DAG-Aware Logic Synthesis of Datapaths

Cunxi Yu, Maciej Ciesielski
University of Massachusetts, Amherst
ECE Department
Amherst, MA, 01003
ycunxi@umass.edu

Mihir Choudhury, Andrew Sullivan
IBM T.J Watson Research Center
Yorktown Heights, NY, 10598
choudhury@us.ibm.com

Abstract - Traditional datapath synthesis for standard-cell designs go through extraction of arithmetic operations from the high-level description, high-level synthesis, and netlist generation. In this paper, we take a fresh look at applying high-level synthesis methodologies in logic synthesis. We present a DAG-Aware synthesis technique for datapaths synthesis which is implemented using *And-Inv-Graphs*. Our approach targets area minimization. The proposed algorithm includes identifying vector multiplexers, searching for common specification logic, and reallocating multiplexers in the Boolean network. We propose an algorithm to identify common specification logic by using subgraph isomorphism. Experimental results show that our technique can provide over 10% area reduction beyond the traditional design flow. The proposed algorithm is tested on industry designs and academic benchmark suits using IBM 14nm technology.

Keywords

Logic synthesis, AIGs, datapaths, resource sharing

1. INTRODUCTION

Modern microprocessors and embedded systems contain datapaths modules which play an important role in computations. Traditionally, datapath synthesis for standard-cell design goes through a series of steps, including extraction of arithmetic operations from RTL code, high-level synthesis (HLS), logic synthesis, and technology mapping [1]. The arithmetic operations such as addition, multiplication, shifting, comparison, and etc., are extracted first and are modeled into the datapaths. High-level synthesis (HLS), which basically consists of scheduling, allocation, and binding is applied when the arithmetic operations are extracted [2]. Finally, the standard-cell netlist is generated after logic synthesis and technology mapping with a given standard-cell library.

A known area-reduction optimization in the high-level synthesis step called *resource sharing*. We illustrate the concept of resource sharing using an example in Figure 1(a).

DAC '16, June 05-09, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4236-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2897937.2898000

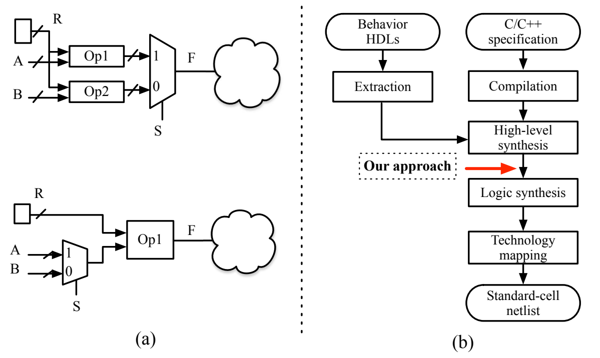

Figure 1: (a) Design flow of datapaths. (b) *Resource sharing* example.

Assuming $Op1$ and $Op2$ are addition operations, function $F = S?(A+R) : (B+R)$. Figure 1(a) shows that the first implementation can be optimized by sharing the adder component. We can see that resource sharing is able to reduce the area and potentially the power. Several attempts have been made to provide optimizations based on deriving the optimized *data flow graph* (DFG). For example, [3][4] proposed an optimization algorithm which was implemented in HDL compilers using behavioral transformations; [5][6] proposed an integer linear programming (ILP) formula based algorithm for binding resources; and in [7], canonical TED representation was introduced to optimize data-flow computations. However, these techniques has several limitations: **1)** The high-level optimization is limited to the capability of extracting the arithmetic operations from the implementation. **2)** *Resource sharing* is only applicable to a high-level component that is included in the library. For example, if *Op1*, *Op2* are combinational logic units that perform the same but the function is not included in the library, the resource sharing will not be applied. **3)** Resource sharing is applicable only if two components are equivalent or equivalent with negation. Another design flow that starts with C/C++ specification is popular and widely used [8]. The C/C++ implementations are initially compiled into a high-level HDL description regardless of the extraction. However, the architecture is implicit in the coding style, which makes the high-level synthesis more difficult because of the coding variety.

In this work, we demonstrate that these limitations can be successfully addressed by adding a design flow component into the current flow between high-level synthesis and logic synthesis (Figure 1(b)). We take a fresh look at applying high-level synthesis methodologies in logic synthesis, particularly the resource sharing. This algorithm initially identifies the vector multiplexers from a Boolean network that provides the boundaries of the logic that could be *shared*. An efficient algorithm of searching *common specification logic* is implemented using graph isomorphism. We demonstrate that the resource sharing method is no longer limited to arithmetic operations. In fact, it is applicable to arbitrary combinational logic if common specification logic can be identified. This means that the resource sharing can be partially applicable even if $Op1$ and $Op2$ are two logic units have partial functional similarity. This paper is organized as follows: Section 2 covers the background. Section 3 presents the methodology and implementation of our technique. Section 4 presents the experimental results.

2. BACKGROUND

2.1 Boolean network

A Boolean network can be represented using directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates [9]. In the AIG, each node has either 0 or two incoming edges. A node with no incoming edges is a primary input (PI). An edge can be complemented to indicates the inversion of the signal. Primary outputs (POs) are represented using specific nodes. Registers in AIG network are considered as PIs and POs. A. Mishchenko et. al [9] claimed that the size (area) of an AIG is the number of its nodes and the depth (delay) is the number of nodes on the longest path from the PIs to the POs. The combinational logic of an arbitrary Boolean network can be factored and transformed into an AIG using DeMorgan's rule [10].

2.2 Logic synthesis

Logic synthesis transforms the Boolean network to reduce the number of nodes (area), logic levels (delay), switching activity (power). Traditional logic synthesis tools such as SIS [11] and ABC [10] target multi-level logic optimization that basically apply removing redundancy, logic simplification, and logic sharing. However, the resource sharing in Boolean network has yet been studied.

3. MULTIPLEXERS RELOCATION

This section describes the methodology and implementation of our technique composed of two parts: *pre processing* and *multiplexer relocation*. The overview of the proposed technique is shown in Figure 2. Our framework takes the gate-level netlist (Verilog, BLIF or AIGER [12]) as input and transforms the design into AIG network. The pre-processing includes collecting vector multiplexers and generating sub-networks (Section 3.1). The sub-networks are the logic cones of of the *vector multiplexers* that end at PIs or latches. These are used for multiplexer relocation. The *multiplexer relocation* process includes searching common specification logic and reallocating the multiplexers (Section 3.2). We check if the design before and after each relocation are functional equivalent by combinational equivalence checking (CEC).

3.1 Pre-processing

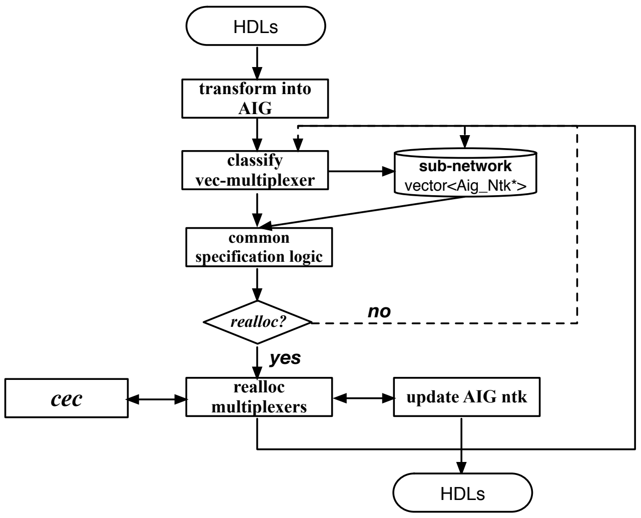

Figure 2: Overview of the proposed technique.

The pre-processing algorithm (Algorithm 1) includes identifying multiplexer nodes, identifying vector multiplexers, and generating the sub-networks.

Algorithm 1 Pre processing

```
Input: Boolean network $\mathcal{N}$
Output: Vector multiplexers and the sub-network
 1: $n \leftarrow$ number of AIG nodes
 2: Initial $map(node, vector[node])$  $\mathcal{M}$
 3: Initial network vector  $\mathcal{V}$
 4: for each node $i \in \mathcal{N}$ do
 5:     if $i$ is multiplexer node then
 6:         Generate pair $\mathcal{P}(s, i)$, $s$ is the select node of $i$
 7:         Insert $\mathcal{P}$ into $\mathcal{M}$, $key$ is $s$
 8:     end if
 9: end for
10: for each key $k$ in map $\mathcal{M}$ do
11:     Create logic cone $\mathcal{N}_k$ of the nodes in $k \rightarrow second$
12:     $\mathcal{V} \leftarrow \mathcal{N}_k$
13: end for
14: return $\mathcal{V}$, $\mathcal{M}$
```

First, all the multiplexer nodes are collected in the Boolean network (*line* 5). We classify a node as a *multiplexer node* if it satisfies three properties: a) it has two complemented children i_1, i_2; and b) nodes i_1, i_2 have a shared child; and c) one of these two nodes has a complemented fanin from the shared child. In Figure 3(a), i_1, i_2 are two completed children of node m. They have a shared node s, which is a fanin of i_2 and s complement is a fanin of i_1. Hence, m is *multiplexer node*. Node s is called the *select node*. The function of z is equivalent to a 2-to-1 multiplexer shown in Figure 3(b). Each AIG node is an AND operation and complemented edge is an inverter. We prove the equivalence in Eq. 1. When a node is identified as a *multiplexer node*, the *select node* is also collected. This is used for classifying the multiplexer nodes into '*vector multiplexers*.

Definition 1. *Vector multiplexer:* In an AIG network, if there is a set of *multiplexer nodes* $\mathcal{N}\{n_1, n_2, ...\}$ such that $\{n_1, n_2, ...\}$ have identical *select node*, then $\mathcal{N}$ is a vector

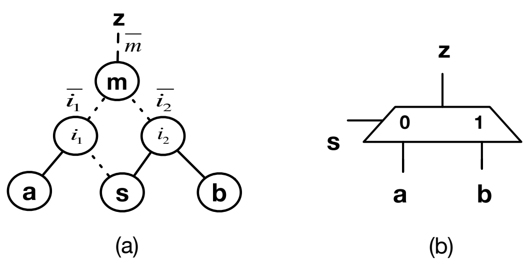

Figure 3: (a) Identify *multiplexer node*. (b) Equivalent 2-to-1 multiplexer.

multiplexer.

In high-level description C/SystemC or behavior Register Transfer level (RTL), the word or vector multiplexer is a multiplexer with word inputs (Figure 4(a)). However, in boolean network, hierarchy and module information is lost when the netlist is flattened during synthesis flow (see Figure 4(b)). We collect all the vector multiplexers based on Definition 1. First, we collect all the multiplexer nodes and the corresponding select nodes by traversing the entire AIG graph. The multiplexer nodes are classified into different sets depending on select nodes(*line* 7). Note that the multiplexer node does not necessary correspond to multiplexer in the design, which means that it is possible to find larger size vector multiplexer than the original design. The sub-network is generated by backward searching the AIG network (*line* 11). It starts from the nodes included in the vector multiplexer and ends at PIs or latches. The last step of pre-processing is ranking the vector multiplexers based on the size of the vector multiplexer (i.e. the number of multiplexer nodes of the vector multiplexer) and the sub-network depth.

$$\begin{aligned} i_1 &= a \cdot \overline{s} \\ i_2 &= b \cdot s \\ \overline{m} &= \overline{\overline{i_1} \cdot \overline{i_2}} = i_1 + i_2 \\ z &= a \cdot \overline{s} + b \cdot s \end{aligned} \tag{1}$$

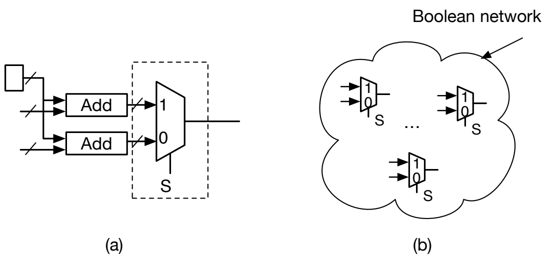

Figure 4: Vector 2-to-1 MUXes in Boolean network.

3.2 Multiplexer relocation

Definition 2. *Common specification logic (CSL):* $\mathcal{L}_1$, $\mathcal{L}_2$,..., $\mathcal{L}_n$ are combinational logic blocks that have the same number of inputs and outputs. Given the identical inputs, if $\mathcal{L}_1, \mathcal{L}_2, ..., \mathcal{L}_n$ are functionally equivalent, they form a **common specification logic** ($n >= 2$).

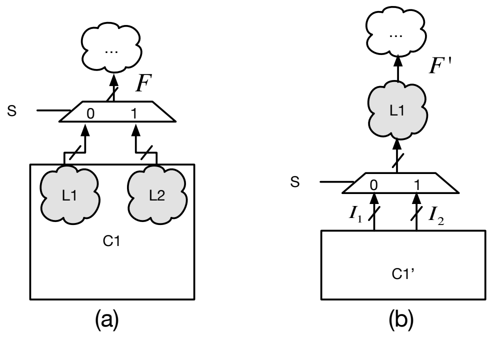

Figure 5: Single vector multiplexer re-allocation; L_1, L_2 are common specification logic (CSL); I_1, I_2 are inputs of L_1 and L_2.

The problem in our context can be defined as follows: Logic cone $\mathcal{L}_1$ is selected by $\bar{s}$ and logic cone $\mathcal{L}_2$ is selected by s(Figure 5 a); starting with a vector multiplexer $\mathcal{V}$ $\{n_1, n_2,...\}$, search if $\mathcal{L}_1, \mathcal{L}_2$ that are *common specification logic*. In other words, $\mathcal{L}_1$ and $\mathcal{L}_2$ are functionally equivalent if the inputs are identical. Note that they are not necessarily functionally equivalent in the design.

If $\mathcal{L}_1$ and $\mathcal{L}_2$ are common specification logic, the design can be minimized by reallocating the multiplexers with inputs $\mathcal{L}_1$ and $\mathcal{L}_2$. As shown in Figure 5(b), $\mathcal{I}_1$ and $\mathcal{I}_2$ are the inputs of the common specification logic $\mathcal{L}_1$ and $\mathcal{L}_2$. To reallocate the multiplexers, we first duplicate the logic of $\mathcal{C}_1$ without $\mathcal{L}_1$ and $\mathcal{L}_2$ (call it $\mathcal{C}_1'$). Then, a set of multiplexers are created to *re-multiplex* $\mathcal{I}_1$ and $\mathcal{I}_2$. These multiplexers are created depending on how logic $\mathcal{L}_1$ and $\mathcal{L}_2$ are selected in the original design. Now, the outputs of the multiplexers will be the inputs of $\mathcal{L}_1$. With this transformation, $\mathcal{F}$ and $\mathcal{F}'$ are functionally equivalent, which can be proved as follows: $\mathcal{F} = f(\mathcal{I}_1) \cdot S + f(\mathcal{I}_2) \cdot \bar{S}$; $\mathcal{F}' = f(\mathcal{I}_1 \cdot S + \mathcal{I}_2 \cdot \bar{S})$. If $S = 0$, $\mathcal{F} = \mathcal{F}' = f(\mathcal{I}_1)$; and if $S = 1$, $\mathcal{F} = \mathcal{F}' = f(\mathcal{I}_2)$, where f is the function of $\mathcal{L}_1$ and $\mathcal{L}_2$.

The most popular technique for checking if the designs are common specification logic is combinational equivalence checking (CEC). This problem has been addressed using BDDs [13], SAT[14] [15], AIG[10] etc. Current CEC methodology ensures that the two combinational circuits are checked for equivalence. However, this is not applicable here because that the boundary of the input is not given.

Eugene Goldberg[1] [16] proposed *common specification verification* technique that is able to identify if two combinational circuits N_1, N_2 have common specification. If N_1, N_2 are two logic with a common specification, they are *toggle equivalent*. However, this technique [16] is applicable to a known specification. Additionally, the technique is implemented by checking the common specification level by level from output to input, which means it requires a known set of signals as inputs.

Our algorithm of identifying common specification logic is shown in Algorithm 2. It takes the sub-network created in pre-processing as the input and returns two set of AIG nodes: common specification logic nodes $\mathcal{V}_C$ and bound nodes $\mathcal{V}_B$. Our algorithm solves this problem using subgraph

[1]Currently, the ACM version has an error about author's name (11.23.2016).

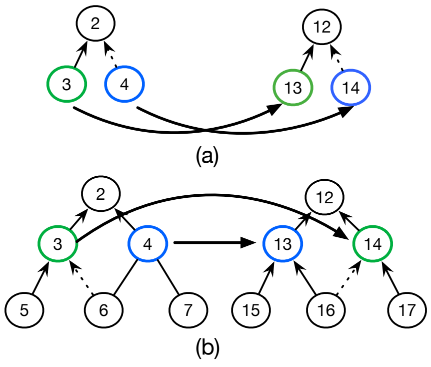

Figure 6: Node pairing in Algorithm 2 (a) case 1 (b) case 2

Algorithm 2 Identify Common Specification Logic

Input: Sub-Boolean network $\mathcal{N}_{sub}$
Output: common specification logic nodes $\mathcal{V}_C$ and bound nodes $\mathcal{V}_B$

```
1: Initialize $\mathcal{V}_{tmp} = Initial(\mathcal{N}_{sub})$
2: while $\mathcal{V}_{tmp} \neq NULL$ do
3:     $\mathcal{V}_C \leftarrow \mathcal{V}_C + \mathcal{V}_{tmp}$
4:     $\mathcal{V}_{tmp} \leftarrow NextPair(\mathcal{V}_{tmp})$
5: end while
6: $\mathcal{V}_B \leftarrow BoundNodes(\mathcal{V}_C)$
7: return $\mathcal{V}_B$, $\mathcal{V}_C$
```

isomorphism.

A digraph $G = (V, A)$ consists of a set of vertices V and arcs $A \subseteq V \times V$. Each vertex $v \subseteq V$ has an in-degree $d^-(v) = \#\{w \subseteq V | (w, v) \subseteq A\}$ and out-degree $d^+(v) = \#\{w \subseteq V | (v, w) \subseteq A\}$. The graphs G and H are isomorphic iff there is a permutation p of V such that for any two vertices $[u, v]$ in $E(G)$, $[p(u),\ p(v)]$ are also in $E(H)$. Determining if two graphs are isomorphic is thought to be neither an *NP-complete* problem nor a *P*-problem. However, the graph isomorphism in practice can be reduced into a polynomial time problem. This is because AIG network is a *Directed acyclic graph.* and the in-degree of each node is limited to a constant (number of fanin equals 2).

Our algorithm determines if two DAG subgraphs $G(V_1, A)$ and $G'(V_2, A)$ are isomorphic by backward searching level-by-level. The two vertices, V_1, V_2 are the two AIG nodes connected to a multiplexer (e.g. Figure 3-a, *node* a, b). At each iteration, we collect the nodes which maintain the isomorphism between the two graphs and pair the nodes for next iteration. Assuming that the complemented edge is 1 and regular edge is 0, there are four types of AIG node: $\{0, 0\}, \{0, 1\}, \{1, 0\}$, and $\{1, 1\}$. For pairing the nodes for checking isomorphism, two cases are considered:

Case 1) {0,1}, {1,0}: If only one of the coming edges of the two given nodes are complemented, the paring solution is unique (Figure 6 a). To make sure that the two graphs are isomorphic, *node* 3 must be paired with *node* 14 and *node* 4 must be paired with *node* 13. In this case, node 2 is added into the common specification logic nodes set. Nodes {3,14} and {4,13} are generated pairs for next iteration.

Case 2) {0,0}, {1,1}: If the two coming edges of the two given nodes are the same, there are two possible pairings. For example, in Figure 6(b), *node* 3 can be either paired with *node* 13 or *node* 14. As we can see, the more common specification logic is detected, the more area reduction can be achieved by sharing the logic. This means that we need to target on the largest subgraphs that are isomorphic. This is done by searching further AIG structures. In Figure 6(b), if *node* 3 is paired with *node* 13 and *node* 4 with *node* 14, the common specification logic searching algorithm stops at the second level and returns only three nodes. However, if *node* 3 is paired with *node* 14, the common specification logic will have three levels. This is because pairing of *node* 3 with *node* 14 makes the two sub-graphs {3,5,6} and {14,16,17} are isomorphic. In this paper, we set k equal to 3.

We illustrate the entire *multiplexer relocation* algorithm using an example shown in Figure 7.

Initialization (line 1): The initialization step returns the initial pairs for identifying common logic. First, the *multiplexer node* 4 is identified (Figure 7(b)). The multiplexer nodes are $\{4, 5, 6\}$ and the select node is S. The *initial nodes*, node 15 and 21, are the input nodes of the multiplexer. Therefore, node 15 and 21 are the two vertices for identifying isomorphic sub-graphs. The initial pairs are generated based the case 2 ($\{0, 0\}, \{1, 1\}$). To maximize the common logic size, the two initial pairs are $\{13, 19\}$ and $\{14, 20\}$. The common logic is $\mathcal{V}_C$=$\{15\}$.

Searching CSL (lines 2-5): This function starts with the initialized pairs and ends if there is no new pairs generated. At the beginning of each iteration, $\mathcal{V}_{tmp}$ stores the current level pairs. Then, $\mathcal{V}_{tmp}$ is overwritten by new pairs generated by $NextPair()$. Meanwhile, the new nodes are collected iteratively (*line* 3). As shown in Figure 7(b), at the first iteration $\mathcal{V}_{tmp} = [\{13, 19\}, \{14, 20\}]$. $NextPair()$ is the function that returns the pairs for next iteration based on the two cases we discussed. It returns the two pairs $[\{11, 18\}, \{12, 17\}]$, which are written into $\mathcal{V}_{tmp}$. Note sets {15,13,14} and {21,19,20} are the current subgraphs that are isomorphic. Nodes 13 and 14 are collected into $\mathcal{V}_C$, so that $\mathcal{V}_C = \{15, 13, 14\}$. At the second iteration, $NextPair()$ returns *null* since it is not able to generate any new pairs. At this iteration, nodes 11 and 12 are collected into $\mathcal{V}_C$. This function ends and returns $\mathcal{V}_C = \{15, 13, 14\} + \{11, 12\}$ $= \{15, 13, 14, 11, 12\}$.

Bound nodes (line 6): $BoundNodes()$ is the function that returns the nodes of the boundary between common specification logic and the remaining logic. It takes the common logic nodes $\mathcal{V}_C$ as input. This is done by collecting all the children of the nodes that have the largest depth in subgraphs ($\mathcal{V}_C$). Additionally, this function returns the relationship between the select node (S) with the *bound nodes.* In Figure 7(c), the *bound nodes* are $\{A_1, B_1\}$ and $\{A_2, B_2\}$ which are selected by $\bar{S}$ and S respectively.

Multiplexer relocation: The relocation function takes the original AIG network, *CSL* nodes ($\mathcal{V}_C$), and *bound nodes* ($\mathcal{V}_B$) as inputs and returns the optimized AIG network. The relocation is done in three steps: 1) Duplicate the non-common logic as $\mathcal{C}'$ when the bound nodes are reached. Note that the bound nodes are included in $\mathcal{C}'$. In Figure 7(d), $\mathcal{C}'$ only includes the PI nodes. 2) A set of multiplexers (AIG nodes) are created to "re-multiplexing" the bound nodes. Nodes {22, 23, 24} and {25, 26, 27} are the two multiplexers in this example; 3) The common specification logic is connected to the multiplexer nodes by matching the selecting function.

3.3 Multiplexers relocation with Retiming

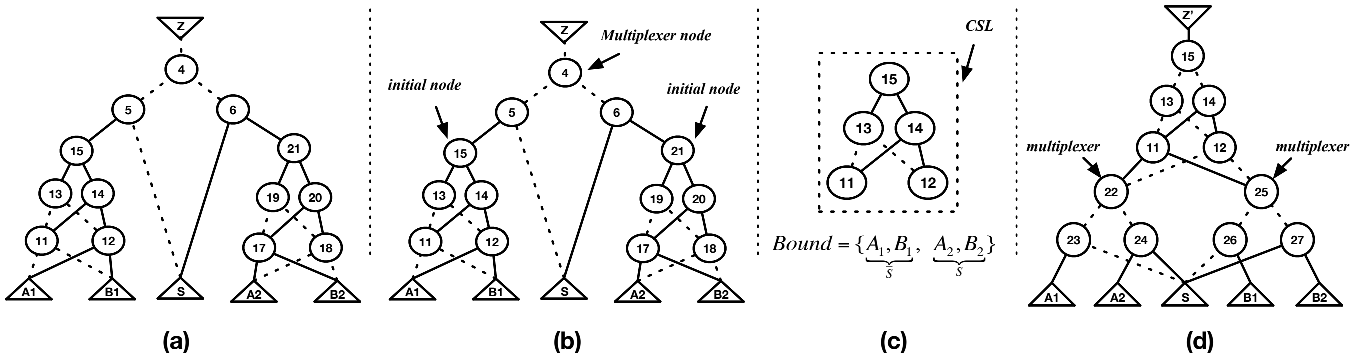

Figure 7: (a) Original AIG network, A_1, A_2, B_1, B_2, S are PIs; Z is output. (b) Initialize multiplexer relocation; multiplexer node is node 4; initial pairs are {13,19},{14,20}. (c) Common specification logic, bound nodes, and how bound nodes are selected. (d) AIG network after multiplexer relocation

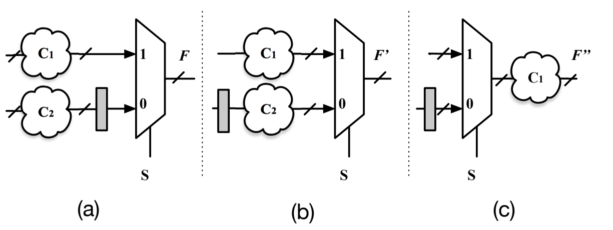

Figure 8: Identifying common specification logic after backward retiming (a) Original design. (b) Backward retiming applied. (c) Multiplexers relocation.

The proposed algorithm in Section 3.2 is applicable to both combinational and sequential designs. However, it does not provide relocations across the latches. This is because pre-processing generates sub-networks between the vector multiplexer and the latches or PIs of the sequential designs. However, this may eliminate some common specification logic when generating the sub-network. C_1 and C_2 are common specification logic in Figure 8 (a). However, our technique cannot identify any common logic. This is because the sub-network generated by pre-processing only includes C_1 and the vector multiplexer. To overcome this limitation, we combine our technique with *retiming*. Backward retiming enables the algorithm to identify more common specification logic (shown in Figure 8(b)). Then, our technique is able to identify C_1 and C_2 as common specification logic. Figure 8(c) shows that the design has been minimized by backward retiming and multiplexer relocation. One limitation of this approach is that the number of latches may increase if only backward retiming is applied.

4. EXPERIMENTAL RESULTS

The proposed algorithm has been implemented in the *ABC* environment [10]. It takes the gate-level netlist as input and outputs the optimized gate-level netlist. The designs tested in Table 1 have been initially implemented in C/C++. These implementations are compiled into RTL-level by IBM high-level synthesis tool and GAUT [17] and are synthesized into gate-level netlist using IBM logic synthesis system BooleDozer [18]. We evaluate our technique by measuring the design area and the delay (as logic levels) after technology mapping. The area and logic levels are obtained by *ABC* mapper (command *map -v*) using IBM 14nm technology library.

Design	*Original*	*Ours*	*Δ area*	*CPU time*
ibm1	24010	20154	16.1 %	320 s
branch1	10759	8920	17.1 %	57 s
branch2	5654	4144	26.7 %	167 s
branch3	3421	3038	9.9 %	1.7 s
polynom1	24648	15676	37.4 %	470 s
polynom2	37254	33220	10.8 %	611 s
loop_safe1	1587	1587	0 %	0.1 s
loop_safe2	2760	2760	0 %	0.1 s

Table 1: Evaluation of *multiplexer reallocation*; comparison with traditional design flow

```
for (i = 0; i <5; i++)
{
  if(x > y && x < 2*y)
      f = f - x;
  else
      f = f + y;
}
```

Table 2: C++ implementation example

In Table 1, *Original* represents the area of the designs that go through compilation, high-level synthesis, logic synthesis, and technology mapping. Logic synthesis and mapping are done using ABC(*command: strash; dch -v; map -v*)[10]. Optimization command *dch*, the most powerful optimization function in ABC) , has been applied multiple times until the number of AND nodes and logic levels cannot be reduced. The experiments in third column represent the area with our *multiplexer relocation* technique. The only difference compared to *Original*, is that we apply multiplexer relocation *before* the logic synthesis. *ibm1* is a datapath design of IBM *Z-series* microprocessor. *branch* designs are branch prediction designs. *polynom* benchmarks are polynomial division algorithm. *loop_safe* are the benchmarks in *loop* track of SV-COMP benchmarks suit.

The results shown in Table 1 demonstrate that the proposed method is able to improve design area beyond the approach offered by traditional design flow. For the *loop_safe*

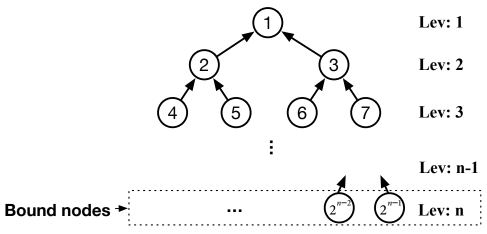

Figure 9: *Fanout-free* AIG

designs, our technique does not provide additional area reduction compared to the traditional flow. This is because that these designs have simple structure so that ABC is able to identify and merge the equivalent nodes during the logic synthesis. The proposed algorithm does not improve the delay (number of levels) directly. This is because relocating the multiplexers does not reduce the logic depth of the critical path. However, our technique can potentially enable other optimization techniques in the design flow. For example, we observe that the number of levels of the critical path of *ibm1* design can be significantly reduced after relocations by other techniques implemented in [18].

4.1 Limitations

This approach has also been evaluated using ISCAS89 and ITC99 benchmarks suits. The benchmarks we tested include $c5315, c7552, s9234, s38584, b04, b14, b22$ circuits. However, the multiplexer relocation technique improves only 4.2% area for $c7552$. For other designs, this technique does not provide reduction. This is because:

a) The technique is applicable to a designs that contain *vector multiplexers* (see Definition 1). **b)** The number of *bound nodes* n_B must be 2x smaller than the number of common logic nodes n_C. This is because our technique needs to insert $n_B/2$ multiplexers to reallocate the original multiplexers. Note that each multiplexer are a two level logic that consists of three AIG nodes.

The worst-case scenario is when the common logic (CSL) is represented as a *Fanout-free* AIG graph (Figure 9). Assuming the common logic includes *level 1* to *level n-1*, n_C is the number of nodes in the common logic.That is, n_C=$2^0 + 2^1 + ...2^{n-2} = 2^{n-1} - 1$, and n_B=2^{n-2}. To reallocate a multiplexer, n_{cost}=$3 * 2^{n-2}$=$2^{n-1} + 2^{n-2}$ extra nodes are inserted. The number of reducible nodes is equal to n_C. Hence, Δn=n_C - n_{cost}=$2^{n-1}-1-(2^{n-1}+2^{n-2}) = -1-2^{n-2}$, which is always negative.This means that the multiplexer relocation always provides negative gain when the logic cone of the vector multiplexer is a *Fanout free* network. Note that this approach is less efficient if the designs have been optimized using logic synthesis. This is because the current method is limited to identify common specification logic and vector multiplexers structurally since AIG is not canonical.

5. CONCLUSION

The paper presented an efficient logic synthesis technique that targets area minimization of datapath designs. The proposed technique combines the high-level synthesis technique, *resource sharing* with logic synthesis. Instead of binding the resources of arithmetic operations in high-level synthesis, our technique is able to binding arbitrary combinational logic if they form common specification logic. We presented an efficient algorithm of identifying common specification logic using graph isomorphism. The experimental results show that our technique can provide additional area reduction beyond traditional design flow. Future work will focus on improving the algorithm of identifying common logic regardless of the structure.

6. REFERENCES

[1] R. Zimmermann, “Datapath synthesis for standard-cell design,” in *2009 19th IEEE International Symposium on Computer Arithmetic*. IEEE, 2009, pp. 207–211.

[2] G. D. Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[3] M. Potkonjak and J. Rabaey, “Optimizing Resource Utilization using Transformations,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 13, no. 3, pp. 277–292, 1994.

[4] M. B. Srivastava and M. Potkonjak, “Optimum and Heuristic Transformation Techniques for Simultaneous Optimization of Latency and Throughput,” *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 3, no. 1, pp. 2–19, 1995.

[5] S. P. Mohanty, N. Ranganathan, and S. K. Chappidi, “An ILP-based Scheduling Scheme For Energy Efficient High Performance Datapath Synthesis,” in *ISCAS*, vol. 5. IEEE, 2003, pp. V–313.

[6] J. Cong and J. Xu, “Simultaneous FU and Register Binding-based on Network Flow Method,” in *Design, Automation and Test in Europe, 2008. DATE’08*. IEEE, 2008, pp. 1057–1062.

[7] M. Ciesielski, D. Gomez-Prado, Q. Ren, J. Guillot, and E. Boutillon, “Optimization of Data-flow Computations using Canonical TED Representation,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 28, no. 9, pp. 1321–1333, 2009.

[8] G. Martin and G. Smith, “High-level Synthesis: Past, Present, and Future,” *IEEE Design & Test of Computers*, no. 4, pp. 18–25, 2009.

[9] A. Mishchenko, S. Chatterjee, and R. Brayton, “DAG-aware AIG Rewriting A Fresh Look at Combinational Logic Synthesis,” in *43rd DAC*. ACM, 2006, pp. 532–535.

[10] A. Mishchenko *et al.*, “ABC: A System for Sequential Synthesis and Verification (2007),” *URL http://www. eecs. berkeley. edu/alanmi/abc*, 2010.

[11] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, “SIS: A System for Sequential Circuit Synthesis,” 1992.

[12] A. Biere, “The AIGER and-inverter graph (AIG) format,” *Available at fmv. jku. at/aiger*, 2007.

[13] R. E. Bryant, “Graph-based Algorithms for Boolean Function Manipulation,” *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[14] A. Kuehlmann and F. Krohm, “Equivalence Checking using Cuts and Heaps,” in *34th DAC*. ACM, 1997, pp. 263–268.

[15] E. Goldberg, M. Prasad, and R. Brayton, “Using SAT for Combinational Equivalence Checking,” in *DATE*. IEEE Press, 2001, pp. 114–121.

[16] E. Goldberg, “Equivalence Checking of Dissimilar Circuits II,” Technical report, Tech. Rep., 2004.

[17] P. Coussy, C. Chavet, P. Bomel, D. Heller, E. Senn, and E. Martin, “GAUT: A High-level Synthesis Tool for DSP Applications,” in *High-Level Synthesis*. Springer, 2008, pp. 147–169.

[18] L. Stok, D. Kung, and et al., “BooleDozer: Logic Synthesis for ASICs,” *IBM Journal of Research and Development*, vol. 40, no. 4, pp. 407–430, 1996.