DELAY OPTIMIZATION BASED ON BDD AND COMMUNICATION COMPLEXITY

by

Minshine Shih

Memorandum No. UCB/ERL M92/117

16 October 1992

DELAY OPTIMIZATION BASED ON BDD AND COMMUNICATION COMPLEXITY

by

Minshine Shih

Memorandum No. UCB/ERL M92/117

16 October 1992

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

DELAY OPTIMIZATION BASED ON BDD AND COMMUNICATION COMPLEXITY

by

Minshine Shih

Memorandum No. UCB/ERL M92/117

16 October 1992

Delay Optimization Based on BDD and Communication Complexity*

Minshine Shih
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
California 94720

October 16, 1992

*This research was sponsored by the National Science Foundation, under Grant No. MIP 88-03711 and the State of California MICRO program.

Abstract

We propose a delay optimization method based on top-down logic decomposition. Given a logic function, we decompose it into simpler sublogics and construct a circuit structure with minimum circuit delay as the primary goal. We generalized the α-G decomposition proposed by Roth & Karp[1] to allow arbitrary α-functions. Instead of their branch and bound exhaustive search, we use Communication Complexity based heuristics to determine the best α and G-functions. Since all our algorithms are operated on BDD (Binary Decision Diagram) data structure, they take much less CPU time than existing methods. In addition to computational efficiency, our method provides much more flexibility in the structures of decomposed circuit, and enables us to reduce circuit delay by controlling decomposition sequence.

A group of circuits from MCNC benchmark set were run and results are given. When compared to standard logic minimization tool (misII), our decomposition method produces circuits which are 41% faster and 20% smaller (area) by using 49% less CPU time on average. Applications of our algorithms including delay optimization for combinational circuit are also given.

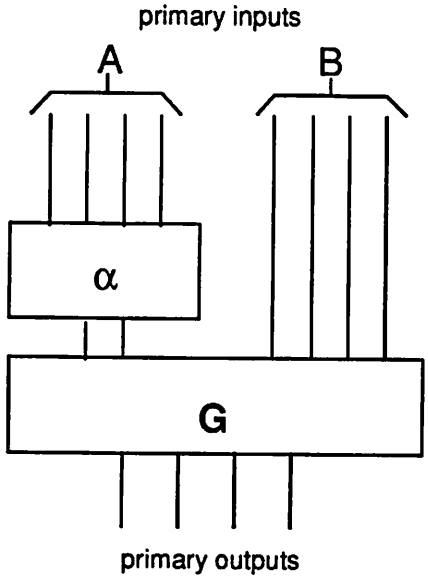


Figure 1:

1 Introductions and Definitions

1.1 Introduction

We propose a delay optimization method based on logic decomposition. Given a logic function, we decompose it into simpler sublogics and construct a circuit structure with minimum circuit delay as the primary goal.

Karp & Roth[1] introduced the concept of decomposition by α and G functions.(see Fig. 1 for example) We generalize their top-down decomposition method and allow α-functions to be any general functions instead of just primitive gates in their algorithm. Instead of their branch and bound exhaustive search, we use Communication Complexity based heuristics to determine α and G-functions.

Our basic strategy is simple: find a "good" subset A of input variables, compute the α and G-functions such that A is the set of input variables to α exclusively. This strategy can be repeated k times on the (rest of the original) input variables, obtaining k α-functions. The resulting logic network would have k branches, each branch having disjoint input variables (see Fig. 2 for example). This strategy can also be applied recursively to α and G-functions.(see Fig. 3 for example) There can also be many combinations of the basic strategy.

In theory we may obtain better results by decomposing into k α-functions on the same level of hierarchy. But for simplicity of implementation, our first attempt focused on decomposing into 2 α-functions, that is,

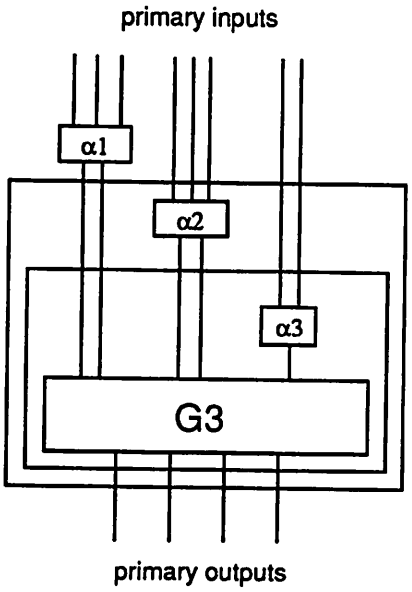


Figure 2:

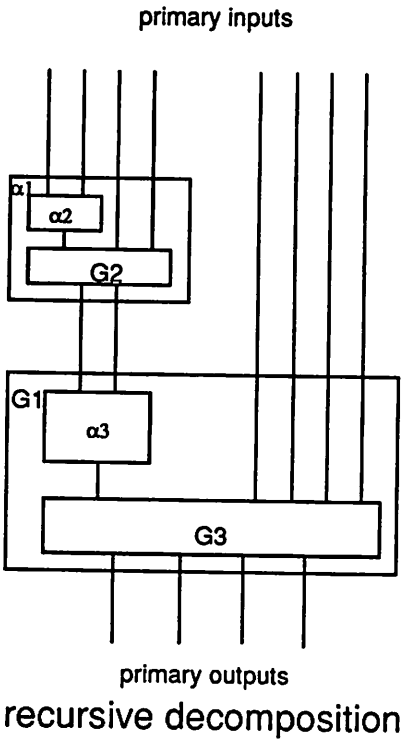


Figure 3:

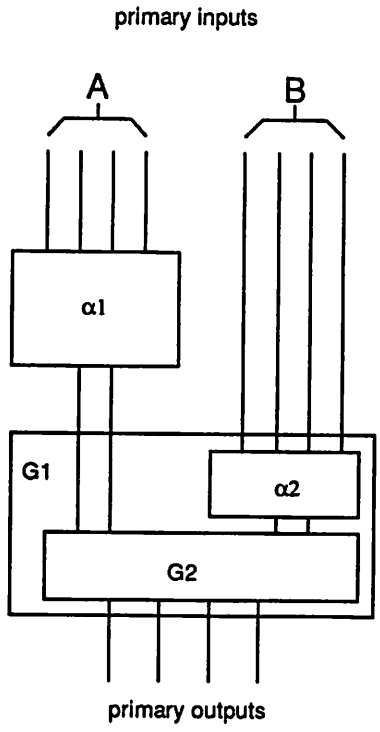


Figure 4:

$k = 2$ (Fig. 4). This immediately translates to grouping the input variables into two subsets. Therefore in the rest of this paper we shall use "input grouping" to mean the 2-way grouping of input variables unless otherwise noted. Our benchmark results are obtained with 2-way groupings of almost equal sizes (balanced grouping) to achieve faster circuits, but our problem formulation and algorithms does not depend on this fact, i.e., we make no assumption on the sizes of the groupings.

The key issue in our approach is how to find a "good" subset of input variables. This is done by the heuristic of minimizing communication complexity. Hwang[2] first proposed the idea of finding a good input grouping by minimizing the rank of the Communication Matrix. However the size of the circuits can be handled is limited because of the exponential size of the matrix. To solve this problem we propose to use BDD (Binary Decision Diagram) introduced by Bryant[3]. We use BDD as our only internal data structure for *both* finding the optimal input grouping *and* computing the α and G-functions efficiently.

Our decomposition approach can be extended to a general k-way style, in this respect it is more general than the work of Hwang[2] since the matrix LDR decomposition is 2-way by nature. Another difference is that our approach can take advantage of a large library of gates for technology mapping, while the matrix decomposition method inherently uses only 2 different gates (corresponding to 'addition' and 'multiplica-

tion' of matrix elements). Since *all* of our algorithms are operating on BDD, our method is much more efficient than Hwang's approach which is operating on a matrix of exponential size.

1.2 Definitions and Notations

Let $f : B^n \longrightarrow B^m$ be a completely specified function with n inputs and m outputs, where $B = \{0, 1\}$.

Definition 1 *$(A : B)$ denotes a 2-tuple of subsets of input variables, where A,B (in that order) are two non-empty disjoint subsets of input variables of f, and $A \bigcup B$ is the set of all input variables of f.*

Furthermore, let A, B be as above, $\{A, B\}$ denotes a 2-way grouping of input variables.

Let $n_A = |A|$ and $n_B = |B|$, then clearly $n_A + n_B = n$.

Notice that by definition $(A : B)$ and $(B : A)$ are different 2-tuples.

Definition 2 *The Communication Matrix $C_f(A : B)$ of a completely specified function f is defined as a matrix of 2^{n_A} rows and 2^{n_B} columns, where each row (column) index corresponds to a minterm from the subset A (B) of input variables. Thus each element c_{ij} in the matrix corresponds to a minterm of the function f. The value of element c_{ij} is an m-bit binary vector and represents the output vector of f when the corresponding minterm is applied to f.*

Definition 3 *We say rows i_1 and i_2 are distinct when there exists a column j such that $c_{i_1 j} \neq c_{i_2 j}$.*

Definition 4 *Given a BDD ordering of input variables and a 2-tuple $(A : B)$, we say the BDD ordering is compatible with the 2-tuple $(A : B)$ if $\forall\, a \in A$ and $\forall\, b \in B$, a proceeds b in the BDD ordering.*

Definition 5 *The Communication Complexity of a function f with respect to a 2-way input grouping $\{A, B\}$ is defined as the sum of the numbers of distinct row patterns of two Communication Matrices $C_f(A : B)$ and $C_f(B : A)$.*

In the case of k-way decomposition, the inputs are grouped into $\{A_1, A_2, A_3, ..., A_k\}$. The definition of Communication Complexity becomes

$$\sum_{i=1}^{k} (number\ of\ distinct\ row\ patterns\ in\ C_f(A_i : (\bigcup_{j \neq i} A_j)))$$

2 Input Grouping Minimizing Communication Complexity

Our first step for logic decomposition is to group the input variables into 2 groups $\{A, B\}$ such that the communication complexity is minimized.

Brayton[4] made the following observation:

Theorem 1 *Given a Communication Matrix $C_f(A : B)$, the number of distinct row patterns can be obtained from any BDD compatible with $(A : B)$ by counting the number of BDD nodes (including terminal nodes) satisfying the following conditions:*

a) not corresponding to an input variable in A and

b) being pointed to by another BDD node corresponding to an input variable in A or by the BDD root.

The BDD nodes satisfying these 2 conditions are called "pattern nodes" since they each corresponds to a unique row pattern in the matrix. Each pattern node corresponds to a *compatible class* of minterms from A as defined in [1].

From this theorem the communication complexity can be computed by simple graph traversals on two BDD's, one with ordering compatible with $(A : B)$ and the other compatible with $(B : A)$.

2.1 Single Output BDD

In order to find a good grouping we need to visit many different groupings. Therefore we built a simple basic operation called "bubbling", which is a local swap of 2 neighboring input variables in the BDD ordering[8]. The exploration of search space is done by repeating this basic operation in an efficient fashion. For example, let [a b c d e f] represent the BDD ordering, we can "bubble" variable 'e' into the 2nd position by 3 bubble operations (d e), (c e) and (b e). The result is [a e b c d f]. This kind of bubbling will be used repeatedly in our algorithm.

Since we need two BDD's in order to compute communication complexity, one for $(A : B)$ and the other for $(B : A)$, we must maintain two BDD's along the search process. One convenient way of doing it is to keep two BDD's in total reverse order w.r.t each other, whenever we need to bubble one BDD, we bubble the other BDD on the same variable pair (and consequently) in the *opposite* direction. When we need to count the communication complexity we simply obtain the sum of the counts on these 2 BDD's.

Given a fixed number n_A for the size of the first grouping, the following exact algorithm *enumerate* finds the optimum grouping of input variables w.r.t. communication complexity.

Let $[a_{-n_A} \quad a_{-n_A+1} \quad \ldots \quad a_{i_A} \quad \ldots \quad a_{-2} \quad a_{-1} \ , \ a_1 \quad a_2 \quad \ldots \quad a_{i_B} \quad \ldots \quad a_{n_B}]$ be the BDD ordering, where $-n_A \leq i_A \leq -1$ and $1 \leq i_B \leq n_B$.

Algorithm 1 *enumerate;*

```
{       if (n_A = n_B) {
            /* In this case, we can fix a_{n_B} in place to avoid checking */
            /* equivalent groupings twice, e.g. [a b c,d e f] and [d e f,a b c] */
            enum(-n_A, n_B - 1);
        } else {
            enum(-n_A, n_B);
        }
        return;
}
enum(i_A, i_B)
/* all variables not between i_A and i_B are fixed */
{   if (i_A = -1 and i_B = 1) {
        count communication complexity;
        exchange a_{-1} and a_1 by 1 bubble operation;
        count communication complexity;
    } else if (i_A = -1) {
        enum(-1, i_B - 1);
        move a_{i_B} to slot index -1 by i_B bubble operations;
        count communication complexity;
    } else if (i_B = 1) {
        enum(i_A + 1, 1);
        move a_{i_A} to slot index 1 by -i_A bubble operations;
        count communication complexity;
    } else {
        enum(i_A + 1, i_B);
        move a_{i_A} into slot index i_B by (i_B - i_A - 1) bubble operations;
        enum(i_A, i_B - 1);
    }
```

return;

}

Theorem 2 *If $n_A \neq n_B$, algorithm "enumerate" visits all $\frac{n!}{n_A!n_B!}$ groupings exactly once.*
If $n_A = n_B$, algorithm "enumerate" visits all $\frac{1}{2} \times \frac{n!}{n_A!n_B!}$ groupings exactly once.

Proof. The proof is based on induction on the sizes of n_A,n_B and examining all 4 cases in subroutine "enum". The detail is omitted here. □

This algorithm is so efficient that it visits a different grouping in less than 2 × 3 bubble operations on average, independent of the total number of input variables[8], where the factor of 2 is due to the fact that we bubble 2 BDD's at once. Notice that if $n_A = 3$, [a b c , d e f] and [f e d , b a c] are considered as the same grouping.

Additionally we developed a heuristic algorithm based on the general framework of Kernighan & Lin[6] partition heuristic to handle circuits whose numbers of inputs are large. This becomes useful when the number of inputs is beyond about 12.

2.2 Multiple Output BDD

We need multiple output BDD (MOBDD) to represent multiple output functions. Unlike others[5], our MOBDD (Figure 5) has a different structure. This structure provides the important information of communication complexity (as defined in this paper) by looking all outputs at once. In the worst case it may have 2^m terminal nodes, where m is the number of outputs. Our MOBDD is built this way so that we can use exactly the same methods of counting communication complexity and selecting optimum input grouping.

2.3 Construction of MOBDD

We first build single output BDD's for each output separately. Then we *merge* two BDD's into one in a recursive fashion starting from the roots of these 2 BDD's and apply the merging procedure to their left children and right children respectively until terminal nodes are reached, where new terminal nodes are created by concatenating the values of the terminal nodes being reached in those 2 original BDD's. Merging 2 BDD's is repeated in a loop until all single output BDD's are merged into the target MOBDD. The overall

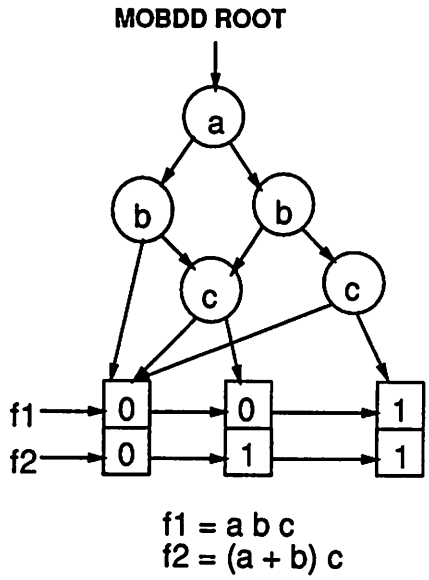


Figure 5:

computation complexity is linear with respect to the size of the final BDD. When merging 2 BDD's, since the same node may be reached by more than one path in a BDD, care must be taken not to merge the same nodes more than once.

3 Logic Decomposition by BDD

After we obtain a good input grouping, the next step is to compute the α and G-functions.

3.1 α-Function

Let $\{A, B\}$ be the optimum input grouping obtained, and n_p be the number of pattern nodes in a BDD whose ordering is compatible with $(A : B)$. It is obvious that we can encode all the input variables in A with $\lceil log_2(n_p) \rceil$ bits. The following algorithm computes the α-function BDD corresponding to input variables in A.

Algorithm 2 *α-function(BDD_ORIGINAL)*

1. n_bits = $\lceil log_2(n_p) \rceil$

2. copy BDD_ORIGINAL to BDD_α.

3. On BDD_α, assign increasing coding numbers (starting from 0) to each one of n_p pattern nodes in an arbitrary order, record the order that pattern nodes are encoded.

4. set values of these pattern nodes equal to their own encoding number and change them into terminal nodes, this is equivalent to terminating BDD_α at pattern nodes.

5. return BDD_α

BDD_α contains all BDD nodes corresponding to variables in A, plus the newly created terminal nodes. It represents an α-function with A as the set of input variables and A' as the set of output variables, where $|A'|$ = n_bits.

3.2 *G*-Function

When extracting the α-function in the previous section, we are essentially extracting the top part of BDD_ORIGINAL. Now we extract the bottom part as described by the following algorithm.

Algorithm 3 *G-function(BDD_ORIGINAL)*

1. create a complete rooted binary tree BDD_G with 2^{n_bits} *leaf nodes.*

2. label the nodes in the binary tree with variables in A' such that nodes at the same level of binary tree are labeled with the same variable in A'.

/ The main loop is starting from the leftmost leaf node of the binary tree and iterates towards the right. */*

3. for ($i = 1; i \leq 2^{n\_bits}; i$++) {

if ($i \leq n_p$) {

substitute BDD_G's ith leaf node (from the left) by the ith pattern node (and consequently all its offsprings) of BDD_ORIGINAL according to the order recorded earlier in algorithm α-function.

} *else* {

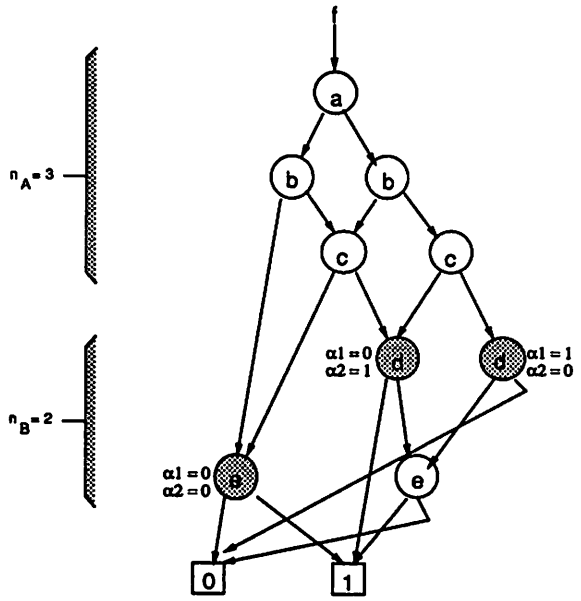
/ This is the don't care situation */*

substitute BDD_G's ith leaf node (from the left) by the rightmost pattern node (and consequently all its offsprings) of BDD_ORIGINAL.

}

}

4. return BDD_G



f = abc'e' + ab'ce' + a'bce' + a'b'e + a'c'e + a'd'e + bcd'e' + b'c'e + b'd'e + c'd'e

There are 3 pattern nodes and requires 2 bits to encode.

Figure 6:

BDD_G corresponds to a G-function with A' and B as the sets of input variables and outputs remain the same as original. Figures 6, 7 and 8 illustrate these two algorithms.

Now we need to do the second time α/G decomposition in order to obtain another α-function for input variables in B and the final G-function.

4 Implementations and Experiments

4.1 Control Flow

Our global strategy is to apply this 2-way decomposition procedure in a recursive fashion to both α and G-functions until we cannot get any simplification, i.e. $|A| = |A'|$. Then we stop recursion and convert the current BDD's into Boolean network. The BDD's may be a single output BDD or MOBDD. In the latter case the conversion is done for one output at a time till all outputs are converted into the Boolean Network. Our current conversion procedure simply starts from the terminal nodes of BDD and works upwards by composing the function at each BDD node from its left child's and right child's respective functions. Since the "0" and "1" terminal nodes for a particular output may be scattered among more than 2 terminal nodes in an MOBDD, we need to do a BDD reduction[3] before we proceed. When the function at the BDD root

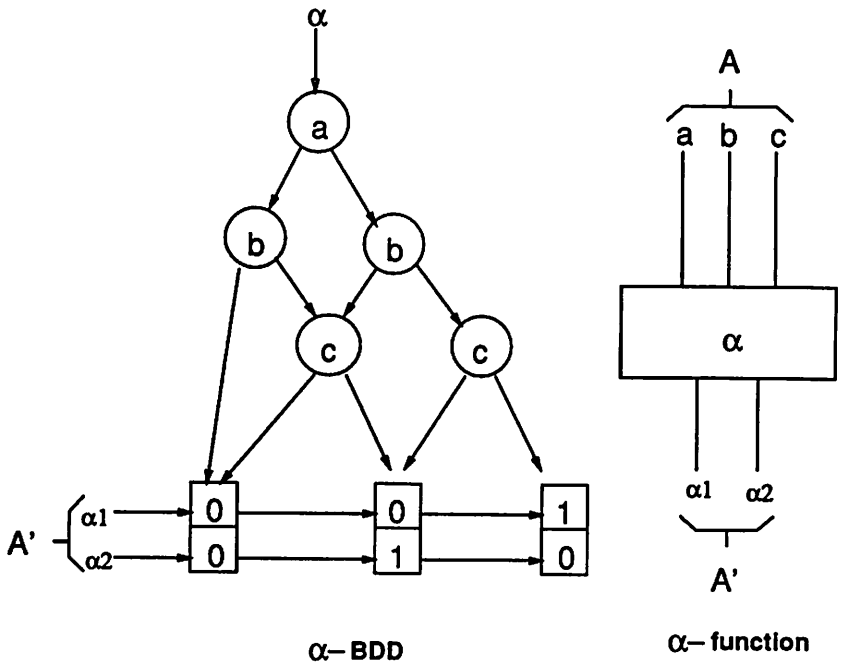


Figure 7:

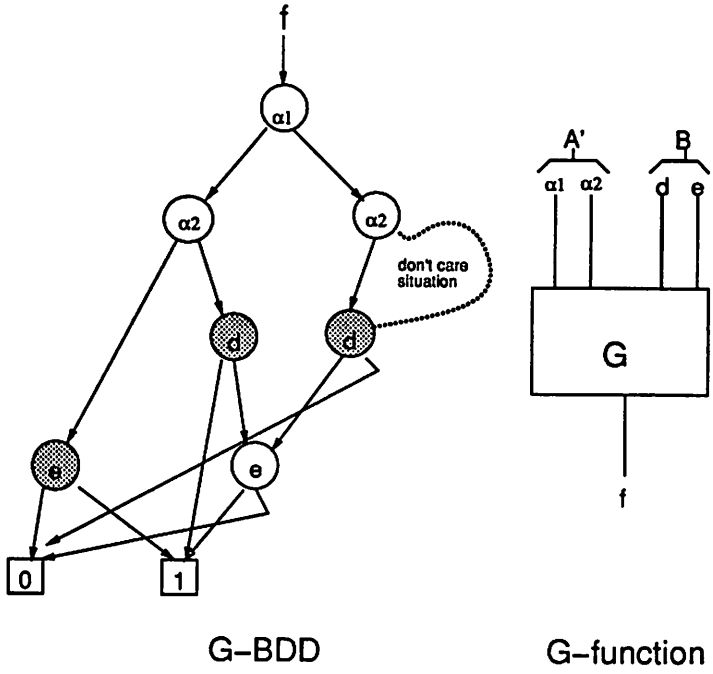


Figure 8:

is composed, we create a Boolean node in the network representing the function.

Once the complete Boolean network is obtained we invoke ESPRESSO to simplify each individual Boolean node function. It is important not to disturb the circuit structure at this stage.

4.2 Advantage of Decomposition

The main advantage of a balanced decomposition is to force *all* input signals to be encoded into a smaller set of signals simultaneously. The *parallel* processing of signals through logic gates makes different branches of the decomposition tree have almost equal path delays. This reduces the difference between the largest and smallest arrival times at the primary outputs and improves circuit speed.

In case that primary input signals arrive at different times, we can decompose the logic in an unbalanced way to allow late-coming signals to be processed at a latter stage. Again the the difference between the largest and smallest arrival times will be minimized at the primary outputs.

4.3 MCNC Benchmark

We compare our decomposition approach against mis2.2 running standard script for a set of MCNC benchmark circuits. The correctness of our results were verified by misII 'verify' command. The area and delay informations came from mis2.2 technology mapper using msu standard cell library 2.2. The results are in Table 1. The average circuit speedup is 41% and circuit area savings is 20%. and the average cpu savings from our method is 49%.

We also tried to combine our method with misII standard script. The results showed that although we can gain some further area savings, the improvements of circuit speed from our method were totally eliminated by running misII. This fact further proved that our method is unique in the way that it generates a fast circuit structure which is not obtainable or even maintainable in other methods.

5 Future Work and Applications

Currently our overall CPU time is dominated by the searching of optimum input grouping using BDD. We use traditional linked list to represent BDD. There should be 1-2 orders of magnitude speedup if we use hashing techniques on BDD[7] (It is feasible but the modification is nontrivial).

Because of the improvement in circuit speed by our approach, we are investigating a possible application, namely, to speed up a large circuit by speeding up a set of small subcircuits along the critical paths. Another possible application is to take full advantage of the generality of α/G decomposition. Since we

have full control on the overall structures of decomposed subcircuits, we can decompose in such a way to allow those signals along the original critical paths to arrive late without being timing-critical. Thus the delay along the original critical path is reduced and the overall speed can be improved.

6 table

		misII standard script				BDD decomposition alone				BDD decomposition + misII script		
circuit	# I/O	cpu	# lit.	area	delay	cpu	# lit.	area	delay	# lit.	area	delay
C17	5/2	0.5	9	136	3.00	0.4	13	192	5.60	9	160	4.20
b1	3/4	0.5	10	128	3.00	0.1	11	152	3.20	10	128	3.00
majority	5/1	0.4	10	200	5.40	0.3	19	240	5.40	10	200	5.40
rd53	5/3	2.3	37	584	12.20	0.6	39	568	6.40	30	488	13.20
z4ml	7/4	3.2	52	880	12.80	3.3	56	768	8.60	48	872	15.80
parity	16/1	1.9	60	664	6.20	16.7	60	712	5.00	60	648	8.60
rd73	7/3	20.3	78	1256	15.00	2.3	79	992	11.80	61	992	20.00
f51m	8/8	9.6	126	2032	36.40	38.2	171	2312	13.20	124	2112	38.60
5xp1	7/10	8.3	129	2272	30.40	19.7	172	2360	13.60	117	1936	25.40
rd84	8/4	119.9	168	2680	24.40	3.4	109	1496	13.20	85	1416	22.00
9symml	9/1	31.4	192	3040	17.60	5.4	91	1352	12.80	77	1320	18.00
TOTAL		198.3	871	13872	166.40	90.4	820	11144	98.80	631	10272	174.20
% change w.r.t misII		0	0	0	0	-49.3	-5.8	-19.6	-40.6	-27.5	-25.9	+4.6

Note 1: Cpu's are in seconds on VAXstation 3100.

Note 2: Boolean script was used as misII standard script.

Note 3: Number of literals were counted in factored form.

Note 4: All "% change"'s are relative to misII standard script.

References

[1] J.P. Roth and R.M. Karp, "Minimization Over Boolean Graphs," *IBM Journal of Research and Development*, Apr., 1962.

[2] T. Hwang, R.M. Owens and M.J. Irwin, "Exploiting Communication Complexity for Multilevel Logic Synthesis," *IEEE Transactions on Computer-Aided Design*, Oct., 1990.

[3] R.E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, Aug., 1986.

[4] private communication, Apr., 1990.

[5] S. Minato, N. Ishiura and S. Yajima, "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation," *Proceedings, 27th Design Automation Conference*, Jun., 1990.

[6] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *The Bell System Technical Journal*, Feb., 1970.

[7] J.R. Brace, R.L. Rudell and R. Bryant, "Efficient Implementation of a BDD Package," *Proceedings, 27th Design Automation Conference*, Jun., 1990.

[8] M. Shih, "BDD and Communication Complexity," unpublished manuscript May., 1990.