP legalization (buffer and splitter insertion) problem is a scheduling problem because once a schedule is given, the minimal-size mapped network can be derived in linear time using an irredundant buffer insertion algorithm Algorithm 1. Then, in Section IV-B, depth-optimal scheduling algorithms are proposed based on the well-known as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) scheduling algorithms. These algorithms provide different starting points for the heuristic optimization algorithms that will be presented in Section V.

### A. Irredundant Buffer Insertion

*Claim 1:* The AQFP legalization problem (Problem 3) is a scheduling problem on the unmapped network.

To elaborate on the above claim, we will first introduce the notion of *irredundant* mapped network. Then, we will present Algorithm 1 to show how buffers can be inserted irredundantly given a schedule of the unmapped network.

*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 buffers whose incoming edges are connected from the same splitter and both of them have out-degrees smaller than $s_b$.

Otherwise, the network is *redundant*.

Notice that the local retiming optimization used in [10] and [25], which pushes buffers from the outputs of a splitter to its input, is subsumed by the definition of irredundant networks. In other words, if a mapped network

**Algorithm 1:** Irredundant Buffer Insertion

```
Input: An unmapped network $N^* = (V^* = I \cup O \cup G^*, E^*)$ and
       a schedule $\mathcal{S}$ for $N^*$
Output: Legalized mapped network $N'$
1  $N' \leftarrow N^*$
2  foreach $n \in I \cup G^*$ do
3      $l_{\max} \leftarrow \max_{n_o \in \mathrm{FO}(n)} \mathcal{S}(n_o)$
4      $A \leftarrow \{n_o \in \mathrm{FO}(n) : \mathcal{S}(n_o) = l_{\max}\}$
5      for $l = l_{\max} - 1$ downto $\mathcal{S}(n) + 1$ do
6          Create $\lceil(|A|/s_b)\rceil$ buffers at level $l$ in $N'$
7          $B \leftarrow$ the set of newly-created buffers
8          for $i = 1$ to $|A|$ do
9              Remove $n$ from $A[i]$'s fanins in $N'$
10             Add $B[\lceil \frac{i}{s_b} \rceil]$ as $A[i]$'s fanin in $N'$
11         $A \leftarrow B \cup \{n_o \in \mathrm{FO}(n) : \mathcal{S}(n_o) = l\}$
12     assert $|A| = 1$
13     Add $n$ as $A[1]$'s fanin in $N'$
14 return $N'$
```

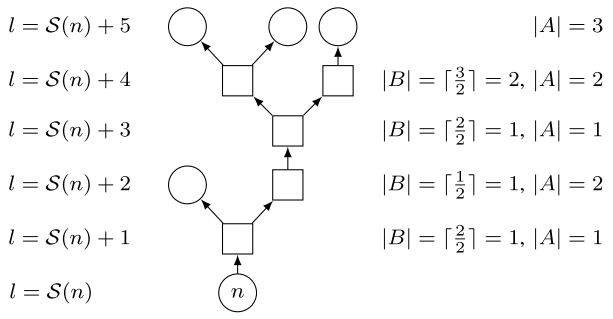
$l = \mathcal{S}(n) + 5$
$l = \mathcal{S}(n) + 4$
$l = \mathcal{S}(n) + 3$
$l = \mathcal{S}(n) + 2$
$l = \mathcal{S}(n) + 1$
$l = \mathcal{S}(n)$
$n$
$|A| = 3$
$|B| = \lceil \frac{3}{2} \rceil = 2, |A| = 2$
$|B| = \lceil \frac{2}{2} \rceil = 1, |A| = 1$
$|B| = \lceil \frac{1}{2} \rceil = 1, |A| = 2$
$|B| = \lceil \frac{2}{2} \rceil = 1, |A| = 1$

Fig. 2. Example subnetwork to illustrate Algorithm 1 ($s_b = 2$).

is irredundant, no optimization can be made with the local retiming technique. This is because local retiming looks for splitters whose fanouts are all buffers and the sum of the fanout counts of these buffers does not exceed the splitting capacity $s_b$, which violates the second condition in Definition 2.

For each PI or gate $n$, Algorithm 1 iterates over all levels $l$ between $n$ its fanouts. Initially, the set $A$ contains the fanouts (gates and POs, if any) of $n$ at the highest level $l_{\max}$. At each level $l$, enough buffers ($|B| = \lceil [|A|/s_b] \rceil$) are inserted, where $|A|$ is the number of nodes at level $l + 1$. Then, $n$ is removed from the fanins of the $i$th element in $A$, and the $\lceil [i/s_b] \rceil$th buffer in $B$ is added instead. Finally, $A$ is updated as the newly created buffers and the fanouts at the current level. Fig. 2 illustrates an example iteration (of the out-most loop) of Algorithm 1, where $s_b = 2$ is assumed.

Algorithm 1 runs in linear time with respect to $\sum_{n \in I \cup G^*} |\mathrm{FO}(n)| \leq |E^*|$. It also verifies whether it is possible to build a properly branched network with the given schedule $\mathcal{S}$. In line 12, the assertion makes sure that the gate or PI $n$ has only one outgoing edge. If this assertion does not hold, then it is impossible to construct a legal mapped network with $\mathcal{S}$ and we say that $\mathcal{S}$ is an illegal schedule. Otherwise, the constructed mapped network is properly branched if the given schedule is legal. It is also path-balanced as each node is connected to a node at exactly one level lower. Moreover, the constructed mapped network is irredundant because in each fanout tree, only the minimum number of buffers is inserted

**Algorithm 2:** Depth-Optimal Single-Node Scheduling

```
**Input**: A node $n$ and a partial schedule $\mathcal{S}$
**Output**: Level $\mathcal{S}(n)$ assigned to node $n$
1  $l_{prev} \leftarrow \max_{n_o \in \mathrm{FO}(n)} \mathcal{S}(n_o)$
2  $edges \leftarrow 0$
3  **foreach** $n_o \in \mathrm{FO}(n)$ *in a descending order of* $l \leftarrow \mathcal{S}(n_o)$ **do**
4      $splitters \leftarrow \lceil (edges/[s_b^{(l_{prev}-l)}]) \rceil$
5      $edges \leftarrow splitters + 1$
6      $l_{prev} \leftarrow l$
7  **while** $edges \neq 1$ **do**
8      $edges \leftarrow \lceil (edges/s_b) \rceil$
9      $l_{prev} \leftarrow l_{prev} - 1$
10 $\mathcal{S}(n) \leftarrow l_{prev} - 1$
11 **return** $\mathcal{S}(n)$
```

at each level $l$ and only at most one of them has fanout count smaller than $s_b$. An irredundant network is size-optimal with respect to the given schedule because no buffer can be removed while keeping the network legal.

In conclusion, a legal schedule on the unmapped network determines an irredundant and legal mapped network, therefore Problem 3 is equivalent to finding a legal schedule whose corresponding irredundant mapped network is minimal.

## B. Depth-Optimal Scheduling

In this and the next section, we present algorithms to obtain a legal schedule on an unmapped network, such that an irredundant legal mapped network can be derived using Algorithm 1. These algorithms are intended to serve as quick initial scheduling methods that will be further optimized later on Section V.

As discussed in Section II-C, common cost metrics to be considered for AQFP circuits are network size and depth. Unlike in many other technologies where circuit area and delay are often inversely related in a Pareto curve and engineers must trade one for the other, we observe that in the AQFP buffer insertion problem, the size of an irredundant mapped network correlates to the depth of the provided schedule. Intuitively, in Problem 3, the unmapped network and any mapped network have roughly the same number of paths and similar logic sharing (slight differences may only exist in how fanouts are split), and the size of a mapped network is the sum of all path lengths, which is the network depth, minus the sizes of the shared cones. In other words, a larger network depth results in longer (balanced) paths and thus larger network size. Hence, we present scheduling algorithms that also optimize for depth besides being fast (having a linear time complexity) and giving legal results.

Given a partial schedule $\mathcal{S}$ where some nodes, including $n$ but excluding all fanouts of $n$, have not been assigned a level, Algorithm 2 computes the value to be assigned to $\mathcal{S}(n)$, such that the fanout tree of $n$ has the minimum-possible height. This algorithm follows a similar strategy as compared to Algorithm 1. Variable *edges* corresponds to $|A|$ in Algorithm 1, counting the number of nodes (thus edges) needed to be connected at each level; variable *splitters* corresponds to $|B|$ in Algorithm 1, computing the number of splitters (buffers)

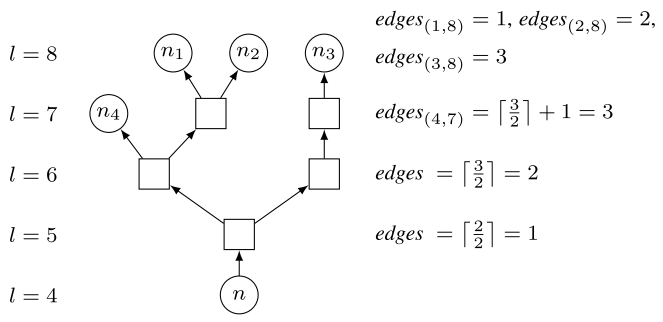
$l = 8$
$l = 7$
$l = 6$
$l = 5$
$l = 4$
$n_1$
$n_2$
$n_3$
$n_4$
$n$
$edges_{(1,8)} = 1$, $edges_{(2,8)} = 2$,
$edges_{(3,8)} = 3$
$edges_{(4,7)} = \lceil \frac{3}{2} \rceil + 1 = 3$
$edges = \lceil \frac{3}{2} \rceil = 2$
$edges = \lceil \frac{2}{2} \rceil = 1$

Fig. 3. Example subnetwork to illustrate Algorithm 2 ($s_b = 2$).

needed at each level. The foreach-loop (lines 3–6) iterates over the fanouts of $n$ in descending order of their levels, and variable $l_{\text{prev}}$ keeps the level of the previous iteration. If the level does not change from the previous to the current iteration, variable *splitters* is equal to *edges* because $l_{\text{prev}} = l$ and ${s_b}^0 = 1$ (line 4). As a result, *edges* is simply increased by 1 in this iteration, counting the fanout itself (line 5). Otherwise, when a fanout at a lower level is encountered, we compute the minimum number of buffers needed at level $l$ to drive *edges* nodes at level $l_{\text{prev}}$ as follows. A complete binary tree of height $h$ has at most $2^h$ leaves. Similarly, a splitter tree rooted at level $l$ can split into at most ${s_b}^h$ fanouts at level $l + h$. To drive *edges* fanouts at level $l_{\text{prev}}$, at least $\lceil[edges/(s_b^{(l_{\text{prev}}-l)})]\rceil$ splitter trees rooted at level $l$ are needed (line 4). Moreover, at most one of them is not full, i.e., they are irredundant. In line 5, this value, plus one for the fanout itself, is used to update variable *edges*. Finally, after all fanouts of $n$ have been processed, the algorithm finds the highest level where *edges* is one to schedule $n$ (lines 7–10).

Fig. 3 shows an example to illustrate Algorithm 2. The node $n$ to be scheduled has four fanouts, assigned, respectively, to levels 8 ($n_1$, $n_2$, $n_3$) and 7 ($n_4$) in the partial schedule. The splitting capacity is $s_b = 2$. In Fig. 3, $edges_{(v,l)}$ indicates the value of variable *edges* in Algorithm 2 when node $n_v$ at level $\mathcal{S}(n_v) = l$ is considered in the foreach-loop (lines 3–6). First, $edges_{(1,8)} = 1$, $edges_{(2,8)} = 2$, and $edges_{(3,8)} = 3$ are computed, essentially counting the number of fanouts at level $l = 8$. When node $n_4$ at a lower level, $l = 7$, is encountered, the number of buffers needed at level 7 to drive all nodes at the previously considered level $l_{\text{prev}} = 8$ is computed by $\lceil 3/2^{8-7} \rceil = 2$. The loop ends with $l_{\text{prev}} = 7$ and $edges = 3$. Finally, in the while-loop (lines 7–9), *edges* is updated two times before it reaches value 1, resulting in $l_{\text{prev}} = 5$. Thus, node $n$ is scheduled at $\mathcal{S}(n) = 4$.

With the following lemma, we show that the computation in line 4 of Algorithm 2 has the equivalent effect in Algorithm 1 on $|A|$ as $l_{\text{prev}} - l$ iterations of line 6.

*Lemma 1:* Let $b$ be a positive integer and $A = a_0, a_1, \ldots, a_n$ be a sequence of $n + 1$ positive integers related by $a_{i+1} = \lceil (a_i/b) \rceil, 0 \leq i < n$. Then, $a_n = \lceil (a_0/b^n) \rceil$.

*Proof:* We first prove that for any positive integers $a$ and $b$, $\lceil (\lceil (a/b) \rceil / b) \rceil = \lceil (a/b^2) \rceil$. Let $x = \lceil (a/b) \rceil$, by definition, we have

$$\frac{a}{b} \leq x \Rightarrow \frac{a}{b^2} \leq \frac{x}{b} \Rightarrow \frac{a}{b^2} \leq \frac{x}{b}.$$

Suppose, for the sake of contradiction, that $\lceil (a/b^2) \rceil < \lceil (x/b) \rceil$, then there must exist an integer $y$ such that $(a/b^2) \leq y < (x/b)$. Multiplying by $b$ and using the definitions, we have

$$\frac{a}{b} \leq b \cdot y < x = \frac{a}{b} \leq b \cdot y = b \cdot y < x$$

which leads to an absurd statement of $x < x$. Thus, by contradiction, $\lceil (a/b^2) \rceil = \lceil (x/b) \rceil = \lceil (\lceil (a/b) \rceil / b) \rceil$ and the statement is proved by induction on $i$. ■

Next, we prove the legality and optimality of Algorithm 2 with the following lemma.

*Lemma 2:* Given a legal partial schedule $\mathcal{S}$, Algorithm 2 assigns the largest value to $\mathcal{S}(n)$ such that $\mathcal{S}$ is still legal.

*Proof:* Let the value returned by Algorithm 2 be $l_n$ and assume, for the sake of contradiction, a schedule $\mathcal{S}'$, where $\mathcal{S}'(n_o) = \mathcal{S}(n_o)\ \forall n_o \in \text{FO}(n)$ and $\mathcal{S}'(n) = l'_n > l_n$. Let $l_m = \min_{n_o \in \text{FO}(n)} \mathcal{S}(n_o)$. If $l'_n \geq l_m$, $\mathcal{S}'$ is obviously illegal. Assume $l'_n < l_m$. Let $e$ be the value of variable *edges* when the foreach-loop in Algorithm 2 (lines 3–6) ends. The while-loop in Algorithm 2 has $l_m - l_n - 1$ iterations, so the value of variable *edges* before the last iteration is, by Lemma 1, $\lceil e/s_b^{(l_m - l_n - 2)} \rceil > 1$.

Now, consider an execution of Algorithm 1 using $\mathcal{S}'$, in particular, the iteration of the outer loop processing the considered node $n$, we have $|A| = e$ after line 11 in Algorithm 1 in the iteration $l = l_m - l'_n$. The loop (lines 5–11 in Algorithm 1) has $l_m - l'_n - 1$ more iterations before it ends, in which line 11 can be replaced by "$A \leftarrow B$" because there are no more fanouts. By the end of the loop, $|A| = \lceil e/s_b^{(l_m - l'_n - 1)} \rceil \geq \lceil e/s_b^{(l_m - l_n - 2)} \rceil > 1$. Thus, we conclude that $\mathcal{S}'$ is illegal, and $l_n$ is indeed the largest possible value for $\mathcal{S}(n)$. ■

By corollary, if all fanouts of a node $n$ are scheduled at the largest level, then the level of $n$ obtained by Algorithm 2 is also the largest. Formally written as follows.

*Corollary 1:* Given a legal schedule $\mathcal{S}$ and a node $n$, let $\mathcal{S}(n)$ be the level of $n$ computed by Algorithm 2. If there does not exist a legal schedule $\mathcal{S}'$ such that $\max_{o \in O} \mathcal{S}'(o) = \max_{o \in O} \mathcal{S}(o)$ and $\exists n_o \in \text{FO}(n), \mathcal{S}'(n_o) > \mathcal{S}(n_o)$, then there does not exist a legal schedule $\mathcal{S}'$ such that $\max_{o \in O} \mathcal{S}'(o) = \max_{o \in O} \mathcal{S}(o)$ and $\mathcal{S}'(n) > \mathcal{S}(n)$.

Algorithm 2 requires that a node is only scheduled after all of its fanouts have been scheduled. In other words, a reversed topological order is required. Thus, it is suitable to use an ALAP scheduling scheme, which first schedules all POs of a network to an upper bound $\lambda$, and then schedules the remaining nodes to the largest-possible level ("ALAP") in a reversed topological order. We present Algorithm 3 for this purpose. It first computes a sufficiently large upper bound $\lambda$ on the depth of the mapped network for ALAP scheduling, assuming each node would need a balanced splitter tree to drive the maximum fanout in the network. POs are first scheduled at $\lambda$. Then, each node is scheduled using Algorithm 2 in a reversed topological order. Finally, to obtain a schedule independent of the value of $\lambda$, post-scheduling correction is applied: PIs are moved to level 0 to fulfill (2), and the levels of all other nodes are reduced by the smallest PI level before correction. This algorithm has a linear time complexity with respect to the network size.

**Algorithm 3:** Depth-Optimal ALAP Scheduling

```
Input: An unmapped network $N^* = (V^* = I \cup O \cup G^*, E^*)$
Output: A schedule $\mathcal{S}$ for $N^*$
1  $\lambda \leftarrow d(N^*) \cdot (1 + \max_{n \in V^*} \lceil \frac{\log(|\text{FO}(n)|)}{\log(s_b)} \rceil)$
2  foreach $o \in O$ do
3      $\mathcal{S}_\lambda(o) \leftarrow \lambda$
4  foreach $n \in I \cup G^*$ in a reversed topological order do
5      $\mathcal{S}_\lambda(n) \leftarrow$ schedule_node($n$, $\mathcal{S}_\lambda$)        // alg. 2
6  $l_{\min} \leftarrow \min_{i \in I} \mathcal{S}_\lambda(i)$
7  foreach $i \in I$ do
8      $\mathcal{S}(i) \leftarrow 0$
9  foreach $n \in O \cup G^*$ do
10     $\mathcal{S}(n) \leftarrow \mathcal{S}_\lambda(n) - l_{\min}$
11 return $\mathcal{S}$
```

We have shown with Corollary 1 that the depth-optimal scheduling problem has optimal substructure when nodes are scheduled in a reversed topological order. Now, we can prove that Algorithm 3 achieves optimal depth.

*Theorem 1:* Given an unmapped network $N^*$, let the schedule for $N^*$ returned by Algorithm 3 be $\mathcal{S}$. The irredundant mapped network $N'$, obtained by running Algorithm 1 with $N^*$ and $\mathcal{S}$ as inputs, is legal and its depth $d(N')$ is minimal.

*Proof:* At line 6 of Algorithm 3, the depth of schedule $\mathcal{S}_\lambda$ is $\max_{o \in O} \mathcal{S}_\lambda(o) = \lambda$ by definition. After the correction in lines 6–10, the maximum level becomes $\lambda - l_{\min}$, which is also the resulting depth $d(N')$. Thus, minimizing depth $d(N')$ is equivalent to maximizing the lowest PI level $l_{\min}$ during scheduling because $\lambda$ is a constant.

In Algorithm 3, levels of POs are maximized to $\lambda$. By Corollary 1, each node is scheduled at the largest level because all of its fanouts are scheduled before it and they are also scheduled at their largest possible levels. By induction, levels of all nodes are maximized and thus depth is minimized. The legality of $\mathcal{S}$ is similarly proved by Lemma 2. ■

In conclusion, Algorithm 3 guarantees to find a legal schedule for an unmapped network. Followed by Algorithm 1, a legal mapped network is obtained in linear time. By Theorem 1, such mapped network is depth-optimal.

## C. Alternative Depth-Optimal Scheduling

The methods presented in the previous section give a depth-optimal mapped network, but size optimality is not guaranteed. Indeed, the AQFP size optimization problem is likely a difficult one without an algorithm that is both optimal and has polynomial time complexity. Thus, we propose to use depth-optimal networks as starting points and further optimize for size with heuristic algorithms presented in Section V. As heuristics are often biased by the starting point, we present in this section an alternative depth-optimal scheduling method based on ASAP instead of ALAP scheduling.

An ASAP scheduling scheme schedules each node, in a topological order, to the lowest-possible level according to the schedule of its fanins. To do so, we define a *mobility function* $\mu : V^* \to \mathbb{Z}_{\geq 0}$ representing the maximum negative displacement that can be made to a node [from $\mathcal{S}_{\text{ALAP}}(n)$ to

**Algorithm 4:** Depth-Optimal ASAP Scheduling

```
Input: An unmapped network $N^* = (V^* = I \cup O \cup G^*, E^*)$ and
       its ALAP schedule $\mathcal{S}_{\mathrm{ALAP}}$
Output: ASAP schedule $\mathcal{S}_{\mathrm{ASAP}}$ for $N^*$
1  foreach $i \in I$ do
2      $\mu(i) \leftarrow 0$
3  foreach $n \in G^*$ do
4      $\mu(n) \leftarrow \infty$
5  $\mathcal{S}_{ASAP} \leftarrow \mathcal{S}_{ALAP}$
6  foreach $n \in G^*$ in a topological order do
7      $\mathcal{S}_{ASAP}(n) \leftarrow \mathcal{S}_{ASAP}(n) - \mu(n)$
8      foreach $n_o \in FO(n)$ do
9          $T(n_o) \leftarrow 0$
10     $l_{prev} \leftarrow \max_{n_o \in \mathrm{FO}(n)} \mathcal{S}_{ASAP}(n_o)$
11     $edges \leftarrow 0$
12     foreach $l = \mathcal{S}(n_o) : n_o \in \mathrm{FO}(n)$ in descending order do
13         $mobility \leftarrow 0$
14         for $l_{prev} - l$ iterations do
15             if $edges = 1$ then
16                 $mobility \leftarrow mobility + 1$
17             $edges \leftarrow \lceil \frac{edges}{s_b} \rceil$
18         foreach $n'_o \in \mathrm{FO}(n) : \mathcal{S}(n'_o) > l$ do
19             $T(n'_o) \leftarrow T(n'_o) + mobility$
20         $edges \leftarrow edges + 1$
21         $l_{prev} \leftarrow l$
22     $mobility \leftarrow 0$
23     for $l = \mathcal{S}_{ASAP}(n)$ upto $l_{prev} - 2$ do
24         if $edges = 1$ then
25             $mobility \leftarrow mobility + 1$
26         $edges \leftarrow \lceil \frac{edges}{s_b} \rceil$
27     foreach $n_o \in \mathrm{FO}(n)$ do
28         $\mu(n_o) \leftarrow \min(\mu(n_o), T(n_o) + mobility)$
29 return $\mathcal{S}_{ASAP}$
```

$\mathcal{S}_{\mathrm{ASAP}} = \mathcal{S}_{\mathrm{ALAP}}(n) - \mu(n)]$ while keeping the schedule legal and depth-optimal. Algorithm 4 computes (a lower bound on) the mobility of each node and uses these values to obtain an ASAP schedule using a given ALAP schedule.

Mobility is initialized to infinite for gates and to 0 for PIs (lines 1–4). For each node $n$ in topological order, first, $n$ is scheduled to a lower level based on its ALAP schedule and mobility (line 7). Then, the mobilities of its fanouts are updated using a similar computation as in Algorithm 2. A map $T$ stores the temporary mobilities of the fanouts of $n$, initialized to zero (lines 8 and 9). The foreach-loop in lines 12–21 is similar to lines 3–6 in Algorithm 2, except that the computation of variable *splitters* in Algorithm 2 is rewritten as a loop (lines 14–17) to compute the local mobility (variable *mobility*), which is the number of buffers needed to balance the splitter tree from level $l_{\mathrm{prev}}$ to $l$, and is added to the temporary mobilities $T$ of all the processed fanouts (lines 18 and 19). Again, the for-loop in lines 23–26 is similar to lines 7–9 in Algorithm 2, where the local mobility is also similarly computed. Finally, $\mu$ is updated for each fanout, but to guarantee a legal schedule, it is only updated if the computed temporary mobility is smaller (lines 27 and 28). In other words, from the perspective of $n_o$, the minimum mobility among the values computed via its different fanins as $n$ will be taken.

## V. Buffer and Splitter Optimization

The scheduling-based legalization approach presented in the previous section allows us to find one (or two) legal mapped network that is (are) depth-optimal. In some scenarios, this may already be good enough, but it is still possible to further optimize the obtained mapped network to reduce its size. In this section, given a mapped network, we attempt to find a better schedule to minimize $|B|$. Two orthogonal heuristic algorithms are proposed in Sections V-A and V-B, and then combined as a portfolio flow in Section V-C.

### A. Chunked Movement

The *chunked movement* technique attempts to move groups of nodes up or down to reduce the total number of buffers. *Moving* a gate $g$ up (down) by $l$ levels means that $\mathcal{S}(g)$ is increased (resp., decreased) by $l$ while the levels of the other gates remain the same. During the process, we always ensure that the network is legal and buffers are inserted irredundantly using Algorithm 1. A movement is *legal* if the network remains legal after the movement. For example, if a gate $g$ has a fanout $g_o$ at level $\mathcal{S}(g_o) = \mathcal{S}(g) + 1$, 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 level $\mathcal{S}(g) + 1$ is not legal because there must be a splitter occupying the only outgoing edge of $g$ at $\mathcal{S}(g) + 1$. 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 together might eventually lead to further reduction. Thus, we propose to identify groups of connected gates and move them together as *chunks*, defined as follows.

A gate $g$ and one of its fanouts $g_o \in \mathrm{FO}(g)$ are said to be *close* if either one of the following conditions holds.

1) $|\mathrm{FO}(g)| = 1$ and $\mathcal{S}(g_o) = \mathcal{S}(g) + 1$.
2) $|\mathrm{FO}(g)| > 1$ and $\mathcal{S}(g_o) = \mathcal{S}(g) + 2$.

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

Algorithm 5 illustrates how a chunk is identified. Starting from an initial gate $g_0$, a chunk is formed by exploring its fanins and fanouts and adding gates into the chunk if they are close (line 8), or recording an II or OI otherwise (line 11). When a new gate is added to the chunk, its fanins and fanouts are also explored (line 9). The queue $Q$ stores the edges to be checked next.

By definition, a chunk has flexibilities at all of its interfaces. Moreover, the set of all chunks in a mapped network forms a partitioning of all gates. Fig. 4 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,

**Algorithm 5:** Chunk Construction

```
   Input: An initial gate $g_0$
   Output: A chunk $C$ and its interfaces $T$
1  $C \leftarrow \{g_0\}$
2  $Q \leftarrow \{(g_0, g) : g \in \mathrm{FI}(g_0) \cup \mathrm{FO}(g_0)\}$
3  $T \leftarrow \emptyset$
4  while $Q \neq \emptyset$ do
5      $(g_c, g_e) \leftarrow \mathrm{pop}(Q)$
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          $Q \leftarrow Q \cup\ \{(g_e, g) : g \in \mathrm{FI}(g_e) \cup \mathrm{FO}(g_e)\}$
10     else
11         $T \leftarrow T \cup \{(g_c, g_e)\}$
12 return $C, T$
```

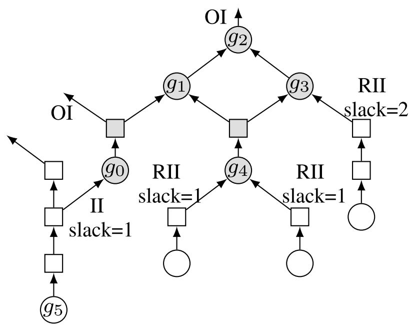

Fig. 4. Example subnetwork showing a chunk (in gray).

although the gate $g_0$ can be legally moved down, moving it alone would only incur more buffers. However, if the entire chunk is moved down together by one level, one buffer is saved, which is analyzed as follows.

To see how many levels a chunk can be moved, a *slack* is computed at each interface. For an II $(g_c, g_e)$

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

For an OI, $g_c$ and $g_e$ are exchanged in (4). When trying to move a chunk down, the maximum number of levels we can move is the minimum slack at all IIs; when moving a chunk up, it is the minimum slack at all OIs.

We further classify IIs as relevant or not. An II $(g_c, g_e)$ is said to be a relevant II (RII) if

$$\forall g_o \in \mathrm{FO}(g_e), g_o \notin C : \mathcal{S}(g_o) > \mathcal{S}(g_c). \tag{5}$$

For example, in Fig. 4, $(g_0, g_5)$ is not an RII because $g_5$ has another fanout at a higher level than $\mathcal{S}(g_0)$, so when $g_0$ is moved, no buffer is added or eliminated at this interface.

We decide to move a chunk up or down on whether there are more OIs or RIIs. If a chunk has $x$ OIs and $y$ RIIs, moving the chunk up by $l$ levels eliminates $l \cdot (x - y)$ buffers (if $x > y$), and moving the chunk down eliminates $l \cdot (y - x)$ buffers (if $y > x$). In Fig. 4, there are three RIIs and two OIs, and the minimum slack at all IIs is 1, thus moving the chunk down by 1 level reduces 1 buffer.

Overall, the chunked movement algorithm iteratively constructs a chunk using Algorithm 5 for each node that 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.

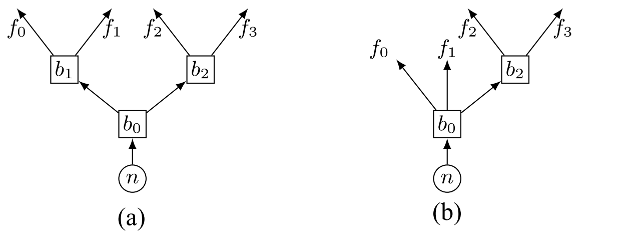

Fig. 5. Example subnetwork for retiming ($s_b = 3$). (a) Before retiming. (b) After moving $b_1$.

## B. Retiming

The optimization of B/Ss in an AQFP circuit is reminiscent of the register minimization problem called retiming. *Minimum register retiming* is the problem of relocating the registers of a circuit in order to minimize their number while preserving the functionality. Retiming is formulated as a linear problem dual to the minimum-cost flow problem for which many polynomial algorithms exist [31].

In this section, we propose the AQFP B/S retiming algorithm, which minimizes B/Ss in an AQFP network, similar to how registers are minimized in minimum register retiming. Previous work applied a retiming-like optimization to AQFP logic [10], [25]. However, their approach does not perform global retiming but moves buffers locally from the output of splitters to the input. This optimization is subsumed by Algorithm 1 in the definition of irredundant mapped networks.

Minimizing the number of buffers can be seen as maximizing sharing of buffers on multiple paths. Without accounting for fanout-branching, e.g., assuming that buffers have an infinite splitting 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. As mentioned in Section III-B, previous works successfully applied this idea to the RSFQ technology family [22], but when the fanout-branching constraint in AQFP comes into consideration, splitter relocation is conditional on respecting the splitting capacity. Hence, retiming is only a heuristic for AQFP B/S optimization instead of an optimal algorithm.

Fig. 5(a) shows an example mapped subnetwork under retiming, where $s_b = 3$ is assumed. This subnetwork is redundant because $b_1$ and $b_2$ have out-degree $2 < s_b$ (Definition 2). Indeed, a mapped network can become redundant temporarily during retiming. Not all buffers can be retimed at the same time, and this example shows two such cases. First, $b_0$ cannot be retimed because its movement would increase the fanout count of $n$ to 2, violating the fanout constraint of gates ($s_g = 1$). Second, only one of the splitters $b_1$ and $b_2$ can be selected for retiming since the movement of both of them would increase the fanout count of $b_0$ to 4, violating the fanout constraint of buffers ($s_b = 3$). Also, fanouts of splitters in the same fanout tree originating from the same gate are exchangeable, and such exchanges may affect possible retiming optimizations. For example, instead of $\mathrm{FO}(b_1) = \{f_0, f_1\}$, $\mathrm{FO}(b_2) = \{f_2, f_3\}$ in Fig. 5(a), $\mathrm{FO}(b_1) =$

**Algorithm 6:** B/S Retiming

```
    Input: Mapped network $N'$
    Output: Optimized mapped network $N'$
 1  while improvement do
 2      select_retimeable_buffers($N'$)
 3      set up retiming direction to forward
 4      maximum_flow($N'$)
 5      get_minimum_cut($N'$)
 6      $N' \leftarrow$ move_retimed_buffers($N'$)
 7  while improvement do
 8      select_retimeable_buffers($N'$)
 9      set up retiming direction to backwards
10      maximum_flow($N'$)
11      get_minimum_cut($N'$)
12      $N' \leftarrow$ move_retimed_buffers($N'$)
13  $N' \leftarrow$ reconstruct_fanout_trees($N'$)
14  return $N'$
```

$\{f_0, f_2\}$, $\text{FO}(b_2) = \{f_1, f_3\}$ is also possible and may unlock more retiming on $b_1$ and $b_2$. Fig. 5(b) shows the fanout tree after the relocation of splitter $b_1$ to its transitive fanout cone (not shown).

The B/S retiming algorithm is shown in Algorithm 6, which takes a legal mapped network as input and outputs an optimized mapped network. The retiming problem is formulated as a binary maximum-flow problem similar to [32], which separates flow computation into forward and backward directions. The algorithm performs two optimization loops in both directions until no more improvements can be made. A loop starts by selecting buffers to be retimed (lines 2 and 8), which are buffers that can be relocated without exceeding the splitting capacity of its fanin node. In the case of mutually exclusive selections (i.e., two splitters cannot be retimed at the same time), one is picked randomly. Each selected buffer is a source and a sink of a unitary flow. Next, the algorithm proceeds by selecting the retiming direction (lines 3 and 9), computing the binary maximum flow using the augmenting path algorithm (lines 4 and 10), getting the minimum cut (lines 5 and 11) and moving the selected buffers to the new position if there is a reduction (lines 6 and 12). Since retiming movements may create redundant fanout trees, the algorithm terminates by reconstructing each fanout tree irredundantly using Algorithm 1 (line 13).

An example of a forward retiming iteration is depicted in Fig. 6. Fig. 6(a) shows the initial subnetwork, where $s_b = 3$. The algorithm selects the buffers in orange to perform retiming. Fig. 6(b) shows the optimized subnetwork after retiming. Two new buffers are inserted (in green). The number of buffers is reduced from 6 to 5 while maintaining the same path lengths.

### C. Buffer and Splitter Optimization Flow

Algorithm 7 describes our optimization flow. It combines chunked movement and retiming to achieve better results than the individual algorithms. Additionally, we use a deterministic randomization function to select a different topological order and to rearrange the fanout of nodes in the network. This helps explore different local optima heuristic retiming may fall into

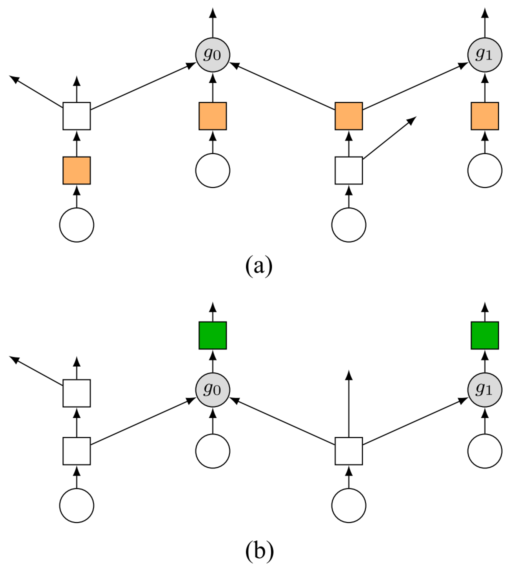

Fig. 6. Example of forward retiming ($s_b = 3$). (a) Initial subnetwork. (b) Optimized subnetwork.

**Algorithm 7:** Buffer and Splitter Optimization

```
   Input: Mapped network $N'_{\text{init}}$
   Output: Optimized mapped network $N'_{\text{opt}}$
1  $N'_{\text{tmp}} \leftarrow$ bs_retiming($N'_{\text{init}}$)            // alg. 6
2  repeat
3      $N'_{\text{opt}} \leftarrow N'_{\text{tmp}}$
4      $N'_{\text{tmp}} \leftarrow$ chunked_movement($N'_{\text{opt}}$)    // alg. 5
5      $N'_{\text{tmp}} \leftarrow$ bs_retiming($N'_{\text{tmp}}$)         // alg. 6
6      $N'_{\text{tmp}} \leftarrow$ randomize($N'_{\text{tmp}}$)
7  until $|N'_{tmp}| \geq |N'_{opt}|$
8  return $N'_{opt}$
```

by randomizing the tie-breaking mechanism in the algorithm. Specifically, the order of processing different fanouts of a gate sometimes leads to different retiming results due to the first-come–first-serve nature of filling a splitter's s_b fanout slots. This can be considered as local heuristic choices made by the algorithm, whose goodness can only be evaluated after the global optimization is done and is hard to predict during the optimization process. Empirically, adding such randomization in the optimization flow enhances the optimization quality of some benchmarks, providing a few percent further reductions on the buffer count on average.

## VI. AQFP Logic Synthesis Toolbox

As discussed in Section II-C, the AQFP technology mapping problem (Problem 1) can be divided into two subproblems: 1) MIG restructuring (Problem 2) and 2) AQFP legalization (Problem 3). Solving the two subproblems together, such as the resynthesis algorithm in [29], leads to a high problem complexity, thus has to rely on a precomputed database and is only locally optimal. Hence, we propose to solve the two subproblems untangled, but mixed and interleaved in multiple iterations to enhance quality of result. It is essential for the algorithms used to solve both subproblems to be efficient, such

**Algorithm 8:** AQFP Technology Legalization Flow (Solves Problem 3)

```
Input: MIG network $N^*$
Output: Mapped network $N'$
1  $\mathcal{S}_{\text{ALAP}} \leftarrow$ ALAP($N^*$)                                   // alg. 3
2  $\mathcal{S}_{\text{ASAP}} \leftarrow$ ASAP($N^*$, $\mathcal{S}_{\text{ALAP}}$)                  // alg. 4
3  $N'_{\text{ALAP}} \leftarrow$ insert_buffers($N^*$, $\mathcal{S}_{\text{ALAP}}$)      // alg. 1
4  $N'_{\text{ASAP}} \leftarrow$ insert_buffers($N^*$, $\mathcal{S}_{\text{ASAP}}$)      // alg. 1
5  $N'_{\text{ALAP}} \leftarrow$ optimize($N'_{\text{ALAP}}$)                     // alg. 7
6  $N'_{\text{ASAP}} \leftarrow$ optimize($N'_{\text{ASAP}}$)                     // alg. 7
7  if $|N'_{ALAP}| < |N'_{ASAP}|$ then
8      return $N'_{ALAP}$
9  else
10     return $N'_{ASAP}$
```

that more iterations can be done in reasonable runtime and achieve better quality.

In this section, we first present a scalable and efficient AQFP technology legalization flow combining the proposed scheduling, buffer insertion, and buffer optimization algorithms. Then, we present an AQFP technology mapping solution combining existing MIG optimization methods and the proposed AQFP technology legalization flow with an on-the-fly design space exploration methodology. Finally, we discuss how verification is done throughout the process.

## *A. Technology Legalization Flow*

In Section IV, we presented algorithms to obtain an initial scheduling (Section IV-B) and to insert buffers irredundantly (Section IV-A). In Section V, we presented optimization algorithms to further reduce the buffer count of a mapped network. Combining everything together, a technology legalization flow is presented in Algorithm 8. Two initial scheduling, ALAP and ASAP, are obtained with the depth-optimal scheduling algorithms and result in two mapped networks by inserting buffers irredundantly. Then, the two mapped networks are optimized independently using the portfolio optimization flow. Finally, the better one with a smaller size is adopted.

## *B. Design Space Exploration for AQFP Technology Mapping*

Imagine a design space consisting of all legal and logically equivalent mapped networks, the optimization problem of AQFP technology mapping is to find the best one in the design space in terms of a cost metric (usually, JJ count or depth). Performing MIG restructuring and AQFP legalization can be seen as moving along two orthogonal directions (or axes) in the design space, exploring first different logically equivalent MIGs without buffers, and then different mapped networks corresponding to the same MIG. This approach confines the degree of freedom of the exploration in order to be more scalable and potentially explore a larger space within the confined regions. However, if the two axes are only explored once each, then still only a small subset of the entire space is explored and the result may be far from the global optimal. The major problem is that during MIG restructuring, buffers are not inserted yet and the algorithm can only decide on the best moves based on a truncated cost metric (usually, MIG size

**Algorithm 9:** AQFP Technology Mapping With Design Space Exploration (Solves Problem 1)

```
Input: Unconstrained network $N$
Output: Optimized mapped network $N'$
1  $N_0^* \leftarrow$ map_into_MIG($N$)                                    // [16]
2  $N_{best}^* \leftarrow$ copy($N_0^*$)
3  $best\_cost \leftarrow \infty$
4  for $restart = 1$ upto $num\_restarts$ do
5      $N_{best\_inner}^* \leftarrow N_0^*$
6      $N_{curr}^* \leftarrow N_0^*$
7      $best\_cost\_inner \leftarrow \infty$
8      $rnd \leftarrow$ new_random_engine()
9      $timer \leftarrow$ start_timer()
10     for $step = 1$ upto $max\_steps$ do
11         $N_{curr}^* \leftarrow$ restructure_MIG_randomly($N_{curr}^*$, $rnd$)
                   // [15], [16], [18], [33], [34]
12         $curr\_cost \leftarrow$ evaluate(legalize($N_{curr}^*$))     // alg. 8
13         if $curr\_cost < best\_cost\_inner$ then
14             $N_{best\_inner}^* \leftarrow N_{curr}^*$
15             $best\_cost\_inner \leftarrow curr\_cost$
16             $last\_impr \leftarrow step$
17         if $step - last\_impr \geq max\_no\_impr$ then break if
           elapsed_time($timer$) $\geq timeout$ then break
18     if $best\_cost\_inner < best\_cost$ then
19         $N_{best}^* \leftarrow N_{best\_inner}^*$
20         $best\_cost \leftarrow best\_cost\_inner$
21 $N' \leftarrow$ legalize($N_{best}^*$)                                   // alg. 8
22 return $N'$
```

or depth) which does not completely correlate to the actual cost metric.

We propose a design space exploration approach, illustrated in Algorithm 9, which performs multiple iterations of MIG restructuring and legalizes the MIGs in every iteration to compute the actual JJ cost, such that the exploration is correctly guided. As formulated in Problem 1, the input is an unconstrained network $N$, so we first map it into an MIG network (line 1). In the rest of the algorithm, four copies of the MIG are maintained: the initial MIG $N_0^*$, the overall best MIG $N_{\text{best}}^*$, the best MIG in the inner for-loop $N_{\text{best\_inner}}^*$, and the current MIG $N_{\text{curr}}^*$. The algorithm explores the design space by starting *num_restarts* times from the initial point $N_0^*$ (the outer for-loop, lines 4–21), each time exploring MIGs along a random trajectory (the inner for-loop, lines 10–18). For each MIG, the second axis of different mapped networks is also explored, and the cost is evaluated on the best mapped network (line 12). The best-seen MIGs are book-marked on the current trajectory ($N_{\text{best\_inner}}^*$, line 14) and on all trajectories ($N_{\text{best}}^*$, line 20). The inner loop is terminated when no improvement is observed for *max_no_impr* steps consecutively (line 17), or when the timeout limit is exceeded (line 18). These two parameters set the effort level of each restart and are the major factors determining the tradeoff between runtime and quality of results.

The key ingredients of Algorithm 9 are the functions map_into_MIG (line 1), restructure_MIG_randomly (line 11), and legalize (lines 12 and 22). In line 1, function map_into_MIG calls a graph mapping algorithm [16]. In the case where $N$ is an AIG, it can also be transformed directly into an MIG by converting each AND2 into a MAJ3 with a constant 0 fanin. In line 11, function restructure_MIG_randomly applies a randomly chosen MIG restructuring script. In our experience, scripts that perform well consist of a drastic restructuring step, such as mapping into $k$-LUT network [33] and then remapping into MIG [16], followed by some MIG optimization steps, such as resubstitution [18], [20], algebraic rewriting [15], and balancing [34]. In line 12, the current MIG is legalized using the proposed legalization flow Algorithm 8 to obtain a mapped network $N'$ for evaluation. Depending on the design objective, the function evaluate may return the JJ count (#JJs $= 6\cdot|G'|+2\cdot|B|$), depth ($d(N')$), or (EDP $=$ #JJs $\cdot\, d(N')$). Line 22 legalizes the best MIG again also using Algorithm 8. If better runtime efficiency is desired, lighter-effort legalization (e.g., by limiting the number of optimization iterations in Algorithm 7) can be used in line 12 for cost evaluation while keeping the final legalization in line 22 the highest-affordable effort.

The advantages of this design space exploration approach are twofold. First, compared to existing approaches, it explores a larger design space, and the frontier of exploration also stretches further. This is thanks to the hill-climbing strategy, where we simply record the best-seen design on the trajectory and keep moving forward when the cost gets worse instead of rolling back. Moreover, the key enabling factor to explore on the orthogonal axis (different mapped networks from the same MIG) is that the legalization runtime is fast enough, which motivates the focus of this article on efficient heuristic buffer optimization methods instead of unscalable exact algorithms. The second advantage of Algorithm 9 is that the design space exploration is done on the fly. That is, no heavy data training, complicated decision making, or human expert intuition is needed to guide the exploration, and the results are not overfitted for a subset of benchmarks. The direction of exploration is guided by the simplest strategy, randomness, and the best transformation sequence is discovered on the fly. As there is a factor of luck involved, the purpose of the outer loop is to mitigate the possibility of a "bad" random seed leading to unsatisfactory results and to increase the chance of meeting at least one "good" random sequence in all restarts. Setting the number of restarts higher would increase the chance of obtaining an even better result, but such improvement saturates at some point and the additional runtime is wasted in repeating similar optimization sequences. Empirically, we have found that setting *num_restarts*$=5$ reaches a good balance between runtime and quality.

### C. Verification

To ensure the correct functionality of the synthesized AQFP circuit, two types of verification should be performed: logic equivalence to the specification and legality with respect to the AQFP technology constraints. These correspond to the first and the third condition in Problem 1. The second condition, i.e., only AQFP-compatible gates are used, is ensured automatically by having used MIG as logic representation in the restructuring step.

For logic equivalence, we apply the well-developed combinational equivalence checking algorithm [35] on the mapped network $N'$ and the original network $N$. For legality verification, we check if the mapped network is indeed path-balanced and properly branched. First, a schedule $\mathcal{S}$ of the mapped network is (re)computed by visiting all nodes in a topological order and assigning

$$\mathcal{S}(n) = \begin{cases} 0, & \text{if } n \in I \\ \max_{n_i\in \mathrm{FI}(n)} \mathcal{S}(n_i) + 1, & \text{otherwise.} \end{cases} \quad (6)$$

Then, we verify if $N'$ is path-balanced by traversing all nodes again and testing (1)–(3). The "for all edges" in (1) is equivalent to checking all fanins $n_1$ of all gates $n_2$. Finally, we verify if $N'$ is properly branched by comparing the number of fanouts of all PIs, gates, and buffers against the parameters $s_i$, $s_g$, and $s_b$, respectively. With our data structure and constraint formulation, the AQFP technology legality verification can be done in linear time.

## VII. Experimental Results

All of the algorithms and flows presented in this article are implemented in the open-source C++ logic synthesis library *mockturtle*[1] [36]. In this section, we present experimental results of our methods solving Problem 3 alone Section VII-A as well as solving the bigger Problem 1 Section VII-C. We also demonstrate in Section VII-B the scalability of the proposed B/S insertion algorithm using much bigger benchmarks. To be consistent with previous works that we compare to, we use $s_b = 4$ for the splitting capacity of buffers. All results are verified with the verification methods described in Section VI-C and published[2] for third-party verification.

### A. Technology Legalization and Buffer Optimization

First, we compare the performance of our B/S insertion and optimization flow Algorithm 8 against the SoTA on solving the same problem [27]. For the sake of completeness, we list all of the benchmarks used in the first work on AQFP B/S insertion [25] in Table I, but the totals are computed only with the benchmarks presented in [27]. The number of gates ($|G^*|$) and the depth ($d(N^*)$) of the initial MIGs, as well as the number of buffers ($|B|$), the JJ count (#JJs), and the depth ($d(N')$) of the mapped networks are listed. Moreover, the runtime (Time) used by our flow is presented. Unfortunately, the runtime data was not presented in [27]. In the last column, we list the known global optimum results obtained by ILP solving [28] to have an idea of how far the heuristics are from optimal. Some of the numbers are only an upper bound because the ILP formulation could not be solved within reasonable time, and some of the benchmarks are too big for the ILP solver to return any partial result.

From Table I, we can see that the heuristic methods achieve optimum for the smaller benchmarks and are fairly close to optimum for most of the benchmarks. While our flow obtains slightly worse results in average size than SoTA, the difference is very small (0.96% in number of buffers and 0.5% in JJ count). Thanks to the depth-optimal scheduling, we obtain a better depth in one benchmark (c7552). Most importantly,

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

TABLE I
TECHNOLOGY LEGALIZATION RESULTS COMPARING TO THE SOTA AND GLOBAL OPTIMUM

| | MIG $N^*$ | | SoTA [27] | | | Ours (Algorithm 8) | | | | Global optimum [28] | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bench. | $\lvert G^*\rvert$ | $d(N^*)$ | $\lvert B\rvert$ | #JJs | $d(N')$ | $\lvert B\rvert$ | #JJs | $d(N')$ | Time (s) | $\lvert B\rvert$ | #JJs | $d(N')$ |
| adder1 | 7 | 4 | - | - | - | 16 | 74 | 8 | 0.00 | 16 | 74 | 8 |
| adder8 | 77 | 17 | - | - | - | 371 | 1204 | 33 | 0.01 | 371 | 1204 | 33 |
| mult8 | 439 | 35 | 1681 | 5996 | 70 | 1690 | 6014 | 70 | 0.18 | $\leq$1724 | $\leq$6082 | $\leq$70 |
| counter16 | 29 | 9 | 66 | 306 | 17 | 65 | 304 | 17 | 0.00 | 65 | 304 | 17 |
| counter32 | 82 | 13 | 156 | 804 | 23 | 154 | 800 | 23 | 0.01 | 154 | 800 | 23 |
| counter64 | 195 | 17 | 351 | 1872 | 30 | 347 | 1864 | 30 | 0.02 | 347 | 1864 | 30 |
| counter128 | 428 | 22 | 755 | 4078 | 38 | 747 | 4062 | 38 | 0.07 | 747 | 4062 | 38 |
| c17 | 6 | 3 | - | - | - | 12 | 60 | 5 | 0.00 | 12 | 60 | 5 |
| c432 | 121 | 26 | 829 | 2384 | 37 | 839 | 2404 | 37 | 0.02 | 829 | 2384 | 37 |
| c499 | 387 | 18 | 1173 | 4668 | 29 | 1173 | 4668 | 29 | 0.09 | 1173 | 4668 | 29 |
| c880 | 306 | 27 | 1536 | 4908 | 40 | 1511 | 4858 | 40 | 0.15 | - | - | - |
| c1355 | 389 | 18 | 1186 | 4706 | 29 | 1184 | 4702 | 29 | 0.06 | 1178 | 4690 | 29 |
| c1908 | 289 | 21 | 1253 | 4240 | 34 | 1234 | 4202 | 34 | 0.09 | 1232 | 4198 | 34 |
| c2670 | 368 | 21 | 1869 | 5954 | 28 | 1912 | 6032 | 28 | 0.32 | $\leq$1804 | $\leq$5816 | $\leq$28 |
| c3540 | 794 | 32 | 1963 | 8690 | 52 | 1943 | 8650 | 52 | 0.81 | $\leq$1926 | $\leq$8516 | $\leq$52 |
| c5315 | 1302 | 26 | 5505 | 18942 | 40 | 5640 | 19092 | 40 | 2.06 | $\leq$6260 | $\leq$20332 | $\leq$42 |
| c6288 | 1870 | 89 | 8832 | 28884 | 179 | 8647 | 28514 | 179 | 2.56 | - | - | - |
| c7552 | 1394 | 33 | 6768 | 21908 | 58 | 7437 | 23238 | 56 | 4.20 | - | - | - |
| sorter32 | 480 | 15 | - | - | - | 480 | 3840 | 30 | 0.06 | 480 | 3840 | 30 |
| sorter48 | 880 | 20 | - | - | - | 880 | 7040 | 35 | 0.20 | 880 | 7040 | 35 |
| alu32 | 1513 | 100 | 13976 | 37030 | 169 | 13836 | 36750 | 169 | 2.74 | - | - | - |
| Total* | | | 47899 | 155370 | 873 | 48359 | 156154 | 871 | 13.38 | | | |

TABLE II
TECHNOLOGY LEGALIZATION RESULTS ON THE LARGEST EPFL BENCHMARKS

| | MIG $N^*$ | | Non-d.-opt. legal.+opt. [12] | | | D.-opt. legal. (alg. 3 + alg. 1) | | | D.-opt. legal.+opt. (alg. 8) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Bench. | $\lvert G^*\rvert$ | $d(N^*)$ | $\lvert B\rvert$ | $d(N')$ | Time (s) | $\lvert B\rvert$ | $d(N')$ | Time (s) | $\lvert B\rvert$ | $d(N')$ | Time (s) |
| div | 57300 | 2217 | 2084772 | 4918 | 271.71 | 1881255 | 4371 | 0.87 | - | 4371 | >300 |
| hyp | 136109 | 8762 | - | 17910 | >300 | 9035578 | 17246 | 2.78 | - | 17246 | >300 |
| log2 | 24456 | 200 | 98047 | 414 | 194.92 | 129547 | 379 | 0.10 | 86705 | 379 | 64.18 |
| multiplier | 19710 | 133 | 79651 | 286 | 13.21 | 102005 | 264 | 0.08 | 63414 | 264 | 43.50 |
| sin | 4303 | 110 | 17470 | 225 | 5.67 | 18905 | 188 | 0.01 | 14886 | 188 | 4.12 |
| sqrt | 23238 | 3366 | 1751742 | 8191 | 5.64 | 1791005 | 6628 | 0.49 | 1343705 | 6628 | 284.10 |
| square | 12180 | 126 | 60552 | 256 | 42.71 | 89516 | 251 | 0.03 | 63630 | 251 | 18.30 |
| arbiter | 7000 | 59 | 31011 | 65 | 5.80 | 27566 | 63 | 0.01 | 25721 | 63 | 1.28 |
| mem_ctrl | 42758 | 73 | 305689 | 182 | 87.86 | 216927 | 114 | 0.27 | 215202 | 114 | 10.55 |
| voter | 7860 | 47 | 18044 | 99 | 5.43 | 19263 | 86 | 0.01 | 15736 | 86 | 0.92 |

these results are obtained using short runtime. Thus, our flow can be used in design space exploration, where legalization is called extensively, such that large improvements can be achieved Section VII-C.

It is also worth mentioning that, out of the 21 benchmarks, the legalization and optimization result starting from an ASAP scheduling is taken (i.e., better than the one from an ALAP scheduling) in 16 benchmarks. We can see that ASAP may provide better quality in more cases, but this is not definitive. Thus, trying both starting points, as in Algorithm 8, helps achieve better results when the runtime budget is sufficient.

## B. Scalable AQFP Legalization

To demonstrate the scalability of our AQFP legalization approach, we use the largest ten benchmarks in the EPFL benchmark suite [37] for experiment, which are 10×–100× in size compared to the benchmarks generally used in previous works on AQFP logic synthesis. The MIGs are obtained using delay-oriented graph mapping [16]. In Table II, we compare our results obtained using a simple depth-optimal legalization flow (Algorithm 3 followed by Algorithm 1, column "D.-opt. legal.") as well as depth-optimal legalization with further optimization (Algorithm 8, column "D.-opt. legal.+opt.") against results of nondepth-optimal legalization with optimization presented in [12] (column "Non.-d.-opt. legal.+opt."). A timeout limit of 300 s is enforced. From this experiment, we can see that simple legalization without optimization is very fast, so such a flow can still be used in design space exploration even when benchmarks are large. Comparing the mapped network depths, the proposed depth-optimal scheduling reduces the depth by about 9% on average.

## C. Technology Mapping

With the proposed design space exploration approach presented in Section VI-B, we present new best-known results on the problem of AQFP technology mapping on the MCNC benchmark suite [38]. In Table III, our results are compared to SoTA [30]. Since [30] outperformed other previous works [11], [19], [24], [29] on all benchmarks and on all

TABLE III
BEST-KNOWN RESULTS ON AQFP TECHNOLOGY MAPPING

| Bench. | SoTA [30] | | | Ours (Algorithm 9) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | #JJs | $d(N')$ | EDP | #JJs | | $d(N')$ | | EDP | | Time (s) | Eval. (s) |
| 5xp1 | 726 | 10 | 7260 | 368 | -49% | 9 | -10% | 3312 | -54% | 66.2 | 0.7 |
| c1908 | 5108 | 34 | 173672 | 4434 | -13% | 29 | -15% | 128586 | -26% | 190.4 | 36.4 |
| c432 | 3098 | 34 | 105332 | 2342 | -24% | 27 | -21% | 63234 | -40% | 68.0 | 2.7 |
| c5315 | 16410 | 30 | 492300 | 13986 | -15% | 24 | -20% | 335664 | -32% | 519.8 | 267.8 |
| c880 | 3876 | 23 | 89148 | 3364 | -13% | 19 | -17% | 63916 | -28% | 100.6 | 14.6 |
| chkn | 3500 | 15 | 52500 | 2238 | -36% | 15 | 0% | 33570 | -36% | 96.5 | 6.0 |
| count | 1400 | 12 | 16800 | 1302 | -7% | 11 | -8% | 14322 | -15% | 77.3 | 1.4 |
| dist | 3536 | 14 | 49504 | 1824 | -48% | 14 | 0% | 25536 | -48% | 116.7 | 6.1 |
| in5 | 3370 | 14 | 47180 | 1602 | -52% | 13 | -7% | 20826 | -56% | 120.2 | 4.4 |
| in6 | 2884 | 11 | 31724 | 1708 | -41% | 12 | +9% | 20496 | -35% | 90.3 | 3.5 |
| k2 | 14748 | 22 | 324456 | 8376 | -43% | 19 | -14% | 159144 | -51% | 404.7 | 102.8 |
| m3 | 2680 | 12 | 32160 | 1600 | -40% | 12 | 0% | 19200 | -40% | 115.6 | 4.3 |
| max512 | 4812 | 16 | 76992 | 2740 | -43% | 14 | -13% | 38360 | -50% | 140.8 | 10.1 |
| misex3 | 11272 | 20 | 225440 | 2634 | -77% | 17 | -15% | 44778 | -80% | 238.1 | 21.9 |
| mlp4 | 2976 | 14 | 41664 | 1588 | -47% | 14 | 0% | 22232 | -47% | 160.0 | 7.3 |
| prom2 | 22326 | 20 | 446520 | 15258 | -32% | 16 | -20% | 244128 | -45% | 788.8 | 286.5 |
| sqr6 | 916 | 10 | 9160 | 710 | -22% | 9 | -10% | 6390 | -30% | 59.3 | 0.7 |
| x1dn | 1208 | 11 | 13288 | 714 | -41% | 10 | -9% | 7140 | -46% | 61.5 | 0.5 |
| Total | 104846 | 322 | 2235100 | 66788 | -36% | 285 | -12% | 1239208 | -44% | 3414.6 | 777.8 |

metrics,[3] data from these works is omitted. We use the same optimization objective as in [30], i.e., minimizing EDP. The parameters used in Algorithm 9 are *num_restarts* $= 5$, *max_steps*[4] $= 1000$, *max_no_impr* $= 50$, and *timeout* $=$ 100 s. With this parameter setting, for most benchmarks, the loop-breaking condition is *max_no_impr* before the *timeout* limit is exceeded. This is an intentional decision to make the experiment more reproducible on different machines.

In addition to #JJs, $d(N')$, and EDP, the last two columns in Table III list, respectively, the total runtime of Algorithm 9 (column "Time") and the runtime for cost evaluation (line 12 in Algorithm 9, column "Eval.") using Algorithm 8. The runtime information of [30] is unfortunately not provided.

Our design space exploration achieves strictly better results than [30] in #JJs and EDP on all benchmarks. In total, 36% improvement in #JJs, 12% improvement in depth, and 44% improvement in EDP are achieved within manageable runtime.

## VIII. CONCLUSION

This article presents a full flow on the AQFP technology mapping problem, focusing mainly on legalization and optimization. We first establish that the AQFP legalization problem is a scheduling problem and propose two depth-optimal scheduling algorithms. Then, the obtained schedules may be further optimized for size using the proposed chunked movement and retiming techniques. As both irredundant buffer insertion and depth-optimal scheduling have linear time complexity, scalability is guaranteed. Finally, we combine our legalization flow with MIG logic optimization and propose an unsupervised design space exploration for AQFP technology mapping, which achieves massive improvement over the SoTA. As AQFP legalization is performed after each MIG optimization trial in design space exploration, one of the key elements to its success is the efficient optimization heuristics in the legalization flow. For the sake of completeness, we also discuss verification methods for legalized AQFP circuits.

[3]References [19] and [29] used different assumptions, i.e., primary inputs do not need to be balanced, so the numbers presented in the papers are different. As both works are open-sourced and flexible to taking different assumptions, we reran the experiment with the same assumptions for a fair comparison.

[4]All restarts end within 200 steps due to the two terminating conditions, so this value is never really reached.

## REFERENCES

[1] N. Takeuchi, D. Ozawa, Y. Yamanashi, and N. Yoshikawa, "An adiabatic quantum flux parametron as an ultra-low-power logic device," *Supercond. Sci. Technol.*, vol. 26, no. 3, 2013, Art. no. 035010.

[2] N. Takeuchi, Y. Yamanashi, and N. Yoshikawa, "Adiabatic quantum-flux-parametron cell library adopting minimalist design," *J. Appl. Phys.*, vol. 117, no. 17, 2015, Art. no. 173912.

[3] N. Takeuchi et al., "Adiabatic quantum-flux-parametron cell library designed using a 10 kA cm$^{-2}$ niobium fabrication process," *Supercond. Sci. Technol.*, vol. 30, no. 3, 2017, Art. no. 035002.

[4] N. Takeuchi, M. Nozoe, Y. He, and N. Yoshikawa, "Low-latency adiabatic superconductor logic using delay-line clocking," *Appl. Phys. Lett.*, vol. 115, no. 7, 2019, Art. no. 072601.

[5] R. Saito, C. L. Ayala, and N. Yoshikawa, "Buffer reduction via N-phase clocking in adiabatic quantum-flux-parametron benchmark circuits," *IEEE Trans. Appl. Supercond.*, vol. 31, no. 6, pp. 1–8, Sep. 2021.

[6] Y. He et al., "Low clock skew superconductor adiabatic quantum-flux-parametron logic circuits based on grid-distributed blocks," *Supercond. Sci. Technol.*, vol. 36, no. 1, 2022, Art. no. 015006.

[7] N. Tsuji, Y. Yamanashi, N. Takeuchi, C. Ayala, and N. Yoshikawa, "Design and implementation of scalable register files using adiabatic quantum flux parametron logic," in *Proc. ISEC*, 2017, pp. 1–3.

[8] C. L. Ayala, T. Tanaka, R. Saito, M. Nozoe, N. Takeuchi, and N. Yoshikawa, "MANA: A monolithic adiabatic integration architecture microprocessor using 1.4-zj/op unshunted superconductor josephson junction devices," *IEEE J. Solid-State Circuits*, vol. 56, no. 4, pp. 1152–1165, Apr. 2021.

[9] Q. Xu, C. L. Ayala, N. Takeuchi, Y. Murai, Y. Yamanashi, and N. Yoshikawa, "Synthesis flow for cell-based adiabatic quantum-flux-parametron structural circuit generation with HDL back-end verification," *IEEE Trans. Appl. Supercond.*, vol. 27, no. 4, pp. 1–5, Jun. 2017.

[10] C. L. Ayala et al., "A semi-custom design methodology and environment for implementing superconductor adiabatic quantum-flux-parametron microprocessors," *Supercond. Sci. Technol.*, vol. 33, no. 5, 2020, Art. no. 054006.

[11] G. Meuli et al., "Majority-based design flow for AQFP superconducting family," in *Proc. DATE*, 2022, pp. 34–39.
[12] S.-Y. Lee, H. Riener, and G. De Micheli, "Beyond local optimality of buffer and splitter insertion for AQFP circuits," in *Proc. DAC*, 2022, pp. 445–450.
[13] A. T. Calvino and G. De Micheli, "Depth-optimal buffer and splitter insertion and optimization in AQFP circuits," in *Proc. ASP-DAC*, 2023, pp. 152–158.
[14] Y. Harada, H. Nakane, N. Miyamoto, U. Kawabe, E. Goto, and T. Soma, "Basic operations of the quantum flux parametron," *IEEE Trans. Magn.*, vol. 23, no. 5, pp. 3801–3807, Sep. 1987.
[15] L. G. Amarù, P. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 35, no. 5, pp. 806–819, May 2016.
[16] A. Tempia Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *Proc. ASP-DAC*, 2022, pp. 410–416.
[17] H. Riener, E. Testa, L. G. Amarù, M. Soeken, and G. De Micheli, "Size optimization of MIGs with an application to QCA and STMG technologies," in *Proc. NANOARCH*, 2018, pp. 157–162.
[18] S.-Y. Lee, H. Riener, and G. De Micheli, "Logic resynthesis of majority-based circuits by top-down decomposition," in *Proc. DDECS*, 2021, pp. 105–110.
[19] E. Testa, S. Lee, H. Riener, and G. De Micheli, "Algebraic and boolean optimization methods for AQFP superconducting circuits," in *Proc. ASP-DAC*, 2021, pp. 779–785.
[20] S.-Y. Lee and G. De Micheli, "Heuristic logic resynthesis algorithms at the core of peephole optimization," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 42, no. 11, pp. 3958–3971, Nov. 2023.
[21] K. K. Likharev and V. K. Semenov, "RSFQ logic/memory family: A new Josephson-junction technology for sub-terahertz-clock-frequency digital systems," *IEEE Trans. Appl. Supercond.*, vol. 1, no. 1, pp. 3–28, Mar. 1991.
[22] N. K. Katam and M. Pedram, "Logic optimization, complex cell design, and retiming of single flux quantum circuits," *IEEE Trans. Appl. Supercond.*, vol. 28, no. 7, pp. 1–9, Oct. 2018.
[23] G. Pasandi and M. Pedram, "PBMap: A path balancing technology mapping algorithm for single flux quantum logic circuits," *IEEE Trans. Appl. Supercond.*, vol. 29, no. 4, pp. 1–14, Jun. 2019.
[24] R. Cai et al., "A majority logic synthesis framework for adiabatic quantum-flux-parametron superconducting circuits," in *Proc. GLSVLSI*, 2019, pp. 189–194.
[25] R. Cai, O. Chen, A. Ren, N. Liu, N. Yoshikawa, and Y. Wang, "A buffer and splitter insertion framework for adiabatic quantum-flux-parametron superconducting circuits," in *Proc. ICCD*, 2019, pp. 429–436.
[26] C. Huang, Y. Chang, M. Tsai, and T. Ho, "An optimal algorithm for splitter and buffer insertion in adiabatic quantum-flux-parametron circuits," in *Proc. ICCAD*, 2021, pp. 1–8.
[27] R. Fu, M. Wang, Y. Kan, N. Yoshikawa, T.-Y. Ho, and O. Chen, "A global optimization algorithm for buffer and splitter insertion in adiabatic quantum-flux-parametron circuits," in *Proc. ASP-DAC*, 2023, pp. 769–774.
[28] D. S. Marakkalage and G. De Micheli, "Fanout-bounded logic synthesis for emerging technologies," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 43, no. 5, pp. 1415–1428, May 2024.
[29] D. S. Marakkalage, H. Riener, and G. De Micheli, "Optimizing adiabatic quantum-flux-parametron (AQFP) circuits using an exact database," in *Proc. NANOARCH*, 2021, pp. 1–6.
[30] R. Fu et al., "BOMIG: A majority logic synthesis framework for AQFP logic," in *Proc. DATE*, 2023, pp. 1–2.
[31] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, nos. 1–6, pp. 5–35, 1991.
[32] A. P. Hurst, A. Mishchenko, and R. K. Brayton, "Fast minimum-register retiming via binary maximum-flow," in *Proc. FMCAD*, 2007, pp. 181–187.
[33] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 26, no. 2, pp. 240–253, Feb. 2007.
[34] A. Mishchenko, R. K. Brayton, S. Jang, and V. N. Kravets, "Delay optimization using SOP balancing," in *Proc. ICCAD*, 2011, pp. 375–382.
[35] A. Mishchenko, S. Chatterjee, R. K. Brayton, and N. Eén, "Improvements to combinational equivalence checking," in *Proc. ICCAD*, 2006, pp. 836–843.
[36] M. Soeken et al., "The EPFL logic synthesis libraries," 2022, *arXiv:1805.05121*.
[37] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. IWLS*, 2015, pp. 1–5.
[38] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0*, Microelectron. Center North Carolina, Durham, NC, USA, 1991.

**Siang-Yun Lee** received the B.Sc. degree from the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, in 2019, and the Ph.D. degree in computer and communication sciences from EPFL, Lausanne, Switzerland, in 2024.

She worked with the Integrated Systems Laboratory led by Prof. G. De Micheli, EPFL. She is a Research and Development Engineer with Cadence Design Systems, Munich, Germany. Her research interests include logic synthesis and design automation for emerging technologies.

**Alessandro Tempia Calvino** (Graduate Student Member, IEEE) received the B.S. degree in computer engineering from the Politecnico di Torino, Turin, Italy, in 2017, and the M.S. degrees in computer engineering from the Politecnico di Torino in 2020 and Télécom Paris, Paris, France, in 2021. He is currently pursuing the Ph.D. degree in computer science with the Integrated Systems Laboratory, Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland.

His research interests include design automation, logic synthesis, and emerging technologies.

**Heinz Riener** received the B.Sc. and M.Sc. degrees from Technical University Graz, Graz, Austria, in 2008 and 2011, respectively, and the Ph.D. degree in computer science from the University of Bremen, Bremen, Germany, in 2017.

He is an Engineer with Cadence Design Systems, Munich, Germany. From 2015 to 2017, he worked with the Group of Avionics Systems, German Aerospace Center, Bremen. From 2017 to 2021, he worked with the Integrated Systems Laboratory, EPFL, Lausanne, Switzerland. His research interests are logic synthesis, formal methods, and computer-aided verification of hardware and software systems.

**Giovanni De Micheli** (Life Fellow, IEEE) received the Nuclear Engineer degree from the Politecnico di Milano, Milan, Italy, in 1979, and the M.S. degree and the Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley, Berkeley, CA, USA, 1980 and 1983, respectively.

He is a Professor and the Director of the Integrated Systems Laboratory, EPFL, Lausanne, Switzerland. His current research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies.

Prof. De Micheli is the recipient of the 2022 ESDA-IEEE/CEDA Phil Kaufman Award, the 2019 ACM/SIGDA Pioneering Achievement Award, and several other awards. He is a member of the Scientific Advisory Board of IMEC and STMicroelectronics. He is a Fellow of ACM and AAAS, a member of the Academia Europaea, and an International Honorary Member of the American Academy of Arts and Sciences.