Optimizing Adiabatic Quantum-Flux-Parametron (AQFP) Circuits using Exact Methods

Dewmini Sudara Marakkalage, Heinz Riener, Giovanni De Micheli

Integrated Systems Laboratory, EPFL, Lausanne, Switzerland

***Abstract*—Adiabatic Quantum-Flux-Parametron (AQFP) is a family of superconducting electronic (SCE) circuits exhibiting high energy efficiency. In AQFP technology, logic gates require splitters to drive multiple fanouts and both the logic gates and the splitters are clocked, requiring path balancing to ensure all fanins of a gate arrive simultaneously. There have been several attempts to optimize the resource usage of AQFP circuits under these constraints, but we identify three main shortcomings of the state of the art: i) lack of consideration for interdependent logic paths, ii) heavy reliance on balanced splitter trees, and iii) lack of support for majority-5 gates. In this work, we present a new approach of optimization for the AQFP technology that alleviates these shortcomings using exact methods. Through explicit, size-bounded enumeration, we generate a database of optimum AQFP circuits for small logic functions under different arrival time patterns of the inputs and use the resulting database to rewrite logic blocks of larger networks. We evaluate our algorithm on a subset of MCNC benchmarks. We show that the proposed method achieves over a 21% decrease in area (the number of Josephson Junctions) and a 35% decrease in delay in the critical path (the number of levels) as compared to existing work when using an area-oriented strategy. Our method achieves over a 19% area reduction when using a delay-oriented approach while decreasing delay by over 40%.**

***Index Terms*—AQFP, majority gates, exact synthesis, logic synthesis**

I. Introduction

Superconducting electronic (SCE) circuits are getting increasingly popular in the electronics industry due to their low energy consumption and high-speed operation. The growing interest in SCE is further fuelled by the escalating challenges and higher costs of transistor downscaling in traditional CMOS technologies. The potential of SCE to revolutionize the electronics industry is widely recognized as evidenced by the growing involvement of the EDA industry in developing tools and synthesis flows for SCE supported by government-funded programs such as IARPA's SuperTools program [1].

SCE circuits are based on superconductive inductors and Josephson Junctions (JJs) [2], and there are several families of SCE circuits. The examples include Rapid Single Flux Quantum (RSFQ) [3], Energy-efficient SFQ (eSFQ) [4], Reciprocal Quantum Logic (RQL) [5], Low-Voltage RSFQ (LV-RSFQ) [6], Dynamic Single Flux Quantum (DSFQ) [7], and Adiabatic Quantum-Flux-Parametron (AQFP) [8]. While most of such families use DC-biased junctions which cause static power dissipation, the technologies such as AQFP achieve superior energy-efficiency using AC-biased junctions, and this work focuses on the AQFP technology.

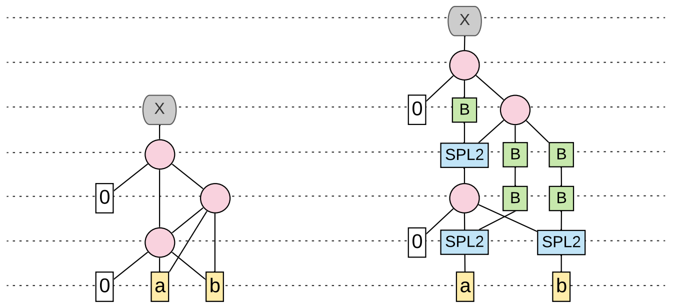

Fig. 1: An example logic network with unit-delay gates (left) and its splitter-inserted, path-balanced version (right).

The AQFP technology provides efficient implementations of majority-3 and majority-5 gates that offer more complex logic functionalities at a comparably low resource usage (see Section III), leading to more compact circuitry. On the other hand, AQFP logic gates cannot directly drive more than one fanout, necessitating clocked splitters to support multiple fanouts. Moreover, the gates are also clocked, requiring each gate's fanins to arrive at the same time via logic paths that are balanced with (clocked) buffers. Figure 1 shows a simple logic network with unit-delay gates (left) and its splitter-inserted, path-balanced version (right). In all figures, green squares labeled "B" denote buffers while blue rectangles labeled "SPL*k*" denote splitters of branching factor *k*.

The path balancing and splitter requirements of AQFP technology pose additional challenges in logic synthesis because, in addition to the gates that implement logic functions, buffers and splitters also significantly affect the area and delay of a circuit. As such, there have been several attempts to optimize the splitter insertion and path balancing of AQFP circuits [9]–[11]. However, existing work suffers from one or more of the following weaknesses: i) lack of consideration for interdependent logic paths [10], ii) the bias towards using balanced splitter trees [9], and iii) the lack of support for more complex logic gates such as majority-5 [9]–[11]. In this work, we show how to mitigate these shortcomings using exact synthesis on small blocks of logic in a given logic network.

Our main idea is to generate a database of minimum area (the number of JJs for example) AQFP circuits for all single-output, 4-input functions and for a set of different input arrival-time patterns, and then use the database to rewrite logic blocks of a larger network in the topological order. Our approach differs from the similar-looking method proposed by Amaru et al. [12] for exact delay synthesis in two key aspects. First, as their goal was to minimize the delay, their database generation ignores possible area improvements that result from logic sharing and enumerates only the tree structures. In contrast, our goal is to minimize the overall area, and logic sharing can both support (by reducing the gate count) and hinder (by increasing the splitter count, the number of levels, and consequently the number of buffers) this goal. Therefore, we enumerate all directed acyclic graph (DAG) structures within a predetermined size bound. Second, the outputs of logic blocks chosen by the algorithm can have multiple fanouts, and our algorithm takes the splitter requirements of such logic blocks into account before synthesizing the other logic blocks that use the outputs of already synthesized logic blocks as inputs. To synthesize such fanout nets, we use balanced splitter trees. But unlike the work of Testa et al. [9] which uses balanced splitter trees on all multi-fanout nodes, we use this strategy on only a small fraction of such nodes (i.e., only on the outputs of the blocks of logic chosen by the algorithm).

We evaluate our synthesis algorithm on the same subset of MCNC benchmarks considered by Testa et al. [9] using two different strategies, area-oriented and delay-oriented. The former strategy improves the area by over 21% while reducing the critical path delay by over 35% on average compared to the optimized results of Testa et al. [9]. Compared to the same results, the latter strategy reduces the area by over 19% while decreasing the critical path delay by more than 40% on average. Note that we use the total number of Josephson Junctions as the measure of area and the number of gate levels (including buffers and splitters) as the delay measure.

The remainder of this paper is organized as follows: In Section II, we discuss the shortcomings of existing approaches which motivated this work, and in Section III, we provide some relevant background. In Section IV, we describe our database generation method and introduce the algorithm for synthesizing AQFP circuits. In Section V, we present our experimental results, and in Section VI, we conclude with a brief discussion on our results and some potential improvements to the proposed method.

II. Motivation

In this section, we discuss existing work on optimizing AQFP circuits and their main drawbacks.

The work of Cai et al. [10] presents a new framework for inserting the optimum number of buffers and splitters for a given fanout net with fixed levels. Both Ayala et al. [11] and Testa et al. [9] proposed new synthesis flows for AQFP circuits. The synthesis flows first apply logic optimizations, then using the required number of splitters on multi-fanout nets, determine levels of the gates, and finally insert buffers as required together with optimizations that move splitters around to decrease the buffer count. In the logic optimization phase, the work of Testa et al. [9] uses depth optimizations based on majority-inverter graphs (MIGs) [13] (see Section III),

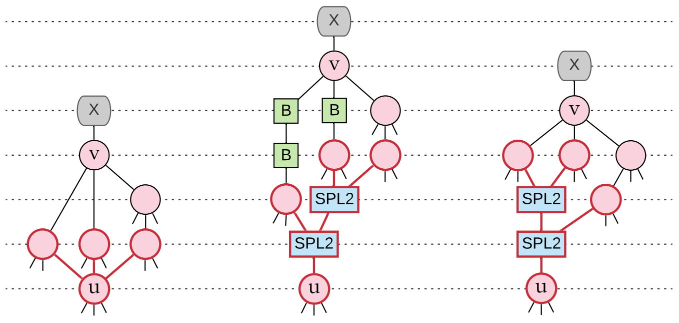

Fig. 2: A part of a logic network with unit-delay gates (left), and its path-balanced versions (middle and right) using two choices of locally optimum splitter-trees with 1-to-2 splitters.

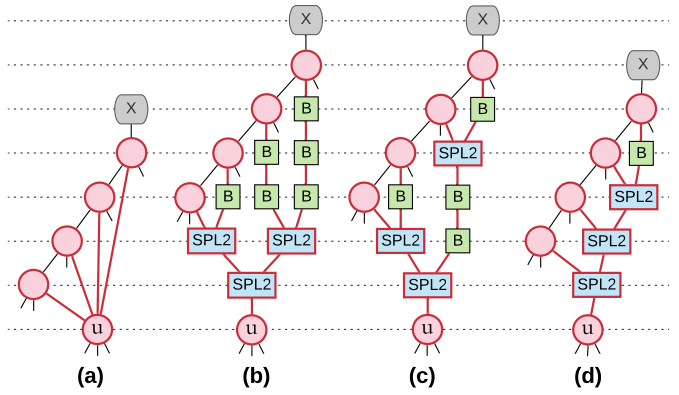

Fig. 3: (a) A logic network with unit-delay gates, (b) its path balanced version using a balanced splitter tree, (c) an optimized version of (a) by pushing one splitter up in the hierarchy, and (d) optimum path balancing with an unbalanced splitter tree.

where the depth optimization algorithms use estimated splitter requirement as the cost heuristic to achieve better results.

We identify three shortcomings in the existing work:

a) Lack of consideration for interdependent logic paths: Consider Figure 2 which shows a logic network on the left, and two possible splitter tree choices for the highlighted fanout net assuming 1-to-2 splitters. (For simplicity, we disregard the splitter/buffer requirement of all unspecified fanins.) The logic path interdependencies make the choice on the right a better option than the choice in the middle. However, the existing algorithms are susceptible to making suboptimal choices in such situations as they reason based solely on the local view and consider both splitter trees as equally good options.

b) Bias for balanced-splitter trees: The approaches in prior work naively use balanced splitter trees for multiple-fanout nets when deciding the levels of those fanouts [9]. However, always using balanced splitter trees can incur additional buffer costs if the fanouts of a particular node have to be placed at uneven depths due to constraints dictated by other shared logic paths. In such cases, an unbalanced splitter tree can be a better match as shown in Figure 3.

For the given logic network (left), the unbalanced splitter tree (right) costs fewer buffers compared to a balanced splitter tree (middle two trees) assuming 1-to-2 splitters. The second network naively uses a balanced splitter-tree and adds buffers on top of it whereas the third network optimizes the buffer count by pushing one of the splitters up in the hierarchy. Nevertheless, the network on the right with the unbalanced splitter tree has a better resource usage.

c) Lack of support for more complex gates: None of the prior works support the generation of netlists with majority-k gates for $k > 3$ although the AQFP technology can support efficient implementations of such gates [14]. For example, consider the logic function of majority-5 itself. Using only AND-2, OR-2, and majority-3 gates, computing this function needs at least four gates which costs at least 24 JJs whereas using a single majority-5 gate uses only 10 JJs. (See Section III.)

Our method alleviates these drawbacks by precomputing exactly optimum circuits for functions with a few variables and using them to rewrite logic blocks of larger networks.

III. Background

In this section, we give background on majority-inverter graphs (MIGs), AQFP logic circuits, NPN equivalence, and exact synthesis.

A. MIG-based logic synthesis

The k-input Boolean majority gate outputs 1 if and only if more than $k/2$ of the inputs are one. Logic synthesis based on majority gates was extensively studied in the past [15]–[17], and recently Amaru et al. [13], [18] proposed MIG as a new paradigm for logic synthesis. An MIG is a directed acyclic graph (DAG) where each internal node represents a majority gate, and each directed edge is either labeled as a regular edge or a complemented edge indicating the absence or presence of an inverter at the respective fanin. MIGs accompany a sound and complete set of algebraic rules [19] which we state below. We use the notation $\langle \ldots \rangle$ to denote the majority operation.

$$\begin{aligned}
\textbf{Commutativity}: &\quad \langle x\,y\,z\rangle = \langle y\,z\,x\rangle = \langle z\,x\,y\rangle,\\
\textbf{Associativity}: &\quad \langle x\,u\,\langle y\,u\,z\rangle\rangle = \langle z\,u\,\langle x\,u\,y\rangle\rangle\\
\textbf{Distributivity}: &\quad \langle x\,u\,\langle y\,z\,v\rangle\rangle = \langle\langle x\,u\,y\rangle\,z\,\langle x\,u\,v\rangle\rangle,\\
\textbf{Majority}: &\quad \langle x\,x\,y\rangle = x \text{ and } \langle x\,\bar{x}\,y\rangle = y,\\
\textbf{Inverter Propagation}: &\quad \langle \bar{x}\,\bar{y}\,\bar{z}\rangle = \overline{\langle x\,y\,z\rangle}.
\end{aligned}$$

B. AQFP logic circuits

Logic gates in the AQFP technology are mainly constructed using superconductive inductors and JJs which are based on the Josephson effect [20]. Takeuchi et al. [21] proposed a simple cell library for AQFP technology based on four primitive cells—buffer, inverter, constant, and branch—where a gate is created using an array of primitive cells together with a branch while a splitter is constructed using a buffer and a branch. The majority-3 gate consists of three buffer cells together with a branch, and different fanin inverted versions of a majority-3 gate are constructed by substituting a subset of buffer cells with inverter cells [21]. The 2-input AND and OR gates are constructed by substituting a buffer cell with a constant cell. Each of the three primitive cells, buffer, inverter, and constant, consists of two JJs, and hence a splitter also uses 2 JJs while all gates—majority-3, AND-2, and OR-2—as well as all their input-inverted versions use 6 JJs each. Additionally, the majority-5 gate and its input-inverted versions can be implemented with 10 JJs each [14]. As a majority-3 gate uses the same resources as an AND-2 or an OR-2 gate, Cai et al. [22] proposed that majority logic synthesis is more suitable for optimizing AQFP circuits.

As stated in Section I, logic gates in AQFP technology cannot drive multiple fanouts and therefore a tree of splitters must be used in case a gate has to feed several fanouts. The gates and the splitters are clocked, and consequently, buffers must be inserted to make the circuit pipelined. Depending on the design of registers and the used clocking schemes, there can be different requirements on whether splitters are needed for primary inputs, whether path balancing is needed for primary inputs, and how the path balancing is done for primary outputs [23].

C. NPN Equivalence

Two functions f and g are NPN equivalent if f can be obtained from g using a combination of input negations, input permutations, and output negation [24], [25]. When generating synthesis databases, the NPN equivalence can be used to significantly reduce the number of database entries. For example, although there are $2^{2^4} = 65536$ different 4-input functions, there are only 222 different 4-input NPN classes.

D. Exact synthesis

Exact synthesis is the process of synthesizing circuits to meet exact specifications. For example, it can be used to synthesize exactly optimum circuit structures for a given Boolean function. Prior work on exact synthesis include [12], [26]–[30] and they use different approaches such as decomposition-based, SAT-based, and explicit enumeration-based methods. Since exact methods are computation-heavy, it is impractical to use them for large networks. Instead, they are typically used to construct databases for logic functions with a few variables, and separate rewriting algorithms are used to optimize small logic blocks of larger networks using the generated databases.

IV. AQFP Resynthesis Approach

In the first half of this section, we describe in detail the generation of the exact synthesis database, and in the second half, we present the overall algorithm for synthesizing path-balanced AQFP circuits using the precomputed database.

A. Generation of the database

The exact synthesis database contains MIGs that give the minimum area (after splitter-insertion and path-balancing) for each 4-input NPN class[1] under different input arrival-time patterns. We identify the input arrival-time pattern of a single-output MIG structure by the depths of its inputs with respect to

[1]We store MIGs that compute the function with the lexicographically smallest truth-table in each NPN class.

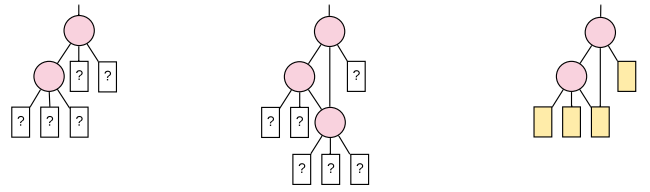

Fig. 4: Two partial DAGs (left and middle) and a DAG (right). The partial DAG in the middle is derived from the one on the left by tying a new gate and connecting it to one uncommitted leaf in each of the existing gates. The DAG on the right is derived from the partial DAG on the left by attaching four inputs (leaf nodes) to the five uncommitted leaves.

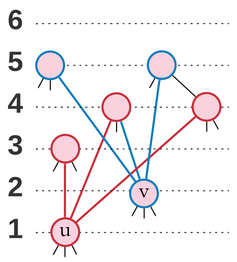

Fig. 5: Two fanout-nets with the same relative fanout levels.

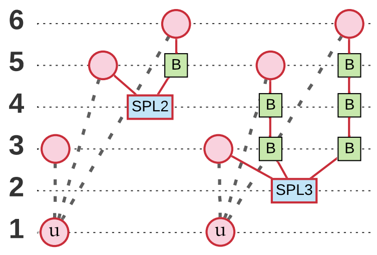

Fig. 6: Computing best area for a relative level configuration using dynamic programming.

the output. For simplicity, we explain the method focusing only on 3-input gates, but it is straightforward to extend it to include k-input majority gates for $k > 3$. The database is generated in three steps which are summarized below. Note that we use the term *DAG* to mean the underlying directed acyclic graph structure of an MIG *without* considering inverters.

First, we enumerate all single-output DAGs with two, three, four, and five leaf nodes (up to four inputs and a constant) and seven[2] gates. Next, we compute the area of realizing those DAGs as splitter-inserted, path-balanced circuits under a set of different input arrival patterns. We use the number of JJs (proportional to the number of primitive cells) to measure the area, but our method can support more general measures such as the physical cell area. Finally, we enumerate the 4-input NPN classes computable by each DAG structure (considering fanin inverters) and store the best-area MIG (i.e., the DAG together with a fanin inverter configuration) for each 4-input NPN class and for a set of input arrival-time patterns. We now describe each of the three steps in detail.

1) Generating DAG structures: To systematically generate DAGs, we first generate *partial DAGs*, which are DAGs where some gates have leaves that are not committed to inputs. This notion of partial DAGs is similar to the one used by Haaswijk [31]. Note that, a partial DAG can be extended to a larger partial DAG by binding new gates to a subset of uncommitted leaves, or they can be converted to a DAG by binding inputs to uncommitted leaf nodes. Figure 4 shows a partial DAG (left), and another partial DAG (middle) and a DAG (right) obtained from the first partial DAG. Note that the uncommitted leaves are designated with question marks and inputs are shown as filled squares.

Starting with a single-gate partial DAG with three uncommitted leaves, we generate other partial DAGs by attaching gates to the uncommitted leaves of existing partial DAGs in a level-by-level fashion. Having generated all partial DAGs with k gate levels, we generate all partial DAGs that have $k+1$ gate levels by extending those with k gate levels in such a way that each newly added gate has at least one fanout that is in level k. To generate DAGs from partial DAGs, we consider all different ways of attaching at most five inputs to available uncommitted leaves of a partial DAG. When enumerating DAGs, we use the *commutativity* and *majority* rules (see Section III-A) to avoid having redundant gates.

2) Computing Area of DAGs: Recall from Section III-B that all input-inverted versions of a logic gate use the same amount of resources in AQFP technology. Thus, to determine the minimum amount of resources needed to realize an MIG as an AQFP circuit, we only need to know its underlying DAG structure ignoring the inverters. We now explain how to find the optimum buffer-splitter insertion for such DAGs. Note that we do not need splitter insertion or path balancing for constant nodes as we have different versions of majority gates with implicit constant fanins [21] (see Section III). Thus the optimum buffer-splitter insertion for a DAG may depend on which leaf (if any) is treated as a constant. However, at the time of database generation, we do not know which leaf of an MIG in the database gets connected to a constant during synthesis. Therefore, for each DAG, we consider all versions of it obtained by assigning a constant to at most one leaf and compute their optimum splitter-inserted path balanced versions separately. Note that treating one leaf as a constant affects all shared logic paths. Hence it can also decrease the splitter-buffer resources needed in other fanout nets.

If we fix the depths of a gate and all of its fanouts, we can find the best buffer-splitter tree for that fanout net using dynamic programming. Once the gate depths are fixed, the best splitter-buffer tree for a given fanout net depends only on the *relative levels of the fanout nodes*. The relative levels of fanout nodes in the fanout net of a node u are the level differences between those fanout nodes and u. For example, Figure 5 shows two fanout nets (of nodes u and v respectively) in red and blue that have the same relative fanout levels $\{2, 3, 3\}$.

For a given multiset $S_{\text{lev}} = \{\ell_1, \ldots, \ell_k\}$ of relative fanout levels, the buffer cost c_b, the splitter cost c_s, and the splitter branching factor f_s, the recursive function in Algorithm 1 computes the optimum buffer-splitter cost or returns ∞ if there is no valid splitter-buffer configuration. In any valid fanout net, all relative fanout levels must be positive (Line 2), and if a node has only one fanout, we only need to add sufficiently many buffers (Line 3). If there are multiple fanouts, the algorithm iterates over fanout choices for the top-most splitter in the splitter tree (Line 5), computes the cost of the

[2]It is known that all 4-input functions can be synthesized with MIGs of at most seven majority-3 gates.

Algorithm 1: Computing minimum cost for a given multiset of relative fanout levels S_{lev}, buffer cost c_b, splitter cost c_s, and splitter branching factor f_s.

```
1  function cost ($S_{\text{lev}} = \{\ell_1, \dots, \ell_k\}$, $c_b$, $c_s$, $f_s$):
2      if $0 \geq \min_{\ell \in S_{\text{lev}}} \ell$ then return $\infty$
3      if $|S_{\text{lev}}| = 1$ then return $c_b(\ell_1 - 1)$
4      $c_{\text{best}} \leftarrow \infty$
5      for all $T \subseteq S_{\text{lev}}$ such that $2 \leq |T| \leq f_s$ do
6          $c_T \leftarrow c_s$, $\ell_{\min} \leftarrow \min T$
7          for $t \in T$ do $c_T \leftarrow c_T + c_b(t - \ell_{\min})$
8          $c_T \leftarrow c_T$ + cost ( $\{\ell_{\min} - 1\} \cup S_{\text{lev}} \setminus T$, $c_b$,
             $c_s$, $f_s$)
9          $c_{\text{best}} \leftarrow \min(c_{\text{best}}, c_T)$
10     return $c_{\text{best}}$
```

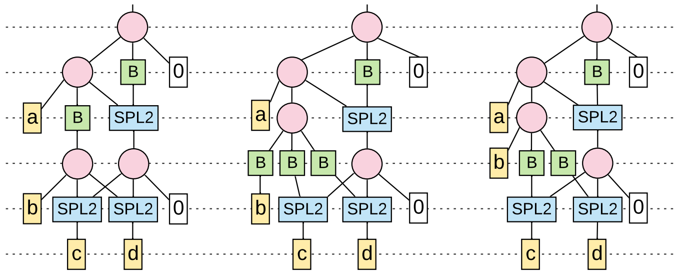

Fig. 7: Three different buffer-splitter configurations to realize the same DAG structure.

chosen splitter and that of the buffers needed on the outputs of the chosen splitter (Lines 6-7), and recursively compute the best cost for the remaining tree (Line 8), while keeping track of the best cost found so far (Line 9). For example, Figure 6 shows two possible choices for the top-most splitter in the splitter tree (Line 5 of Algorithm 1) assuming that the splitter branching factor is at least three. We can speed-up Algorithm 1 by caching the already computed optimum buffer-splitter costs for different relative level configurations.

To find the minimum area for a DAG structure, we employ a depth bounded search algorithm. We consider all possible ways of assigning depths (relative to the output node) to the gates within a gradually increasing bound on the maximum depth. For each assignment of depths to gates, we compute the best splitter-buffer tree for each fanout net using Algorithm 1. If no valid splitter-buffer configuration is found for the considered bound on the maximum depth, we increase the bound and try again. Once we reach a depth bound that gives at least one depth assignment with a valid splitter-buffer configuration, we stop increasing the maximum depth. However, we still consider all possible depth combinations for the inputs (leaf nodes) and find the best cost for each such combination over all depth assignments to other gates.

For example, Figure 7 shows three different depth assignments to gates and inputs for the same underlying DAG structure. One can verify there is no valid splitter-buffer configuration that achieves a maximum depth of 4, but there are several valid assignments for a maximum depth of 5 including the three shown in the figure. In the first two cases, the inputs (a, b, c, d) have the depths $(2, 4, 5, 5)$. (I.e., the input a is at depth 2, the input b is at depth 4, etc., assuming the top node has depth 0. This correspond to an arrival-time pattern where inputs a and b, respectively, arrive 3 and 2 unit-delays later than inputs c and d.) However, the first case uses only $(6 \cdot 4 + 2 \cdot 3 + 2 \cdot 2) = 34$ JJs whereas the second case uses $(6 \cdot 4 + 2 \cdot 3 + 2 \cdot 4) = 38$ JJs. Hence, the minimum cost for the considered DAG under the input depths $(2, 4, 5, 5)$ is 34. In the third case depicted in Figure 7, the inputs (a, b, c, d) have depths $(2, 3, 5, 5)$ and the total cost is 36 JJs. Since this is the only valid buffer-splitter configuration for this input depth pattern, the minimum cost for the DAG under this input depth pattern is 36.

3) Enumerating NPN classes and constructing the database: To generate the database, for each DAG, we compute the different 4-input NPN-classes computable by it. We do this by considering all possible inverter configurations (i.e., different ways of inverting fanins of gates), evaluating the functions computed by the DAG under these inverter configurations, and determining the corresponding NPN classes together with the respective NPN transformation (i.e., how to permute and negate inputs and outputs). For a given majority-3 gate, although there are eight different inverter configurations for its fanins, we need to explore only half of them due to the inverter propagation property (see Section III-A). Thus, in total, we explore 4^n different inverter configurations for a DAG with n gates. We use a pre-computed look-up table to efficiently find the NPN classes of 4-input functions and their associated NPN transformations.

Recall that, in the cost computation step, we computed costs of DAGs for different input arrival-time patterns identified by the pattern of the input depths. To construct the database, we go over different NPN classes computable by the DAG (under a suitable inverter configuration), and for each NPN class, we go over the different patterns of input depths and associated costs. For each NPN class f computable by a DAG g and for each input depths d and associated cost c, we then compute the new input depth pattern d' we would get if we use DAG g to compute f (this is computed by permuting d according to the input permutation of the associated NPN transformation). Finally, we update the database entry for (NPN class, input depth pattern) pair (f, d') by c and the respective input permuted DAG if there is no existing entry for the pair (f, d') or if c is smaller than the cost of the existing entry for the pair (f, d').

B. Synthesis Algorithm

We now describe our algorithm for synthesizing a large logic network as an AQFP circuit using the generated database. Given a logic network, it outputs a new MIG (which we refer to as the AQFP circuit) together with an assignment of levels to each gate. The best splitter-buffer configuration can then be recovered from the level assignment by adapting Algorithm 1

Algorithm 2: Algorithm to synthesize a given logic network as an AQFP circuit.

```
1  $\mathrm{ntk}_{\mathrm{aqfp}} \leftarrow$ Empty AQFP circuit.
2  $\mathrm{ntk}_{\mathrm{lut}} \leftarrow$ ABC_LUT_MAP($\mathrm{ntk}_{mig}$).
3  $m_{\mathrm{lev}} \leftarrow$ Empty map from $\mathrm{ntk}_{\mathrm{aqfp}}$ nodes to integers.
4  $m_{\mathrm{sig}} \leftarrow$ Empty map from $\mathrm{ntk}_{\mathrm{lut}}$ nodes to $\mathrm{ntk}_{\mathrm{aqfp}}$ signals.
5  foreach primary input $p$ of $\mathrm{ntk}_{\mathrm{lut}}$ do
6      Create new primary input $p'$ in $\mathrm{ntk}_{\mathrm{aqfp}}$.
7      $m_{\mathrm{lev}}[p'] \leftarrow 0$, $m_{\mathrm{sig}}[p] \leftarrow p'$.
8  foreach node $n \in \mathrm{ntk}_{\mathrm{lut}}$ in topological order do
9      $n_i \leftarrow$ The $i$-th fanin of $n$ for $i = 1, ..., 4$.
10     $\ell_i \leftarrow$ Adjusted level of $m_{\mathrm{sig}}[n_i]$ for $i = 1, ..., 4$.
11     $h \leftarrow$ Node function of $n$.
12     $f \leftarrow$ NPN class of $h$.
13     $\sigma \leftarrow$ Input permutation to get $h$ from $f$.
14     $(\ell'_1, \ldots, \ell'_4) \leftarrow \sigma(\ell_1, \ldots, \ell_4)$.
15     $g \leftarrow$ Best DAG from DB for $f$ and $(\ell'_1, \ldots, \ell'_4)$.
16     $g \leftarrow$ Input permuted $g$ according to inverse of $\sigma$.
17     $g \leftarrow$ Fanin inverted $g$ such that it computes $h$.
18     Create $g$ in $\mathrm{ntk}_{\mathrm{aqfp}}$ with inputs $(m_{\mathrm{sig}}[n_1], \ldots, m_{\mathrm{sig}}[n_4])$ and let $n'$ be the root node.
19     Update $m_{\mathrm{lev}}[n']$ and $m_{\mathrm{sig}}[n]$.
```

to output the minimum area splitter-buffer configuration instead of only returning the minimum area.

The algorithm, outlined in Algorithm 2, first maps the input network to a 4-LUT circuit using ABC's [32] LUT mapping [33]. Then, it maintains two mappings, one from the 4-LUT nodes to the corresponding signals (a signal denotes the output of nodes or its complement) in the new AQFP circuit, and another from the majority nodes in the new AQFP circuit to their assigned levels. Initially, it replicates all primary inputs in the target AQFP network, assigns level 0 to each primary input, and initializes the two mappings accordingly. Then it traverses the 4-LUT nodes in the topological order and resynthesizes each 4-LUT node according to a DAG structure chosen from the exact synthesis database. To choose the best DAG structure for a given 4-LUT node n with fanins $n_1, \ldots, n_4$, the algorithm does the following: First, it computes the node function h (i.e., h such that the output of n is $h(n_1, \ldots, n_4)$), its NPN class f, and the input permutation σ that describes how to permute the inputs $n_1, \ldots, n_4$ in order to compute h from f. Then it computes the current levels of the signals in the AQFP circuit that correspond to nodes $n_1, \ldots, n_4$. In case any fanin n_i has multiple fanouts, its level is computed assuming a nearly balanced splitter tree at its output. For example, suppose that n_1 has 10 fanouts and we have 1-to-4 splitters. Let $n_1^{(\mathrm{aqfp})}$ be the corresponding node in the new AQFP network. Then, the algorithm assumes a splitter tree with three splitters and two levels at the output of $n_1^{(\mathrm{aqfp})}$. Consequently, if $n_1^{(\mathrm{aqfp})}$ is at some level ℓ, for two of n_1's fanouts, we assume n_1 is available at level $\ell + 1$ (the first splitter in the tree has two slots remaining), and for the other eight of n_1's fanouts, we assume n_1 is available at level $\ell + 2$ (after two successive splitters). After finding the input levels of the fanin nodes, we permute those levels according to σ.

Next, in the database, we go over the entries with different input depth patterns for NPN class f. Suppose that for NPN class f and input depth pattern d, we have some DAG g with cost c. If we were to use DAG g to resynthesize node n, the cost would be c plus the cost of buffers needed to fill in the gaps between the permuted fanin levels and the input depths d of DAG g. Furthermore, we can also compute the level of the new node in the AQFP circuit that would represent the 4-LUT node n, using the permuted input levels and the input depth pattern d. At this point, to choose the best DAG, we propose two strategies: Either we use an *area-oriented* strategy where we choose the DAG that minimizes the area and break ties using the level we would get for the output node, or we use a *delay-oriented* strategy where we choose the DAG that minimizes the level of the output node and break ties using the area cost that would be incurred.

V. Experimental Results

In this section, we present the experimental results obtained from our AQFP synthesis algorithm and compare them with the results obtained by the AQFP synthesis flow presented in [9]. We compare with [9] as other flows [10], [11] are not open source and hence their results cannot be reproduced. We consider the same subset of 18 MCNC benchmarks [34] used in that work.

Similarly, we also use the same AQFP cell library discussed in Section III and use the number of JJs in the synthesized network as the area measure and the number of levels in the critical path as the delay measure. We construct two exact synthesis databases assuming we have 1-to-4 splitters:

DB1 A database that uses DAGs with up to seven 3-input gates without any 5-input gates.

DB2 DB1 extended with a) DAGs with up to three 5-input gates and three 3-input gates with a limit of four on the total number of gates and b) DAGs with up to three 5-input gates, four 3-input gates, and three levels with a limit of five on the total number of gates.

To generate the database, we used a cluster with 48 cores of Intel Xenon E5-2680 v3 CPUs running at 2.5GHz, and 256GB main memory, and each of the three steps was executed using 48 parallel threads. The generation of all DAGs for DB1 using the method described in Section IV-A1 consumed ∼20 minutes, and the output consists of 440 million DAGs (including different versions obtained by designating one leaf node as the constant node). The cost computation step Section IV-A2 took ∼1.5 hours whereas enumerating computable NPN classes and constructing the final database took ∼30 hours. Extending DB1 to DB2 using the DAGs with the given constraints took ∼25 hours in total. After removing redundant input depths patterns, DB1 and DB2 consist of only 5744 and 4317 DAGs respectively over all 222 4-input NPN classes.

We perform two experiments with the two databases: In the first experiment, we first synthesize the initial MIG as an

TABLE I: Results after 10 iterations of the proposed algorithm under the assumption that no splitter-buffer insertion is need for primary inputs but primary outputs need path-balancing. The *reference* column shows the optimized results from [9].

	Reference		**All iterations use DB1**								**Last iteration uses DB2**							
			Area-Oriented				**Delay-Oriented**				**Area-Oriented**				**Delay-Oriented**			
Benchmark	Delay (Levels)	Area (#JJs)	Delay (Levels)	Area (#JJs)	Delay Impr. %	Area Impr. %	Delay (Levels)	Area (#JJs)	Delay Impr. %	Area Impr. %	Delay (Levels)	Area (#JJs)	Delay Impr. %	Area Impr. %	Delay (Levels)	Area (#JJs)	Delay Impr. %	Area Impr. %
5xp1	8	824	8	716	0.00	13.11	8	742	0.00	9.95	8	674	0.00	18.20	8	730	0.00	11.41
c1908	53	5242	39	4512	26.42	13.93	36	5204	32.08	0.72	36	4082	32.08	22.13	32	4498	39.62	14.19
c432	50	2198	35	2178	30.00	0.91	36	2944	28.00	-33.94	32	1994	36.00	9.28	35	2696	30.00	-22.66
c5315	49	18932	33	14976	32.65	20.90	30	16312	38.78	13.84	31	14410	36.73	23.89	29	14850	40.82	21.56
c880	36	4520	24	3406	33.33	24.65	21	3678	41.67	18.63	22	3200	38.89	29.20	20	3402	44.44	24.73
chkn	28	4022	18	3312	35.71	17.65	14	3398	50.00	15.51	15	2900	46.43	27.90	13	2988	53.57	25.71
count	18	1426	13	1184	27.78	16.97	11	1346	38.89	5.61	13	1126	27.78	21.04	11	1326	38.89	7.01
dist	17	4208	13	3802	23.53	9.65	11	3990	35.29	5.18	12	3502	29.41	16.78	10	3480	41.18	17.30
in5	20	4312	15	3522	25.00	18.32	13	3754	35.00	12.94	12	3042	40.00	29.45	12	3116	40.00	27.74
in6	17	3472	12	2978	29.41	14.23	10	2952	41.18	14.98	10	2572	41.18	25.92	8	2552	52.94	26.50
k2	29	18294	19	16380	34.48	10.46	18	16306	37.93	10.87	16	14326	44.83	21.69	16	14372	44.83	21.44
m3	13	3118	12	2964	7.69	4.94	10	3016	23.08	3.27	10	2654	23.08	14.88	9	2680	30.77	14.05
max512	19	5536	14	5018	26.32	9.36	13	5334	31.58	3.65	13	4610	31.58	16.73	12	4636	36.84	16.26
misex3	29	14996	18	11922	37.93	20.50	15	12598	48.28	15.99	17	10580	41.38	29.45	14	10584	51.72	29.42
mlp4	19	3622	13	3222	31.58	11.04	11	3326	42.11	8.17	11	2938	42.11	18.88	10	2998	47.37	17.23
prom2	22	28774	16	26300	27.27	8.60	14	27302	36.36	5.12	14	24374	36.36	15.29	13	24586	40.91	14.55
sqr6	11	1102	9	962	18.18	12.70	8	978	27.27	11.25	8	896	27.27	18.69	7	902	36.36	18.15
x1dn	15	1296	11	1126	26.67	13.12	10	1148	33.33	11.42	10	988	33.33	23.77	10	1010	33.33	22.07
Total	453	125894	322	108480	**28.92**	**13.83**	289	114328	**36.20**	**9.19**	290	98868	**35.98**	**21.47**	269	101406	**40.62**	**19.45**

TABLE II: Results after 10 iterations of the proposed algorithm under the assumption that primary inputs need splitters to support multiple fanouts and primary outputs need path-balancing.

	All iterations use DB1				**Last iteration uses DB2**			
	Area-Oriented		**Delay-Oriented**		**Area-Oriented**		**Delay-Oriented**	
Benchmark	Delay (Levels)	Area (#JJs)	Delay (Levels)	Area (#JJs)	Delay (Levels)	Area (#JJs)	Delay (Levels)	Area (#JJs)
5xp1	10	870	10	888	10	830	10	884
c1908	37	5972	37	6328	37	5580	35	5810
c432	41	4002	39	4200	37	3714	35	3944
c5315	31	17600	32	19518	30	17166	31	17914
c880	25	4984	24	5012	23	4070	23	4086
chkn	16	3812	16	3912	13	3496	14	3556
count	12	1632	12	1660	12	1574	12	1592
dist	14	4332	15	4456	13	4060	13	4036
in5	15	3942	14	4070	13	3544	13	3520
in6	12	3360	12	3292	11	3010	10	2930
k2	22	17634	22	17632	20	15696	19	15444
m3	13	3334	13	3320	13	3108	12	3168
max512	17	5798	15	5796	15	5324	14	5306
misex3	21	13160	20	13512	20	11928	19	11996
mlp4	15	3714	14	3820	13	3480	13	3538
prom2	19	29362	19	30238	18	27608	18	27946
sqr6	11	1154	10	1134	10	1084	9	1070
x1dn	12	1348	11	1356	10	1214	10	1214
Total	343	126010	335	130144	318	116486	310	117954

AQFP circuit using our proposed algorithm with DB1. Then using the underlying MIG in the synthesized AQFP circuit as the input, the same algorithm was repeatedly applied for a total of 10 iterations, and considered the best result obtained among all iterations. The second experiment is the same as the first one except that we make sure the last iteration of the algorithm is run with DB2. To elaborate, for each $i \in 1, \ldots, 10$, we run $i-1$ iterations of the synthesis algorithm using DB1 and one iteration with DB2, and we pick the best result out of the 10 versions. The reason not using DB2 in the intermediate iterations is that, if it creates majority-5 gates, the subsequent LUT mapping operation can potentially increase the overall size since a single 4-LUT cannot compute majority-5.

The two experiments were done using both the area-oriented and delay-oriented strategies for selecting an appropriate DAG from the database. When using the area-oriented strategy, we select the circuit with the minimum area over the 10 iterations as the best result. Similarly, when using the delay-oriented strategy, we select the circuit with the minimum critical-path length as the best result.

We first perform all experiments under the same assumptions used by Testa et al. [9] that no splitters or buffers are needed on primary inputs but all primary outputs have to be path-balanced using buffers. The result are shown in Table I together with the improvements as compared to the results of Tests et al. [9]. As seen from Table I, the repeated application of our proposed algorithm reduces the delay by 40.62% and decreases the area by 19.45% when the delay-oriented strategy was used, while achieving a 35.98% reduction in the delay and a 21.47% decrease in area when the area-oriented strategy was used. It is evident that having majority-5 gates in the database allows the algorithm to achieve significantly better area and delay improvements as compared to the case where only majority-3 gates were allowed in the database.

In Table II, we present the results for the same experiments under the assumption that splitters are needed for primary inputs to support multiple fanouts and primary outputs have to be balanced. We still assume that path balancing is not needed for primary inputs. It is noteworthy that, even when using splitters for primary inputs, our algorithm achieves better area and delay as compared to the reference work that did not use splitters on primary inputs. We also remark that our flow supports other assumptions on the need of buffers and splitters on primary inputs and outputs, but due to spaces constraints we are unable to present the results.

VI. Conclusion

As seen from the results in Section V, our proposed algorithm for the exact synthesis of AQFP circuits achieves much improved circuits in terms of both area and delay. These improvements are enabled by holistic and simultaneous optimizations of logic and path balancing resources that capture more optimization opportunities compared to prior work in the field. The exact synthesis method employed in this work is able to use unbalanced splitter trees effectively in most parts of the circuit, and the balanced splitter tree assumption is used only on outputs of blocks of logic.Our results also demonstrate that having majority-5 gates can help significantly improve the resource usage.

Moreover, our database generation method is not restricted to using the number of JJs as the area cost. Instead, it can also work with more general cost functions such as the cell area and consider multiple types of splitters with varying branching factors and area with minor modifications to Algorithm 1.

Finally, we note that we can improve our database by considering more DAG structures at the expense of the one-time computational cost of generating the database. Also, the current algorithm depends on an external LUT mapping algorithm and hence a better LUT mapping algorithm can yield better overall results. Alternatively, it is interesting to see if we can directly integrate the database with a technology mapper to achieve even better results.

Acknowledgments

This research was supported by the SNSF grant Supercool 200021_1920981 and the EPFL Open Science Fund.

References

[1] S. R. Whiteley and J. Kawa, "Progress toward VLSI-capable EDA tools for superconductive digital electronics," in *2019 IEEE International Superconductive Electronics Conference (ISEC)*, 2019, pp. 1–3.

[2] D. S. Holmes, A. M. Kadin, and M. W. Johnson, "Superconducting computing in large-scale hybrid systems," *Computer*, vol. 48, no. 12, pp. 34–42, 2015.

[3] K. K. Likharev and V. K. Semenov, "RSFQ logic/memory family: a new josephson-junction technology for sub-terahertz-clock-frequency digital systems," *IEEE Transactions on Applied Superconductivity*, vol. 1, no. 1, pp. 3–28, 1991.

[4] O. A. Mukhanov, "Energy-efficient single flux quantum technology," *IEEE Transactions on Applied Superconductivity*, vol. 21, no. 3, pp. 760–769, 2011.

[5] Q. P. Herr, A. Y. Herr, O. T. Oberg, and A. G. Ioannidis, "Ultra-low-power superconductor logic," *Journal of applied physics*, vol. 109, no. 10, p. 103903, 2011.

[6] M. Tanaka, A. Kitayama, T. Koketsu, M. Ito, and A. Fujimaki, "Low-energy consumption RSFQ circuits driven by low voltages," *IEEE transactions on applied superconductivity*, vol. 23, no. 3, pp. 1 701 104–1 701 104, 2013.

[7] G. Krylov and E. G. Friedman, "Asynchronous dynamic single-flux quantum majority gates," *IEEE Transactions on Applied Superconductivity*, vol. 30, no. 5, pp. 1–7, 2020.

[8] N. Takeuchi, D. Ozawa, Y. Yamanashi, and N. Yoshikawa, "An adiabatic quantum flux parametron as an ultra-low-power logic device," *Superconductor Science and Technology*, vol. 26, no. 3, p. 035010, 2013.

[9] E. Testa, S.-Y. Lee, H. Riener, and G. De Micheli, "Algebraic and Boolean optimization methods for AQFP superconducting circuits," in *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ser. ASPDAC '21, New York, NY, USA, 2021, p. 779–785.

[10] 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 *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE, 2019, pp. 429–436.

[11] C. L. Ayala, R. Saito, T. Tanaka, O. Chen, N. Takeuchi, Y. He, and N. Yoshikawa, "A semi-custom design methodology and environment for implementing superconductor adiabatic quantum-flux-parametron microprocessors," *Superconductor Science and Technology*, vol. 33, no. 5, p. 054006, 2020.

[12] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, P.-E. Gaillardon, J. Olson, R. Brayton, and G. De Micheli, "Enabling exact delay synthesis," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2017, pp. 352–359.

[13] L. Amarù, P. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[14] C. Ayala and N. Yoshikawa, personal communication.

[15] S. B. Akers, "Synthesis of combinational logic using three-input majority gates," in *3rd Annual Symposium on Switching Circuit Theory and Logical Design (SWCT 1962)*. IEEE, 1962, pp. 149–158.

[16] H. S. Miller and R. O. Winder, "Majority-logic synthesis by geometric methods," *IRE Transactions on Electronic Computers*, vol. EC-11, no. 1, pp. 89–90, 1962.

[17] S. Amarel, G. Cooke, and R. O. Winder, "Majority gate networks," *IEEE Transactions on Electronic Computers*, vol. EC-13, no. 1, pp. 4–13, 1964.

[18] L. Amarú, P. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016.

[19] L. Amarú, P.-E. Gaillardon, A. Chattopadhyay, and G. De Micheli, "A sound and complete axiomatization of majority-n logic," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 7. 2889–2895, 2016.

[20] D. S. Holmes, A. L. Ripple, and M. A. Manheimer, "Energy-efficient superconducting computing—power budgets and requirements," *IEEE Transactions on Applied Superconductivity*, vol. 23, no. 3, pp. 1 701 610–1 701 610, 2013.

[21] N. Takeuchi, Y. Yamanashi, and N. Yoshikawa, "Adiabatic quantum-flux-parametron cell library adopting minimalist design," *Journal of Applied Physics*, vol. 117, no. 17, p. 173912, 2015.

[22] R. Cai, O. Chen, A. Ren, N. Liu, C. Ding, N. Yoshikawa, and Y. Wang, "A majority logic synthesis framework for adiabatic quantum-flux-parametron superconducting circuits," in *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, 2019, pp. 189–194.

[23] S.-Y. Lee, H. Riener, and G. De Micheli, "Irredundant Buffer and Splitter Insertion and Scheduling-Based Optimization for AQFP Circuits," in *[Proceedings of the 30th International Workshop on Logic & Synthesis (IWLS 2021)]*, no. CONF, 2021.

[24] S. Muroga, *Threshold Logic and its Applications*. New York: Wiley-Interscience, 1971.

[25] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko, "Fast Boolean matching based on NPN classification," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 310–313.

[26] J. P. Roth and R. M. Karp, "Minimization over Boolean graphs," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 227–238, 1962.

[27] E. Davidson, "An algorithm for NAND decomposition under network constraints," *IEEE Transactions on Computers*, vol. C-18, no. 12, pp. 1098–1109, 1969.

[28] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, "Finding efficient circuits using SAT-solvers," in *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 2009, pp. 32–44.

[29] D. E. Knuth, *The art of computer programming, volume 4A: combinatorial algorithms, part 1*, 2011.

[30] W. J. Haaswijk, "SAT-based exact synthesis for multi-level logic networks," Ph.D. dissertation, EPFL, Lausanne, 2019.

[31] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "SAT-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Trans. on CAD of Integrated Circuits and Systems*, pp. 1–1, 2019.

[32] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.

[33] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 354–361.

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

adding up the complexity of the nodes:

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

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:

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

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

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

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:

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

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:

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

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

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

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:

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

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