A Method for Finding Good Ashenhurst Decompositions and Its Application to FPGA Synthesis

Ted Stanion and Carl Sechen
Department of Electrical Engineering
University of Washington

Abstract—In this paper, we present an algorithm for finding a good Ashenhurst decomposition of a switching function. Most current methods for performing this type of decomposition are based on the Roth-Karp algorithm. The algorithm presented here is based on finding an optimal cut in a BDD. This algorithm differs from previous decomposition algorithms in that the cut determines the size and composition of the bound set and the free set. Other methods examine all possible bound sets of an arbitrary size. We have applied this method to decomposing functions into sets of k-variable functions. This is a required step when implementing a function using a lookup table (LUT) based FPGA. The results compare very favorably to existing implementations of Roth-Karp decomposition methods.

I. Introduction

This paper introduces a new method for the decomposition of switching functions. Decomposition is the representation of a logic function by a set of other functions. For example, given a function, $f(X)$, which depends on n variables, $X = \{x_1, \ldots x_n\}$, we wish to find a set of functions, $\{f', g_1, \ldots, g_m\}$, where $f(X) = f'(g_1(X_b), \ldots, g_m(X_b), X_f)$. We call the functions, $\{g_1, \ldots, g_m\}$, *auxiliary* functions. The set of variables $X_b = \{x_1, \ldots, x_l\}$ is called the *bound set*, while the set $X_f = \{x_k, \ldots, x_n\}$ is called the *free set*. If $k = l + 1$, *i.e.* X_b and X_f are disjoint, then the decomposition is *disjunctive*. If $m = 1$, then the decomposition is *simple*. Every function admits a trivial simple disjunctive decomposition where $l = 1$ by virtue of the Shannon decomposition, namely $f = f'(g(x_1), x_2, \ldots, x_n)$ where $g(x_1) = x_1$ and $f' = g \cdot f_{x_1} + g \cdot f_{\bar{x}_1}$. In general, however, larger bound sets provide better decompositions.

The purpose of decomposition is to transform a function which is hard or impossible to implement into a set of functions which are easier to implement. The complexity of implementing a function may be measured in many ways. One possible measure is the size of the support of the function. This measure is important when implementing the function using a lookup table based FPGA architecture where each function must be expressed as a set of functions which depend on k or fewer variables. A typical value for k is 5.

The algorithms in this paper use the binary decision diagram (BDD) as their principle data structure. We refer the reader who is unfamiliar with this data structure to the papers by Bryant [4] and Brace *et al.* [2]. In this paper, the variable associated with a node in the BDD is the *decision variable* of the node. The decision variable of the root of a BDD is the *top variable* of the BDD. If the decision variable of a node is x, then the outgoing edge associated with the assignment $x = 1$ is the *positive edge*. The edge associated with $x = 0$ is the *negative edge*.

Ashenhurst [1], Curtis [6], and Roth and Karp [11] described the first techniques for decomposing functions. We discuss this work in Section II. In Section III and Section IV we present a set of new algorithms for performing decomposition. Like the work of Cong and Ding [5] and Lai *et al.* [8], this method is also based on the BDD data structure. Unlike their work, however, it uses an algorithm for implicitly enumerating all of the cut sets in a BDD to determine the bound set and free set. We introduce this new algorithm in Section III. In Section IV, we apply this algorithm to find good decompositions of switching functions implemented as sets of k-input subfunctions. In Section V, we provide results obtained by this new algorithm.

II. Previous Work

Ashenhurst reported some of the earliest work on decomposition in 1959 [1]. He gave a procedure for determining whether a given set of bound variables admitted a simple disjunctive decomposition. The method forms a *map* of the function which contains one row for each term formed using the variables in the free set and one column for each term formed using the variables in the bound set. Every entry in the table corresponds to one minterm. In Ashenhurst's method we insert a 1 in the table if the corresponding minterm is contained by the function, otherwise we insert a 0. He reduces the table by merging all of the identical columns. Ashenhurst showed that there is a simple disjunctive decomposition of the function with the given bound set if and only if there were at most two distinct columns. In general, a decomposition with m auxiliary functions can be found if there are at most 2^m distinct columns.

The problem with this approach is that a new table must be built for every partition of the variables we consider. If we consider all partitions, an n variable function would require $O(2^n)$ tables. Even if we limit the size of the set of bound variables to k, the number of required tables is still $O(C(n, k))$ where $C(n, k)$ is the number of ways to choose k out of n distinct objects. Furthermore, each table initially contains 2^n entries. Because of this exponential growth, this method is not feasible except for functions with very few variables.

Roth and Karp extended this method in several ways [11]. First, they employ *covers* to represent functions. A cover is a matrix representation of a sum-of-products formula. Using manipulations of the covers, the method partitions the terms formed from the bound set into equivalence classes. These equivalence classes are in one-to-one correspondence with the distinct columns of the previous method. Therefore, a function has a simple disjunctive decomposition if and only if there are at most two equivalence classes for a given partition of the variables. Roth and Karp also generalized this method to per-

32nd ACM/IEEE Design Automation Conference ®

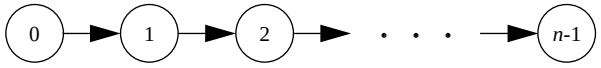

Figure 1: A dag with an exponential number of cut sets.

form non-disjunctive decompositions on incompletely specified functions.

By using covers rather than tables, Roth and Karp were able to reduce the average amount of memory required, although processing time is still a problem except for functions with a small number of variables. The SIS system uses this method as a part of its suite of commands for synthesizing Xilinx FPGAs [9]. To increase the speed of the Roth-Karp algorithm in SIS, this implementation limits the size of the bound set to k or less.

Lai, Pedram and Vrudhula [8] recently described a faster method for implementing the Roth-Karp algorithm: Assume that we wish to find a disjunctive decomposition with a bound set X_b and a free set X_f. Lai creates a BDD such that if $x \in X_b$ and $y \in X_f$ then $\pi(x) < \pi(y)$ where π is a permutation of the variables corresponding to the variable order. The function has a simple disjunctive decomposition if and only if there are at most two source nodes in the subdag induced by nodes with decision variables in the free set. In fact, these nodes are in one-to-one correspondence to the distinct columns of the reduced map in Ashenhurst's method and the equivalence classes of the Roth-Karp. Lai *et al.* implemented this scheme using a variation of BDDs called EVBDDs and demonstrated that it is faster than the cover-based approach implemented in SIS. However, searching over all possible partitions of the variables is still impracticable for functions with a large number of variables. Because of this, Lai also restricts the partitions to those with a bound set of size k or less. Since this algorithm differs from standard Roth-Karp in implementation only, the quality of the results are the same.

The source nodes in the induced subdag of Lai's decomposition method form a cut in the original dag. Lai showed how any given cut in a BDD induces a certain decomposition of the function represented by the BDD where the variables above the cut define the bound set, and the variables below the cut define the free variables. In this case, the decomposition need not be disjunctive. For a given cut it should be possible, therefore, to either calculate or estimate the cost of implementing this decomposition in an FPGA. This observation suggests an alternative method for finding a good decomposition of a function using BDDs. If we estimate the cost of all possible cuts in the BDD, we could then choose the best cut consistent with a given variable ordering. We propose such a method in Section IV, but first we discuss the problem of enumerating cuts in a dag.

III. Enumerating Cuts in a Graph

In order to implement a decomposition method based on finding an optimum cut in a BDD, we need to enumerate all of the cuts in a dag. However, this is a potentially intractable problem. Consider the family of dags represented in Figure 1. For this dag there are $2^n - 1$ non-empty subsets of the nodes. Each one of these subsets is a cut for the graph. Therefore, it is impossible to enumerate all of the cuts unless n is very small. However, we are interested in the minimal cuts of a graph, *i.e.*, a cut which is not a subset of any other cut. In this example, the number of minimal cuts is n. While the number of minimal cuts is often much smaller than the number of cuts, in the worst case it may still be exponential with respect to the number of nodes in the graph. Because of this, we wish to avoid explicitly enumerating the cuts or even the minimal cuts of a dag. We do this by implicitly enumerating the cuts using a characteristic function.

```
procedure cutset(G) {
    s = source of G;
    if FO(s) is empty then
        xi = x_s;
    else
        xi = 1;
        for each v in FO(s) do
            H = subgraph rooted at v;
            xi = xi * cutset(H);
        rof;
        xi = x_s + xi;
    fi;
    return xi;
} end procedure cutset;
```

Figure 2: Procedure for calculating the characteristic function of the cuts of a dag.

Given a dag, $G = (V, E)$, the characteristic function that we wish to build is of the form $\chi : 2^V \rightarrow \{0, 1\}$, a function from the power set of V onto the set $\{0, 1\}$. Furthermore, we require this function to evaluate to 1 when its argument defines a cut set of the dag G and to 0 when it does not. We can establish a bijection between the set 2^V and the set $\{0, 1\}^{|V|}$ as follows. Let $U \in 2^V$. Then the element $x \in \{0, 1\}^{|V|}$ which corresponds to U is $x = (x_1, x_2, \ldots, x_{|V|})$ where $x_i = 1$ if $v_i \in U$ and $x_i = 0$ if $v_i \notin U$. The inverse operation from an element in $\{0, 1\}^{|V|}$ to 2^V is obvious. Therefore, we may represent this characteristic function as a switching function, $\xi : \{0, 1\}^{|V|} \rightarrow \{0, 1\}$. For a given dag G, we denote this as ξ_G.

There are several facts about ξ_G which will be useful in the next section. First, the function is positively unate in all of its variables. Second, the prime implicants of this function are in one-to-one correspondence to the minimal cut sets of the dag. Third, since the characteristic function is unate, any unate sum-of-products expressions for this function is guaranteed to contain all prime implicants. Therefore, since prime implicants correspond to minimal cuts, any unate sum-of-products representation is guaranteed to contain a representation of all of the minimal cut sets.

We use the following propositions to derive a procedure for constructing the characteristic function of the cuts of a dag.

Proposition 1 : Let $G = (\{v_1\}, \varnothing)$ be a dag with one node and no edges. Then the characteristic function of the cuts of G is $\xi_G(x_1) = x_1$.

Proposition 2 : Let $G = (V, E)$ be a dag with a single source node, s. Let the fanout of s be $FO(s) = \{v_1, \ldots, v_m\}$ and G_j be the subdag rooted at v_j, $1 \le j \le m$. Suppose that ξ_{G_j} is the characteristic function for the cut sets of G_j. Then the characteristic function for the cut sets of G is $\xi_G = x + \prod_{j=1}^{m} \xi_{G_j}$ where x is the variable corresponding to node s.

These two propositions provide a recursive step and a terminating step of a procedure for calculating our characteristic function. We outline this procedure in Figure 2.

IV. Finding Decompositions in a BDD

As we mentioned in Section II, Lai *et al.* showed how every cut in a BDD implies a decomposition of a function. Unfortunately, using their method we cannot determine any of the functions in the decom-

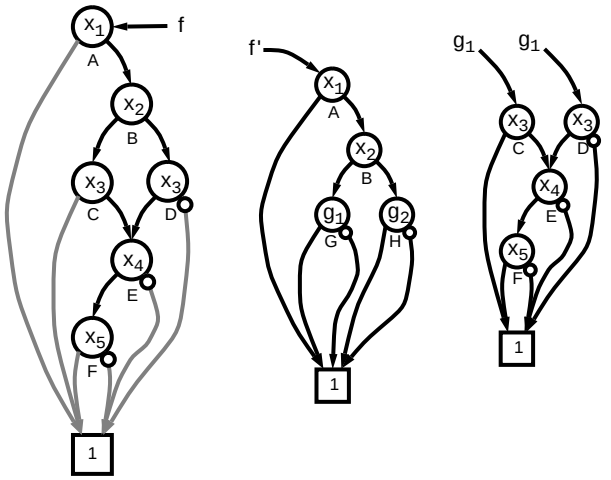

Figure 3: BDDs for function in text and its decomposition.

```
procedure best_cut( ξf , cur_soln, cur_score, k) {
    if ( ξf = 1 ) then
        cur_score = cur_score +
                score(|vars above cut| +
                        |nodes in cut|, k);
        if (cur_score < best_score) then
            best_score = cur_score;
            best_soln = cur_soln;
        fi;
    else if ( ξf ≠ 0 ) then
        x = top( ξf );
        best_cut( (ξf)x̄ , cur_soln, cur_score);
        cur_score = cur_score +
                    score(|support( vx )|, k);
        if (cur_score < best_score) then
            best_cut( (ξf)x , cur_soln ∪ { vx },
                        cur_score);
        fi;
    fi;
} end best_cut;
```

Figure 4: Branch-and-bound procedure for enumerating cuts.

position until we know the entire cut. However, there is another way to determine a decomposition using a cut in a BDD. Recall that Lai places all of the variables above the cut in the bound set and all the variables in the cut and below in the free set. When we perform a decomposition, we make the opposite assignment of variables to the bound and free sets. The nodes in the cut become the auxiliary functions of the decomposition. We create f 'by replacing the nodes in the cut with nodes representing literal functions.

For example, consider the BDD for the function $f = x_1 + x_2(x_3 + x_4x_5) + x_3x_4x_5$ (Figure 3). Edges to the constant node are lightened to indicate that they are not considered when finding a cut. Therefore, the set $\{C, D\}$ forms a cut. The functions represented by the nodes C and D become the BDDs for functions g_1 and g_2 which are extracted from f. We create f 'by replacing the nodes C and D by nodes representing the functions g_1 and g_2, respectively. Now we have created the following decomposition:

$$f' = x_1 + x_2g_1 + x_2g_2,$$
$$g_1 = x_3 + x_4x_5,$$
$$g_2 = x_3x_4x_5.$$

Figure 3 also shows the BDDs corresponding to the functions in this decomposition.

There is a problem with this decomposition, however. The original function f is unate in all of its variables. but the new function f 'is binate in x_2. This is because we replaced the subfunctions with literals ignoring the fact that $g_2 \le g_1$. If we simplified f 'with respect to the satisfiability don't care set implied by the functions in $\{g_1, \ldots, g_m\}$, then this unateness property would be restored.

The advantage of making this assignment of variables to the bound and free sets lies in the fact that the nodes in the cut become functions in the decomposition. Unlike the other method, this allows us to bound the cost of a partial solution, *i.e.* a partial cut, as soon as we add a node to the cut. Therefore, we can give a branch-and-bound procedure for finding an optimal decomposition of a BDD using the characteristic function for the set of cuts. However, we first need to define the objective function we wish to optimize.

Since we wish to implement the function using k-variable LUTs, we must decompose the function into a set of subfunctions such that each subfunction depends on at most k variables. If a function depends on more than k variables, what is the minimum number of lookup tables that is required to implement the function? While we know of no way to easily answer this question exactly, we can place a lower bound on the number of LUTs required. Assume that the function depends on $n > 1$ variables. In order to implement this function, we need at least $\lceil (n-1)/(k-1) \rceil$ LUTs. This is the number of internal nodes in a k-ary tree with n leaves. If n **mod** $k = k-1$, then we reach this bound only if we are always able to perform simple disjunctive decompositions of size k. Since this is a hard lower bound, we define the cost of a function to be

$$\mathbf{cost}(f) = \left\lceil \frac{|\mathbf{sup}(f)| - 1}{k-1} \right\rceil^2 \qquad (1)$$

where $\mathbf{sup}(f) = \{x \mid f_x \neq f_{\bar{x}}\}$ is the set of variables in the support of f. We define the cost of a decomposition $\{f', g_1, \ldots, g_m\}$ as $\mathbf{cost}(f) + \sum \mathbf{cost}(g_i)$. By using the square of the minimum number of required LUTs, we favor decompositions which are closer to meeting our stopping criteria.

We can now give our branch-and-bound procedure for searching for the optimal cut of a BDD. Starting with a BDD for a function f, we create the characteristic function, which we call ξ_f, describing all cuts in the BDD for f. Every path from the root of the BDD of ξ_f to the node representing the constant 1 function represents an implicant of ξ_f. We call these paths *1-paths*. For a given 1-path, we create the corresponding cut set using the following procedure:

1) Initially, the cut set is the empty set.
2) When traversing a positive edge from node u to node w in the BDD for ξ_f, we add node v_i to the cut set where v_i is a node from the BDD for f and x_i is the decision variable of u.
3) When traversing a negative edge, we add no nodes to the cut set.

If we use this procedure while enumerating all 1-paths, we will obtain all of the minimal cuts of the BDD for f. At any point while traversing the path, we can bound the cost of the current cut by computing the cost of the nodes currently in the cut. If their cost exceeds the cost of the best cut found, we abandon the search of this path and try another. This process is presented in Figure 4.

```
procedure decomp(f, k) {
    if (|sup(f)| <= k) then return {fn};
    f = optimize_bdd(f);
    ξ = cutset(f);
    cut = best_cut( ξ , Ø, 0, k);
    nu_fns = do_decomp(f, cut);
    rtn = Ø;
    for each g in nu_fns do
        if (|support(g)| > k) then
            rtn = rtn ∪ decomp(g,k);
        else
            rtn = rtn ∪ {g}
        fi;
    rof;
    return rtn;
} end decomp;
```

Figure 5: Procedure for decomposing a function into *k*-variable LUTs.

The procedure ***best_cut*** is called with four parameters: the characteristic function of the set of cuts, ξ_f, the current partial solution, `cur_soln`, the score of the current partial solution, `cur_score` and the maximum number of variables in an LUT, `k`. Initially, we call this procedure with a current solution consisting of the empty set and current score of zero. The procedure ***top*** returns the top variable of a BDD. Since every variable, x, in ξ_f corresponds to a node in the BDD of f, we denote this node and its corresponding function as v_x. The procedure ***score***(n, k) returns the square of the minimum number of LUTs required to implement an n variable function with k variable LUTs as determined by Eq. (1). If the characteristic function passed to the procedure represents the constant 1 function, then the current solution is a complete cut. The current score at this point represents the minimum number of LUTs required to implement the auxiliary functions represented by the nodes in the cut. In order to compute the total cost of the cut, we need to add the cost of the function f' which the cut induces. Since we add a new variable for each auxiliary function, the number of variables in f' is the number of variables which appear above the cut plus the number of nodes in the cut. If the score of this cut is better than the best found so far, then we save it as the best found so far.

If we reach the node of the constant 0 function, then the current path does not represent an implicant of the characteristic function. Otherwise, we are at a non-sink node of the BDD. In this case, we first assume that the node associated with the top variable of the characteristic function will not be added to the current partial solution and make the appropriate call to ***best_cut***. Following that, we assume that the node associated with the top variable is added to the current partial solution. If adding this node to the cut does not cause the score of the new partial solution to exceed the best found so far, then we call ***best_cut*** again. This bounds the search process by terminating paths which are guaranteed to result in cut sets representing decompositions worse than the best already found. In spite of this bounding procedure, it is sometimes not feasible to run the branch-and-bound procedure to completion. To combat the possible combinatorial explosion, the user may limit the number of cuts that the algorithm enumerates.

We use the method for finding an optimal cut in a BDD in an algorithm for decomposing a function of n variables into a set of functions of k variables. This is outlined in Figure 5. Given a function f and a parameter k, the procedure ***decomp*** returns a set of functions which implement f such that each function in the set has a support of k or less. If the support of f is less than or equal to k, we return f itself. Otherwise, we minimize the BDD of f by reordering its variables. Next, we calculate the characteristic function of all cuts of the minimized BDD. We find an optimal cut based on this characteristic function and the objective function mentioned earlier. The procedure ***do_decomp*** returns the set of functions corresponding to the decomposition implied by the optimum cut. If any of the functions in the decomposition do not meet the decomposition criteria, then they are decomposed further with a recursive call to ***decomp***.

V. Results

To test the efficacy of the algorithm described in Figure 5, we decomposed functions in 27 MCNC benchmark circuits into sets of 5-variable functions. First, we minimized the circuits using standard scripts provided with SIS 1.2 [12]. We then decomposed the circuits using both SIS and our program, Catamount.

We implemented the algorithm described in this paper as follows. We limit the branch-and-bound procedure to examining at most 5000 cuts. Also, if a BDD for a function was excessively large, we do not generate the characteristic function. Instead, we choose a default decomposition with a single variable in the free set such that our objective function is minimized. This default decomposition produces Shannon decompositions of the type $f = f'(g_1, g_2, x) = x \cdot g_1 + \bar{x} \cdot g_2$ where $g_1 = f_x$ and $g_2 = f_{\bar{x}}$.

Once we produce the decompositions, we then simplify all functions in the network. Finally, we look for nodes in the decomposed network which can be collapsed into all of its fanouts without violating the support constraint. Whenever we find such a node, we immediately collapse it into its fanouts. (Since the order in which the nodes are collapsed affects which other nodes may be subsequently collapsed, this greedy procedure may not be optimum.) At the end of this step, we have a decomposed network which may be implemented by 5-variable LUTs.

To compare the results obtained by this algorithm to the normal implementation of Roth-Karp decomposition, we also decomposed the same functions into 5-variable functions using the SIS `xl_k_decomp` command. This command implements classical Roth-Karp decomposition. After SIS obtained its initial decomposition, we also performed simplification and collapsing of collapsible nodes as above. We base this sequence on the recommended procedure for FPGA synthesis outlined in the SIS Technical Report from UC, Berkeley [12]. In neither case did we perform merging of nodes into two output functions as some FPGA architectures allow.

Table I shows the results of this experiment. The number of nodes originally in each circuit is given in the column titled *Ori*. The column titled *Inf* gives the number of infeasible nodes in each circuit. This is the number of functions whose support exceeded five and required decomposition. The column titled *SIS* gives the number of LUTs required in the final decompositions found by SIS. The column titled *Cat* gives the same number for the decompositions found by Catamount. The column titled *Ratio* gives the ratio of the number of LUTs required by Catamount to the number required by SIS.

These results demonstrate that Catamount's decomposition algorithm is very effective compared to SIS's implementation of Roth-Karp. For all circuits, Catamount produced a superior decomposition. Sometimes this difference is quite large. For six circuits, the Catamount decomposition required half of the number of LUTs required

TABLE I
COMPARISON WITH ROTH-KARP DECOMPOSITION IN SIS.

Circuit	Ori	Inf	SIS	Cat	Ratio
5xp1	19	6	81	26	0.32
9sym	11	2	35	31	0.89
9symml	16	3	94	49	0.52
C499	56	24	84	78[1]	0.93
C5315	321	95	618	344	0.56
C880	79	13	241	112	0.46
alu2	46	9	187	117	0.63
alu4	122	20	211[2]	182	0.86
apex2	52	8	184	109	0.59
apex6	147	28	265	182	0.69
apex7	61	8	63	57	0.90
b9	29	7	61	37	0.61
bw	34	12	62	39	0.63
clip	15	5	90	32	0.36
count	31	15	32	31	0.97
DES	562	185	2435	1072[1]	0.44
duke2	77	18	228	134	0.59
e64	76	19	95	84	0.88
f51m	15	3	43	13	0.30
misex1	10	4	17	13	0.76
misex2	26	6	35	30	0.86
rd73	16	2	28	17	0.61
rd84	18	2	39	24	0.62
rot	161	22	291	176	0.60
sao2	13	4	123	39	0.32
vg2	10	7	29	26	0.90
z4ml	8	2	10	6	0.60
Total	2031	529	5681	3060	0.54

[1.] Simplification was not performed due to excessive CPU time.
[2.] The `xl_k_decomp` command did not complete within one CPU hour. The `xl_imp` command was used instead. Besides Roth-Karp, this command uses several other decomposition techniques.

by SIS. Overall, 46 percent fewer LUTs were required to implement all benchmark circuits.

Although SIS and Catamount both perform Ashenhurst style decomposition, the two methods choose their bound sets quite differently. In SIS, the Roth-Karp decomposition is limited to decompositions with a bound set of size $k = 5$. In Catamount's method, no limit is placed on the size of the bound set or the free set. There is a restriction that the bound set and free set must be consistent with a given variable ordering. However, we may use any decomposition consistent with this ordering. We feel that the limitation of using decompositions with bound and free sets consistent with a given ordering is offset by the flexibility in choosing sets of any size.

Another difference in the two algorithms is the way they encode the auxiliary functions, $\{g_1, \ldots g_m\}$, when forming the function f'. In the implementation of Roth-Karp in the latest version of SIS (Version 1.2), these functions are encoded serially. In our algorithm, the structure of the BDD above the cut implies the encoding. Murgai *et al.* have recently introduced an extension to the existing implementation of Roth-Karp which also performs smarter encoding of the auxiliary functions [10]. By doing this, they were able to reduce the LUT count by 21 percent over the naive serial encoding (*cf.* the 46 percent improvement reported here).

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented two ideas. First, we show how it is possible to implicitly enumerate all of the cuts of a dag. We do this by constructing a characteristic function for the set of cuts. Because this function is unate, it is also possible to easily enumerate all minimal cuts. Second, we showed how a cut in a BDD implies a decomposition of the function represented by the BDD. Since a BDD is a dag, we use the previous algorithm to generate the characteristic function of the set of cuts in the BDD. We then use a branch-and-bound procedure in order to find the cut which produce the best decomposition given a certain objective function. This procedure is able to significantly outperform SIS when decomposing functions from MCNC benchmark circuits.

We feel that we can apply these algorithms to other problems. First, the task of analyzing dags is one which occurs often in other CAD problems. The cut enumeration algorithm can be useful in other applications where we need to enumerate cuts in a graph. Second, the optimization criteria that we use for our decompositions are designed to produce good decompositions for implementing a function using LUTs. It should be easy to produce different decompositions which are optimized for other purposes by changing the objective function and stopping criteria in the procedure in Figure 5. One possible application would be decomposition for delay optimization. At each step in the decomposition we would choose the cut which minimized the expected number of levels in the circuit.

REFERENCES

[1] R.L. Ashenhurst, "The decomposition of switching functions," *Ann. Comp. Lab., Harvard Univ.*, vol. 29, 1959, pp. 74-116.

[2] K.S. Brace, R.L. Rudell and R.E. Bryant, "Efficient implementation of a BDD package," *Proc. 27th ACM/IEEE DAC*, June 1990, pp. 40-5.

[3] R.K. Brayton, G.D. Hachtel and A.L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proc. IEEE*, vol. 78, no. 2, Feb. 1990, pp. 265-300.

[4] R.E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Computers.*, vol. 35, no. 8, August 1986, pp. 677-91.

[5] J. Cong and Y. Ding, "Beyond the combinatorial limit in depth minimization for LUT-based FPGA designs," *Proc. 1993 IEEE Intl. Conf. on CAD*, November 1993, pp. 110-4.

[6] H.A. Curtis, *A New Approach to the Design of Switching Circuits*, D. Van Nostrand Co., Princeton, NJ, 1962.

[7] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill, New York, 1970.

[8] Y.-T. Lai, M. Pedram and S.B.K. Vrudhula, "BDD based decomposition of logic functions with applications to FPGA synthesis," *Proc. 30th ACM/IEEE DAC*, June 1993, pp. 642-7.

[9] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," *Proc. 27th ACM/IEEE DAC*, June 1990, pp. 620-5.

[10] R. Murgai, R.K. Brayton and A. Sangiovanni-Vincentelli, "Optimal functional decomposition using encoding," *Proc. 31st ACM/IEEE DAC*, June 1994, pp. 408-13.

[11] J.P. Roth and R.M. Karp, "Minimization over Boolean graphs," *IBM J. Res. Dev.*, April 1962, pp. 227-38.

[12] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan and R.K. Brayton, *SIS: A system for sequential circuit synthesis*, Technical report UCB/ERL M92/41, University of California, Berkeley, May 1992.