

Scalable Synthesis and Clustering Techniques using Decision Diagrams

Andrew C. Ling, *Member, IEEE*, Jianwen Zhu, *Member, IEEE*, and Stephen D. Brown, *Member, IEEE*

Abstract—BDDs have proven to be an efficient means to represent and manipulate Boolean formulae [1] and sets [2] due to their compactness and canonicity. In this work, we leverage the efficiency of BDDs for new areas in FPGA CAD flow including cut generation and clustering by reducing these problems to BDDs and solving them using Boolean operations. As a result, we show that this leads to more than 10x reduction in runtime and memory use when compared to previous techniques as reported in [3] and [4]. This speedup allows us to apply our work to new areas in the FPGA CAD flow previously not possible. Specifically, we introduce a new method to solve the logic synthesis elimination problem found in FBDD, a recently reported BDD synthesis engine with an order of magnitude speedup over SIS. Our new elimination algorithm results in an overall speedup of 6x in FBDD with no impact on circuit area.

Index Terms—FPGAs, Cut Generation, Clustering, BDD.

I. INTRODUCTION

AS the FPGA capacity grows with each chip generation, the scalability of FPGA CAD tools is a growing concern. This is a result of the exponential space and time complexity many CAD algorithms have in relation to the circuit size, n . Scalability problems have traditionally been handled by divide and conquer techniques where the circuit is partitioned into several smaller circuits ([5], [6]). This reduces the problem size and, as a result, dramatically reduces the solution space the CAD tool must explore. Although partitioning has proven to improve the scalability of CAD algorithms, partitioning a design will lead to solutions much further from optimal when compared to non-partitioning based techniques.

As an alternative to partitioning techniques, we propose using heuristics that improve the scalability of CAD algorithms by removing redundant operations and data. Specifically, in this work we improve the scalability of cut generation and FPGA LUT clustering. By recognizing that both cut generation and clustering can be reduced to set operations, we can represent cut and cluster sets as reduced-ordered binary-decision diagrams (BDDs) and, as a consequence of this, leverage efficient BDD managers to solve the cut generation and clustering problem ([7], [8]).

To generate the cut and cluster set for a given node v , a dynamic programming approach is devised in [9] and [4] respectively. In [9], cuts for node v are created by using the cuts from the fanin nodes of v and combining them to form larger cuts. An example of this is illustrated in Fig. 1. Here,

to generate a cut for node g , a cut from node e and f are duplicated and combined to form a larger cut.

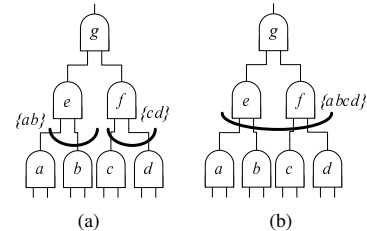


Fig. 1. Illustration of cut generation through dynamic programming. (a) Two cuts for nodes e and f . (b) Larger cut created by duplicating and concatenating cuts ab and cd for node g .

In [4], a similar dynamic programming approach is adapted to clustering and is shown in Fig. 2 where the large cluster shown in Fig. 2b is formed by duplicating and combining the smaller clusters shown in Fig. 2a.

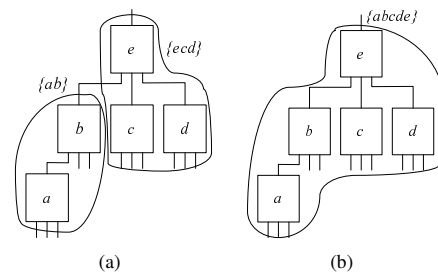


Fig. 2. Illustration of cluster generation through dynamic programming. (a) Two clusters rooted at node b and e . (b) Larger cluster created by duplicating and combining clusters ab and ecd .

Unfortunately, both of these approaches duplicate subsets to form larger sets of data which leads to scalability problems for large circuits and set sizes. In contrast, we represent our cut and cluster sets as a BDD. BDDs are generated using a dynamic programming framework, as in the case of cut and cluster generation. However, unlike cut generation or clustering, the BDDs are not duplicated when creating larger BDDs. This property is illustrated in Fig. 3. Here, two smaller BDDs, g and h , are joined to create function f . However, the subfunctions are not duplicated to form the larger function f . This is possible since old BDDs can be referenced from multiple sources. Thus the top node, c , in function f , simply references to the original locations of function g and h without needing to recreate and store the original functions. This saves both runtime and memory when creating large BDDs. We will

Manuscript received December 20, 2006. This work was supported by the IEEE.

Andrew C. Ling, Jianwen Zhu, and Stephen D. Brown are with the University of Toronto. Stephen D. Brown is also with Altera Corporation.

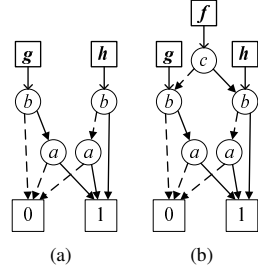


Fig. 3. Illustration of BDD creation using dynamic programming. (a) Two BDDs representing function $g = ab$ and $h = a + b$. (b) New BDD representing function $f = \bar{a}b + ca + cb = \bar{c}g + ch$; note that previous BDDs g and h do not have to be duplicated.

show in later sections how we reduce cut and cluster sets to BDDs such that we can leverage this important property of BDDs to improve the scalability of cut generation and clustering. As a result, our approach is an order magnitude better than previous approaches, both in terms of runtime and memory.

The rest of the paper will be organized as follows. Section II will give problem preliminaries and background material. Section III will describe our cut generation approach. Section IV will describe our clustering approach. Section V will give an overview of our results followed by some concluding remarks in Section VI.

II. BACKGROUND

A. Terminology

Before we can describe our problem, we first review some basic terminology here. The combinational portion of a Boolean circuit can be represented as a directed acyclic graph (DAG) $G = (V_G, E_G)$. A node in the graph $v \in V_G$ represents a logic gate, primary input or primary output, and a directed edge in the graph $e \in E_G$ with head, $u = head(e)$, and tail, $v = tail(e)$, represents a signal in the logic circuit that is an output of gate u and an input of gate v . The set of *fanin edges* for a node v , $fanine(v)$, is defined to be the set of edges with v as a tail. Similarly, the set of *fanout edges* for v , $fanoute(v)$, is defined to be the set of edges with v as a head. A *primary input* (PI) node has no fanin edges and a *primary output* (PO) node has no fanout edges. The set of distinct nodes that supply fanin edges to v are referred to as *fanins* and is denoted $fanin(v)$. Similarly, the set of distinct nodes that connect to fanout edges from v are referred to as *fanouts* and is denoted $fanout(v)$. A node v is *K-feasible* if $|fanin(v)| \leq K$. If every node in a graph is *K-feasible* then the graph is *K-bounded*.

Each edge e has an associated delay, $delay(e)$. The length of a path is the sum of the delays of the edges along the path. At a node v , the *depth*, $depth(v)$, is the length of the longest path from a primary input to v and the *height*, $height(v)$, is the length of the longest path from a primary output to v . Both the depth for a PI node and the height for a PO node are zero. At an edge e , the *depth*, $depth(e)$, is the length of the longest path from a primary input to e and the *height*, $height(e)$, is the length of the longest path from a primary output to e . Both

the depth and the height of an edge include the delay due to the edge itself. The depth or height of a graph is the length of the longest path in the graph.

A *cone* of v , C_v , is a subgraph consisting of v and some of its nonPI predecessors such that any node $u \in C_v$ has a path to v that lies entirely in C_v . Node v is referred to as the *root* of the cone. The size of a cone is the number of nodes and edges in the cone. At a cone C_v , the set of fanin edges, $fanine(C_v)$, is the set of edges with a tail in C_v and the set of fanout edges, $fanoute(C_v)$, is the set of edges with v as a head. The set of fanins to the cone are also known as a *cut* in a graph. Thus, there is a one to one correlation between all cuts and cones in a graph. With fanin edges and fanout edges so defined, a cone can be viewed as a node, and notions that were previously defined for nodes can be extended to handle cones. Notions such as $fanin(\cdot)$, $fanout(\cdot)$, $depth(\cdot)$, $height(\cdot)$ and *K-feasibility* all have similar meanings for cones as they do for nodes.

B. Problem Description and Related Work

1) *Cut Generation*: While cut generation has been traditionally applied to iterative FPGA technology mappers, such as DAOMap [10] and IMap [11], there has been a renewed interest in the cut generation problem [3], [12] due to its growing use in several other CAD problems including:

- Boolean matching of PLBs [13], [14]
- resynthesis of LUTs [15]
- synthesis rewriting [16]
- synthesis elimination [17], [18]

One of the first pieces of work to define the cut generation problem was in [19] where the authors define the set relation to generate all *K-feasible* cuts shown in equation 1. For a detailed explanation of equation 1, please refer to [19]. This contrasts with incremental cut generation methods based on network flow [20], [21] and has proven to be much faster.

$$\Phi(v) = \{c_u * c_w \mid c_u \in \{\{u\} \cup \Phi(u) \mid u \in fanin(v)\}, \quad (1)$$

$$c_w \in \{\{w\} \cup \Phi(w) \mid w \in fanout(v), u \neq w, \|c_u * c_w\| \leq K\}$$

In equation 1, $\Phi(v)$ represents the cut set for node v ; $\{u\}$ represent the trivial cut (contains u only); c_u represents a cut from the cut set $\{\{u\} \cup \Phi(u)\}$; and $\Phi(u)$ represents the cut set for fanin node u . Traditional methods generate cuts by visiting each node in topological order from PIs to POs and merging cut sets as defined by equation 1. Two cut sets are merged by performing a concatenation ($c_u * c_w$) of all cuts found in each fanin cut set, and removing any newly formed cuts that are no longer *K-feasible* ($\|c_u * c_w\| \leq K$). For example, referring to Fig. 4, cut c_2 is generated by combining the cut c_1 with the trivial cut v_4 ($c_2 = c_1 * v_4 = v_1 v_2 v_4$). Generating cuts this way is not scalable to large cut sizes ($K \geq 6$) and for circuits containing a large degree of reconvergent paths. For example, in IMap [11], which utilizes a popular technology mapping framework, cut generation takes more than 99% of the runtime for $K = 7$. In [9], the authors address this problem by selectively pruning cuts that they deem to be wasteful.

However, for large cut sizes, pruning tends to remove too many cuts that may be valuable in the final mapping solution.

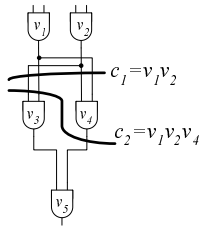


Fig. 4. Example of two cuts in a netlist for node v_5 where c_2 dominates c_1 ($K = 3$).

The main reason equation 1 is not scalable for large cut sizes is because subcuts must be duplicated every time a new cut is generated. For example, referring to Fig. 4, cut c_1 must be duplicated to generate cut c_2 . Furthermore, equation 1 can generate redundant cuts. A cut, c_2 , is redundant if it completely contains all the input nodes of another cut, c_1 , in which case c_2 is known as a *dominator* cut. Fig. 4 illustrates this relation. These cuts can be removed because they will not affect the final quality of a mapping solution. In ABC [3], the authors address this problem by assigning all cuts a signature such that dominator cuts can be quickly identified and removed. This, along with several other optimization, results in an order of magnitude runtime reduction over previous techniques. As a consequence, ABC is currently the fastest LUT technology mapper available with competitive depth and area results. However, even with its clever heuristics, ABC cut generation time slows down significantly for cuts sizes of 8 or larger. Although this is not a problem for commercial FPGAs that restrict their LUT size to 6 or less [22], migrating the covering problem to larger problems requires a more scalable cut generation solution. As a solution, we will show that reducing cut generation to BDDs dramatically improves its scalability and as a result, expands the application of cut generation to new problems previously not thought possible. As a practical example, we will show how we apply cut generation in the elimination step in a BDD-based synthesis flow which leads to a 6x speedup in runtime without any degradation to circuit area.

2) *Clustering of K-LUTs*: Modern FPGAs are hierarchical in nature where LUTs are grouped into regular logic array blocks (LABs), also known as clustered logic blocks (CLBs) or more simply, clusters. Deciding how to pack a given LUT netlist into an array of clusters is the clustering problem and it has a significant impact on the final performance of the circuit. Clustering has typically not been a bottleneck during the CAD flow where traditionally clustering has been solved using greedy algorithms [23]. In [23], LUTs are successively packed together such that routability or delay is optimized. Although this produces good clustering solutions in a reasonable amount of time, a recently reported study has shown that solving the clustering problem from a global perspective leads to significant performance gains [4]. In [4], the authors combine clustering and technology mapping into a single phase. During this process, several alternate clustering

solutions are stored and evaluated. This allows the tool to explore a much larger solution space than solving technology mapping and clustering disjointly. Also, while evaluating each clustering solution, an optimal delay value is maintained. The results are fairly impressive where the authors are able to get a 12.3% improvement in circuit delay on average. However, solving technology mapping and clustering together explores a much larger search space than disjoint methods. Furthermore, a large set of clustering solutions must be computed and explored during the forward traversal of the algorithm. Both these factors has led to a 100x runtime penalty when compared to previous techniques. Since FPGA design sizes are reaching an order of 100K LUTs, we feel that a 100x runtime penalty will be a barrier for the practical application of [4] without heuristics to improve its runtime.

In our approach, we adopt a similar global heuristic as in [4] since we feel that this is a significant factor leading to performance gains found in [4]. However, we perform clustering as a disjoint step after technology mapping to reduce the search space of our clustering tool. We suspect that maintaining global information during clustering is an important factor to improve the final performance of the circuit. however, this requires storing a large set of clustering solutions which is the main factor for the runtime penalty reported in [4]. To alleviate this problem, we propose using zero-suppressed BDDs (ZDDs), which are extremely efficient in representing sets, to represent our clusters. In sections IV and V, we will prove that this will have a significant runtime advantage when compared to [4] while maintaining some of its performance gains.

C. The Covering Problem

The covering problem seeks to find a set of covers to cover a graph such that a given characteristic of the final covered graph is optimized. For example, when applied to K -LUT technology mapping, the covering problem returns a covered graph such that the number of distinct covers in the graph is minimized where each cover gets mapped directly into a single LUT. This is illustrated in Fig. 5. We will show in later sections how to adapt the covering problem for the synthesis elimination step.

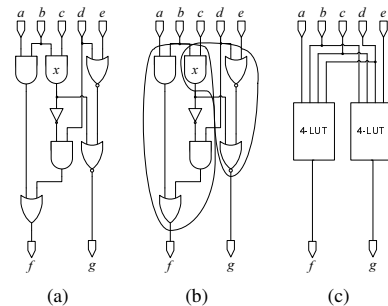


Fig. 5. Illustration of the covering problem when applied to K -LUT technology mapping. (a) Initial network. (b) A covering of the network. (c) Conversion of the covering into 4-LUTs.

A common framework to solve the covering problem is shown in Fig. 6. The covering problem starts by generating all

```

1 GENERATECUTS( $K$ )
2 for  $i \leftarrow 1$  upto  $MaxI$ 
3   TRAVERSEFWD()
4   TRAVERSEBWD()
5 end for

```

Fig. 6. High-level overview of network covering.

K -feasible cuts in the graph (line 1). This is followed by a set of forward and backward traversals (line 3-4) which attempt to find a subset of cuts to cover the graph such that a given cost function is minimized. Iteration is necessary ($MaxI > 1$) if the covering found in TRAVERSEBWD() influences the cost function used in TRAVERSEFWD(). A detailed description of this algorithm when applied to technology mapping can be found in [11].

```

1 foreach  $v \in$  TSORT( $G(V, E)$ )
2    $cut_v \leftarrow$  MINCOSTCUT( $v$ )
3    $cost_v \leftarrow$  COST( $cut_v$ )
4 end foreach

```

Fig. 7. High-level overview of forward traversal.

1) *Forward Traversal:* Fig. 7 illustrates the high-level overview of the forward traversal. Here, each node is visited in topological order from PIs to POs. For each node, the minimum cost cut is found (line 2). After the minimum cost cut is found, the cost of the root node v is assigned the cost of the cut (line 3). Note that MINCOSTCUT is dependent on the goal of the algorithm. In later sections, we will describe the cost function used when we apply the covering problem to elimination.

```

1 MARKPOASVISIBLE()
2 foreach  $v \in$  RTSORT( $G(V, E)$ )
3   if VISIBLE( $v$ )
4     foreach  $u \in$  fanin( $cut_v$ )
5       MARKASVISIBLE( $u$ )
6   end if
7 end foreach

```

Fig. 8. High-level overview of backward traversal.

2) *Backward Traversal:* Fig. 8 illustrates the high-level overview of the backward traversal. First, all POs are marked as visible (line 1). Next, the graph is traversed in reverse topological order. If a node is visible, its minimum cost cut found in the preceding forward traversal, cut_v , is selected and all of its fanins are marked as visible (line 4-5). After the backward traversal completes, the minimum cost cuts of all visible nodes in the graph are converted to cones to cover the network.

III. BDDCUT: SCALABLE CUT GENERATION

As described in section I, there is a growing need for scalable cut generation. Prior to this work, cut generation has generally been limited to applications requiring small cuts where K is smaller than 6 [10], [11], [16]. In this section, we will explain how to reduce cut generation to BDDs which will later prove to dramatically improve its scalability.

As described in equation 1, cuts are generated by combining the subcuts in every possible way. This is extremely inefficient since subcuts are duplicated every time they are used to generate a new cut. Our BDD-based approach solves this problem by sharing subcuts between larger cuts. Referring back to our original cut expression in equation 1, we can rewrite our equation as a Boolean expression.

$$f_v = \prod_{u \in fanin(v)} (u + f_u) \quad (2)$$

Equation 2 is very similar to the set relation shown in equation 1; however, in contrast with previous approaches, we maintain cut set representations as a Boolean function. In our approach, we map a unique Boolean variable to each node v found in our netlist and represent cuts by the conjunction of the fanin node variables. Thus, our cut set f_v will be a Boolean expression in SOP form where each cube will represent a cut. To join cut sets, we replace the set union operation (\cup) with a logic OR. Furthermore, the \prod operation can be thought as the logical AND of all clauses ($u + f_u$). For example, consider Fig. 9. Here, each node is represented by a Boolean variable

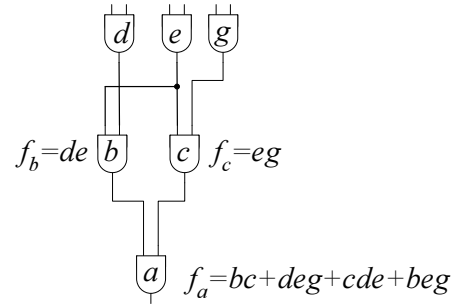


Fig. 9. Symbolic representation of cut sets.

where each product term in the function represents a cut. Also, notice that the cut set function f_a is the conjunction between the clauses $(c + f_c)$ and $(b + f_b)$.

A problem with using cubes to represent our cut set is that it suffers from similar scalability problems as traditional cut generation methods since each cut needs to be stored separately as a cube and no subcut sharing occurs. A solution to this is to represent our cut set as a BDD, (for a detailed description of the BDD data structure, please refer to [1]). BDDs are DAGs which represent a Boolean function where each node in the DAG represents one variable. Node edges represent positive (1) or negative (0) assignments to the variable where each edge points to the associated cofactor. For example, referring back to Fig. 9, the BDD used to represent the cut set f_a is shown in Fig. 10. Here, positive edges are represented by a solid line and negative edges are represented by a dotted line.

Notice that representing cut sets as a BDD allows subcuts to be shared as cofactors. Thus, subcuts can be reused in expressing larger cuts. For example, consider Fig. 11 which shows the BDDs representing the cut set functions shown in Fig. 9. In Fig. 11a, two small BDDs are shown. In Fig. 11b, the BDDs shown in Fig. 11a are reused as cofactors to build

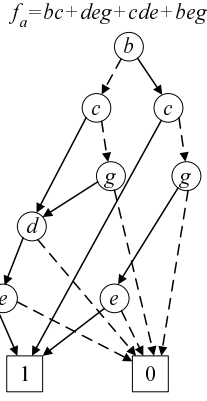


Fig. 10. BDD representation of cut set in Fig. 9.

the BDD for function f_a . Thus, subcuts de and eg do not have to be duplicated to form larger cuts for node a .

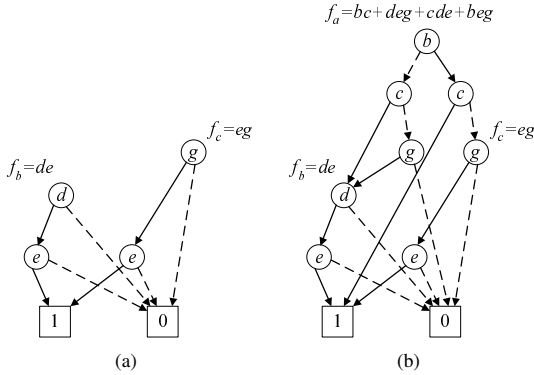


Fig. 11. Illustration of reusing BDDs to generate larger BDDs. (a) Small BDDs representing cut set function f_b and f_c . (b) Reusing BDDs in (a) as cofactors within cut set function f_a .

BDDs can also share cofactors within a single cut set. For example, consider Fig. 12. Notice that in the BDD

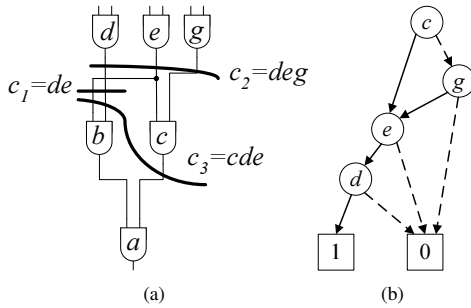


Fig. 12. BDD representation of node a cuts c_1 , c_2 , and c_3 ($K = 3$).

representation, the subcut $c_1 = de$ is a positive cofactor for variable c and g , and is shared by two larger cuts $c_3 = cde$ and $c_2 = deg$. The benefit of subcut sharing is very sensitive to variable ordering. For example, in the previous example, c_1 could not be shared if variables d and e were found at the top of the BDD. Hence, to ensure

that subcut sharing is maximized, we assign BDD variables to nodes such that fanin node variables are always found below their fanout node variables in the BDD cut set. This is stated formally in lemma 3.1 and proposition 3.2.

Lemma 3.1: Consider two functions f_1 and f_2 represented as BDDs where f_1 is composed of f_2 and some other variables (i.e. $f_1 = g(f_2, x_0, \dots, x_n)$). Also, let θ be the set of variables found in f_1 which are not in f_2 . The BDD graph f_2 can exist as a subgraph in f_1 if and only if all the variables in f_2 are below all variables in θ .

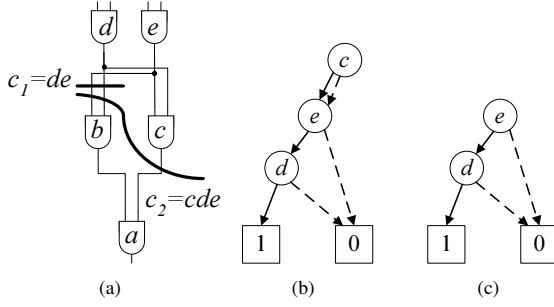
An intuitive explanation to lemma 3.1 can be seen in Fig. 11 where the BDD representing f_b is a subgraph in f_a , which would not be possible if variables d and e were not at the bottom of the BDD in f_a .

Cut sharing will only occur if BDD variables are assigned nodes in the following manner. If the variable assigned to node v and the variable assigned to node u appear in the same BDD, f , and if $u \in \text{fanin}(v)$, the variable assigned to node v should appear above the variable assigned to node u in f . For example, this relationship is shown in Fig. 12. Here, the node labeled c in Fig. 12a has fanins labeled e and g . This translates to the BDD shown in Fig. 12b where BDD variable c appears above variables e and g . This variable ordering condition is stated formally in proposition 3.2.

Proposition 3.2: Cut set sharing can only occur if the variable assigned to nodes is such that variables assigned to a node will appear above variables assigned to their transitive fanins if those variables appear in the same BDD cut set.

Proof: Proof by contradiction. Recall from equation 2 that the Boolean expression to generate the cut set for a given node v is $f_v = \prod_{u \in \text{fanin}(v)} (u + f_u)$. First we will look at a single fanin to v . Let us assign $g = (u + f_u)$ and assume the BDD f_u exists as a subgraph in g and hence is shared. Also assume that u is not the top variable in g . However, if u is not the top variable in g , f_u cannot exist in g by lemma 3.2. Thus by contradiction, if f_u is shared in g , u must be the top variable in g . Since all variables in f_u are assigned to the transitive fanin nodes to the node u , the variables assigned to the transitive fanin nodes appear below u in function g . A same argument can be applied to all fanin nodes $u \in \text{fanin}(v)$. ■

Another benefit of using BDDs is that redundant cuts, such as dominator cuts, are automatically removed. For example, consider Fig. 13a containing the cut c_1 and the dominator cut c_2 . As a BDD, c_1 and c_2 are shown in Fig. 13b. Since BDD node c is now redundant, it can be removed as in Fig. 13c which removes the dominator cut c_2 . BDD managers will perform this redundancy removal during the BDD construction. If both the positive and negative cofactor of a node are common, it is removed and the parent edge to the redundant node will be adjusted to point to the common cofactor. For example, in Fig. 13b, if a parent edge to c existed it would be adjusted to point to node e after c is removed. Both dominator cut removal and subcut sharing substantially reduces the space complexity of a BDD represented cut set.


 Fig. 13. BDD representation of node a cuts c_1 and c_3 ($K = 3$).

As we will show in later sections, zero-suppressed BDDs (ZDDs) are another method to represent sets efficiently. These can also be used to represent our cut sets; however, the benefit of redundant cut removal is not possible with ZDDs. For example, in Fig. 13, a ZDD will not remove the node c in Fig. 13b since it will treat the dominator cut as a unique element in the cut set. Furthermore, since we can represent a cut directly as a cube, BDDs can be directly leveraged without the need to use the ZDD structure.

A. Symbolic Cut Generation Algorithm

Fig. 14 illustrates our cut generation algorithm. First, the

```

CutGeneration()
1   $G(V, E) \leftarrow \text{TSORT}()$ 
2  foreach  $v \in G(V, E)$ 
3       $f_v \leftarrow 1$ 
4       $b_v \leftarrow \text{CREATENEWBDDVARIABLE}()$ 
5  end foreach
6  foreach  $v \in G(V, E)$ 
7      foreach  $u \in \text{fanin}(v)$ 
8           $f_x \leftarrow \text{BDDOR}(b_u, f_u)$ 
9           $f_v \leftarrow \text{BDDANDPRUNE}(f_v, f_x, K)$ 
10     end foreach
11 end foreach
    
```

Fig. 14. High-level overview of symbolic cut generation algorithm.

netlist is sorted in topological order (line 1). Next, the cut set function, f_v , for each node in the graph is initialized to a constant 1 and assigned a unique variable (line 2-5). Finally, for each node, v , its cut set is formed following equation 2 (line 7-10). When forming the cut set for node v , each fanin node, u , is visited (line 7) and a temporary cut set is formed by the logical OR of the trivial cut u and its cut set f_u . Next, the temporary cut set is conjoined to the cut set of v using the logical AND operation (line 9). When forming larger cuts with the logical AND operation, it is possible to form cuts larger than K , thus BDDANDPRUNE is also responsible for pruning cuts that are not K -feasible. It does so by removing all cubes that are contain more than K positive literals which will be explained in the detail in the following section.

B. Ensuring K -Feasibility

When conjoining two cut sets together using the logical AND operation, we must ensure that all cuts remaining in

the new cut set are K -feasible. We achieve this by modifying the BDD AND operation to remove cubes with more than K literals. This recursive algorithm is illustrated in Fig. 15. Notice that the only difference in this algorithm compared to

```

<  $f_z$  > BddAndRecurPrune( $f_x, f_y, K, n$ )
1  if ISCONSTANT( $f_x$ ) AND ISCONSTANT( $f_y$ )
2      return <  $f_x$  AND  $f_y$  >
3   $b \leftarrow \text{GETTOPVAR}(f_x, f_y)$ 
4   $fn_x \leftarrow f_x(b = 0)$ 
5   $fn_y \leftarrow f_y(b = 0)$ 
6   $fp_x \leftarrow f_x(b = 1)$ 
7   $fp_y \leftarrow f_y(b = 1)$ 
8   $fn_b \leftarrow \text{BDDANDRECURPRUNE}(fn_x, fn_y, K, n)$ 
9  if  $n \leq K$ 
10      $fp_b \leftarrow \text{BDDANDRECURPRUNE}(fp_x, fp_y, K, n + 1)$ 
11 else
12      $fp_b \leftarrow 0$ 
13 SETTOPVAR( $f_z, b$ )
14 SETCOFACTORS( $f_z, fn_b, fp_b$ )
15 return <  $f_z$  >
    
```

 Fig. 15. High-level overview of BDD AND operation with pruning for K .

the recursive definition of a BDD AND operation is the check in line 9. The algorithm starts off by checking the trivial case where both BDD cut sets are constant functions (line 1). If not the trivial case, the top most variable of both cut sets is retrieved (line 3). Next, the cofactors relative to the variable b are found for the cut sets f_x and f_y (line 4-8). This is followed by recursive calls to find the negative and positive cofactors of the new cut set f_z (line 9-12). When constructing the positive cofactor, we make sure that the number of positive edges seen is less than or equal to K (line 9-10). If not, we prune out all cubes that form due to that branch in the BDD. This works since our cut sets, f_x and f_y , only contain positive literals and n is initialized to zero in the first call to BDDANDPRUNERECUR. Thus, we can assume n is equivalent to the size of the cube in the current branch of the BDD. Finally, we join the cofactors and form a new cut set, f_z , and return (line 13-15).

C. Finding the Minimum Cost Cut

In general, the cost of a given cut is usually defined recursively with the form as shown in equation 3.

$$\text{cost}_c = \sum_{u \in \text{fanin}(c)} \text{cost}_{\min}(c_u) \quad (3)$$

In equation 3, u is a fanin of cut c , c_u is the minimum cost cut associated with node u , and $\text{cost}_{\min}(c_u)$ is the cost of the cut c_u . For traditional cut generation methods where subcuts are not shared, each cut has to be traversed independently to determine the minimum cost cut. Conversely, since we represent our cut set as a BDD where we share subcuts, we can leverage dynamic programming to calculate the cut cost and find the minimum cut cost. This is illustrated in the recursive algorithm in Fig. 16. In MINCUTCOSTRECUR, the minimum cost cut, c_{\min} , and its cost, cost , from the cut set f_v is returned. Notice that c_{\min} is returned as a cube where each positive literal in the cube represents a fanin node to the cut. First, if the cut set is trivial ($f_v \equiv 1$), the algorithm returns an

```

<  $c_{min}, cost$  > MinCutCostRecur( $f_v$ )
1  if  $f_v \equiv 1$ 
2    return < 1, 0 >
3  else if  $f_v \equiv 0$ 
4    return <  $\phi, \phi$  >
5  // dynamic programming step
6  if CACHED( $f_v$ )
7    return < LOOKUP( $f_v$ ) >
8   $b \leftarrow \text{TOPVAR}(f_v)$ 
9   $fn_v \leftarrow f_v(b = 0)$ 
10  $fp_v \leftarrow f_v(b = 1)$ 
11 <  $cn_{min}, cost_n$  >  $\leftarrow$  MINCUTCOSTRECUR( $fn_v$ )
12 <  $cp_{min}, cost_p$  >  $\leftarrow$  MINCUTCOSTRECUR( $fp_v$ )
13  $cost_p \leftarrow cost_p + \text{GETNODECOST}(b)$ 
14 if VALID( $cn_{min}$ ) AND VALID( $cp_{min}$ )
15   if  $cost_n < cost_p$ 
16     CACHE( $f_v$ , <  $cn_{min}, cost_n$  >)
17   else
18      $f_x \leftarrow \text{BDDAND}(cp_{min}, b)$ 
19     CACHE( $f_v$ , <  $f_x, cost_p$  >)
20   end else
21 else if VALID( $cp_{min}$ )
22    $f_x \leftarrow \text{BDDAND}(cp_{min}, b)$ 
23   CACHE( $f_v$ , <  $f_x, cost_p$  >)
24 else
25   CACHE( $f_v$ , <  $cn_{min}, cost_n$  >)
26 return < LOOKUP( $f_v$ ) >

```

Fig. 16. Find the minimum cost cut in a given cut set.

empty cube (const 1) with zero cost (line 1-2). If the cut set is empty, an invalid cube is returned (line 3-4). If the cut set is not an empty set, the algorithm checks if this cut set has been visited already, and if so, returns the cached information (line 6-7). If the cut set has not been visited previously, two recursive calls are done to find the minimum cost cut and cost for the cofactors (line 9-12). Next, the positive cofactor cost is modified with the node cost of the current variable (line 13). Finally, the minimum cost cut set and cost are returned (line 14-26). Note that when the minimum cost cut is found, it is cached for future reference (line 15-25).

D. A Practical Application: Elimination

As a practical driver for our cut generation technique, we look at applications that require large cut sizes ($K > 6$). Although we listed several applications requiring cut generation [14], [15], [16] in section I, in this work we only focus on synthesis elimination found in FBDD.

FBDD is a BDD based synthesis engine [24] which has proven to be an order of magnitude faster than SIS with competitive area results. In FBDD several logic transformations, such as decomposition or shared extraction [17], were sped up significantly and as a result, elimination emerged as the primary bottleneck for scalability and has been reported to take up to 70% of the runtime [17]. Removing the elimination bottleneck will further increase the speedup experienced by FBDD. FBDD currently adopts an elimination scheme similar to SIS. In FBDD elimination, regions are grown from a given seed node where its fanins are successively collapsed into the node in a greedy fashion. If the new logic representation simplifies after the collapse operation, the collapse is committed into the netlist, otherwise the collapse is undone.

For BDDs, this collapse and uncollapse operation is relatively slow compared to other BDD operations. A solution to this is to treat elimination as a covering problem, as opposed to a greedy algorithm. Here, each elimination region is created by covering the netlist where each cover has at most 8 inputs. Following this, each cover is collapsed into a single node.

The cost function used to derive our covers is similar to the area flow heuristic described in [11]. However, here we adapt area flow to elimination and rename it as *edge flow*. Edge flow attempts to minimize the total cut size of the final covering (i.e. minimize the number of edges in the final graph). Edge flow is defined recursively in equation 4 and is denoted $ef(\cdot)$. The

$$ef(C_v) = \sum_{e \in \text{fanin}(C_v)} ef(e) \quad (4)$$

$$ef(e) = A_e + \frac{ef(\text{head}(e))}{\|\text{fanout}(C_{\text{head}(e)})\|} \quad (5)$$

edge flow of cover C_v is defined as the sum of the edge flows of the fanin edges to C_v . The edge flow of an edge e is the weight of an edge, A_e , plus the edge flow of its head divided by the number of fanouts of the cover rooted at the head of e . Picking covers which minimize the overall edge flow of the final covering leads to covers which capture a high degree of reconvergence. This will remove any redundancies within the reconvergent cone after the collapse of the cover is done. We will show later that solving elimination this way results in a significant speedup in FBDD with no sacrifice to area.

IV. ICLUSTER: ITERATIVE CLUSTERING

Here, we describe our iterative clustering algorithm. As in [4], we maintain an optimal delay during clustering to improve performance, however, to represent our clustering sets, we propose using zero-suppressed BDDs (ZDDs [2]).

Our clustering algorithm works in two phases. First, an iterative approach is applied where the clustering tool does a forward and backward traversal of the LUT netlist to form an intermediate clustering solution. This is followed by a packing phase to recover area where duplication is reduced and each cluster is filled to its full capacity.

A. Iterative Phase

The iterative phase of our algorithm is responsible for finding an optimal global delay value of our LUT netlist. This will act as a bound when selecting various clusters to cluster the LUT netlist. This is analogous to the iterative step during technology mapping to LUTs. Fig. 17 illustrates

ITERATIVEPHASE

```

1  GENERATECLUSTERS()
2  for  $i \leftarrow 1$  upto  $MaxI$ 
3    TRAVERSEFWD()
4    TRAVERSEBWD()
5  end for

```

Fig. 17. ICluster iterative phase framework.

the entire iterative phase flow. First, all clusters containing N or less LUTs are generated (line 1). During the iterative phase, only clusters that form a cone rooted at a single node are generated (this is not the case during the packing phase following the iterative phase). Following cluster generation, a series of forward and backward traversals are applied (line 3 and 4). The forward traversal is responsible for finding a cluster for each node such that delay and area is minimized and the backward traversal is responsible for finding a final cluster covering of the LUT netlist.

1) *Cluster Generation (GENERATECLUSTERS)*: As stated previously, the iterative phase is responsible for finding an intermediate clustering of the LUT netlist. To do this, we need to generate a large set of clusters that will form the solution space of our intermediate clustering solution. We define a cluster for a LUT, v , as all cones containing at most N LUTs rooted at node v . Although in reality, an FPGA cluster can implement any structure of N LUTs (i.e. not necessarily forming a cone), we ignore these structures in the iterative portion of our algorithm. To generate all cone based clusters, we traverse the netlist in topological order for each node v and generate clusters for v as described in the set relation shown in equation 6.

$$\Phi(v) = \left\{ \bigcup_{i=1..K} \{v\} \times \Phi(u_1) \times \dots \times \Phi(u_j) \mid u_j \in \text{fanin}(v), \|C_v\| \leq N \mid C_v \in \Phi(v) \right\} \quad (6)$$

In equation 6, all clusters for node v are generated by combining the clusters rooted at the fanin nodes of v in every possible way, and discarding all resulting clusters that are larger than size N . $\Phi(v)$ is the set of all cones rooted at node v containing at most N LUTs. K are the number of fanin combinations for node v (e.g. if there are 3 fanins, $K = 6$). C_v represents a cone rooted at fanin node u_j , v is the trivial cone containing only node v , $\Phi(u_j)$ represents the set of cones for fanin u_j , and N is the maximum number of LUTs that can fit into a cluster. The \times operator represents the Cartesian product where taking the Cartesian product of two sets combines all elements within each set in every possible way. Equation 6 combines the elements of all fanin cone sets, $\Phi(u_j)$, in every possible way where all elements with more than N nodes are discarded. These cones form the set of clusters used in the iterative portion of our clusterer. For example, in Fig. 18, $\Phi(v)$ represents our cluster set for node v and it is generated by combining the cluster sets of its fanins. Note that $\Phi(u) \times \Phi(w)$ represents the Cartesian product of sets $\Phi(u)$ and $\Phi(w)$.

$$\Phi(v) = (\{v\} \times \Phi(u)) \cup (\{v\} \times \Phi(w)) \cup (\{v\} \times \Phi(u) \times \Phi(w))$$

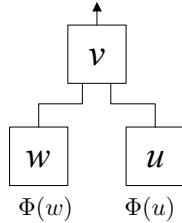


Fig. 18. Cluster set formulation example.

Although in this work, we determine cluster capacity to be the number of distinct LUTs within a single cluster, other constraints such as input constraints (i.e. the number of distinct inputs allowed into a single cluster) can also be added to the formulation with added complexity. Storing clusters explicitly using equation 6 is not scalable due to the exponential number of clusters that need to be stored. As a solution, we propose generating and storing our cluster sets as ZDDs. The benefit of using ZDDs to store clusters is that subclusters can be shared as cofactors within our ZDD. This is similar to the cut generation case where subcuts were shared as cofactors within the BDD.

ZDDs are similar to BDDs, however, nodes whose positive edges point to zero are “suppressed” and will not appear in the graph while nodes whose positive and negative edge point to the same node are kept in the graph (it is recommended that those not familiar with ZDDs should refer to [2]). This makes ZDDs efficient at representing and manipulating sets [2]. For example, consider Fig. 19. To represent the set, Φ , the Boolean function F is formed. Here, F is known as the *characteristic function* for set Φ , where F evaluates to one if the variables are set to a valid element found in set Φ . Set operations on Φ such as the union operator (\cup) can be applied to F using standard Boolean operations. In F , each product term represents a set element where each positive literal represents that the given node exists in that element. Representing F as a ZDD as opposed to a BDD is beneficial since nodes whose positive edges point to zero can be removed in the ZDD. This is clearly shown by looking at Fig. 19 and as a result the BDD representing F is much larger than the ZDD representation of F . As with BDDs, ZDDs can be combined together to form larger ZDDs. This leads to a compact representation of cluster sets.

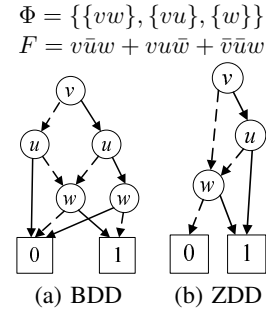


Fig. 19. Representing sets as BDDs versus ZDDs.

2) *Forward Traversal (TRAVERSEFWD)*: After all clusters have been generated, a series of forward and backward traversals of the LUT netlist is done. During the forward traversal, for each node v , a cluster rooted at v is chosen to cluster v and some of its predecessors. For the first traversal, the cluster with minimal delay is chosen where after the entire netlist has been traversed, an optimal delay, $ODelay$, of the circuit is found. A backward traversal follows which produces an intermediate clustering solution and establishes a height for all nodes in the graph. The height, $height(C_v)$, is used in conjunction with the optimal delay in successive forward traversals as a

depth bound when selecting clusters for each node as shown in equation 7. Of the clusters that meet equation 7, the clusters with the lowest area flow are selected.

$$ODelay \geq delay(C_v) + height(C_v) \quad (7)$$

Lemma 4.1: From equation 7, after the forward traversal, all clusters selected for each node will not exceed the optimal delay, $ODelay$, of the entire circuit.

Area flow gives a close estimation of the area found in the final mapped solution. It was first described in [11], and it is described recursively by equations 8 and 9 using the symbol $af(\cdot)$.

$$af(C_v) = A_v + \sum_{e \in fanin(C_v)} af(e) \quad (8)$$

$$af(e) = \frac{af(head(e))}{\|fanoute(head(e))\|} \quad (9)$$

Here, the area flow of a cluster rooted at node v is the sum of the area flow of the fanins to cluster C_v plus the area associated with the cluster C_v denoted as A_v . The area flow of a fanin edge is the area of the head node of the edge divided evenly with its fanouts.

3) *Backward Traversal (TRAVERSEBWD)*: The backward traversal is responsible for finding an intermediate clustering of the LUT netlist. First, each PO is marked as visible. Then, for each visible node, the minimum cost cluster found in the preceding forward traversal is used to cluster that node and some of its predecessors. The fanins of the minimum cost clusters are marked as visible and the process continues until the PIs are reached. This process is illustrated in Fig. 20 where the primary outputs of the circuit are located at the top of the picture.

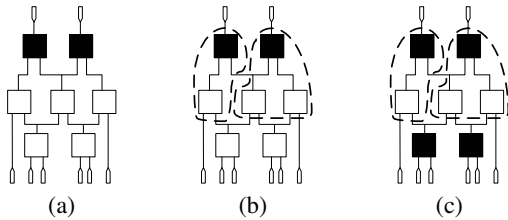


Fig. 20. Illustration of the steps taken during the backward traversal. (a) LUTs feeding the primary outputs are marked as visible. (b) Cluster found in previous forward traversal are used to cover visible nodes and some of its predecessors. (c) Fanins feeding clusters selected previously are marked as visible.

The backward traversal is also responsible for updating the internal heights of each node. This is done by looking at all the fanouts of a given node, v , and setting its height to the largest height of its fanout edges. This is shown in Fig. 21 where $delay(e_1)$ represents an inter-cluster delay and $delay(e_2)$ represents an intra-cluster delay.

After an intermediate clustering solution is found, the root node of all clusters are checked to see if they can be merged into existing clusters to save area. Clusters can be merged if the merge will not increase the delay of the critical path. For example, consider Fig. 22. In Fig. 22a, two clusters are shown, where the root node with height h_3 and some of its

$$h_3 = \max\{h_1 + delay(e_1), h_2 + delay(e_2)\}$$

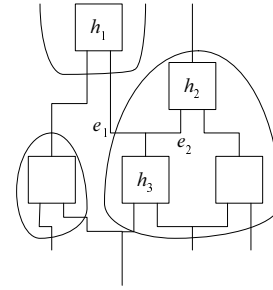


Fig. 21. Updating height of node v to h_3 .

successors must be duplicated. However, if h_3 is greater than $h_2 + delay(e_2)$, the node with height h_3 has some slack and the two clusters can be merged together as in Fig. 22b. Merging clusters is possible since clusters have multiple outputs to support the output of all LUTs found within the cluster.

Lemma 4.2: Merging clusters is guaranteed to not violate the optimal delay constraint.

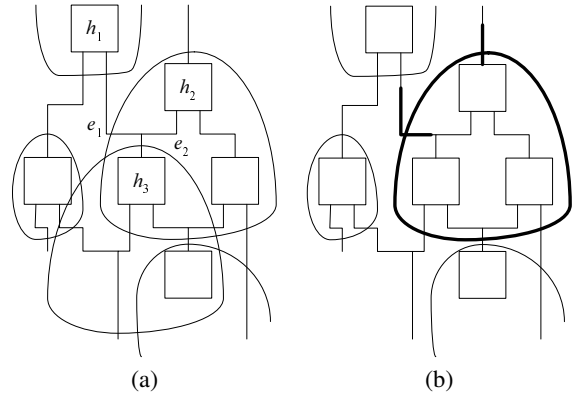


Fig. 22. Leveraging the multi-outputs of a given cluster by merging clusters.

Theorem 4.3: After the backward traversal, a clustered circuit will be returned with an optimal delay, $ODelay$.

Proof: In the forward traversal, all clusters created will have a delay less than or equal to $ODelay$ from lemma 4.1. Since the backward traversal can only select clusters created from the forward traversal, no cluster will have a delay greater than $ODelay$. Furthermore, since merging clusters will not increase the delay of a cluster beyond $ODelay$ by lemma 4.2, all clusters will obey the delay of $ODelay$. Thus, the final clustered circuit will have a delay of $ODelay$. ■

B. Packing Phase

Since the iterative flow described previously will lead to much unnecessary duplication and will not necessarily fill all clusters to their full capacity, a packing phase is applied to the intermediate clustered netlist. The packing phase will break the previous assumption that all clusters will form a cone by packing clusters in various ways. The packing heuristics are very similar to those described in [4] and consist of:

- Pack Fanin - Pack a fanin cluster with its fanout cluster. Fanins found on the critical path are clustered before other fanins.
- Pack Duplicated - Pack clusters which share the same LUTs. This reduces the number of duplicated LUTs in the final netlist.
- Pack General - Do a bin packing of unfilled clusters.

The conjunction of these packing heuristics dramatically reduces the area overhead of our iterative clusterer.

V. RESULTS

Here, we evaluate our cut generation technique where we create a tool called **BddCut** followed by an evaluation of our iterative clustering tool called **ICluster**.

To evaluate our cut generation technique, we look at two aspects. Since **BddCut** can be plugged into any iterative technology mapper to generate cuts and achieve exactly the same area and delay, our first evaluation focuses on its scalability against two representative, state-of-the-art mappers: **IMap**, one of the earliest mappers to use an iterative strategy; and **ABC**, the most recently reported iterative mapper that employs a scalable cut generation algorithm. Our second evaluation attempts to measure the benefits of the proposed method under the context of a complete logic synthesis flow. To this end, we embed **BddCut** as a replacement of the elimination procedure in **FBDD**, and evaluate its impact on runtime and area.

Following this, we look at the scalability and performance impact of our iterative clustering approach. Here we compare **ICluster** against the traditional VPR flow [25] which utilizes **t-vpack** [23] to cluster its LUTs. All of our experiments were run on a Pentium D 3.2 GHz machine with 2GB of RAM. We used the Somenzi’s CUDD BDD package [8] and applied our algorithms to the MCNC [26] and IWLS [27] benchmark (includes **ISCAS89**, **ITC**, and several large circuits) suite.

A. BddCut: Scalable Cut Generation

To investigate our symbolic approach to cut generation, we compare the cut generation time of **BddCut** against **IMap**’s [11] and **ABC**’s [3] cut generation time. Note that all technology mappers were set to generate all possible cuts (i.e. no pruning) and there was no sacrifice to solution quality, hence final mapping results are omitted. Table I shows detailed results for select circuits, followed by Table II and III with summarized results for the entire **ITC** and **ISCAS89** benchmark suite. We also show a peak memory use comparison between **ABC** and **BddCut** as shown in Table IV. Finally, we compared **BddCut** with **ABC** for one of the largest IWLS circuits which is shown in Table V. In cases that the technology mapper ran out of memory, the circuit time is marked as **n/a**.

The results in the previous table clearly indicate that due to subcut sharing and redundant cut removal, our symbolic approach scales better than traditional techniques where **IMap** is more than an order of magnitude slower. When compared against **ABC**, our technique scales much better where our average speedup and relative

TABLE II
AVERAGE RATIO OF $\frac{IMap}{BddCut}$ CUT GENERATION TIMES. IMap COULD NOT BE RUN FOR $K \geq 8$.

Benchmark	K=6	K=7
ITC	27.8x	46.5x
ISCAS89	12.2x	26.5x

TABLE III
AVERAGE RATIO OF $\frac{ABC}{BddCut}$ CUT GENERATION TIMES.

Benchmark	K=6	K=7	K=8	K=9	K=10
ITC	0.512x	1.07x	1.77x	4.25x	11.2x
ISCAS89	0.781x	1.08x	1.59x	2.39x	4.87x

TABLE IV
AVERAGE RATIO OF $\frac{ABC}{BddCut}$ MEMORY USAGE.

Benchmark	K=6	K=7	K=8	K=9	K=10
ITC	0.22x	0.60x	1.58x	2.12x	6.13x
ISCAS89	0.14x	0.24x	0.73x	1.55x	2.71x

TABLE V
RUNTIME COMPARISON OF **BddCut** WITH **ABC** ON CIRCUIT **leon2** (CONTAINS 278,292 4-LUTS).

Cut Size	runtime (sec)		memory (GB)	
	BddCut	ABC	BddCut	ABC
6	23.3	77.9	0.43	1.41
7	58.9	n/a	0.45	n/a
8	152.9	n/a	0.60	n/a
9	547.6	n/a	0.81	n/a

reduction in memory use improves as K gets larger. Unlike runtime, the improvement in memory does not occur until K gets large ($K > 7$). We found that because there is a fixed overhead in the CUDD BDD manager, and therefore only at the larger cut sizes was this overhead amortized through subcut sharing or if the benchmark is large as in the case of **leon2** shown in Table V. Without subcut sharing, memory runs out for a few of the larger benchmark circuits when $K = 10$. This is also true for extremely large benchmark circuits as shown in Table V where **ABC** runs out of memory in circuit **leon2** for $K > 6$. Fortunately, **ABC** supports cut dropping which has proven to reduce the memory usage by several fold, but, from our experience, cut dropping increases the cut computation time so we did not turn on this feature. For example, with cut dropping enabled, **ABC** took more than 12 hours to generate 10-input cuts for circuit **b20**, whereas **BddCut** takes less than 15 minutes.

Although **ABC** outperforms **BddCut** for small cut sizes, the longest 6-input cut generation time in **BddCut** was 2.8 seconds. For small cut sizes, the overhead in storing and generating BDDs is not amortized when generating cut sets symbolically, thus **ABC** is still the better approach for smaller values of K . The exception to this trend occurs for circuits with a high degree of reconvergence such as for circuit **C6288** (**C6288** is a multiplier). For these circuits, our relative speedup is much larger for all values of K because reconvergent paths dramat-

TABLE I

DETAILED COMPARISON OF BDDCUT CUT GENERATION TIME AGAINST IMAP AND ABC. IMAP COULD NOT RUN FOR $K \geq 8$.

Circuit	K=6 (sec)			K=7 (sec)			K=8 (sec)		K=9 (sec)		K=10 (sec)	
	BddCut	IMap	ABC	BddCut	IMap	ABC	BddCut	ABC	BddCut	ABC	BddCut	ABC
C6288	0.20	40.64	0.52	0.67	660.76	5.66	2.48	14.49	9.91	150.13	41.86	1758.44
des	0.36	10.46	0.19	0.70	294.05	3.34	9.05	10.70	74.66	105.16	828.44	1126.50
i10	0.22	14.27	0.25	1.58	98.27	2.00	2.83	6.06	11.41	57.17	50.78	581.09
b20	1.84	81.89	0.88	8.27	890.67	8.69	42.01	73.53	200.27	889.92	895.63	n/a
b21	1.91	86.84	0.94	8.59	929.90	8.66	44.03	80.34	205.25	942.88	920.22	n/a
b22.1	2.17	107.16	1.38	8.81	n/a	10.3	41.22	84.36	180.58	924.38	766.63	n/a
s15850.1	0.11	3.96	0.13	0.33	38.32	0.75	1.08	7.61	4.11	16.69	17.94	192.72
s38417	0.45	13.68	0.31	1.39	133.83	0.72	4.31	6.19	14.19	58.09	47.97	536.84
s4863	0.11	19.27	0.11	0.36	269.07	0.84	1.45	4.99	6.53	50.66	30.77	555.59
s6669	0.11	15.73	0.09	0.33	197.76	0.63	1.20	3.53	5.88	32.63	31.61	295.38
Ratio Geomean		63x	0.83		225x	1.8x		2.5x		4.9x		10x

ically increase the number of cut duplications in conventional cut generation methods.

A concern one could raise with our symbolic approach is the effect of BDD representation of cuts on the cache. Since the CUDD package represents BDDs as a set of pointers, the nodes in each BDD may potentially be scattered throughout memory. Thus, any BDD traversal would lead to cache thrashing, which would dramatically hurt the performance of our algorithm. However, CUDD allocates BDD nodes from a continuous memory pool leading to BDDs that exhibit good spatial locality. Our competitive results support this claim and indicate that good cache behaviour is maintained with CUDD.

1) *Elimination: A Practical Application:* After ensuring our symbolic cut generation approach was scalable, we applied our cut generation to elimination and evaluated our elimination scheme against greedy based elimination schemes. To compare the two approaches, we replaced the folded elimination step in FBDD with our covering-based elimination algorithm and compared both the area and runtime of the original FBDD flow against our new flow. Logic folding exploits the inherent regularity of logic circuits by sharing transformations between equivalent logic structures. This has a huge impact on runtime where it has been shown to reduce the number of elimination operations by 60% on average. Thus, comparing against the folded version of elimination has much more value. We also compare against SIS for a common reference point. For ease of readability, we will refer to our flow which uses covering-based elimination as $FBDD_{new}$. Starting with un-optimized benchmark circuits, we optimized the circuits with $FBDD_{new}$, FBDD, and SIS. To compare their area results, we technology mapped our optimized circuits to two technologies: the SIS standard cell library (*map*) [18] and 4-LUTs using the technology mapping algorithm described in [11]. When optimizing the circuits in SIS, we used *script.rugged* [18]. Table VI illustrates detailed results for a few benchmark circuits. Column *Circuit* lists the circuit name, column *Time* lists the total runtime in seconds, column *Std Cell* lists the standard cell area when mapped to SIS' default standard cell library, and column *4-LUT* lists the 4-LUT count. Note a few circuits caused SIS to run out of memory and are marked as n/a. The final row lists the geometric mean of the ratio when compared against $FBDD_{new}$.

For the circuits shown in Table VI, our new flow is

significantly faster than the original FBDD with an average speedup of over 5x and an order of magnitude speedup over SIS. The results also show that this speedup comes with no area penalty.

We also explored the effect of the maximum cut size used in our elimination algorithm on runtime and area where we varied the cut size from 4 to 10. This is shown in Table VII where we applied our new flow to the entire ITC, ISCAS89, and select IWLS benchmarks and take the geometric mean ratio of the FBDD result over $FBDD_{new}$. Column *K* lists the cut size used in $FBDD_{new}$ when generating resynthesis regions, column *Time* is the time ratio, column *Std Cell* is the final standard cell area ratio, and column *4-LUT* is the final 4-LUT area ratio. Each ratio column is given a benchmark heading indicating the benchmark suite used. As Table VII shows, it appears that using a cut size of 4 or 6 has a substantial speedup of more than 10x in many cases; however, this comes with an area penalty, particularly in the IWLS benchmarks. This implies that the elimination regions created with these cut sizes are too small and does not capture large enough resynthesis regions in a single cone. In contrast, a cut size of 8 still maintains a significant average speedup of more than 6x for all benchmarks with negligible impact on the final area when compared to the original FBDD.

B. ICluster: Iterative Clustering

A runtime, area overhead, and placed and route delay comparison between ICluster against the traditional VPR flow is shown in Table VIII and Table IX. Table VIII shows detailed results for a few select circuits and Table IX shows a summarized result for the ITC and MCNC benchmark suite and also the benchmark circuits used in [4] labeled SMAC. T-vpack is used in the traditional VPR flow to cluster LUTs where both t-vpack and VPR are set to timing driven mode and all circuits are optimized and technology mapped to 4-LUTs with ABC prior to clustering [3]. The clustering architecture we used contained 10 LUTs per cluster. When comparing the results, the results of the ICluster-VPR flow are divided by the results of the traditional tvpac-VPR flow. The results show that our technique is approximately 50% slower with a 15% area overhead and an improvement of 4-8% place and route delay on average. In [4], the authors report a 23% area overhead with a 12.3% improvement in place and

TABLE VI
DETAILED COMPARISON OF AREA AND RUNTIME OF $FBDD_{new}$ AGAINST FBDD AND SIS FOR $K = 8$.

Circuit	Time (sec)			Std Cell Area			4-LUT Area		
	$FBDD_{new}$	FBDD	SIS	$FBDD_{new}$	FBDD	SIS	$FBDD_{new}$	FBDD	SIS
s38417	1.9	7.2	58.0	15992	15711	18617	3560	3559	4052
s38584	3.0	13.7	3927.3	17388	17783	16846	4289	4152	4174
s35932	3.9	4.1	n/a	18630	17806	n/a	3264	3360	n/a
s15850	0.8	9.1	68.8	5707	5605	5735	1282	1270	1329
b20	5.5	44.8	154.5	20280	20002	20776	4514	4324	4773
b22_1	6.2	38.4	202.4	26402	29725	25265	5788	6505	5664
b17	8.9	102.8	583.1	44355	41115	46701	10722	9896	11574
systemcdes	3.1	11.3	123.1	5582	5683	5276	1152	1207	1143
vga_lcd	38.9	585.2	n/a	18435	178033	n/a	40680	40676	n/a
wb_conmax	18.6	104.2	1313.5	76719	82514	77329	19135	19479	19726
Ratio Geomean		5.7x	70x		1.00	1.01		1.00	1.03

TABLE VII
COMPARISON OF AREA AND RUNTIME OF FBDD AGAINST $FBDD_{new}$ FOR VARIOUS VALUES OF K .

K	ITC Ratios			ISCAS89 Ratios			IWLS Ratios		
	Time	Std Cell	4-LUT	Time	Std Cell	4-LUT	Time	Std Cell	4-LUT
4	12.4x	0.978	1.001	11.9x	0.975	0.982	12.8x	0.964	0.913
6	8.76x	1.00	1.00	9.26x	0.965	0.984	8.72x	0.950	0.921
8	6.16x	0.995	1.00	6.24x	0.994	0.987	6.84x	0.968	0.971
10	2.55x	1.02	0.991	2.62x	0.987	0.984	2.76x	0.966	0.964

TABLE VIII
DETAILED COMPARISON OF RUNTIME, AREA, AND DELAY OF ICLUSTER-VPR FLOW AGAINST THE TVPACK-VPR FLOW.

Circuit	t-vpack		Icluster	
	area # CLBs	P&R delay (ns)	area # CLBs	P&R delay (ns)
b14	172	69.9	211	64.3
b15	320	96.2	374	79.8
des	146	33.1	214	32.3
i9	27	22.4	29	19.9
too_large	21	27.9	23	25.3
C3540	37	42.7	49	37.4

TABLE IX
AVERAGE COMPARISON OF RUNTIME, AREA, AND DELAY OF ICLUSTER-VPR FLOW AGAINST TVPACK-VPR FLOW.

Benchmark	Time	area	P&R delay
ITC	0.43	1.14	0.92
MCNC	0.67	1.13	0.96
SMAC	0.56	1.15	0.93

route delay. However, their technique is 100x slower than the traditional clustering flow. Our original hypothesis was proven here where we showed that maintaining global information during clustering can improve the circuit performance. By solving clustering disjointly from technology mapping and using ZDDs to represent our cluster sets, our performance improvement was achieved with a significantly smaller runtime impact than [4]. Though, we recognize that the improvement is not as dramatic as the numbers reported in [4]. However, our approach does not have the 100x runtime penalty reported in [4] which is a limitation for its practical application on large designs with 100K LUTs or more.

VI. CONCLUSION

We introduced a novel BDD-based reduction technique for problems important to CAD including cut generation and clustering of LUTs. Prior to our work, these algorithms were facing scalability issues particularly when the circuit size or the problem parameters were large. The primary benefit of our approach is its generality. For example, there has been extensive research done on pruning the solution space of cut generation to reduce its runtime and memory use [9], [12]. Interestingly, all these techniques are orthogonal to our work. Since the benefit of our approach is how we represent our cut sets, we can apply the same pruning techniques used in previous work on top of our BDD cut generation approach to get a further speedup.

Our approach, however, does come with limitations. In section III-C, we illustrated an algorithm to evaluate our cut sets using dynamic programming. It should be emphasized that this can only work when the cost of each cut is independent of elements found within the cone of logic created by the cut. For example, when mapping to generalized PLBs, the function of each cut needs to be evaluated [14]. Thus, in such cases, a dynamic programming approach to the cut set traversal will not be applicable. In this case, the cuts must be traversed individually which is the case in current cut generation algorithms. Furthermore, cut generation can only be applied to single output cones. As future work, we would like to investigate possible means to generate multi-output cones efficiently. This would be valuable for many logic transformations applicable to multi-output functions.

Overall, we have shown a general technique that leverages BDDs to represent sets. We have shown that this idea is general enough to be applied to cut generation and clustering where we proved this leads to an order of magnitude speedup on

average.

REFERENCES

- [1] F. Somenzi, "Binary decision diagrams," pp. 303–366, 1999. [Online]. Available: citeseer.ist.psu.edu/somenzi99binary.html
- [2] S. Minato, "Zero-suppressed bdds for set manipulation in combinatorial problems," in *Design Automation Conference*. New York, NY, USA: ACM Press, 1993, pp. 272–277.
- [3] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs," in *Field-Programmable Gate Arrays*. ACM Press, 2006.
- [4] J. Y. Lin, D. Chen, and J. Cong, "Optimal simultaneous mapping and clustering for fpga delay optimization," in *Design Automation Conference*. New York, NY, USA: ACM Press, 2006, pp. 472–477.
- [5] S. Areibi and A. Vannelli, "Advanced search techniques for circuit partitioning," in *Quadratic Assignment and Related Problems*, P. Pardalos and H. Wolkowicz, Eds. AMS, 1994, vol. 16, pp. 77–96. [Online]. Available: citeseer.ist.psu.edu/areibi94advanced.html
- [6] —, "An efficient clustering technique for circuit partitioning," 1996. [Online]. Available: citeseer.ist.psu.edu/areibi96efficient.html
- [7] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986. [Online]. Available: citeseer.ist.psu.edu/bryant86graphbased.html
- [8] F. Somenzi, "CUDD: CU decision diagram package release," 1998.
- [9] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: enabling a general and efficient FPGA mapping solution," in *Field-Programmable Gate Arrays*. ACM Press, 1999, pp. 29–35.
- [10] D. Chen and J. Cong, "DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs," in *International Conference on Computer-Aided Design*, Washington, DC, USA, 2004, pp. 752–759.
- [11] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," *IEEE Journal on Technology in Computer Aided Design*, vol. 25, pp. 2331–2340, Nov. 2006.
- [12] S. Chatterjee, A. Mishchenko, and R. Brayton, "Factor cuts," in *International Conference on Computer-Aided Design*, 2006. [Online]. Available: <http://www.eecs.berkeley.edu/~alanmi/abc/>
- [13] J. Cong and Y.-Y. Hwang, "Boolean matching for complex PLBs in LUT-based FPGAs with application to architecture evaluation," in *Field-Programmable Gate Arrays*, 1998, pp. 27–34. [Online]. Available: citeseer.ist.psu.edu/cong98boolean.html
- [14] A. C. Ling, D. P. Singh, and S. D. Brown, "FPGA PLB evaluation using quantified boolean satisfiability," in *Field-Programmable Logic and Applications*, Aug. 2005, pp. 19–24.
- [15] —, "FPGA technology mapping: a study of optimality," in *Design Automation Conference*. New York, NY, USA: ACM Press, 2005, pp. 427–432.
- [16] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG Rewriting: A fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–536. [Online]. Available: <http://www.eecs.berkeley.edu/~alanmi/abc/>
- [17] D. Wu and J. Zhu, "FBDD: A folded logic synthesis system," in *International Conference on ASIC*, Shanghai, China, Oct. 2005.
- [18] 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," Electrical Engineering and Computer Sciences, University of California, Berkeley, Tech. Rep., 1992. [Online]. Available: citeseer.ist.psu.edu/sentovich92sis.html
- [19] J. Cong and Y. Ding, "On area/depth trade-off in LUT-based FPGA technology mapping," in *Design Automation Conference*, 1993, pp. 213–218. [Online]. Available: citeseer.ist.psu.edu/article/cong94areadepth.html
- [20] —, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Journal on Technology in Computer Aided Design*, vol. 13, no. 1, pp. 1–13, Jan. 1994.
- [21] J. Cong and Y.-Y. Hwang, "Simultaneous depth and area minimization in LUT-based FPGA mapping," in *Field-Programmable Gate Arrays*, 1995, pp. 68–74.
- [22] Altera Corporation, *Stratix II Device Handbook*, Oct. 2004.
- [23] A. S. Marquardt, V. Betz, and J. Rose, "Using cluster-based logic blocks and timing-driven packing to improve fpga speed and density," in *Field-Programmable Gate Arrays*. New York, NY, USA: ACM Press, 1999, pp. 37–46.
- [24] C. Yang, M. J. Ciesielski, and V. Singhal, "BDS: a BDD-based logic optimization system," in *Design Automation Conference*, 2000, pp. 92–97.
- [25] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Field-Programmable Logic and Applications*, W. Luk, P. Y. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, 1997, pp. 213–222. [Online]. Available: citeseer.ist.psu.edu/betz97vpr.html
- [26] S. Yang, "Logic synthesis and optimization benchmarks user guide version," 1991.
- [27] "IWLS 2005 Benchmarks." [Online]. Available: <http://iwls.org/iwls2005/benchmarks.html>