A Dynamic Programming-Based, Path Balancing Technology Mapping Algorithm Targeting Area Minimization

Conference Paper · November 2019

DOI: 10.1109/ICCAD45719.2019.8942053

2 authors:

Ghasem Pasandi
University of Southern California
41 PUBLICATIONS **506** CITATIONS
SEE PROFILE

Massoud Pedram
University of Southern California
926 PUBLICATIONS **25,904** CITATIONS
SEE PROFILE

A Dynamic Programming-Based, Path Balancing Technology Mapping Algorithm Targeting Area Minimization

Ghasem Pasandi and Massoud Pedram
Ming Hsieh Department of Electrical and Computer Engineering
University of Southern California (USC), Los Angeles, CA 90089.
{pasandi, pedram}@usc.edu

***Abstract*—Path balancing technology mapping is a method of mapping a technology-independent logical description of a circuit, such as a Boolean network, into a technology-dependent, gate-level netlist. For a gate-level netlist generated by the path balancing mapper, the difference between lengths of the longest and the shortest paths in the circuit is minimized. To achieve full path balancing, it may be necessary to add buffers on signal paths, and in such a case, the cost of buffers must be properly accounted for. This paper presents a dynamic programming-based technology mapping algorithm that generates a minimum-area mapping solution which is guaranteed to be fully path balanced. The fully path balanced mapping solution is essential to conventional superconductive single flux quantum circuits, which will fail otherwise. The balanced mapping solution is also useful in CMOS circuits to avoid (or minimize) unwanted hazard activity and the resulting wasteful dynamic power dissipation as well as to achieve the maximum throughput in a wave-pipelined circuit. Experimental results show that our path balancing technology mapping algorithm decreases total area, static power consumption, and path balancing overhead of single flux quantum circuits by large factors. For example, it reduces the circuit area by up to 111% and by an average of 26.3% compared to state-of-the-art technology mappers.**

I. Introduction

With increasing challenges to the downscaling of Complementary Metal Oxide Semiconductor (CMOS) devices and the impending end of the Moore's law [1], new device, circuit, and architectural solutions are required to keep up with the ever increasing demand for high-speed and low-power circuits and systems. Carbon nanotubes [2], graphene [3], and spin wave [4] devices are some device level solutions which have been recently explored to replace the CMOS technology. Superconductive Single Flux Quantum (SFQ), with a switching delay of *1ps* and switching energy of $10^{-19}J$ is another beyond-CMOS technology candidate for achieving high-performance and energy-efficient circuits and systems [5]. The first family of SFQ logic: Rapid Single Flux Quantum (RSFQ), was developed in the 1980s and is as fast as *770GHz* at *T=4K* [6], [7]. SFQ circuits are made of Josephson Junctions (JJs), which are superconductive devices operating based on the Josephson effect [8].

SFQ devices are pulse-based logic which use the presence or absence of a single quantum of magnetic flux ($\Phi_0 = h/2e = 2.07mV \times ps$) to represent the binary "1" or "0" information, respectively. Most of SFQ gates receive a clock signal; indeed an SFQ logic gate can be modeled as a normal (combinational) logic gate with a clocked D-Flip-Flop (DFF) attached to its output. If a gate has more than one fanout, special clockless gates called *splitters* are inserted at the output of this gate to create multiple fanouts [9]. Finally, SFQ circuits need to be path balanced [10], [11], which means the length of any path from a Primary Input (PI) to any Primary Output (PO) in terms of the (clocked) gate count must be the same. This requirement is essential to the correct operation of the SFQ circuit. Fig. 1 shows the circuit diagram of an SFQ NOT gate and corresponding waveforms showing its functionality. As seen, after the clock pulse comes, when there is no input pulse, a pulse is generated at the output of the gate representing a "1". On the other hand, when there is an input pulse, no pulses are generated at the output, meaning a "0". For more details about SFQ circuits, see [6]–[12].

We have observed that if the path lengths of a given SFQ circuit are not controlled during the technology mapping, we may end up having a circuit that needs the insertion of many path balancing DFFs to ensure the said path balancing requirement. Fig. 2 compares original logic gate and required path balancing DFF counts for a few benchmark circuits. These circuits are mapped using a state-of-the-art open-source academic logic synthesis tool called ABC [13], and are path balanced and retimed [14] similar to [9]. For IntDiv8 circuit (an 8-bit integer divider), the DFF count is as high as $4.5\times$ that of the original logic gate count in the circuit. On the other hand, if the balancing requirement of the circuit is considered during the technology mapping, it is possible to come up with much better mapping solutions which need fewer path balancing DFFs. For example, for a 2-bit adder, as shown in Fig. 3, two mapping solutions with the same gate count and area exist. However, one of them requires 10 path balancing DFFs whereas the other one needs only three path balancing DFFs.

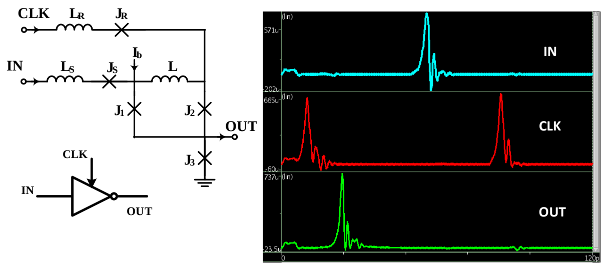


Fig. 1: Schematic of an SFQ NOT gate and its waveforms. In case of not having any input pulses, an output pulse is generated after arrival of the clock pulse, representing a "1". However, when there is an input pulse, no pulses are generated at the output, meaning a "0".

In this paper, we present a technology mapping algorithm which takes the balancing overhead into account and tries to find the most cost-efficient mapping solutions for a given circuit. Thanks to this algorithm, the total gate count (accounting for both the original logic gates and the path balancing DFFs) is reduced, which results in decreasing the total area and static power consumption of the circuit (static power is the main source of power consumption in SFQ circuits [15]). The proposed algorithm provides an optimal solution for balanced tree mapping, and a modified version of it acts as a very effective heuristic for path balancing general Directed Acyclic Graph (DAG) mapping. The optimality of the algorithm for tree mapping is proven by developing models relating the number of required path balancing DFFs at each level of a given subject tree to the leaf node count of this tree and to its height.

Our path balancing technology mapping algorithm can be used in wave-pipelined circuits to increase the rate at which data can propagate through the circuit (throughput) by decreasing the differences between lengths of the shortest and longest paths [17], [18]. It also can be used in any technologies or design methods that require full path balancing. For example, in stateful logic [19], each gate combines logic and memory and if all logic paths have equal length, the throughput of the circuit can be increased by orders of magnitudes at the expense of an area overhead of required path balancing buffers [19]. Our path balancing technology mapping

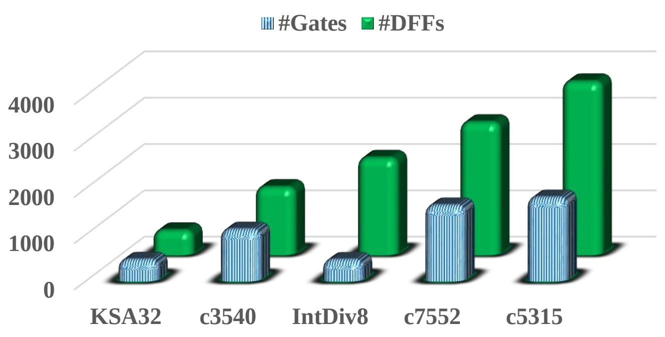


Fig. 2: Path balancing DFF count versus original gate count for a few benchmark circuits. KSA32 is a 32-bit Kogge-Stone adder; IntDiv8 is an 8-bit integer divider and c5315, c7552, and c3540 are chosen from ISCAS benchmark suite [16]. These circuits are mapped using *map* command of ABC [13], and path balanced and retimed using full path balancing [9], [10] and retiming [14] algorithms.

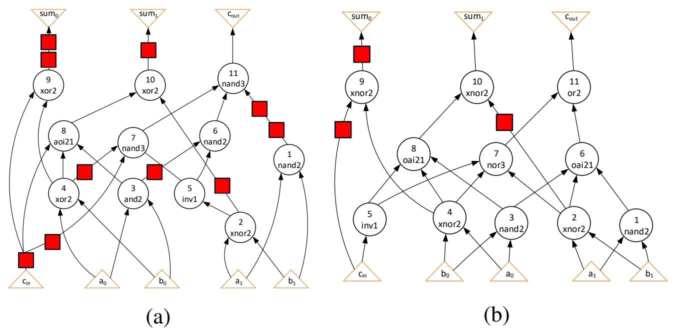


Fig. 3: Two mapping solutions for a 2-bit Kogge-Stone adder using mcnc.genlib library of gates (red squares are path balancing DFFs): (a) Consuming 11 gates with area of 37.0 units and requiring 10 path balancing DFFs , (b) Consuming 11 gates with the same area but requiring only three path balancing DFFs. It is obvious that the total area of gates and path balancing DFFs in the second circuit is much less than the same in the first one.

algorithm helps reducing this area overhead. In the rest of the paper, we use path balancing DFFs or buffers to refer to the elements that should be inserted onto short paths to satisfy the full path balancing property.

The main contributions of this paper are as follows:

- Capturing the cost of required path balancing buffers during technology mapping and developing a dynamic programming-based, balance-aware technology mapping algorithm.
- Developing new formulas relating the number of required path balancing buffers at each level of a tree to the leaf node count of this tree and its height.
- Proving the optimality of the presented technology mapping algorithm for tree-like structures.
- Presenting an effective DAG mapping heuristic for area minimization in path balancing technology mapping.

II. Related Work

Keutzer [20] developed *DAGON*: a technology binding tool with local optimizations. DAGON partitions general DAGs into forests of trees and finds optimal solutions for these trees by pattern matching. Cong and Ding [21] developed *FlowMap*: the first load-independent delay optimal DAG mapping algorithm for FPGAs. Chaudhury and Pedram [22] presented an optimal area-delay mapper by constructing the pareto optimal frontier using dynamic programming. Mishchenko *et al.* [23] developed a priority-cut-based technology mapping tool in which the priority of selecting matches for individual nodes can be chosen as delay, area, or any other desired metric. In [24], majority-based logic synthesis is introduced and is further developed in [25]. Soeken *et al.* [26] proposed an effective algorithm for exact logic synthesis of Boolean networks using majority-inverter graphs, which improved both area and delay after lookup table (LUT)-based technology mapping. In [27], a gate sizing and buffer insertion algorithm is proposed to achieve path balancing in CMOS circuits; this algorithm reduces load capacitance and glitches at the same time. Pasandi *et al.* [12] developed a depth minimization with path balancing algorithm for technology mapping, which provides optimizations for the product of the worst stage delay and length of the longest path, as well. In [10], a path balancing technology mapping algorithm is presented for single flux quantum logic circuits with the goal of minimizing *total required number of path balancing DFFs*; this algorithm is proven to provide optimal tree mapping solutions in case of having up to 2-input gates in the given library. In [28], an *SOP-balancing* algorithm by generalizing an AND-balancing approach is presented; while the AND-balancing algorithm is limited to multi-input AND gates, the SOP-balancing algorithm supports more complex functions, therefore, it has opportunities for further delay minimization.

Regarding area minimization in technology mapping, there are several published papers; Farrahi and Sarrafzadeh [29] proved that even restricted cases of the lookup-table count minimization as a measure of area in FPGAs are NP-complete for *DAGs*. However, these authors presented a polynomial time algorithm for minimum-area *tree mapping* and presented a polynomial time heuristic for area minimization in general Boolean networks. Chaudhary and Pedram [30] presented a near optimal algorithm for technology mapping that minimizes area under delay constraints; this is achieved by generating area-delay curves in a topological ordering traversal of the given subject graph and selecting solutions from these curves in a reverse topological ordering traversal. Chen and Cong [31] studied the technology mapping problem for FPGAs targeting area minimization; they considered the potential node duplication during the cut-enumeration process such that the mapping cost is encoded into cuts, and after the timing constraints are met, the constraints on non-critical paths are relaxed in order to minimize the area. Manohararajah *et al.* [32] presented IMap, an iterative technology mapping tool that supports depth-oriented, area-oriented, and duplication free mapping modes; in the depth-oriented mapping mode, area is a secondary objective, while in the area-oriented mapping mode, area is the first objective.

In this paper, we present a technology mapping framework targeting area minimization in logic circuits which require full path balancing. For this purpose, our new technology mapping cost function captures the effect of the path balancing buffers and our algorithm minimize total area including the area of gates and these buffers. We will prove that our algorithm provides optimal tree mapping solutions and a modified version of it, by capturing node duplication encoded in the information of logic levels, acts as a very effective heuristic for DAG mapping.

There are two main differences between this paper and the path balancing technology mapping tool presented in [10]: (i) In [10], total number of path balancing DFFs is minimized and total gate count is not considered in optimizations. This may increase the sum of gate and DFF counts, which potentially can increase the total area. To solve this problem, in this paper, we define and minimize a new cost function which captures effect of gates and path balancing buffers (DFFs) at the same time, resulting in ensuring total area reduction. (ii) In [10], the optimality of technology mapping algorithm is proven for trees, assuming that the gates in the library have up to two inputs. However, in this paper, we provide proof of optimality for a general case of having any multi-input gates in the library. In addition, in this paper we propose a novel heuristic for mapping general DAGs by encoding duplication of nodes into their logic levels, which helps generating more area efficient solutions for DAGs. Moreover, in favor of CMOS circuits, we present a modified version of path balancing technology mapping algorithm which preserves the best obtained critical path delay, while it still reduces the path balancing overhead.

III. Path Balancing Technology Mapping

Given a Boolean network, technology decomposition [33] generates a DAG consisting of 2-input (N)AND and inverters called the *subject graph*. This DAG can be decomposed into trees and in a special case it can be a tree itself. Technology mapping binds gates from a given library to a node or set of nodes of the subject graph, generating a mapping DAG (a mapping tree in the special case).

A. Terminology and Notation

Buffer Node: A single-fanin, single-fanout node which is added by the path balancing technology mapper to some selected edges of the mapped circuit to ensure that the circuit is fully path balanced. A buffer node is replaced by a DFF in SFQ circuits.

And-Inverter Graph (AIG): A subject graph in which nodes are 2-input AND gates. Inverters are modeled as a field in the data structure of the node [13]. Therefore, if the AIG is a tree, it will be a binary tree rooted at node t with every node having exactly two inputs.

Reverse level R_i of node i in a tree T: Length of the longest path (in terms of the node count including the node itself) from node i to the root node t in T. The root is at reverse level r=R_t=1.

Leaf node of a tree T: An input to the tree (we assume a tree input is modeled as a fanin-free, dummy node).

Logic level L_i of node i in a tree T: Length of the longest path (in terms of the node count including the node itself) from any tree inputs to this node. The tree inputs are at logic level 0.

Height H of a tree T: Number of nodes on the longest path from any tree inputs to the root (this is the same as the largest reverse level of any node).

n**:** Denotes the number of leaf nodes in tree T.

N**:** Denotes the number of internal nodes of tree T including the root node t.

$y(z,r,i)$**:** Denotes the number of nodes with z inputs at reverse level r of tree T rooted at node i.

$Y(z,i)$**:** Denotes the number of nodes with z inputs in tree T rooted at node i: $Y(z,i)$=$\sum_{r=1}^{H} y(z,r,i)$.

B. Mapping Algorithm

The balanced tree mapping problem is solved using dynamic programming (DP). The input of the technology mapper is an AIG with tree structure, and the pattern graphs are gates in the library [34], or supergates[1] [35] generated using these gates. Matches at node i are obtained by enumerating all *k-feasible* cuts [21], [23] of the node and examining all supergates that can implement function of this node based on inputs of each computed cut. In the path balancing technology mapping, cost of any added buffer nodes (DFFs in SFQ circuits) are considered in the total area cost of the mapping solution. The cost of mapping a sub-tree rooted at node i in a given subject tree can thus be written as follows:

$$Cost^*(i) = min\{Cost(m,i)\}\,\forall m \in matches(i)$$

$$Cost(m,i) = Cost(m) + SLD(SuppC(m,i)) \times Cost_{BFR} + \sum_{\forall k \in SuppC(m,i)} Cost^*(k) \quad (1)$$

where $Cost^*(i)$ is the dynamic programming (i.e., minimum) cost of the mapping of the AIG sub-tree rooted at node i, $Cost(m,i)$ is the cost of a particular match m at node i, and $matches(i)$ and $SuppC(m,i)$ denote the possible matches for node i and the support (boundary nodes) of the cut (among *k-feasible* cuts of node i) that corresponds to the match m. In the above equation, $SLD(S)$ denotes a function that returns the sum of the absolute value of level differences of pairs of nodes in a node set S. For example, $SLD(\{i,j,k,l\})$ where $L_i = 1, L_j = L_k = 2, L_l = 4$ is equal to $(4-1)+2\times(4-2) = 7$. Cost of a path balancing buffer is denoted by $Cost_{BFR}$. The product of $SLD(SuppC(m,i))$ and $Cost_{BFR}$ thus gives the total cost of the require path balancing buffers.

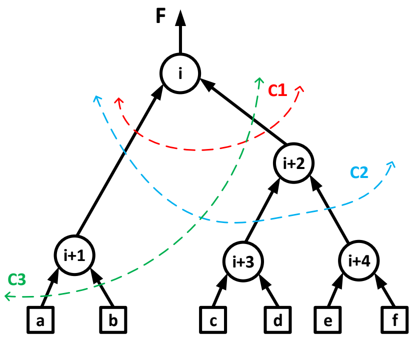


Fig. 4: An AIG tree with five nodes. 3-$feasible$ cuts (C_1, C_2, and C_3) of node i in this subject tree are shown. Leaf nodes are shown with squares. Nodes are labeled in a Breadth First Search (BFS) order (root, left, right).

Example 1: Consider the AIG tree shown in Fig. 4. The value of the optimal solution for mapping the tree rooted at node i when k=3 (in the cut computation procedure) is calculated as follows:

$$\begin{aligned}
&Cost^*(i) = min\{\\
&Cost(mC_1) + SLD(\{i+1, i+2\}) \times Cost_{BFR} + Cost^*(i+1)\\
&+ Cost^*(i+2),\\
&Cost(mC_2) + SLD(\{i+1, i+3, i+4\}) \times Cost_{BFR} + Cost^*(i+1)\\
&+ Cost^*(i+3) + Cost^*(i+4),\\
&Cost(mC_3) + SLD(\{a, b, i+2\}) \times Cost_{BFR} + Cost^*(i+2)\\
&+ Cost^*(a) + Cost^*(b)\\
&\}
\end{aligned} \quad (2)$$

where mC_1, mC_2, and mC_3 are costs of supergates (there can be several for each) which implement function of node i based on inputs of cuts C_1, C_2, and C_3, respectively. $Cost^*()$ is equal to 0 for leaf nodes. Squares in Fig. 4 denote leaf nodes. Please note that in the rest of this paper (Figs. 5, 6, 7), the leaf nodes are not shown.

The minimum-cost fully-path-balanced mapping solution for a given tree is generated using the following algorithm which is called *BalancedMap*:

First, k-$feasible$ cuts and their truth-tables are computed for each node [21], [23]. Next, in a topological ordering traversal starting from the PIs of the tree, nodes are visited and the best solution for each node which gives the least value for the said cost function is computed using the dynamic programming algorithm with Eq. 1 as its value of the optimal solution. When the root is visited, the optimal mapping solution for the whole tree is computed which is generated by a reverse topological ordering traversal from root to leaf nodes of the tree. Finally, splitters and buffers are inserted.

Algorithm 1 shows the pseudo code of our path balancing technology mapping algorithm. In lines 1, the mapping manager is constructed and a few pre-processing steps such as generating supergates, computing k-feasible cuts and their truth tables are done. These steps are similar to ABC [13]. In lines 2-3, the most cost efficient solutions minimizing the cost function in Eq. 1 are calculated. In line 4, the optimally mapped circuit is obtained by traversing the tree back from its root to its PIs. In line 5, the mapped circuit is given to a function to insert path balancing buffers (DFFs) wherever it is needed[2] and

[1]Supergates are small trees, which are generated by exhaustively concatenating original gates in the given library.

[2]This function traverses over all gates; for a gate, it finds maximum logic level among its immediate fanin gates (L_{max}), and inserts $L_{max} - L_i$ buffers to the immediate fanin gate G_i, where L_i is the logic level of this fanin gate.

Algorithm 1: BalancedMap

```
Input: Tree $T = (V, E)$ rooted at node $i$ comprising node sets
       $V$ and edge set $E$, and gate library $\mathscr{L}$
Output: The optimally mapped, path balanced circuit with
        inserted splitters, $N_{Map}$
1 Start the mapping manager and perform pre-processing steps.
2 for each node v in V do
3   Find the most cost efficient mapping solution based on Eq. 1.
  // generating the mapping tree:
4 $T_i^*$ = Network_From_Map ($T$)
  // Inserting path balancing DFFs and
     performing retiming:
5 $N_2$ = add_Buffers_Retime($T_i^*$)
  // inserting splitters:
6 $N_{Map}$ = InsertSplitters($N_2$)
7 return $N_{Map}$
```

to perform standard retiming [14][3]. Finally, if this algorithm is used for mapping SFQ circuits, in line 6, splitters are inserted to outputs of gates with more than one fanouts; this step can be ignored for technologies that do not require splitters for generating fanouts more than one.

The complexity of computing $k\text{-}feasible$ cuts is $O(KMN)$, where K is a constant, N is the node count, and M is the edge count [21]. The complexity of selecting the best solution for a node after having its $k\text{-}feasible$ cuts is $O(K'Np)$, in which, K' is the maximum number of $k\text{-}feasible$ cuts for a node in the given subject graph, and p is the number of gates in the library. The complexity of generating the mapped circuit is $O(N)$, because each node will be visited once at maximum. The complexity of adding path balancing buffers is $O(M+N)$, because each node is visited once and for a visited node each input edge is visited one time. The complexity of retiming is $O(MN \times log(N))$ [14]. Since in the splitter insertion process each node is visited once, the complexity of inserting splitters is $O(p' \times N)$, where p' is a constant value representing the required time for inserting splitters at the output of a gate. Therefore, the complexity of the whole algorithm is determined by the retiming step to be $O(MN \times log(N))$.

C. Proof of Optimality

In this section, we will prove that the principle of optimality of dynamic programming[4] is satisfied in our problem formulation and therefore the aforesaid algorithm produces the optimal tree mapping solution in polynomial time. For this purpose, we need to first develop a few models relating height, leaf node count, and path balancing buffer count in a tree.

Total leaf node count of a full binary tree with height H is 2^H. For a *full b-ary tree*[5], this number is equal to b^H. A tree which is not full has fewer leaf nodes. If a node with in-degree of $b' < b$ is present at reverse level r of a tree, it will contribute in reduction of leaf node counts by the following amount: $b^{H-r+1} - b' \times b^{H-r}$. As a sanity check, a node with maximum in-degree (b'=b) does not contribute in reduction of leaf node counts. Based on this fact and using some basic properties of trees, a closed form formula for the leaf node count of a tree T rooted at t based on its height and number of nodes (with more than one input) and buffers at different levels of the tree can be obtained, written as follows:

$$b^H - (b-1) \times \{y(1,2,t) \times b^{H-2} + y(1,3,t) \times b^{H-3} + ... + y(1,H,t)\} \\ -(b-2) \times \{y(2,2,t) \times b^{H-2} + y(2,3,t) \times b^{H-3} + ... + y(2,H,t)\} - ... \\ = n \quad (3)$$

[3]To minimize the register count; path balancing buffers are treated as registers in the retiming algorithm.

[4]Optimal solution for a subset of the problem should be built from the optimal solutions for its sub-problems.

[5]The tree with maximum number of nodes for the given height in which all nodes have in-degree of b.

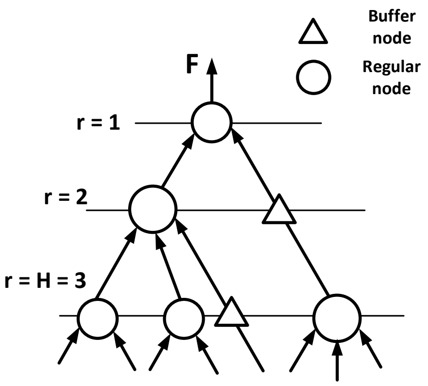


Fig. 5: An example to verify the correctness of Eq. 4 for giving total leaf node count of a general tree. The leaf node count of this tree is : $2 \times 3^2 - 2 \times \{1 \times 3 + 1\} + 1 \times \{0 + 2\} = 18 - 8 - 2 = 8$ ✓.

The above equation starts with the leaf node count of a full $b\text{-}ary$ tree (b^H) and first accounts for contributions of nodes with in-degree of b-1 in reduction of the total leaf node count of the tree, then it accounts for nodes with in-degree of b-2, ..., all the way to nodes with in-degree of one. This equation can be rewritten as follows:

$$\sum_{z=1}^{b-1} \left\{ (b-z) \times \sum_{r=2}^{H} \{y(z,r,t) \times b^{H-r}\} \right\} = b^H - n \quad (4)$$

Note that in the above equation, it is assumed that the root node t has the in-degree of b. If the root node has fewer number of inputs (e.g. $b' < b$), the right hand side of Eq. 4 should be replaced with $b' \times b^{H-1} - n$.

Example 2: Suppose that there are up to 3-input gates in the library; the left hand side of Eq. 4 will be $\sum_{z=1}^{2}\{(3-z) \times \sum_{r=2}^{H}[y(z,r,t) \times 3^{H-r}]\}$. Fig. 5 shows a tree with height H=3. The leaf node count of this tree is calculated as follows:

$$2 \times \{1 \times 3 + 1\} + 1 \times \{0 + 2\} = 2 \times 3^2 - n \\ \Rightarrow n = 18 - 8 - 2 = 8 \quad \checkmark \quad (5)$$

Fig. 5 verifies the correctness of these calculations.

Lemma 3.1: The total leaf node count of a tree with N internal nodes is calculated as follows:

$$n = \left\{ \sum_{i=1}^{N} (b_i - 1) \right\} + 1 \quad (6)$$

where, b_i is the in-degree of the internal node i.

Proof by induction: *Base case:* a tree with only one internal node (node 1) has b_1 leaf nodes. *Induction step:* assume that a tree with j internal nodes has $\{\sum_{i=1}^{j}(b_i - 1)\} + 1$ leaf nodes. If a leaf node is replaced with a new internal node (node j+1), one leaf node will be lost, but b_{j+1} new leaf nodes will be generated by the added node. In total, b_{j+1}-1 leaf nodes will be added. So, the total leaf node count will be as in Eq. 6.■

A special case of the above lemma (as mentioned in *lemma 1* in [10]) for b=2 is: n=N+1. Instead of going through all internal nodes and summing up their *in-degree minus one* values (as in Eq. 6), we can traverse a tree level by level and account for contributions of gates with an specific in-degree in total leaf node count and repeat it for all in-degree values larger than or equal to two (we start with in-degree of two, because for in-degree of one, b-1 = 0). Therefore, using the $y(z,r,t)$ terminology, for a tree with height H, Eq. 6 can be equivalently rewritten as follows:

$$n = \sum_{z=2}^{b} \left\{ (z-1) \times \sum_{r=1}^{H} y(z,r,t) \right\} + 1 \quad (7)$$

Furthermore, the cost function introduced in Eq. 1 can be simplified as follows. Let T_i^* denote the mapping solution corresponding to $Cost^*(i)$ in Eq. 1. By assuming that the cost of a library gate is proportional to the number of its inputs, the normalized cost of T_i^* (denoted by $nCost^*(i)$) may be calculated as follows:

$$nCost^*(i) = \sum_{z=1}^{b} z \times Y(z, i) \tag{8}$$

where b is the maximum in-degree of any library gates. Please note that since the iterator z starts from 1, cost of the path balancing buffers (as single input nodes) are taken into account. This cost function is equal to the total edge count in T_i^*. Clearly if i is selected to be the root node t, then $nCost^*(t)$ denotes the total cost of the best mapping solution for the given *AIG tree*. As mentioned before, by assuming that the area of a library gate with $2z$ inputs is $2\times$ the area of a gate with z inputs, Eq. 8 gives the total area cost of the best mapping solution for an AIG tree (including any path balancing buffers).

The above formula can be further rewritten as follows:

$$nCost^*(i) = \sum_{z=1}^{b} \left\{ z \times \sum_{r=1}^{H} y(z, r, i) \right\} \tag{9}$$

Using Eqs (7, 8), the above cost function is rewritten as follows:

$$nCost^*(i) = \sum_{r=1}^{H} y(b-1, r, i) + 2\times\sum_{r=1}^{H} y(b-2, r, i) + 3\times\sum_{r=1}^{H} y(b-3, r, i)$$
$$+ \ldots + (b-1)\sum_{r=1}^{H} y(1, r, i) + Const. =$$
$$\sum_{r=1}^{H} \left\{ \sum_{z=1}^{b-1} (b-z) \times y(z, r, i) \right\} + Const. \tag{10}$$

where $Const.$ refers to some constant values with no impact on the minimization procedure, thus, can be ignored. Therefore, the final form of the cost function for mapping a tree rooted at node t is expressed by Eq. 11.

$$nCost^*(t) = \sum_{r=1}^{H} \left\{ \sum_{z=1}^{b-1} (b-z) \times y(z, r, t) \right\} \tag{11}$$

On the other hand, Eq. 4 which relates the total leaf node count of a tree to its height and number of nodes at different reverse levels, can be rewritten as follows:

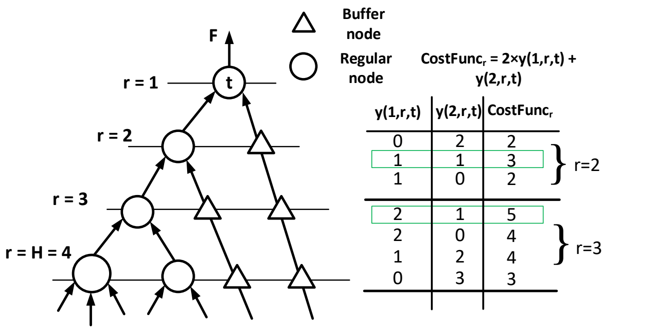
$$\sum_{r=2}^{H} \left\{ \left\{ \sum_{z=1}^{b-1} (b-z) \times y(z, r, t) \right\} \times b^{H-r} \right\} = b^H - n \tag{12}$$

The expression inside braces in Eq. 11 is the portion of the total cost function which corresponds to the reverse level r, hence, can be denoted by $CostFunc_r$. Eq. 12 acts as a constraint for minimizing value of the cost function in Eq. 11. The next two lemmas introduce trees which give lower/upper bounds for the cost function.

Lemma 3.2: Among all b-*ary* trees with height H and total leaf node count of n, the tree which *maximizes* $CostFunc_r$ at each reverse level $1 \le r \le H$ (starting with smaller reverse levels) subject to satisfying Eq. 12 for the whole tree, gives the *lower bound* for the value of the cost function in Eq. 11.

Outline of the Proof: The intuition behind this lemma is that since the right hand side of Eq. 12 is fixed, in order to minimize the cost function (which is equal to sum of all $CostFunc_r$ at different reverse levels), $CostFunc_r$ that is multiplied by larger power of b in the left hand side of the Eq. 12 (which corresponds to smaller reverse levels) should have larger value. Therefore, if we start with smaller reverse levels and maximizes their $CostFunc_r$, the total cost function will be minimized. Please note that if maximizing $CostFunc_r$ for reverse level r gives rise to a final tree topology that violates Eq. 12, it is not accepted and a smaller value should be used for that.

Example 3: Suppose that we want to generate the most cost efficient tree with n=7, H=4 rooted at t, and having b=3. Having these values, $CostFunc_r$ will be $2 \times y(1, r, t) + 1 \times y(2, r, t)$. For the root, a 2-input gate should be used. For the next level, there are some options for choosing gates; three of them are mentioned in the first set of



y(1,r,t)	y(2,r,t)	CostFunc$_r$	
0	2	2	r=2
1	1	3	
1	0	2	
2	1	5	r=3
2	0	4	
1	2	4	
0	3	3	

Fig. 6: An example of the most cost efficient tree with n=7, H=4, and b=3. The right-most column in the table shows portions of the total cost function for a specific reverse level (r).

numbers in the table shown in Fig. 6. The second option which gives the highest valid value for $CostFunc_2$ is selected. This is implemented by using one 2-input gate and one buffer node at reverse level r=2. Note that at this reverse level, there are more options that are not valid. For example, someone may use two 3-input gates at r=2. This will generate six leaf nodes at the reverse level 2. We have to generate two more reverse levels (to reach H=4) by putting at least one 2-input gate per level. This will generate at least two more leaf nodes for the final tree according to lemma 3.1. Since six leaf nodes are generated up to now, only one more leaf node can be generated to reach n=7. Thus, using two 3-input gates at reverse level 2 of this tree, the maximum achievable height for n=7 will be H=3, therefore, using two 3-input gates at reverse level 2 does not lead to a valid solution. For reverse levels 3 and 4, similar gate assignment procedure can be used to generate a valid tree. Due to space limitations, in the table of Fig. 6, options for reverse level 4 are not shown. Using Eq. 11, the value of the cost function for the tree shown in Fig. 6 is $(1 \times 1) + (2 \times 1 + 1 \times 1) + (2 \times 2 + 1 \times 1) + (2 \times 2 + 1 \times 1) = 14$.

Lemma 3.3: Among all b-*ary* trees with height H and total leaf node count of n, the tree which *minimizes* $CostFunc_r$ at each reverse level $1 \le r \le H$ (starting with smaller reverse levels) subject to satisfying Eq. 12 for the whole tree, gives the *upper bound* for the value of the cost function in Eq. 11.

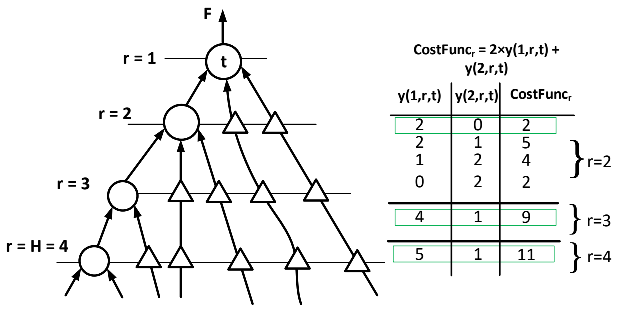
Outline of the Proof: With similar explanations given for the lemma 3.2, the intuition in this lemma is to give smaller values for $CostFunc_r$ at smaller reverse levels because they are multiplied by larger powers of b in Eq. 12, resulting in maximization of the sum of all of them, which is equivalent to maximizing the cost function.

Example 4: The least balanced tree with n=7, H=4 rooted at t, and b=3 (the same values as in *Example 3*) is shown in Fig. 7. Note that for reverse level r=2, there is another option which makes $CostFunc_2$ equal to two, but it is not selected as the final choice. This option corresponds to having one 3-input and two 2-input gates at this reverse level. This will generate seven leaf nodes up to now. We cannot generate more leaf nodes and still two more reverse levels have to be generated, which is not possible. Thus, this option is not valid. The value of the cost function for this example is $(0) + (2 \times 2) + (2 \times 4 + 1 \times 1) + (2 \times 5 + 1 \times 1) = 24$, which is larger than 14 in *Example 3*.

In the above two lemmas, it is assumed that the buffer nodes are only used for path balancing. Note that in Eqs. 11, 12, the upper bound for iterator z can be increased to b without changing values of the equations ($b - b = 0$). This allows selection of gates with the maximum in-degree b in the above lemmas.

Lemma 3.4: If T_1 (with height X) and T_2 (with height $X + p$) are two valid solutions for mapping a tree rooted at node t of a given subject tree, and $Cost_1$ and $Cost_2$ are their cost functions based on Eq. 11, $Cost_2 - Cost_1$ cannot be less than p, where p is a positive integer.

Outline of the Proof: It is enough to prove the lemma for the following T_1 and T_2 trees with the mentioned cost functions below



y(1,r,t)	y(2,r,t)	CostFunc$_r$	
2	0	2	
2	1	5	r=2
1	2	4	
0	2	2	
4	1	9	r=3
5	1	11	r=4

Fig. 7: An example of the least cost efficient tree with n=7, H=4, and b=3.

(the proof is removed):
T_1**:** The least cost efficient tree with height X.

$$Cost_1 = b \times n_b + 0 = b \times \frac{b^X - 1}{b - 1} \tag{13}$$

T_2**:** The most cost efficient tree with height $X + p$.

$$Cost_2 = 2{\times}n_2{+}b{\times}n_b{+}BFRcnt = 2{\times}(X{+}p{-}1){+}b{\times}(X{+}p){+} (X + p - 2) \times (X + p - 1)/2 \tag{14}$$

where $BFRcnt$ is the total path balancing buffer count.

Theorem 3.5: The presented minimum-cost fully-path-balanced tree mapping algorithm satisfies the principle of optimality of DP, therefore, it gives the optimal solution in polynomial time.

Proof: Let S_t refers to the optimal solution for mapping a tree rooted at node t. For simplicity and without loss of generality, suppose that S_t is built of two sub-parts i and i'. Let's call the part of S_t related to mapping the sub-tree rooted at i (i'), S_i ($S_{i'}$). The above theorem claims that S_i ($S_{i'}$) is the optimal solution for mapping the sub-tree rooted at i (i'). By contradiction, suppose that there is another solution for one of these sub-problems (S'_i for i) which is more expensive than the original optimal solution (S_i) for mapping the sub-tree rooted at this node, but it gives rise to a better solution for mapping the tree rooted at node t. This can only happen if using S'_i results in returning a value by $SLD()$ in Eq. 1, which is smaller than the difference between cost of S_i and S'_i. Based on lemma 3.4, this cannot happen. Therefore, the theorem is proven ■.

As mentioned in Section III-B, the above proof is based on the assumption of having linear relation between cost of a gate and its number of inputs (which is a reasonable assumption). If this relation is not linear (e.g. exponential), then the optimality of the proposed algorithm cannot be guaranteed.

D. DAG Mapping

Most of the terminology and methods presented for trees in this section can be easily extended for DAGs. For example, the reverse level for node i can be defined as length of the longest path from node i to any root node of the graph. For DAG mapping, we developed the following heuristic: After computing $k\text{-}feasible$ cuts and their functions for each node, similar to what is presented in Section III-B, in a topological ordering traversal, the value of the cost function is computed for each node using Eq. 1. The main difference is that while choosing the best cut for a node, if the following inequality holds for one of the inputs (boundary nodes) of a cut (e.g. q), $Cost^*(q)$ in Eq. 1 will be set to *0*.

$$(rep(q) - 1\ ) \times \frac{L_q}{D} \geq 1 \tag{15}$$

where D is the depth of the AIG representation of the given Boolean network, which is defined as the largest logic level among its nodes, $rep(q)$ is the number of times node q has been used up to now, and L_q is the logic level of node q. The intuition behind this heuristic is to prevent replication of big cones of logic, which are already implemented and used for implementing other nodes. Note that multiplication by the normalized logic level (i.e., $\frac{L_q}{D}$) should not be removed from the said inequality, otherwise, the generated mapping solutions will be very unbalanced. For mapping DAGs, line 3 of the Algorithm 1 should be modified to capture the revision given by Eq. 15.

IV. Experimental Results

The presented path balancing technology mapping algorithm is implemented inside ABC [13]. We implemented two versions of path balancing mappers, *BalancedMap (BM)*, and *BalancedMapDelay (BMD)*. In BM, the most cost efficient solutions as in Section III are computed. BMD chooses the most cost efficient matches for each node of the network subject to not degrading the best achieved delay in a prior delay optimization pass. For comparison, we have included results from [10], which is referred to as *PBMap* from herein, and also extracted experimental results using the synthesis approach presented in [9] for SFQ circuits; in this synthesis approach, circuits are mapped using default cut-based technology mapper of ABC, followed by inserting path balancing DFFs, applying standard retiming to reduce the total DFF count, and finally, inserting splitters. In the rest of this paper, we refer to this baseline synthesis approach by *Base*.

An SFQ library of gates as in [34] consisting of *and2*, *or2*, *xor2*, *not*, *splitter*, *JTL*, *DFF*, and *MUX21* logic gates is used. Several benchmark circuits from ISCAS [16], EPFL [36] benchmark suites, and some arithmetic circuits are used. The complexity of these circuits ranges from *s38584* with 12/278 I/Os, 19407 nodes, 32910 edges, and 25185 cubes, *sin* with 24/25 I/Os, 5416 nodes, 10832 edges, and 5416 cubes to *dec* with 8/256 I/Os, 304 nodes, 608 edges, and 304 cubes. Table I shows results for area and delay. For *Area*, two sets of numbers are reported; the numbers inside parenthesis are the total area of gates, path balancing DFFs, and splitters, while the area numbers outside parenthesis does not include area of splitters. The reason behind reporting two sets of results for area is to show that our algorithms not only reduce the total area of gates and DFFs as our cost function demands, but it also provides area reduction in case of considering area of splitters too. *Delay* (in ps) is the critical path delay, which is the sum of delays of gates in the critical path of the mapped circuit.

Figs. 8, 9, and 10 compare the value of the cost function (Eq. 8), normalized total static power consumption, and total Josephson junction count for BM, PBMap, and Base. In the following, some statistics are reported. Similar to the method in Table I, for area two sets of numbers are reported; the one inside parenthesis is for area of gates, DFFs, and splitters. To save space, the results for total number of path balancing DFFs is not included in this paper, but their statistics are mentioned in the following. For the *priority* circuit, the cost function, area, total JJ count (#JJs), total number of required path balancing DFFs (#DFFs), and static power consumption are reduced by 1.07×, 1.12× (1.11×), 1.09×, 1.18×, and 1.10× compared to the Base, but its delay is degraded by 6.0%. On average for all benchmark circuits, BM decreases the value of the cost function, area, #JJs, #DFFs, and static power consumption by 86.1%, 29.5% (26.3%), 25.6%, 27.8%, and 25.1% over Base, while the average delay is increased by 5.6%. As seen, the average area reduction in case of considering the area of splitters is reduced from 29.5% to 26.3%. This is because on average the path balancing algorithm generates circuits with more fanout counts, therefore, it requires more number of splitters. BM also shows superiority over PBMap; it reduces area, #JJs, and static power consumption by an average of 6.0%, 6.2% (5.9%), and 5.8% and up to (for ISCAS c1908 benchmark circuit) 24.6%, 24.9% (24.7%), and 24.3% compared with PBMap. BMD is able to offer savings on all of these metrics over the baseline without degrading the critical path delay. On average for all benchmark circuits, BMD reduced the value of the cost function, area, #JJs, #DFFs, and static power consumption by 5.7%, 3.4% (2.3%), 3.8%, 6.0%, and 3.3% compared to the Base.

TABLE I: Area and delay results for BalancedMap (BM), baseline (Base) [9], and PBMap [10]. For area two sets of numbers are reported; numbers inside parenthesis include area of gates, path balancing DFFs, and splitters, while the area numbers outside parenthesis do not include area of splitters. Since delays of circuits generated by PBMap is very close to the same generated by BM and due to the lack of space, they are removed from the table.

	Area (mm^2)			Delay (ps)	
Circuits	Base	PBMap	BM	Base	BM
c5315	26.7 (42.1)	23.4 (37.3)	20.8 (33.3)	173.6	182.2
c3540	13.6 (21.8)	12.5 (20.4)	12.5 (20.5)	204.8	226.6
c7552	23.6 (37.4)	21.4 (34.4)	19.6 (31.5)	125.4	142.6
c2670	16.2 (25.1)	14.7 (23.1)	13.9 (21.7)	123.8	136.8
c1908	5.9 (9.3)	5.5 (8.8)	4.4 (7.0)	125.8	126.8
c6288	25.0 (39.9)	26.2 (41.6)	24.6 (34.4)	526.8	533.6
s35932	57.4 (97.4)	44.1 (76.2)	41.7 (72.4)	225.6	234.3
s38584	119.2 (194.4)	105.2 (173.9)	100.8 (167.1)	273.6	305.6
s5378	17.6 (28.4)	15.6 (25.6)	13.7 (22.2)	162.2	165.3
sin	113.0 (177.0)	97.9 (154.9)	94.2 (150.4)	1133.2	1274.2
dec	3.4 (6.2)	1.9 (4.0)	1.9 (4.0)	29.4	30.0
priority	100.9 (152.3)	47.5 (72.0)	47.7 (72.0)	1890.4	2004.8
i2c	21.1 (33.7)	19.7 (31.6)	19.5 (31.5)	128.4	129.4
KSA32	5.4 (8.9)	5.4 (8.8)	5.4 (8.8)	88.0	88.0
IntDiv8	12.6 (19.4)	10.5 (16.2)	9.9 (15.3)	619.4	640.4

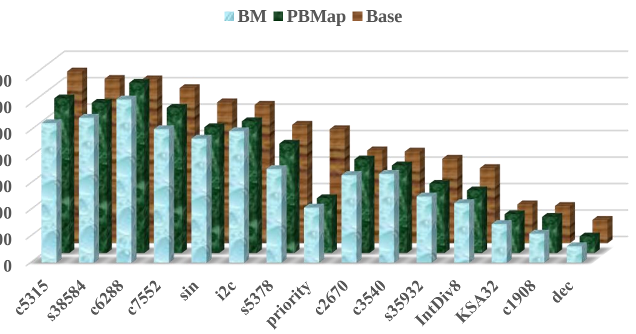


Fig. 8: Value of the cost function (Eq. 11) for different circuits generated by three different mappers. For better exhibition purposes data for *sin*, *priority*, *s35932*, and *s38584* circuits are scaled down by a factor of five.

To verify the correct functionality of circuits generated by our algorithm, we simulated a few circuits generated by our algorithm including a 2-bit Kogge-Stone Adder (KSA2) using JSIM [37]. Fig. 11 shows the corresponding waveforms for this adder. Four sets of random inputs (a_0=1010, a_1=1100, b_0=0101, b_1=1001, c_{in}=0011) and their correct outputs generated by this adder (S_0=1100, S_1=0110, C_{out}=1001) are shown in this figure. Please note that in these waveforms, presence of a pulse means “1” and absence of a pulse means ‘0”. Finally, post place-and-route results show that circuits generated by our algorithm will provide a considerable improvement on total chip area as well. For example, chip area for the *dec* circuit from the EPFL suite mapped by BM is reduced by around 33% compared to the Base. Fig. 12 shows the post place-and-route result of the *dec* circuit.

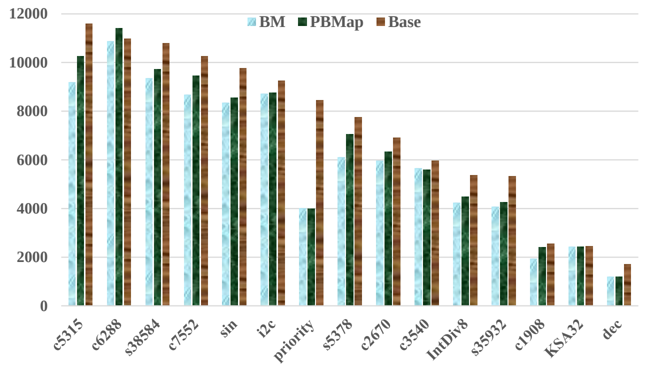


Fig. 9: Normalized total static power consumption (the main source of power consumption in SFQ circuits [15]). For better exhibition purposes data for *sin*, *priority*, *s35932*, and *s38584* circuits are scaled down by a factor of five.

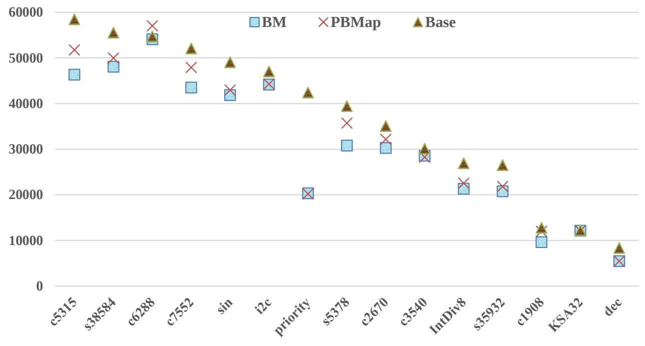


Fig. 10: Total number of Josephson junctions for gates, path balancing DFFs, and splitters. For better exhibition purposes data for *sin*, *priority*, *s35932*, and *s38584* circuits are scaled down by a factor of five.

V. Conclusions

In this paper, a dynamic programming-based path balancing technology mapping algorithm is presented. This algorithm is designed to minimize area of gates and required path balancing D-Flip-Flops (DFFs) in Single Flux Quantum (SFQ) logic circuits, and it can be used for reducing full path balancing overhead in any technology which requires having the same length for all logic paths. The optimality of the algorithm is proven for circuits with tree structure and it is shown that a modified version of the algorithm acts as a very effective heuristic in generating cost efficient solutions for general Directed Acyclic Graphs. Experimental results in SFQ technology showed that on average for 15 benchmark circuits, our technology mapper reduced area, Josephson junction count, DFF count, and static power consumption by 26.3%, 25.6%, 27.8%, and 25.1% compared to the state-of-the-art academic technology mappers.

Acknowledgement

The research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via the U.S. Army Research Office grant W911NF-17-1-0120. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation herein. This project is also supported in part by a grant from the Software and Hardware Foundations program of the National Science Foundation.

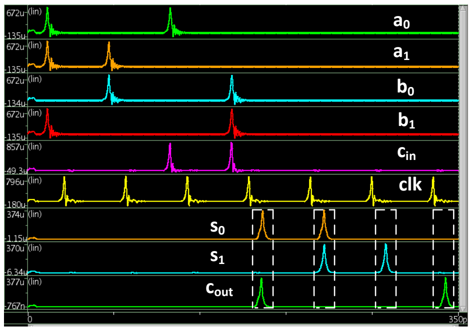


Fig. 11: Simulation waveforms for a 2-bit Kogge-Stone Adder (KSA2) generated by MB. For four sets of random inputs, correct outputs for S_0, S_1, and C_{out} are shown.

Fig. 12: Post place-and-route of *dec* circuit which is mapped by BM. Dimensions are $3960\mu m \times 3310\mu m$. The dimensions are increased to $4870\mu m \times 3990\mu m$ when the circuit is mapped using the map command of ABC.

The presented place-and-route results (Fig. 12) are generated by using software tools provided by S. N. Shahsavani and T. Lin from the University of Southern California.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[2] H. Golnabi, "Carbon nanotube research developments in terms of published papers and patents, synthesis and production," *Scientia Iranica*, vol. 19, no. 6, 2012.

[3] Y. Zhu, S. Murali, W. Cai, X. Li, J. W. Suk, J. R. Potts, and R. S. Ruoff, "Graphene and graphene oxide: synthesis, properties, and applications," *Advanced materials*, vol. 22, no. 35, pp. 3906–3924, 2010.

[4] A. Khitun and K. L. Wang, "Non-volatile magnonic logic circuits engineering," *Journal of Applied Physics*, vol. 110, no. 3, p. 034306, 2011.

[5] 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, 2013.

[6] K. Likharev and V. Semenov, "RSFQ logic/memory family: A new josephson-junction technology for sub-terahertz-clock-frequency digital systems," *IEEE Transactions on Applied Superconductivity*, vol. 50, no. 1, 1991.

[7] W. Chen, A. Rylyakov, V. Patel, J. Lukens, and K. Likharev, "Rapid single flux quantum T-flip flop operating up to 770 GHz," *IEEE Transactions on Applied Superconductivity*, vol. 9, no. 2, pp. 3212–3215, 1999.

[8] M. H. Volkmann, A. Sahu, C. J. Fourie, and O. A. Mukhanov, "Experimental investigation of energy-efficient digital circuits based on eSFQ logic," *IEEE Trans. Appl. Supercond*, vol. 23, no. 3, p. 1301505, 2013.

[9] N. Katam, A. Shafaei, and M. Pedram, "Design of complex rapid single-flux-quantum cells with application to logic synthesis," in *16th International Superconductive Electronics Conference, ISEC 2017*. IEEE, 2017.

[10] G. Pasandi and M. Pedram, "PBMap: A path balancing technology mapping algorithm for single flux quantum logic circuits," *IEEE Transactions on Applied Superconductivity*, vol. 29, no. 4, pp. 1–14, 2019.

[11] N. Katam, A. Shafaei, and M. Pedram, "Design of multiple fanout clock distribution network for rapid single flux quantum technology," in *22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 384–389.

[12] G. Pasandi, A. Shafaei, and M. Pedram, "SFQmap: A technology mapping tool for single flux quantum logic circuits," in *International Symposium on Circuits and Systems (ISCAS)*. IEEE, May 27, 2018.

[13] A. Mishchenko et. al, "ABC: A system for sequential synthesis and verification," *Berkeley Logic Synthesis and Verification Group*, 2018.

[14] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

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

[16] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[17] W. P. Burleson, M. Ciesielski, F. Klass, and W. Liu, "Wave-pipelining: a tutorial and research survey," *IEEE Transactions on very large scale integration (vlsi) systems*, vol. 6, no. 3, pp. 464–474, 1998.

[18] L. Cotton, "Maximum rate pipelining systems," in *Procs. AFIPS Spring Joint Computer Conference, 1969*, 1969.

[19] D. Strukov, A. Mishchenko, and R. Brayton, "Maximum throughput logic synthesis for stateful logic: A case study," in *Reed-Muller 2013 Workshop*, 2013.

[20] K. Keutzer, "DAGON: technology binding and local optimization by dag matching," in *24th Conference on Design Automation*. IEEE, 1987, pp. 341–347.

[21] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.

[22] K. Chaudhary and M. Pedram, "Computing the area versus delay trade-off curves in technology mapping," *IEEE Trans. on Computer Aided Design*, vol. 14, no. 12, pp. 1480–1489, 1995.

[23] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *ICCAD*, 2007, pp. 354–361.

[24] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "BDS-MAJ: A BDD-based logic synthesis tool exploiting majority logic decomposition," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 47.

[25] L. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016.

[26] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 11, pp. 1842–1855, 2017.

[27] S. Kim, J. Kim, and S.-Y. Hwang, "New path balancing algorithm for glitch power reduction," *IEE Proceedings-Circuits, Devices and Systems*, vol. 148, no. 3, pp. 151–156, 2001.

[28] A. Mishchenko, R. Brayton, S. Jang, and V. Kravets, "Delay optimization using SOP balancing," in *Computer-Aided Design (ICCAD), 2011 IEEE/ACM International Conference on*. IEEE, 2011, pp. 375–382.

[29] A. H. Farrahi and M. Sarrafzadeh, "Complexity of the lookup-table minimization problem for FPGA technology mapping," *IEEE TCAD*, vol. 13, no. 11, pp. 1319–1332, 1994.

[30] K. Chaudhary and M. Pedram, "A near optimal algorithm for technology mapping minimizing area under delay constraints," in *Proceedings of the 29th DAC*. IEEE Computer Society Press, 1992, pp. 492–498.

[31] D. Chen and J. Cong, "DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs," in *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*. IEEE Computer Society, 2004, pp. 752–759.

[32] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 11, pp. 2331–2340, 2006.

[33] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Springer Science & Business Media, 2006.

[34] N. Katam et. al. (2017) SPORT Lab SFQ logic circuit benchmark suite. [Online]. Available: http://coldflux.usc.edu

[35] A. Mishchenko, S. Chatterjee, R. Brayton, X. Wang, and T. Kam, "Technology mapping with boolean matching, supergates and choices," in *ERL Technical Report, EECS Dept., UC Berkeley*, March, 2005.

[36] L. Amaru et. al. (2017) The EPFL combinational benchmark suite. [Online]. Available: https://lsi.epfl.ch/benchmarks

[37] JSIM. (2017) The berkeley superconducting spice simulator. [Online]. Available: https://github.com/coldlogix/jsim