chips

MDPI

Article

Directed Acyclic Graph-Based Datapath Synthesis Using Graph Isomorphism and Gate Reconfiguration

Liuting Shang, Sheng Lu, Yichen Zhang, Sungyong Jung and Chenyun Pan *

Department of Electrical Engineering, University of Texas at Arlington, Arlington, TX 76019, USA; liuting.shang@mavs.uta.edu (L.S.); sheng.lu@mavs.uta.edu (S.L.); yichen.zhang@uta.edu (Y.Z.); jung@uta.edu (S.J.)

* Correspondence: chenyun.pan@uta.edu; Tel.: +1-817-272-5720

Abstract: Datapath synthesis is a crucial step in synthesis flow and aims at globally minimizing an area by identifying shareable logic structures. This paper introduces a novel Directed Acyclic Graph (DAG)-based datapath synthesis method based on graph isomorphism and gate reconfiguration. Unlike algorithms that identify common specification logic, our approach simplifies the problem by focusing on searching for common topology. Leveraging the concept of gate reconfiguration, our algorithm extends the applicability of DAG-based datapath synthesis by transforming a topology-equivalent network into a specification-equivalent network. Experimental results demonstrate up to 23.6% improvement when optimizing the adder–subtractor circuit, a scenario not addressed by existing DAG-based datapath synthesis algorithms.

Keywords: datapath synthesis; logic synthesis; reconfigurable logic gates; graph isomorphism; area optimization

check for updates

Citation: Shang, L.; Lu, S.; Zhang, Y.; Jung, S.; Pan, C. Directed Acyclic Graph-Based Datapath Synthesis Using Graph Isomorphism and Gate Reconfiguration. *Chips* **2024**, *3*, 182–195. https://doi.org/10.3390/chips3020008

Academic Editor: Gaetano Palumbo

Received: 2 April 2024
Revised: 22 May 2024
Accepted: 28 May 2024
Published: 4 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Due to a surge in computing-intensive applications, computational hardware faces significant challenges arising from the increased complexity of hardware systems in design, verification, and synthesis [1,2]. As the performance gains from the technology scaling become more challenging, electronic design automation (EDA) tools play an increasingly vital role in enhancing hardware performance in terms of delay, energy, and area efficiency, as well as reducing labor costs [1,3].

A traditional design flow typically includes several synthesis steps. Beginning with a behavioral definition in a high-level programming language, a fully optimized netlist is generated through high-level synthesis (HLS), logic synthesis, and technology mapping. HLS is primarily responsible for extracting arithmetic operations such as addition, multiplication, shifting, and comparison from the hardware description. The synthesizer models these operations as block modules [4,5] and processes them through scheduling, allocation, and binding phases, ultimately producing a netlist using standard libraries for subsequent logic synthesis and technology mapping [6–8].

Datapath synthesis is fundamentally considered a subset of HLS as it is intricately linked with HLS techniques such as behavioral transformations [6] and resource binding [9]. Unlike subsequent circuit optimization stages in logic synthesis, datapath synthesis operates at a higher level of abstraction, allowing for global optimization before delving into low-level details. However, datapath synthesis performs at lower abstraction levels, such as the logic circuit level, which can also uncover beneficial optimization opportunities.

For instance, bit-level optimization, a form of datapath synthesis, can be applied to HLS-generated logic circuits to identify resource-sharing opportunities [10]. As depicted in Figure 1, this optimization step occurs post-HLS and pre-logic synthesis to identify common structures. This method is based on logic circuits, where the structure or topology can be represented as a Directed Acyclic Graph (DAG). Techniques such as [10,11] and are

implemented on logic circuit networks represented as And-Inverter Graphs (AIGs) [12] and networks mapped onto standard libraries.

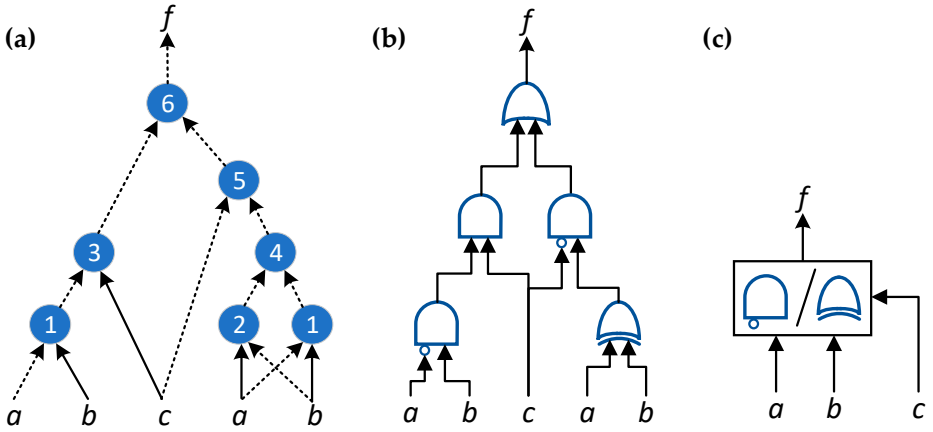


Figure 1. Three Boolean network representations for $f = (\bar{a} + b) \times c + (a \oplus b) \times \bar{c}$: (**a**) an AIG network, in which every node is the AND gate and the dotted line represents an inverter, (**b**) a network mapped with standard two-input logic gates, and (**c**) a network mapped with reconfigurable gates.

DAG-based methods aim to discover graph isomorphism, i.e., equivalent topology, to identify common specification modules and optimize datapaths [10,11]. Despite the successful reduction in the circuit area achieved by DAG-based methods, they are limited in their applicability. The effectiveness of the datapath rearrangement is primarily observed in specification-equivalent modules. In essence, searching for equivalent graphs (structures) is equivalent to searching for equivalent specifications. Unfortunately, modern HLS tools offer robust support for optimizing datapaths for fully equivalent modules, diminishing the practicality of DAG-based datapath synthesis methods.

Reconfigurable logic gates represent a class of emerging devices capable of implementing multiple logic operators within a single cell [13–28]. Drawing inspiration from various physical mechanisms, they have the potential to enhance the area efficiency of logic circuits by enabling a greater number of functions to be performed using a fixed number of cells. In addition to simply leveraging reconfigurable logic gates as compact logic models within libraries, several methodologies have been proposed to uncover circuit-level resource sharing through gate-level reconfiguration. For instance, ref. [29] introduces a satisfiable SAT-based exact synthesizer that utilizes reconfigurable logic gates to realize two arbitrary functions within a single network. While this approach is area-optimal, it is primarily applicable to small-scale circuits. By combining SAT-based synthesis methods with heuristic logic synthesis algorithms, ref. [30] demonstrates the creation of reconfigurable circuits that partially share logic resources. This logic synthesis technique aims to optimize a given circuit without actively identifying large-scale sharable resources, such as arithmetic operations in modules. This distinction arises from the fundamentally different objectives of datapath synthesis and logic synthesis. However, gate-level reconfigurability still presents an opportunity to enhance the tolerance of DAG-based datapath synthesis to specification inequivalence.

In this work, we develop new algorithms to overcome the limitation of DAG-based datapath synthesis by using reconfigurable gates. The major contributions of this work are highlighted as follows:

- Instead of conducting DAG-based datapath synthesis on a standard netlist, we introduce the concept of emerging reconfigurable logic gates to loosen the constraint of specification equivalence. This novel approach enables automatic transformation from common topology into common specification.

- We propose a novel algorithm for identifying common topology while considering the utilization of reconfigurable logic gates. The problems of logic loop and area-efficient inverter/wire removal are investigated.
- A synthesis flow is constructed encompassing high-level synthesis, logic synthesis, and technology mapping. An experimental analysis demonstrates a significant improvement of 23.6% and 26.7% in areas when optimizing the adder–subtractor circuit and parity-or circuit, respectively.

Section 2 introduces the background of conventional DAG-based datapath synthesis algorithms and necessary knowledge. Section 3 explains the proposed synthesis scheme utilizing gate reconfiguration. Section 4 provides the implementation of the proposed datapath synthesis method. Section 5 describes the experiment setup and provides the results demonstrating the achieved area improvement compared to a conventional scheme along with the synthesis flow setup. Section 6 provides a summary of the achieved results.

2. Background

2.1. Network Representation

The representation of a logic network is termed a Boolean network, characterized by its topology expressed as a DAG with nodes symbolizing logic gates and directed edges representing wires connecting the gates [31]. However, most real-world circuits with memory components are sequential and do not adhere to the acyclic nature of a DAG. Logic synthesis facilitates optimization for these circuits by employing techniques such as retiming and extracting combinational logic partitions. This is achieved by considering the memory components as primary inputs (PI) and primary outputs (PO) for the circuit partitioning process.

The representation format of a Boolean network significantly influences logic optimization strategies and outcomes. Different formats, such as DAGs, lead to varied optimization approaches. Among DAGs, the most prominent is AIG. In an AIG (illustrated in Figure 1a), all nodes represent AND gates, excluding PI and PO. Additionally, edges in an AIG are weighted, allowing for optional inversions (depicted by dotted lines), thereby representing either wires or inverters. AIGs are widely favored for logic optimization due to their excellent compatibility with logic gate technologies and fine-grained logical expression. The earliest DAG-based datapath synthesis methods were based on AIG networks. To enhance runtime efficiency in identifying common specification logic, ref. [10] employs a mapped netlist as the DAG for datapath synthesis. This approach no longer requires identical edges as a criterion for determining whether a node is common in two datapaths. It is believed that using a mapped netlist enhances the likelihood of successfully identifying common specifications, as transformations from a mapped network to AIGs may alter the original structure. In this context, we adopt a netlist mapped with a library containing arbitrary two-input gates as the circuit representation. This choice offers runtime advantages, as illustrated in Figure 1b, where the inclusion of an XOR gate reduces the number of gates.

2.2. Specification Equivalence

Specification equivalence refers to two combinational circuits with exactly the same function. The sharable circuit partition is known as common specification logic [32]. In a previous study, the datapath aimed to identify common specification logic within the context of the following task: given the output boundaries of two logic cones, determine the input boundaries that maximize the alignment of PI signals. The number of aligned/merged PIs between the original datapaths indicates the final area improvement after the logic synthesis and technology mapping followed by the datapath synthesis. In the practice of datapath synthesis, this purpose is represented as finding as many common logic nodes as possible. Typical techniques for checking if two designs conform to common types of specification logic are based on combinational equivalence checking (CEC), such as existing methods, BDDs [33], SAT [34,35], and AIG [36]. However, the CEC methods are not universal for this task because of the unknown input boundaries of the designs,

the unknown relationships (Boolean matching) of those inputs, and/or an explosive runtime in large-scale arithmetic networks.

2.3. Graph Isomophism

In the graph context, an isomorphism between graphs G and H denotes a bijection between the vertex sets V(G) and V(F), f:V(G)→V(F), such that any two vertices u and v of G are adjacent in G if and only f(u) and f(v) are adjacent in H [37]. Although graph isomorphism is a well-known NP problem of computational complexity, it can be solved in linear time complexity when the graph is a planar Boolean network, as an acyclic planar graph, which can find isomorphism efficiently via heuristic algorithms [10]. This process can be further accelerated by employing fan in–fan out information to filter out partial potential searching space [10].

2.4. Reconfigurable Gate

The reconfigurable gate refers to a type of emerging device technology that can be switched between multiple operators. Thanks to field advances in polymorphic electronics, especially the arrival of unconventional, post-silicon, or beyond-CMOS materials, many of them cost less with area, delay, and power overhead in realizing multiple Boolean operators in one cell compared to conventional CMOS schemes [13–21]. Particularly, some works have achieved a complete reconfigurable gate set—they provide an efficient implementation of any pair of two-input Boolean functions. For example, ref. [38] proposes eight sets of efficient bi-functional two-input reconfigurable gates based on emerging double-gate ambipolar transistors, which switch N and P polarities according to a control signal. With a slightly increased overhead, refs. [29,39] provides logic gate modules that reconfigure between arbitrary two-input Boolean functions by manipulating the Valley Pseudospin degree of freedom. Figure 1c demonstrates a brief example of how reconfigurable gates can simplify circuits. The reconfigurable logic gates not only stimulate research in reconfigurable circuits but the concept can also be leveraged to increase the tolerance of specification differences in datapath synthesis.

3. Proposed Methodology

The overall methodology of our scheme consists of two steps. First, the synthesizer generates and locates all the 2-to-l multiplexers (MUXs), identifying the optimizable datapath by locating those root multiplexers and treating them as the starting points of two datapaths. In this paper, the synthesizer takes a Boolean network as the input, which is technology-mapped with a library including 2-to-l multiplexers, arbitrary two-input logic gates, and inverters. The multiplexers in the network are collected, excluding those with any of their data inputs having a fanout. Each multiplexer has two data inputs, indicating the starting points of two datapaths, which are also represented as two combinational logic cones.

In the second step, a graph isomorphism searching algorithm is applied to each root multiplexer to identify the common topology. Once the boundary of the common topology is determined, a plan for multiplexer relocation is generated, moving the multiplexers from the root of the logic cone to the input of the common topology. An evaluation script is then executed to determine if the plan results in an area improvement. If the plan yields a positive gain, the multiplexer relocation is deployed to realize the datapath adjustment and re-configurable logic gates are assigned to the required positions to avoid changing the network's functionality. After traversing all the multiplexers, the modified netlist undergoes the remaining logic synthesis steps to further enhance the area of improvement for datapath optimization. Detailed algorithms and examples are described below.

3.1. Basic Searching Method for Common Specification

One of the fundamental functions of the datapath is to explore common specifications among datapaths delineated by multiplexers. As illustrated in Figure 2, the multiplexer

routes of two data inputs each correspond to a logic cone with unknown boundaries and Boolean matching relationships. Despite being termed 'searching graph isomorphism' in existing DAG-based datapath synthesis algorithms, they effectively assign this operation the task of searching for specification equivalence. The search process entails comparing nodes layer by layer to determine if they realize exactly identical functions. It terminates upon reaching a layer devoid of pairable nodes (e.g., all nodes in the layer are primary inputs) or encountering any nodes that are not pairable. As a result, modules rotated during the previous synthesis can be identified and captured as shareable logical resources. In Figure 2b, the realization of resource sharing is depicted by relocating the multiplexer backward to the boundary of the circuit partition with common specifications. Nodes identified as having the same specification are termed 'paired' nodes. After the following resource sharing and multiplexer relocation, only one of the paired nodes remains. For instance, nodes n2 in cone 1 and n3 in cone 2 are paired during the search process, with only n2 persisting in Figure 2b.

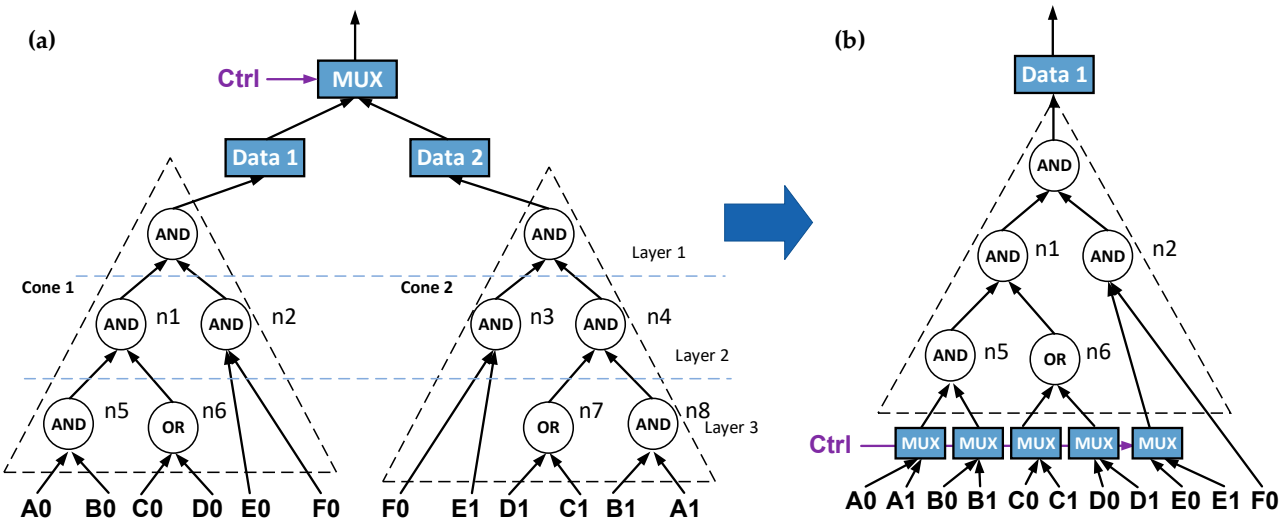


Figure 2. (**a**) The original circuit with two logic cones, i.e., datapaths that are connected to the same MUX, and (**b**) the circuit that has been processed by the original datapath synthesis. This is the basic method to use graph isomorphism and it requires specification equivalence between logic cone 1 and logic cone 2.

An important challenge in maximizing common logic is determining the optimal node pairings when a layer offers multiple choices. For instance, in Figure 2a, layer 2 presents the following two viable options: {n1-n3, n2-n4} and {n1-n4, n2-n3}. It is evident that {n1-n3, n2-n4} offers a superior opportunity for pairing nodes in layer 3. To tackle this challenge, the algorithm employs a look-ahead heuristic approach. It explores three levels deeper into the searching space, selecting pairings that maximize the shared logic gates.

3.2. Approximate Pairing Using Gate Reconfiguration

In Section 3.1, it can be found that the basic method is searching for a common specification instead of a common topology, and the common specification requires an exact equivalent function. This raises a demand for adding tolerance to specification differences when finding sharable resources between logic cones. By reducing the requirement of exact equivalent specifications, common resources can be discovered between cones realizing different functions. In existing studies, an approach of approximate pairing has been proposed to try to handle this problem [10]. Specifically, this approach tolerates the different usage of inverters between logic cones.

As shown in Figure 3a, the circuits in the two logic cones are highly similar except for an extra inverter in the left cone. The approach enables the pairing to take place by replacing the inverter/wire with an XOR gate with one of its inputs from the control signal of the

multiplexer in Figure 3b. However, this approximate pairing only slightly improves the tolerance of specification difference. One observation is that it excels in reusing precisely equivalent functions but exhibits reduced efficiency when dealing with functions that exhibit variation, even if they are very similar. For example, a 38% improvement was achieved in optimizing 64-bit A+B:A+C while the improvement reduced to 0.5% when the algorithm performed on 64-bit A+B:A−C. This is because of several limitations to the current scheme:

1. This scheme does not apply to the specification difference in two-input nodes, which is its major disadvantage. For a netlist mapped with a standard library, the difference probably reflects the two-input logic gates instead of the inverters. For example, the two cones in Figure 4a implement $\overline{\overline{a}bc}$ and $\overline{\overline{a'}b'} + c'$ in the same DAG. The current scheme judges if the pairing fails at the first layer because the gates AND and OR have different operators. Applying the De Morgan law, the implementation can be easily transformed into $\overline{\overline{a'}b'\overline{c'}}$ which can be recognized as the common specification in the current scheme. However, such transformation is not provided in DAG-based synthesis, and the case in Figure 4 widely exists in high-level synthesis-generated networks that have different descriptions.
2. The tolerance of the specification difference is realized in a significant overhead. In the example in Figure 4b, the resource sharing is achieved at the cost of inserting three XOR gates, let alone those which are usually much larger than AND or OR gates in the context of conventional CMOS technology. Although some of those XOR gates can be optimized in subsequent logic synthesis, the overhead is not negligible.
3. Once the approximate method is applied to the network, i.e., XOR gates are inserted into a logic cone, and the topology of this cone (module) is permanently changed. Since the datapath synthesis is performed multiplexer by multiplexer in order, common logic can only be shared once before its topology changes. For example, when optimizing A+B:A−C:A+D, the topology is changed because of the insertion of XOR gates during the generation of A±(B:C). Then, the network of A+(B:C) has a different topology from A+D and cannot pair with it.

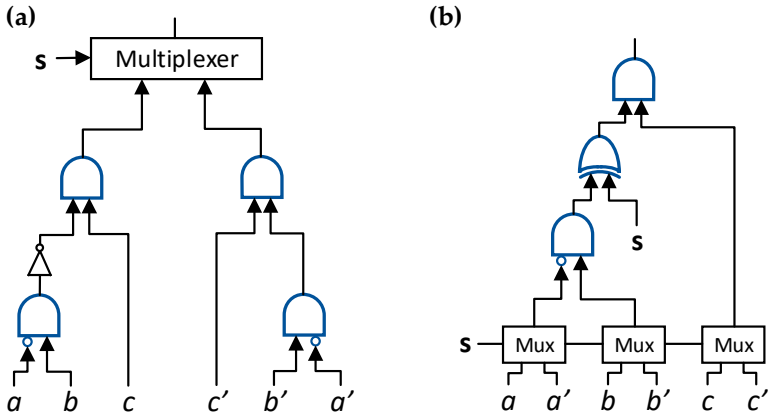


Figure 3. The existing approach of approximate pairing. The XOR gate in (**b**) is added to compensate for the inverter that only exists in the left cone in (**a**).

In this work, we propose a novel approximate pairing scheme based on gate-level reconfiguration to overcome the aforementioned issues that restrict searching for sharable resources. As shown in Figure 5, the new scheme aims for generic graph isomorphic searching, and the paired DAG nodes with different logic operators are transformed into sharable logic gates by assigning reconfigurable logic gates. This approach prevents the pairing of early ceased layer-wise nodes in common specification searching. In the previous scheme, one different logic gate can stop the current attempt at datapath optimization even though massive approximate logic is buried in deeper layers. In the example in Figure 5, the search for equivalent specifications stops at the first layer due to the logic difference

between XOR and AND. However, the two logic cones are fully graph-isomorphic and can be leveraged by searching for equivalent topology. Nevertheless, we keep the approach of replacing the inverter/wire because of its efficiency in some scenarios. This issue is further discussed in Section 4.4.

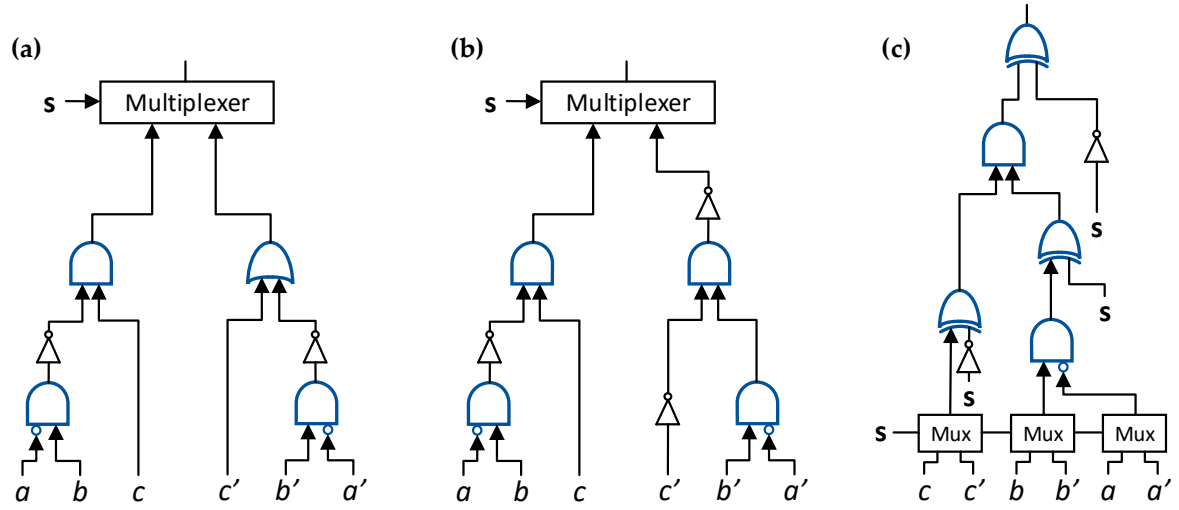


Figure 4. An example showing the limits of the existing approximate pairing method. (**a**) A case that the specification difference reflects in the two-input logic gate, (**b**) a transformed implementation from (**a**), and (**c**) the implementation generated by datapath synthesis using existing approximate pairing.

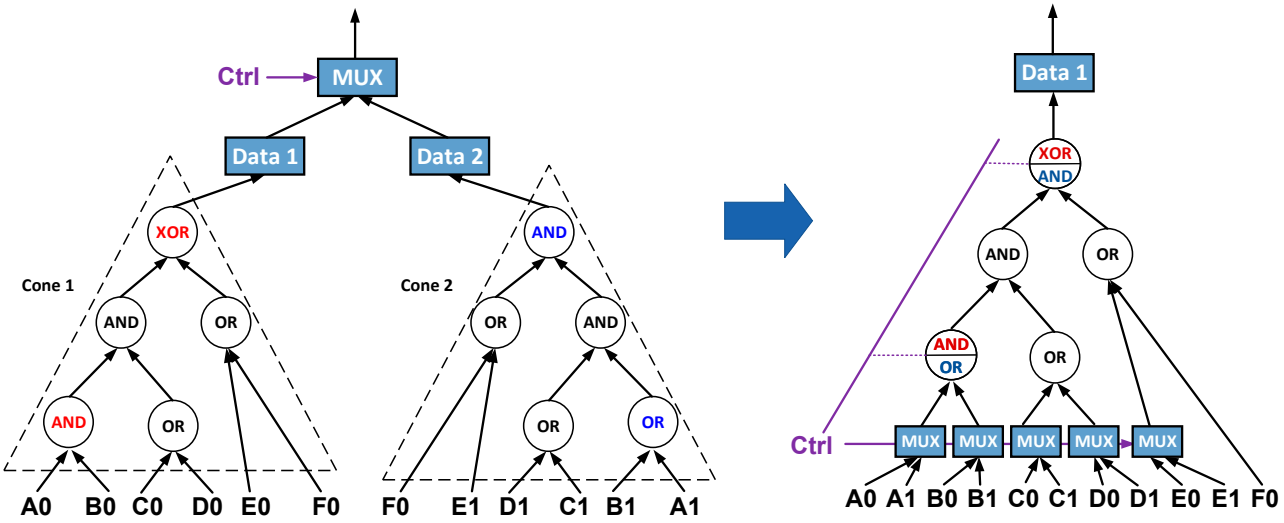


Figure 5. The novel approximate pairing scheme for leveraging gate-level reconfiguration. The colored gates mark the gates that are in the same topological position but with different operators.

4. Implementation

There are several challenges to be solved in implementing reconfigurable-based approximate isomorphic searching. In this section, the examples, detailed solutions, and algorithms are described. The pseudocode in Algorithm 1 outlines the proposed method.

4.1. Priority of Pairing

The critical function of this method is to identify the maximum common specification logic between the neighboring datapath. This function is described in the function CommonLogicBoundary in Algorithm 1. This function returns the pairings of the boundary signals B that maintain the isomorphism class, which is used for relocating the root multiplexers to the boundary of the common logic. It also returns all the positions of conditional pairs ($R0$, $R1$) that need to be assigned a reconfigurable gate.

Algorithm 1 Datapath Optimization

Input: Preprocessed subcircuit C
Output: A datapath-optimized netlist with reconfigurable gates
Datapath_Optimize(*C*)
1: *B*, (*R0*, *R1*) = CommonLogicBoundary(*PO*)
2: P_{inv} = invDiffPosition(*PO*, *B*)
3: **if** evaluateAdjustment(*C*, *B*, (*R0*, *R1*), P_{inv}) = 0 **then**
4: **exit**
5: *C* ← relocation multiplexer to level *B*
6: *C* ← Replace the pairs (*R0*, *R1*) with reconfigurable gates
7: P_{inv} = invDiffPosition(*PO*, *B*)
8: *C* ← handleInvDiff(P_{inv})
9: *C* ← insert XORs to P_{inv}
10: **return** *C*
CommonLogicBoundary(*PO*)
1: $m \leftarrow Layers(PO) - 1$; inverter is considered as 0 layer.
2: **while** $m \geq 0$, perform
3: $L0_m$, $L0_m'$ ← the gates in ($s = 0$) logic at layer m
4: $L1_m$, $L1_m'$ ← the gates in ($s = 1$) logic at layer m
5: $PI0_m$, $PI1_m$ ← pairPIs($L0_m$, $L1_m$)
6: PI0 ← PI0 + $PI0_m$, PI1 ← PI1 + $PI0_m$
7: $L0_m \leftarrow L0_m - PI0_m$; $L1_m \leftarrow L1_m - PI1_m$
8: $U0_m$, $U1_m$ ← uniqueFanoutPairs($L0_m$, $L1_m$)
9: $L0_m \leftarrow L0_m - U0_m$; $L1_m \leftarrow L1_m - U1_m$
10: $E0_m$, $E1_m$ ← exactPairs($L0_m$, $L1_m$)
11: $L0_m \leftarrow L0_m - E0_m$; $L1_m \leftarrow L1_m - E1_m$
12: $R0_m$, $R1_m$ ← reconfigurePairs($L0_m$, $L1_m$)
13: $R0_m$, $R1_m$ ←$L0_m \leftarrow L0_m - R0_m$; $L1_m \leftarrow L1_m - R1_m$
14: R0, R1 ← (R0, R1) + ($R0_m$, $R1_m$)
15: $L0_{m+1}$, $L1_{m+1}$ ← ($U0_m$, $U1_m$) + ($E0_m$, $E1_m$) + ($R0_m$, $R1_m$)
16: **if (**Size($L0_m$) ≠ Size($L0_m'$)**) then**
17: **exit**
18: **end while**
19: **return** (layer(PO-1-*m*), ($L0_{m-1}$ + PI0, $L1_{m-1}$ + $PI1_m$), (R0, R1)
invDiffPosition(*PO*, **boundary**)
1: P_0 ← the positions of all inverts up to the *boundary* layer
2: P_1 ← the positions of all inverts up to the *boundary* layer
3: return $P_0 \cap P_1$

The topology-based common specification determination suffers from the explosion of searching space. Since the pairing is in the context of topology, any nodes that share paired parent nodes have the same fanin number, are in the same node type (e.g., logic node or PI), and can form a pair and operate in a common node. In order to reduce runtime, the layer-wise node pairing is performed in the following order of priority:

- All nodes to be paired have paired parent nodes to ensure functional continuity between layers.
- Nodes that are PIs are paired in advance. Note that the identical PIs are put into a pair with priority since they do not need a multiplexer for datapath adjustment. For example, the pairs {*F0-F0*} in Figure 5.
- The nodes that have the identical operator.
- The nodes that have identical fanins.
- The nodes that have the same number of fanouts [10].
- If several candidates for pairing have equal priority, they are evaluated and sorted according to their potential to maximize the logic sharing. This potential evaluation is a three-layer look-ahead heuristic algorithm adopted from [10].

Note that the inverter is not considered as a gate or a node in the DAG. Additionally, only the nodes that have an identical number of fanins can be paired.

4.2. Elimination of Logic Loop

The utilization of datapath synthesis introduces the potential risk of generating a logic loop within the network. An illustrative example of the creation of a logic loop is demonstrated in Figure 6, where node 2 concurrently resides in the logic cones of data input 0 and data input 1. In such a scenario, if achieving common logic between data 0 and data 1 necessitates pairing node 7 with node 2 and assigning them as a reconfigurable node, a loop forms, as indicated by the red circle in the figure. However, due to the absence of memory elements, loops are prohibited in combinational logic networks. This challenge can be addressed by decoupling the two cones undergoing datapath optimization. Specifically, we temporarily duplicate the overlapping nodes in the two logic cones and assign independent indices to the duplicated nodes. While this temporary measure may momentarily increase the total area, it can be reversed in subcircuits and not updated in the network if the datapath adjustment is evaluated as unworthy.

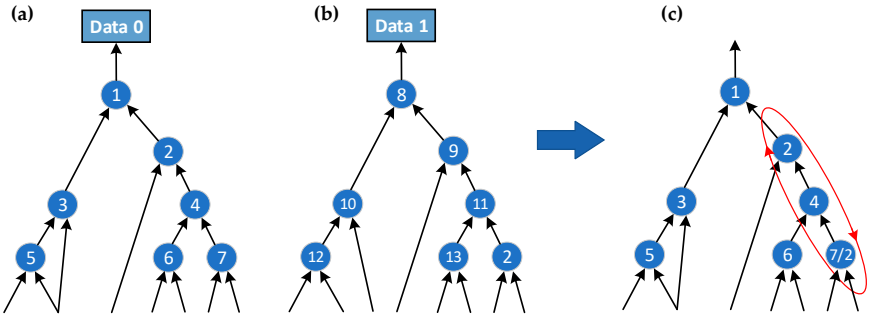


Figure 6. The example of creating a loop using reconfiguration-based approximate pairing. Assume all nodes in (**a**,**b**) are the AND gate except node 7, which is the OR gate; then, the common specification is generated as (**c**). The red circle marks the created loop.

4.3. Evaluation Function

After the boundary of common specification logic has been found, the algorithm needs to evaluate if such resource sharing is worthy in terms of area. The evaluation function first calculates and records the circuit area within the datapaths in optimization. Then, it generates the post-adjustment area by taking the additional area as follows:

$$Gain = A_{Removed} - A_{Rec} - A_{MuxRelocate} - A_{Xor} \quad (1)$$

where $A_{Removed}$ is the saved area by removing the network on one of the logic cones, and the function of the removed cone is undertaken by the other logic cone; A_{Rec} is the area overhead for the assigned reconfigurable gate, which comes from the slightly larger size of the reconfigurable logic gate compared to the two-input logic gate; $A_{MuxRelocate}$ is the cost of relocating the root 2-to-1 multiplexer to the boundaries, and this term usually dominates the overhead; and A_{Xor} is the area of consumption of adding an XOR gate to replace the inverter/wire.

4.4. Remove Inverter/Wire

During the search for common logic, inverters are disregarded, yet some of them introduce specification differences in the form of inverter/wire combinations. The algorithm records the positions of inverters/wires and handles them in two ways. First, it attempts to combine the inverter/wire with the nearest reconfigurable gate. As depicted in Figure 7, an inverter solely resides in the left cone, and the driven gate AND is converted into a reconfigurable gate following datapath adjustments. Consequently, the inverter/wire can be merged with the AND/OR gate by inversing one of the inputs of the AND operator.

If the inverter is not connected to a reconfigurable gate, we transform the conventional gate driven by the inverter into a reconfigurable gate that dynamically switches the input complementation status based on the control signal. This approach offers area advantages compared to inserting an XOR gate, especially considering that input/output negation is at a lower area cost in certain technologies, such as Valley Spin devices [29]. However, if the inverter drives multiple gates, merging the inverter/wire with the driven gates may not be area-efficient. Moreover, if the gate g driving the inverter also controls gate x outside of the logic cone, this method becomes infeasible because gate x could require a statically inverted signal g, disregarding the control signal selecting this logic cone. Therefore, without the loss of the main advantage of leveraging gate-level reconfiguration, in Algorithm 1 handleInvDiff, we still handle the inverter/wire by inserting XOR when the inverter drives or is driven by multiple gates.

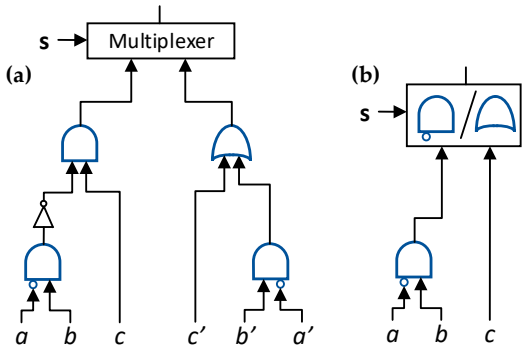


Figure 7. The example of removing inverter/wire combination using reconfiguration-based approximate pairing. (**a**) A circuit with an inverter difference between two datapaths, and (**b**) the solution that addresses the inverter difference by employing a reconfigurable gate.

5. Experiment Setup and Results

We implement the proposed algorithms outlined in Section 4 in C and integrate them with the logic synthesis tool ABC [36]. The benchmark circuit chosen is A+B:A−C, which previously exhibited almost no area optimization using the original datapath synthesis scheme [10]. We use open-source synthesis tools, such as Yosys [40], to perform the high-level synthesis.

First, the A+B:A−C netlist is implemented using the Verilog code '*if O=A+B; else O=A−C;*' with an assigned operant bit-width. Subsequently, the RTL description is translated into a gate-level combinational netlist using the command 'read; hierarchy -top; proc; techmap;' in the synthesis tool Yosys [40]. This netlist is then mapped into a standard library and serves as the initial network for subsequent processes, which is also set as the baseline scheme in this work.

Our algorithm, logic synthesis, and technology mapping are then applied to the netlist. In addition to the baseline, four other schemes are implemented, and the results are shown in Table 1. First of all, a library is built with standard gates as well as a set of reconfigurable gates capable of switching between arbitrary two-input operators. We set the delay and area of standard two-input gates to be 1 and assumed the reconfigurable gates to have the same delay and 1.5× relative area of standard two-input gates. By mapping the baseline network with the library with reconfigurable gates, the scheme 'Rec-library' was deployed to assess the impact of using added reconfigurable gates. In the scheme 'ABC flow', the baseline network experiences the traditional logic synthesis and technology mapping in ABC by executing the commands 'strash; dch -v; map -v' with the new library. This scheme aims to investigate the performance of the synthesis that does not leverage datapath synthesis. The 'original datapath' scheme first processes the initial network with the existing datapath synthesis algorithm, then uses 'strash; dch -v; map -v' to fully explore and leverage the benefits of deploying structural optimization. Based on the same process, the proposed scheme replaces the original datapath synthesis with the proposed datapath

that can tolerate functional differences. The experiments are performed on bit-widths ranging from 8 to 64 to investigate the impact of the scale.

Table 1. Experimental area results for five schemes of implementing A+B:A−C with different bit-widths.

Bit-Width	Baseline	Rec-Library	ABC Flow	Original Datapath	Proposed	Improvement
8-bit	109	95.5	91.5	94.5	77	22.7%
16-bit	239	209.5	202.5	202.5	171	18.4%
32-bit	505	443.5	455.5	430.5	380	13.3%
64-bit	1043	917.5	898.5	904.5	801.5	12.9%

From the experimental results summarized in Table 1 'Rec-Library', which maps a network with advanced reconfigurable devices, causes limited improvement in the area. Applying the ABC optimization flow only improves the area to a very small extent. The original datapath scheme can change the network and occasionally improve the area compared to previous schemes. This is because several topology-identical nodes can still be found in this case. As the partial structure of the benchmark circuit illustrated in Figure 8 shows, the small logic cones of O0 and O1 are optimizable because they have identical topology and gate operators except for the inverter/wire that can be tolerated. However, the red components represent specification differences, and almost none of them can be tolerated by existing algorithms. In contrast, the proposed scheme leverages the common topology by transforming them into common specifications through the replacement of the red components with reconfigurable gates. The results demonstrate the effectiveness of the proposed algorithm by an area improvement of up to 22.7% compared to the original datapath scheme. For the aspect of delay, as shown in Table 2, the two datapath synthesis schemes have the same performance, and both of them increase the delay by one compared to none datapath synthesis schemes. Moreover, since the proposed algorithm is developed in work [10] and retains the main framework, their runtimes are comparable.

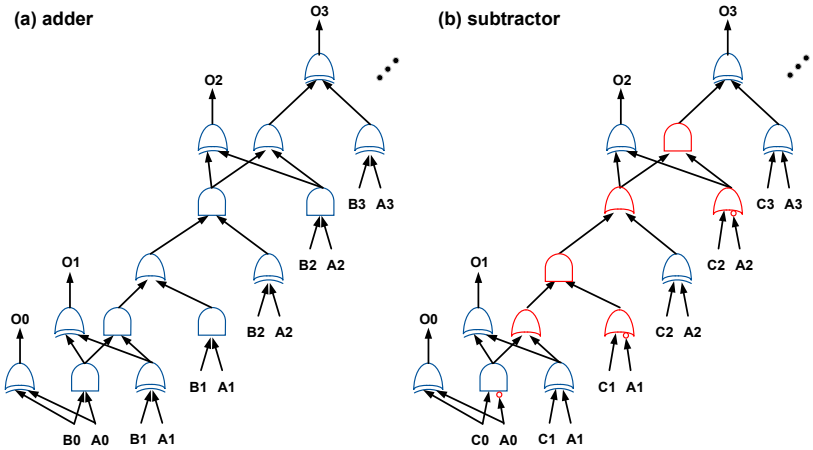


Figure 8. The partial structure of the benchmark circuit. Red components indicate the specification difference.

In the experiments above, we assume the area overhead of gate-level reconfiguration to be 1.5×. However, the cost of realizing reconfigurable gates significantly varies depending on the device technology. Here, a simulation is performed on the 64-bit case to reveal how the area overhead of reconfigurable gates affects the advantage of the proposed scheme. As shown in Figure 9, the proposed scheme becomes more advantageous with lower reconfiguration costs. The highest area of improvement at 23.6% is achieved when no overhead is applied. Meanwhile, it can be observed that the other four schemes are not sensitive to the area of reconfigurable gates. This observation provides further evidence that these existing approaches cannot effectively leverage the reconfigurable gates that have a stronger representative ability even though the area overhead is removed.

Table 2. Experimental delay results for five schemes of implementing A+B:A−C with different bit-widths.

Bit-Width	Baseline	Rec-Library	ABC Flow	Original Datapath	Proposed
8-bit	12	10	10	11	11
16-bit	16	14	14	15	15
32-bit	20	18	18	19	19
64-bit	24	22	22	23	23

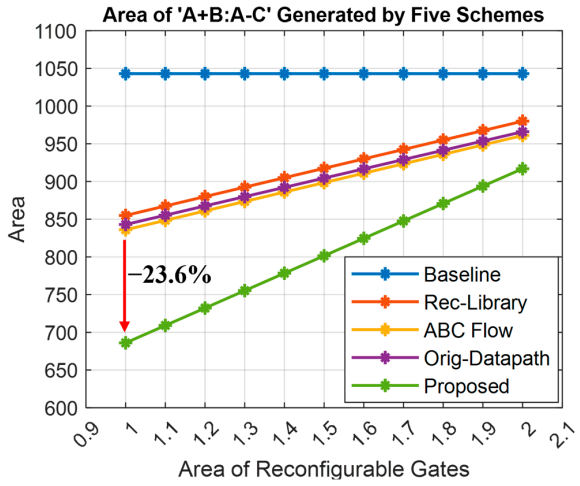


Figure 9. Area of 64-bit 'A+B:A−C' generated by the five schemes with different areas overhead of gate-level reconfiguration.

In addition, another example, 'parity([A [0:x/2–1] B[0:x/2–1]]):A[0:x/2–1]|C[0:x/2–1]' with the bit-width $x = 2^n$ is provided to demonstrate the effectiveness of the proposed scheme. The other experiment setup is the same as the previous one. The delay results of the proposed scheme and other schemes are n and $n + 1$, respectively. The delay of the proposed scheme and the original datapath synthesis scheme are no longer equal since the latter cannot perform any optimization on the circuit. The results for this area are shown in Figure 10. It can be observed that up to 26.7% of area improvements are achieved with negligible delay overhead.

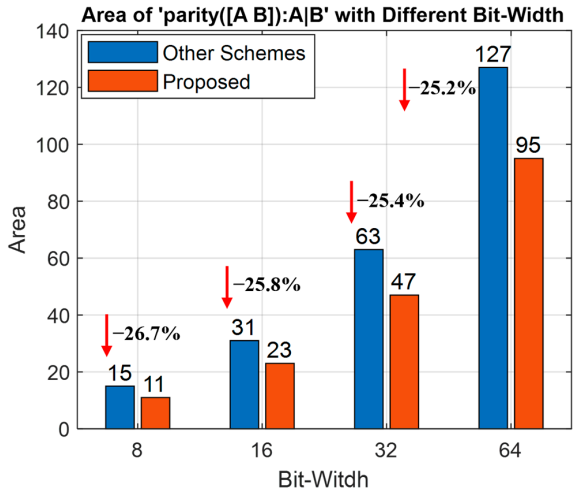


Figure 10. Area of 'parity([A B]):A|C' generated by the five schemes.

6. Conclusions

This paper introduces an advanced DAG-based datapath synthesis method to utilize the unique reconfigurable feature of emerging devices and minimize the overall circuit-

level area. It achieves logic-level resource sharing with greatly enhanced tolerance for specification differences, thereby enabling the effective utilization of identical topology. A range of solutions are proposed to address issues, such as cone pairing priority, loop prevention, and inverter/wire removal. The proposed approach demonstrates its ability to identify more opportunities in datapath optimization and effectively tackle circuits that were previously challenging to address. This leads to a significant reduction in areas within a design flow.

Author Contributions: Conceptualization, L.S., S.L. and C.P.; methodology, L.S. and C.P.; programming, L.S.; validation, L.S., C.P. and Y.Z.; investigation, L.S., S.L., Y.Z., S.J. and C.P.; writing—original draft preparation, L.S.; writing—review and editing, L.S., Y.Z., S.J. and C.P.; supervision, C.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work is funded by the Advanced Scientific Computing Research (ASCR) program of the Department of Energy (DOE) through award DE-SC0022881, IMEC, and the National Science Foundation (NSF) under grant CCF-2219753.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Data is contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. The EDA Industry Council. *EDA Roadmap Taskforce Report—Design of Microprocessors*; Silicon Integration Initiative Inc.: Austin, TX, USA, 1999.
2. Kahng, A.B. Open-source eda: If we build it, who will come? In Proceedings of the 2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC), Salt Lake City, UT, USA, 5–7 October 2020; IEEE: New York, NY, USA, 2020; pp. 1–6.
3. Coward, S.; Constantinides, G.A.; Drane, T. Automatic datapath optimization using e-graphs. In Proceedings of the 2022 IEEE 29th Symposium on Computer Arithmetic (ARITH), Lyon, France, 12–14 September 2022; IEEE: New York, NY, USA, 2022; pp. 43–50.
4. Stok, L. Data path synthesis. *Integration* **1994**, *18*, 1–71. [CrossRef]
5. De Micheli, G. *Synthesis and Optimization of Digital Circuits*; McGraw Hill: New York, NY, USA, 1994.
6. Potkonjak, M.; Rabaey, J. Optimizing resource utilization using transformations. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **1994**, *13*, 277–292. [CrossRef]
7. Srivastava, M.B.; Potkonjak, M. Optimum and heuristic transformation techniques for simultaneous optimization of latency and throughput. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **1995**, *3*, 2–19. [CrossRef]
8. Cong, J.; Xu, J. Simultaneous FU and register binding based on network flow method. In Proceedings of the Design, Automation and Test in Europe, Munich, Germany, 10–14 March 2008; ACM: New York, NY, USA, 2008; pp. 1057–1062.
9. Mohanty, S.P.; Ranganathan, N.; Chappidi, S.K. An ilp-based scheduling scheme for energy efficient high performance datapath synthesis. In Proceedings of the 2003 International Symposium on Circuits and Systems (ISCAS'03), Bangkok, Thailand, 25–28 May 2003; IEEE: New York, NY, USA, 2003. pp. V–V.
10. Yu, C.; Choudhury, M.; Sullivan, A.; Ciesielski, M. Advanced datapath synthesis using graph isomorphism. In Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), Irvine, CA, USA, 13–16 November 2017; IEEE: New York, NY, USA, 2017; pp. 424–429.
11. Yu, C.; Ciesielski, M.; Choudhury, M.; Sullivan, A. Dag-aware logic synthesis of datapaths. In Proceedings of the 53rd Annual Design Automation Conference, Austin, TX, USA, 5–9 June 2016; ACM: New York, NY, USA, 2016; pp. 1–6.
12. Mishchenko, A.; Chatterjee, S.; Brayton, R. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In Proceedings of the 2006 43rd ACM/IEEE Design Automation Conference, San Francisco, CA, USA, 24–28 July 2006; ACM: New York, NY, USA, 2006; pp. 532–535.
13. Kvatinsky, S.; Belousov, D.; Liman, S.; Satat, G.; Wald, N.; Friedman, E.G.; Kolodny, A.; Weiser, U.C. MAGIC—Memristor-aided logic. *IEEE Trans. Circuits Syst. II Express Briefs* **2014**, *61*, 895–899. [CrossRef]
14. Berrettini, G.; Simi, A.; Malacarne, A.; Bogoni, A.; Poti, L. Ultrafast integrable and reconfigurable XNOR, AND, NOR, and NOT photonic logic gate. *IEEE Photonics Technol. Lett.* **2006**, *18*, 917–919. [CrossRef]
15. Nishimoto, S.; Yamanashi, Y.; Yoshikawa, N. Design method of single-flux-quantum logic circuits using dynamically reconfigurable logic gates. *IEEE Trans. Appl. Supercond.* **2015**, *25*, 1301405. [CrossRef]
16. Zhang, Y.; Yan, B.; Wu, W.; Li, H.; Chen, Y. Giant spin hall effect (GSHE) logic design for low power application. In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 9–13 March 2015; IEEE: New York, NY, USA, 2015; pp. 1000–1005.

17. Winograd, T.; Salmani, H.; Mahmoodi, H.; Gaj, K.; Homayoun, H. Hybrid STT-CMOS designs for reverse-engineering prevention. In Proceedings of the 53rd Annual Design Automation Conference, Austin, TX, USA, 5–9 June 2016; ACM: New York, NY, USA, 2016; pp. 1–6.
18. Angizi, S.; He, Z.; Chen, A.; Fan, D. Hybrid spin-CMOS polymorphic logic gate with application in in-memory computing. *IEEE Trans. Magn.* **2020**, *56*, 3400215. [CrossRef]
19. Zhao, R.; Zhao, X.; Liu, H.; Shao, M.; Feng, Q.; Liu, T.; Lu, T.; Wu, X.; Yi, Y.; Ren, T.-L. Reconfigurable logic-memory hybrid device based on ferroelectric $Hf_{0.5}Zr_{0.5}O_2$. *IEEE Electron Device Lett.* **2021**, *42*, 1164–1167. [CrossRef]
20. Lin, Y.-M.; Appenzeller, J.; Knoch, J.; Avouris, P. High-performance carbon nanotube field-effect transistor with tunable polarities. *IEEE Trans. Nanotechnol.* **2005**, *4*, 481–489. [CrossRef]
21. Murapaka, C.; Sethi, P.; Goolaup, S.; Lew, W. Reconfigurable logic via gate controlled domain wall trajectory in magnetic network structure. *Sci. Rep.* **2016**, *6*, 20130. [CrossRef] [PubMed]
22. Rai, S.; Trommer, J.; Raitza, M.; Mikolajick, T.; Weber, W.M.; Kumar, A. Designing efficient circuits based on runtime-reconfigurable field-effect transistors. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2018**, *27*, 560–572. [CrossRef]
23. Trommer, J.; Heinzig, A.; Baldauf, T.; Slesazeck, S.; Mikolajick, T.; Weber, W.M. Functionality-enhanced logic gate design enabled by symmetrical reconfigurable silicon nanowire transistors. *IEEE Trans. Nanotechnol.* **2015**, *14*, 689–698. [CrossRef]
24. Raitza, M.; Märcker, S.; Trommer, J.; Heinzig, A.; Klüppelholz, S.; Baier, C.; Kumar, A. Quantitative characterization of reconfigurable transistor logic gates. *IEEE Access* **2020**, *8*, 112598–112614. [CrossRef]
25. Galderisi, G.; Mikolajick, T.; Trommer, J. The RGATE: An 8-in-1 Polymorphic Logic Gate Built from Reconfigurable Field Effect Transistors. *IEEE Electron Device Lett.* **2023**, *45*, 496–499. [CrossRef]
26. Wind, L.; Fuchsberger, A.; Demirkiran, Ö.; Vogl, L.; Schweizer, P.; Maeder, X.; Sistani, M.; Weber, W.M. Reconfigurable Si Field-Effect Transistors with Symmetric On-States Enabling Adaptive Complementary and Combinational Logic. *IEEE Trans. Electron Devices* **2024**, *71*, 1302–1307. [CrossRef]
27. Chen, J.; Li, P.; Zhu, J.; Wu, X.-M.; Liu, R.; Wan, J.; Ren, T.-L. Reconfigurable MoTe 2 field-effect transistors and its application in compact CMOS circuits. *IEEE Trans. Electron Devices* **2021**, *68*, 4748–4753. [CrossRef]
28. Rai, S.; Riener, H.; De Micheli, G.; Kumar, A. Preserving Self-Duality During Logic Synthesis for Emerging Reconfigurable Nanotechnologies. In Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; IEEE: New York, NY, USA, 2021; pp. 354–359.
29. Shang, L.; Naeemi, A.; Pan, C. Towards Area Efficient Logic Circuit: Exploring Potential of Reconfigurable Gate by Generic Exact Synthesis. *IEEE Open J. Comput. Soc.* **2023**, *4*, 50–61. [CrossRef]
30. Fiser, P.; Simek, V. Optimum polymorphic circuits synthesis method. In Proceedings of the 2018 13th International Conference on Design & Technology of Integrated Systems In Nanoscale Era (DTIS), Taormina, Italy, 9–12 April 2018; IEEE: New York, NY, USA, 2018; pp. 1–6.
31. Keutzer, K. DAGON: Technology binding and local optimization by DAG matching. In Proceedings of the 24th ACM/IEEE Design Automation Conference, Miami Beach, FL, USA, 28 June–1 July 1987; ACM: New York, NY, USA, 1987; pp. 341–347.
32. Goldberg, E.; Novikov, Y. Equivalence checking of dissimilar circuits. In Proceedings of the 12th International Workshop on Logic and Synthesis, Laguna Beach, CA, USA, 28–30 May 2003.
33. Bryant, R.E. Graph-based algorithms for boolean function manipulation. *Comput. IEEE Trans.* **1986**, *100*, 677–691. [CrossRef]
34. Kuehlmann, A.; Krohm, F. Equivalence checking using cuts and heaps. In Proceedings of the 34th annual Design Automation Conference, Anaheim, CA, USA, 9–13 June 1997; ACM: New York, NY, USA, 1997; pp. 263–268.
35. Goldberg, E.I.; Prasad, M.R.; Brayton, R.K. Using SAT for combinational equivalence checking. In Proceedings of the Design, Automation and Test in Europe. Conference and Exhibition 2001, Munich, Germany, 13–16 March 2001; IEEE: New York, NY, USA, 2001; pp. 114–121.
36. ABC: A System for Sequential Synthesis and Verification. 2007, Volume 17. Available online: https://people.eecs.berkeley.edu/~alanmi/abc/ (accessed on 3 May 2020).
37. Fortin, S. The Graph Isomorphism Problem (Tech. Rep. No. TR96-20). Available online: https://era.library.ualberta.ca/items/f8153faa-71bf-4b64-9eb4-f0c6d3b529dd (accessed on 11 August 2023).
38. Nevoral, J.; Šimek, V.; Ružicka, R. PoLibSi: Path towards intrinsically reconfigurable components. In Proceedings of the 2019 22nd Euromicro Conference on Digital System Design (DSD), Kallithea, Greece, 28–30 August 2019; IEEE: New York, NY, USA, 2019; pp. 328–334.
39. Tao, L.; Naeemi, A.; Tsymbal, E.Y. Valley-spin logic gates. *Phys. Rev. Appl.* **2020**, *13*, 054043. [CrossRef]
40. Wolf, C. Yosys Open Synthesis Suite. 2016. Available online: https://yosyshq.net/yosys/about.html (accessed on 11 August 2023).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.