BddCut: Towards Scalable Symbolic Cut Enumeration

Andrew C. Ling
Department Electrical and Computer Engineering
University of Toronto
Toronto, Canada
e-mail: aling@eecg.toronto.edu

Jianwen Zhu
Department Electrical and Computer Engineering
University of Toronto
Toronto, Canada
e-mail: jzhu@eecg.toronto.edu

Stephen D. Brown
Altera Corporation
Toronto Technology Centre
Toronto, Canada
e-mail: sbrown@altera.com

Abstract— While the covering algorithm has been perfected recently by the iterative approaches, such as DAOmap and IMap, its application has been limited to technology mapping. The main factor preventing the covering problem's migration to other logic transformations, such as elimination and resynthesis region identification found in SIS and FBDD, is the exponential number of alternative cuts that have to be evaluated. Traditional methods of cut generation do not scale beyond a cut size of 6. In this paper, a symbolic method that can enumerate all cuts is proposed without any pruning, up to a cut size of 10. We show that it can outperform traditional methods by an order of magnitude and, as a result, scales to 100K gate benchmarks. As a practical driver, the covering problem applied to elimination is shown where it can not only produce competitive area, but also provide more than 6x average runtime reduction of the total runtime in FBDD, a BDD based logic synthesis tool with a reported order of magnitude faster runtime than SIS and commercial tools with negligible impact on area.

I. Introduction

In CAD, the network covering problem has been successfully leveraged by technology mapping for K-LUTs to produce extremely good solutions in terms of area and delay [1, 2]. The covering problem attempts to find a set of cones to cover a network such that a given optimization goal is satisfied. For example, when applied to K-LUT technology mapping, the covering problem attempts to minimize the number of cones in its solution to reduce area of the LUT network. This process is illustrated in Fig. 1.

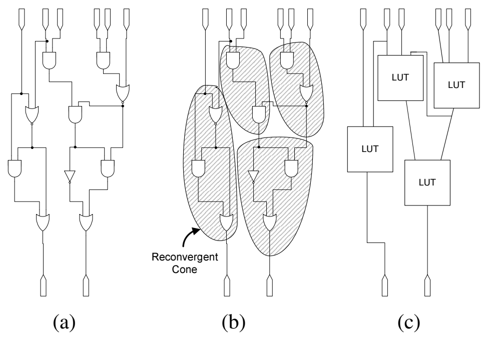

Fig. 1. Treating technology mapping as a covering problem. (a) Initial network. (b) Possible covering. (c) Final mapping to LUTs.

One important step during the covering problem is the generation of all cuts in the network that are used to derive a set of cones to cover the network. The cut generation step is the primary bottleneck of the covering problem and has limited the application of the covering problem framework to technology mapping for K-LUTs with small values of K (4 or 5), even though many other problems in CAD can be represented as a covering problem. One such problem is *elimination* [3, 4], which is found in gate-level synthesis.

Elimination [3, 4] is one of the first area optimizations in the synthesis flow that collapses redundant nodes into their transitive fanouts. Nodes are considered redundant if their removal through resynthesis operations does not change the functionality of the circuit. Elimination also defines regions for succeeding logic optimizations to be applied on since collapsed nodes form large resynthesis regions. Current methods for elimination rely on trial-and-error to define resynthesis regions and, hence, will not scale as circuits reach and surpass the 100K gate mark. In particular, in a binary decision diagram (BDD) based synthesis engine called FBDD, where logic transformations were sped up significantly [4], elimination emerged as the primary bottleneck for scalability and has been reported to take up to 70% of the runtime [4]. Being able to solve elimination as a covering problem would treat elimination as a global optimization problem rather than a greedy based heuristic and, more importantly, dramatically reduce the runtime of the elimination step. However, since each elimination region is relatively large (8 or more inputs), migrating the covering problem to elimination is not feasible due to the exponentially large number of cuts that need to be generated and stored. Thus, for applications that require large values of K, traditional methods for cut generation cannot be used.

As a solution, we propose a novel scalable symbolic cut generation method using BDDs that, unlike previous methods, scales to cut sizes of up to 10 without the need of pruning. The primary benefits of our symbolic approach are summarized as follows:

- Subcuts are shared between larger cuts and do not need to be duplicated in different cut sets. This dramatically reduces time to produce cuts and the storage requirements to hold cut sets.
- Subcut sharing allows cuts to be evaluated simultaneously. This increases efficiency by removing the need to evaluate all cuts independently.
- Redundant cuts are automatically removed from the cut set which further reduces the complexity to generate cuts.

We will show that our symbolic cut generation method is more than 20x faster than traditional cut generation techniques. Also, we will show that our approach scales better than current cut generation methods found in ABC, the fastest technology mapper reported recently. As a result, we can generate large cut sizes sizes up to 10 and can successfully apply the covering problem to elimination in FBDD: a BDD based synthesis engine that has been shown to produce an order of magnitude speedup over SIS [4] with little impact to area. As a consequence of replacing elimination in FBDD with our cover-based elimination algorithm, we get an average 6x speedup in total runtime with no penalty to area when technology mapped to standard cells or 4-LUTs.

The rest of the paper is organized as follows: section II discusses the cut generation in more detail along with previous work; section III describes our symbolic cut generation method; section IV describes our cover-based elimination; section V shows the results of our approach; and section VI concludes with a brief summary of our work and future directions.

II. Background and Previous Work

A. Terminology

A circuit, as a DAG $G = (V, E)$, represents functions, primary inputs and outputs (PIs, POs) as nodes, $u \in G(V)$. Each directed edge, $e \in G(E)$, with head, $u = head(e)$, and tail $v = tail(e)$, represents a signal output from node v and entering node u. A *cone*, C_v, rooted at node v is a subgraph in a circuit where all nodes, u, in C_v have a path from u to v. Additionally, if C_v is found in the final cover of a mapping solution, the root node v is known to be *visible*. For example, in Fig. 1b, the bottom right cover forms a cone and the OR-gate is the visible root node of the cone. The fanins of a cone (node) are the set of nodes feeding the cone (node) and fanouts of a cone (node) are the nodes fed by the cone (node). PIs are nodes with no fanins and POs are nodes with no fanouts. A fanout free cone (FFC), C_v, is a cone that has a fanout only at the root node v, such as the cones in Fig. 1b. A maximum FFC (MFFC) of node v is the largest possible FFC rooted at v. The *cut* of a cone C_v is the set of cone fanin nodes, $u \in fanin(C_v)$, and the cut size, $\|fanin(C_v)\|$, of a cone is known as the number of distinct nodes feeding the cone. For example, looking at the bottom right cone in Fig. 1b, the nodes feeding the NOT-gate and AND-gate are the cone fanins and also form the cut for the cone with a cut size of 2. A cone is derived from a cut by taking the subgraph rooted at a single node whose fanin nodes are identical to the cut nodes. A cone (cut) is thought as *K-feasible* if it has K or less distinct fanin nodes (cut nodes). Traversing a graph in *topological order* implies a node's fanins will be visited before itself.

B. Cut Generation

One of the first works to use cut generation was presented in [5]. Here, 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 [5]. This contrasts with incremental cut generation methods based on network flow [6, 7] and has proven to be much faster.

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

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 ($\|c_u * c_w\| \leq K$). 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 [2], which utilizes a popular technology mapping framework, cut generation takes more than 99% of the runtime for $K = 7$. In [8], 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.

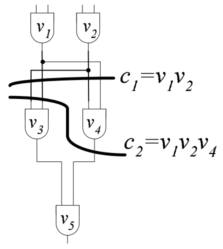

Fig. 2. Example of two cuts in a netlist where c_2 dominates c_1.

A side effect of Equation 1 is the generation of 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. 2 illustrates this relation. These cuts can be removed because they will not affect the final quality of a mapping solution. In ABC [9], 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 [10], migrating the covering problem to elimination requires a more scalable cut generation solution.

C. FBDD

As stated previously, the primary motivation of solving elimination as a covering problem is to remove the elimination bottleneck experienced by FBDD [4]. FBDD is BDD based synthesis engine [11] which has proven to be an order of magnitude faster than SIS with competitive area results. 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. This greedy approach to elimination in FBDD is very slow and as a result, elimination in FBDD takes up more than 70% of the runtime. As we show later, we solve this problem by treating elimination as a covering problem which results in a significant speedup in FBDD with no sacrifice to area.

III. Scalable Symbolic Cut Generation

Before we can treat elimination in FBDD as a covering problem, a scalable cut generation approach is required. Traditional methods for cut generation cannot be used as they do not scale to cut sizes of 6 or more without pruning [8]. We want to avoid pruning since this may remove valuable cuts, particularly when K becomes large (8 to 10). As described in Equation 1, cuts are generated by concatenating subcuts in every possible way. This is extremely inefficient since subcuts are duplicated every time they are used to generate a new cut. Our symbolic 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 in symbolic form.

$$f_v = \Pi_{u \in fanin(v)}(u + f_u) \tag{2}$$

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, f_v. 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 use the Π operation that can be thought as the logical AND of all clauses $(u + f_u)$. Here, f_u is the Boolean function cut set representation for fanin node u, and u is the trivial cut. For example, consider Fig. 3a. Here, each node is represented by a Boolean variable. Also, notice that the cut set f_a is the conjunction between the clauses $(c + f_c)$ and $(b + f_b)$.

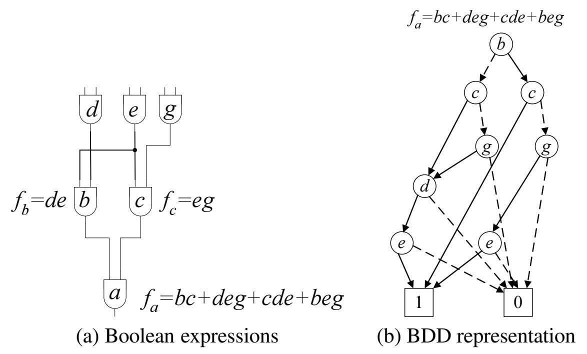

Fig. 3. Symbolic representation of cut sets.

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 reduced order binary decision diagram, which we will simply refer as a BDD for convenience (for a detailed description of the BDD data structure, please refer to [12, 13]). 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. 3a, the BDD used to represent the cut set f_a is shown in Fig. 3b. 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. 4. Notice that in the

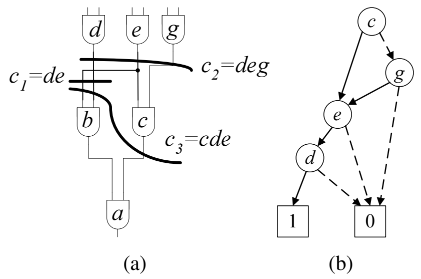

Fig. 4. BDD representation of cuts c_1, c_2, and c_3.

BDD 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 $c_2 = deg$. Thus, instead of requiring 8 BDD nodes to store cuts c_1 to c_3, only 4 BDD nodes are required.

Another benefit of using BDDs is that redundant cuts, such as dominator cuts, are automatically removed. For example, consider Fig. 5a containing the cut c_1 and the dominator cut c_2. As a BDD, c_1 and c_2 are shown in Fig. 5b. Since BDD node c is now redundant, it can be removed as in Fig. 5c which removes the dominator cut c_2. This example illustrates how redundant cuts are automatically removed in BDD representations. This, along with the subcut sharing, substantially reduces the runtime and storage requirements of cut generation.

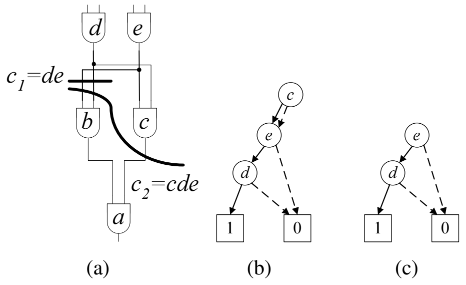

Fig. 5. BDD representation of cuts c_1 and c_3.

A. Symbolic Cut Generation Algorithm

Fig. 6 illustrates our cut generation algorithm. First, the 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 (empties the cut set) and is assigned a unique variable for cut representation (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, $u = fanin(v)$, 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, ($f_x = (u+f_u)$, where f_x, u, and f_u are represented as BDDs). Next, the temporary cut set is conjoined to the cut set of v using the logical AND operation (line 9, $f_v = f_v \cdot f_x$.). This merges the cut sets of all the fanin nodes to form new cuts. 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.

```
**CutGeneration()**
1   $G \leftarrow$ SORT$_t$()
2   **foreach** $v \in G$
3       $f_v \leftarrow 1$
4       $b_v \leftarrow$ CREATENEWBDDVARIABLE()
5   **end foreach**
6   **foreach** $v \in G$
7       **foreach** $u \in fanin(v)$
8           $f_x \leftarrow$ BDDOR($b_u, f_u$)
9           $f_v \leftarrow$ BDDANDPRUNE($f_v, f_x, K$)
10      **end foreach**
11  **end foreach**
```

Fig. 6. High-level overview of symbolic cut generation algorithm.

B. Ensuring K-Feasibility and Finding Minimum Cost Cut

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. 7. Notice that

```
$\langle f_z \rangle$ **BDDANDPRUNE**($f_x, f_y, K, n$)
1   **if** ISCONSTANT($f_x$) **AND** ISCONSTANT($f_y$)
2       **return** $\langle f_x$ **AND** $f_y \rangle$
3   $b \leftarrow$ GETTOPVAR($f_x, f_y$)
4   $fn_b \leftarrow$ BDDANDPRUNE($f_x(b=0), f_y(b=0), K, n$)
5   **if** $n \leq K$
6       $fp_b \leftarrow$ BDDANDPRUNE($f_x(b=1), f_y(b=1), K, n+1$)
7   **else**
8       $fp_b \leftarrow 0$
9   **return** $\langle$CREATEBDD($b, fn_b, fp_b$)$\rangle$
```

Fig. 7. High-level overview of BDD AND operation with pruning for K.

the only difference in this algorithm compared to the recursive definition of a BDD AND operation is the check in line 5. It is recommended that those not experienced with BDD operations please refer to [12, 13]. 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). This is followed by recursive calls to find the negative and positive cofactors of the new cut set f_z (line 4-6). When constructing the positive cofactor, we make sure that the number of positive edges seen is less than or equal to K (line 5-8). 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 BDDANDPRUNE. 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 and return (line 9).

Fig. 8 is a simplified algorithm to find the minimum cost cut from our BDD cut set. 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. Lines 1-4 are the trivial cases when a trivial cut (line 1) or invalid cut (line 3) is encountered. If the cut set is not an empty set, the algorithm

```
$< c_{min}, cost >$ **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   **if** VISITED($f_v$)
6       **return** $< f_x, cost >$
7   $b \leftarrow$ TOPVAR($f_v$)
8   $< cn_{min}, cost_n > \leftarrow$ MINCUTCOSTRECUR($f_v(b=0)$)
9   $< cp_{min}, cost_p > \leftarrow$ MINCUTCOSTRECUR($f_v(b=1)$)
10  $cost_p \leftarrow cost_p +$ GETNODECOST($b$)
11  **if** $cost_n < cost_p$
12      **return** $< cn_{min}, cost_n >$
13  **else**
14      $f_x \leftarrow$ BDDAND($cp_{min}, b$)
15      **return** $< f_x, cost_p >$
```

Fig. 8. Find the minimum cost cut in a given cut set.

checks if this cut set has been visited already, and if so, returns the cached information (line 6-7). This step prevents the need to explicitly enumerate all cuts and dramatically reduces the runtime. If not visited, the algorithm recursively finds the minimum cost cut for the positive and negative cofactors (line 8-10) and returns the cube representing the minimum cost cut (although not shown, the algorithm should also check if the cut is a valid cut).

IV. Covering Problem applied to Elimination

Applying the covering problem to elimination allows us to treat elimination as a global optimization problem and can dramatically reduce the runtime of elimination in BDD based synthesis engines. As illustrated in Fig. 1, the covering problem attempts to cover a given graph with a set of cones such that the covering minimizes a cost metric. A common framework to solve the covering problem is described in [2] and is not described here. When applied to elimination, each cover is collapsed into a single node to remove any redundancies. Since each cover is fairly large (up to 10 inputs), without our scalable cut generation approach, applying the covering problem to elimination would not be practical.

V. Results

TABLE I
Detailed comparison of BddCut cut generation time against ABC.

	K=8 (sec)		K=9 (sec)		K=10 (sec)	
Circuit	*BddCut*	*ABC*	*BddCut*	*ABC*	*BddCut*	*ABC*
`C6288`	2.5	14.5	9.9	150.1	41.9	1758.4
`des`	9.1	10.7	74.7	105.2	828.4	1126.5
`i10`	2.8	6.1	11.4	57.2	50.8	581.1
`b20`	42.0	73.5	200.3	889.9	895.6	n/a
`b21`	44.0	80.3	205.2	942.8	920.2	n/a
`b22_1`	41.2	84.3	180.5	924.3	766.6	n/a
`s15850.1`	1.0	7.6	4.1	16.6	17.9	192.7
`s38417`	4.3	6.2	14.2	58.1	48.0	536.8
`s4863`	1.5	5.0	6.5	50.7	30.8	555.6
`s6669`	1.2	3.5	5.9	32.6	31.6	295.4
Ratio Geomean		2.5x		4.9x		10x

We evaluate the proposed method in two aspects. Since *Bd-*

dCut 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. 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 [15] and applied our algorithms to the MCNC [16] and IWLS [17] benchmark (includes ISCAS89, ITC, and several large circuits) suite.

A. Cut Generation

To investigate our symbolic approach to cut generation, we compare the cut generation time of BddCut against IMap's [2] and ABC's [9] 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. In cases that the technology mapper ran out of memory, the circuit time is marked as n/a.

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	0.781	1.08x	1.59x	2.39x	4.87x

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 improves as K gets larger. Also, for K=10, because ABC does not share any subcuts, it runs out of memory for a few of the larger benchmark circuits. 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 dramatically increase the number of cut duplications in conventional cut generation methods.

One 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.

B. Elimination

B.1 Area and Runtime Impact

After ensuring our symbolic cut generation approach suited our needs for elimination, we 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. As mentioned in section C, logic folding 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 unoptimized 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*) [3] and 4-LUTs using the technology mapping algorithm described in [2]. When optimizing the circuits in SIS, we used $script.rugged$ [3]. Table IV 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 IV, 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 V where we applied our new flow to the entire ITC 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-*

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

	Time (sec)			*Std Cell Area*			*4-LUT Area*		
Circuit	$FBDD_{new}$	*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

LUT is the final 4-LUT area ratio. Each ratio column is given a benchmark heading indicating the benchmark suite used. As Table V 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.

TABLE V
COMPARISON OF AREA AND RUNTIME OF FBDD AGAINST $FBDD_{new}$ FOR VARIOUS VALUES OF K ON THE ITC BENCHMARKS.

K	*Time*	*Std Cell*	*4-LUT*
4	12.4x	0.978	1.001
6	8.76x	1.00	1.00
8	6.16x	0.995	1.00
10	2.55x	1.02	0.991

VI. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a scalable symbolic approach to cut generation using BDDs. We have shown that using BDDs to generate and store cut sets for K-LUT technology mappers has a significant speedup in terms of runtime when compared against current methods and thus is scalable to large cut sizes. As a result, we have been able to apply the covering problem to elimination and we have shown that our approach is competitive with current synthesis tools in terms of both area and runtime where we get a more than 6x speedup with no area penalty when applied to FBDD and an order of magnitude speedup over SIS.

As an additional step, we would like to explore resynthesis region identification for timing driven synthesis using our cover-based elimination. The hope is that we could adapt the elimination algorithm to optimize for circuit delay, rather than solely optimize for area. In conclusion, we have found a scalable approach to cut generation and as a result have found an interesting and useful application of the covering problem to synthesis elimination.

REFERENCES

[1] D. Chen and J. Cong, "DAOmap: a depth-optimal area optimization mapping algorithm for FPGA designs," in *ICCAD '04*, Washington, DC, USA, 2004, pp. 752–759.

[2] V. Manohararajah, S. D. Brown, and Z. G. Vranesic, "Heuristics for Area Minimization in LUT-Based FPGA Technology mapping," *IEEE Trans. Computer-aided Design*, vol. 25, no. 11, pp. 2331–2340, Nov. 2006.

[3] 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

[4] D. Wu and J. Zhu, "FBDD: A folded logic synthesis system," in *International Conference on ASIC*, Shanghai, China, Oct. 2005.

[5] 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

[6] ——, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. Computer-aided Design*, vol. 13, no. 1, pp. 1–13, Jan. 1994.

[7] J. Cong and Y.-Y. Hwang, "Simultaneous depth and area minimization in LUT-based FPGA mapping," in *FPGA*, 1995, pp. 68–74.

[8] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: enabling a general and efficient FPGA mapping solution," in *FPGA'99*. ACM Press, 1999, pp. 29–35.

[9] A. Mishchenko, S. Chatterjee, and R. Brayton, "Improvements to technology mapping for LUT-based FPGAs," in *FPGA'06*. ACM Press, 2006.

[10] Altera Corporation, *Stratix II Device Handbook*, Oct. 2004.

[11] C. Yang, M. J. Ciesielski, and V. Singhal, "BDS: a BDD-based logic optimization system," in *DAC*, 2000, pp. 92–97.

[12] R. Bryant " Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computer*, vol. 35, no. 8, pp. 677–691, 1986.

[13] F. Somenzi, "Binary decision diagrams," pp. 303–366, 1999. [Online]. Available: citeseer.ist.psu.edu/somenzi99binary.html

[14] N. Vemuri, P. Kalla, and R. Tessier, "BDD-based logic synthesis for LUT-based FPGAs," pp. 501–525, 2000.

[15] F. Somenzi, "CUDD: CU decision diagram package release," 1998.

[16] S. Yang, "Logic synthesis and optimization benchmarks user guide version," 1991.

[17] "IWLS 2005 Benchmarks." [Online]. Available: http://iwls.org/iwls2005/benchmarks.html