B4.

# One-Phase Technology Mapping for EPGAs Using Extended GBDD Hash Tables

Cheng-Hsing Yang Sao-Jie Chen
Dept. of Electrical Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.

Jan-Ming Ho
Institute of Information Science
Academia Sinica
Taipei, Taiwan, R.O.C.

Chia-Chun Tsai
Dept. of Electronic Eng.
National Taipei Institute of Technology
Taipei, Taiwan, R.O.C.

## Abstract

*An efficient and novel algorithm for technology mapping of electrically programmable gate arrays (EPGAs) is proposed. In this algorithm, the personalization of an uncommitted logic module is done by stuck-at fault and/or bridging fault assignments. This technology mapping algorithm combines decomposition and covering into one phase. In order to do boolean matching fast, an extended GBDD hash table constructed by permuting and bridging inputs of a module is developed to test whether a portion of given combinational logic circuit can be implemented by personalizing a module quickly. Some experimental results on standard benchmarks are reported.*

## 1. Introduction

*Electrically programmable gate arrays* (EPGAs), logic devices consisting of arrays of identical logic modules, are becoming increasingly important for rapid system prototyping because of their short turnaround time, low cost of manufacturing, and full testability. Figure 1 shows two logic modules, ACT-1 and ACT-2, developed by the Actel Inc. An uncommitted module can be configured to implement a logic function by assigning some of its inputs with logic low or logic high or by bridging inputs [1].

There are some basic phases of synthesizing a circuit using EPGA: logic optimization, technology mapping, placement, and routing [2]. The phase of technology mapping is responsible to transform a given optimized expression into a circuit of EPGA logic modules. The most widely used methods for EPGAs technology mapping are library-based approach [3, 4] and direct approach [5–11]. The objective is to minimize the total number of blocks required, the maximum length of the time-critical paths, or the power dissipation of the resulting circuit. A detailed survey of synthesis methods for EPGAs can be found in [12].

Algorithms for technology mapping were pioneered by Keutzer [13], Rudell [14], and Detjens [15]. Mailhot [7] proposed a framework of technology mapping consisting of three major tasks:

- **Partitioning:** Partition a network into an interconnection of single-output sub-networks.
- **Decomposition:** Decompose each sub-network into an interconnection of two-input functions.
- **Covering:** Cover each decomposed network by library cells (e.g., by committed modules of EPGAs).

In this paper, we develop a one-phase technology mapping algorithm which combines decomposition phase and covering phase. The reason of combining these two phases is that decomposition actions can be adjusted directly by the actions of covering. Assume that a given Boolean network is modeled by a directed acyclic graph (DAG), called *subject graph*. The matching method of our algorithm is based on checking whether a subgraph of the subject graph can be matched by an uncommitted logic module, that is, on programming the inputs of a logic module to match the function associated with the subgraph. The programming techniques include assigning stuck-at faults to and/or bridging some inputs of an uncommitted logic module. We use an *extended GBDD hash table* extended from the structure called *Global Boolean Decision Diagram* (GBDD) [10] to do the matching. The benefits of our structure are:

- The matching operation is very fast.
- The matching operation owns the capability of a full library description.
- This structure supports generic EPGAs using fuse/anti-fuse technology.

The paper is organized as follows: Section 2 describes the matching algorithm used in our technology mapping approach. The one-phase technology mapping is presented in Section 3. Results on a set of benchmarks are reported in Section 4. And conclusion is drawn in Section 5.

## 2. The Matching Algorithm.

The major work of matching is to check whether the function of a subgraph in the subject graph, referred to as *cluster function*, can be implemented by personalizing the logic module of EPGA [7, 10]. The matching problem can be described as follows:

†This work was supported by the National Science Council R.O.C under Grant NSC 84-2215-E002-027.

*Personalize a logic module to match a cluster function by using a set of stuck-at faults, a set of bridging faults, and an input variable assignment.*

For example, from the uncommitted module ACT-1 as shown in Fig. 1, its module function $G = (a + b)(ce + \bar{c}f) + \overline{(a + b)}(dg + \bar{d}h)$ can be personalized to implement the cluster function $xy + \bar{x}\bar{y}$ by making the input connections $a = x$, $b = 0$, $c = y$, $d = y$, $e = 1$, $f = 0$, $g = 0$, and $h = 1$. That is, in this case, we set **Stuck_at_0** $= \{b, f, g\}$, **Stuck_at_1** $= \{e, h\}$, **Bridge** $= \{\{c, d\}\}$, and assign variable $a$ with $x$, variable $c$ with $y$. Where **Stuck_at_0** represents the set of input variables having stuck-at 0 fault, **Stuck_at_1** represents the set of input variables having stuck-at 1 fault, and **Bridge** represents the set of input variable subsets where variables in each subset are bridged together.

## 2.1. Matching Using BDD.

*Binary Decision Diagrams* (BDDs) have been used to represent Boolean functions [16, 17]. A BDD is a two-terminal DAG with a single root node, where terminal nodes are used to represent the values 0 and 1, non-terminal nodes to represent Boolean subfunctions, and root to denote the Boolean function of this BDD. Each non-terminal node has an associated Boolean variable and two outgoing edges, labeled 0 and 1. The Boolean subfunction represented by a non-terminal node is specified by its cofactors with respect to its associated Boolean variable. If the sequence of variables along any path in a BDD is restricted to a given precedence order and if no isomorphic subgraphs exist within the same BDD, then the result is a canonical form known as a *Reduced Ordered* BDD (ROBDD) [17].

Let $BDD_1$ be a ROBDD representing the Boolean function of a logic module, $BDD_2$ be a ROBDD representing the cluster function. If a subgraph of $BDD_1$ is isomorphic to $BDD_2$, they represent the same Boolean function. That is, the logic module can be personalized to implement the cluster function. To simplify description, we assume that a dummy module THREE-AND having a module function of $(ab + cd + ef)$ is given, and that an input ordering of $(a, b, c, d, e, f)$ is used to construct the BDD of a THREE-AND module. In Fig. 2, since the BDD of a given cluster function $xy$ is isomorphic to a subgraph in the BDD of a THREE-AND module, the THREE-AND module can be personalized to implement this cluster function by making stuck-at faults $a = 0$ and $c = 0$, and input assignments $e = x$ and $f = y$. Note that the assignments of inputs $b$ and $d$ do not matter.

Unfortunately, the BDD structure of a module function depends on its input ordering, therefore, most of time we need to consider all different input orderings (even though some input orderings may construct the same BDD).

The consideration of bridging faults can also be done by modifying the BDD of a module function to reflect the effect of bridging faults, then by detecting whether there exists a subgraph isomorphic to the BDD of a cluster function. Figure 3 shows the edges of a BDD that need to be adjusted by bridging two adjacent variables $a$ and $b$ together (adjacent variables means that their ordering positions are adjacent) and the result after adjusting edges, respectively. In Fig. 3, let the Boolean function represented by the BDD be $F$. We have $F = \bar{a}(\bar{b}F_{\overline{a}\overline{b}} + bF_{\bar{a}b}) + a(\bar{b}F_{a\bar{b}} + bF_{ab})$, where $F_{\overline{a}\overline{b}}$, $F_{\bar{a}b}$, $F_{a\bar{b}}$, and $F_{ab}$ are the cofactors of $F$ with respect to the variables $a$ and $b$. The dashed curves in Fig. 3 mean that whenever $a$ is 0, $b$ is 0 and whenever $a$ is 1, $b$ is 1. In a similar way, the edges which need to be adjusted by bridging three adjacent variables are shown in Fig. 4.

After adjusting the edges as described above, it also needs to reduce the BDD again. We give an example by bridging $b$, $c$, and $d$ together from the module function BDD of Fig. 2. The results after edges adjusting and BDD reducing are shown in Fig. 5. Note that the node associated with variable $a$ and the node associated with variable $b$ disappear after reducing.

To detect matching by bridging faults, we must consider all possible bridge sets. But if we process all different BDDs of a module function which are constructed by considering all the input permutations, then we need just to consider the bridge sets which contain only adjacent variables.

## 2.2. Matching Using Extended GBDD Hash Table.

From above discussion, we know that it would take a lot of time to detect one matching if the basic BDD structure and direct approach are used. Below, we will show how to construct a data structure, called *extended GBDD hash table*, from the idea of GBDD [10], which can be used to detect matching quickly and thus is very suitable to the matching problem on EPGAs.

The GBDD was defined in [10] as follows:

*A GBDD is a two-terminal DAG with k roots, whose subgraph induced by the nodes reachable from each root node is a BDD of the function for some variable ordering.*

The GBDD for a module function can be constructed by performing a *reduce* [17] operation on all its BDDs. Therefore, any two subgraphs in the GBDD will not be isomorphic to each other, and the number of roots is equal to the number of different BDDs. Since the GBDD shares common subgraphs, it saves time from not checking isomorphism between those isomorphic subgraphs and cluster BDD redundantly. For ACT-1 and ACT-2 modules, the number of nodes needed to be maintained in GBDDs structures are controllable.

Recall that the ROBDD is a canonical form to represent a Boolean function. There are no two subgraphs isomorphic to each other in a ROBDD. Therefore, each node in a ROBDD represents a unique Boolean function. Given a non-terminal node $v$ in a ROBDD, let $index(v)$ denote the index corresponding to the variable which is associated with node $v$ (the index of a variable means the ordering position of this variable), $low(v)$ denote the child of $v$ connected by edge 0, and $high(v)$ denote the child of $v$ connected by edge

1 [17]. Thus any node $v$ can be denoted by a triplet (index($v$), low($v$), high($v$)). Using the triplet information, we can construct a hash table for nodes in this ROBDD, referred to as *unique table* [18]. Similarly, we can construct a hash table for nodes in a GBDD.

Furthermore, we can combine the operation of bridging faults into the hash table. As stated in the previous section, we can apply bridging operation (just consider adjacent variables) on BDD by adjusting edges and reducing the BDD again. Following shows the algorithm of constructing an extended GBDD hash table.

```
Generate_Extended_GlobalBdd_Table( G )
{  Initiate_HashTable(TABLE);
   for (each ordering φ of input permutation in the given
   function G)
   {  BDD_G = Generate_Bdd(G, φ);
      Map_Bdd_To_HashTable(BDD_G, TABLE);
      for (each feasible adjacent bridge set Δ in function G)
      {  New_BDD_G = Bridge_Bdd(BDD_G, Δ);
         Map_Bdd_To_HashTable(New_BDD_G, TABLE);
      }
   }
}
```

In Algorithm *Generate_Extended_GlobalBdd_Table*, *Initiate_HashTable* sets some initial values for the extended GBDD hash table. Procedure *Generate_Bdd* generates the BDD when giving a Boolean function $G$ and one of its input orderings $\varphi$. Then procedure *Map_Bdd_To_HashTable* records nodes in this BDD into the hash table. Note that some nodes of this BDD may have been in the hash table before recording, for they represent the same Boolean functions. Also, procedure *Bridge_Bdd* applies bridging operation on this BDD using certain kind of bridge sets and returns the bridged BDD. There are 106 different kinds of feasible adjacent bridge sets for an ACT-1 module function. Using the above algorithm, an extended GBDD hash table needs only be constructed once at the beginning for one type of logic module and is saved as a data file for further reference. Then the file will be loaded whenever technology mapping is called. Following shows the matching algorithm which uses the hash table to detect whether a cluster function $F$ can be matched by a module function $G$.

```
Matching( F, G )
{  BDD_F = Generate_Bdd(F, ψ);
   /* ψ is a certain input ordering of F */
   difference_index = | sup(G) | − | sup(F) |;
   hash_id = BDD_Hashing(root(BDD_F), difference_index);
   if (hash_id is valid)
      return(TRUE);
   else
      return(FALSE);
}
```

In the above algorithm, firstly, we generate a BDD for a cluster function $F$ with any input ordering $\psi$. Then we use procedure *BDD_Hashing* to detect whether this BDD is contained in the hash table, and to return a valid hashing *id* if it is. There are also other function and variables needed to be described. Function *root*(BDD_F) returns the root node of BDD_F. | *sup*($G$) | and | *sup*($F$) | denote the number of input variables corresponding to a module function $G$ and to a cluster function $F$, respectively. Variable *difference_index* is used to store the difference value between | *sup*($G$) | and | *sup*($F$) |. As to the procedure *BDD_Hashing* which is the main part of above algorithm, it will be explained below.

```
BDD_Hashing( v, difference) /* v is a node of a BDD */
{  if (v is a terminal node)
      return(the hashing id of this terminal node);
      /* terminal 0 and terminal 1 have public hashing ids */
   low_id = BDD_Hashing(low(v), difference);
   high_id = BDD_Hashing(high(v), difference);
   if (either low_id or high_id are unvalid)
      return(NULL);
   found_id = Hashing(index(v) + difference, low_id, high_id)
   if (found_id is valid)
      return(found_id);
   else
      return(NULL);
}
```

The procedure *BDD_Hashing* detects whether a BDD is in the hash table by calling itself recursively to detect all its nodes in a bottom-up manner. For each node $v$, its triplet (index($v$), low_id, high_id) forms an *id* value in the hash table. After we have generated the extended GBDD hash table, each node generated from the module function $G$ has its *id* value recorded in the hash table. Thus, when we call the procedure *Hashing* for a node $v$ with its triplet (index($v$), low_id, high_id), we can obtain its *id* value. This *Hashing* procedure will return a real *id* value if the node $v$ is contained in the hash table. Furthermore, since the

extended GBDD hash table is generated by the module function $G$ which may have more variables than $F$, we need a value of *difference_index* to adjust the index($v$) when calling *Hashing* for a node $v$ belonging to a BDD of the cluster function $F$. For example, in Fig. 2, since variables $a$, $b$, $\cdots$, $f$ of the module function have index values 1, 2, $\cdots$, 6, and variables $x$ and $y$ of the cluster function have index values 1 and 2, respectively; the value of *difference_index* in this case is 4. Note that node $f$ and node $y$ represent the same function. When we want to detect whether this cluster function exists in the GBDD, we need adjust the index values of $y$ and $x$ to 6 and 5, respectively. Then we can find that $y$ is isomorphic to $f$ and $x$ is isomorphic to $e$.

The extended GBDD hash table constructed above can be used to play the role of matching in the technology mapping phase which will be described in the following section.

## 3. One-Phase Technology Mapping.

We present a new approach to technology mapping, which combines decomposition phase and covering phase. The input Boolean network used here is an AND-OR network, i.e., each node in the network is one of AND gate, OR gate, or inverter, and this Boolean network is modeled by a directed acyclic graph (DAG).

Given a DAG $G$, a *supernode* is a sub-DAG which just contains one node with 0 outdegree. The definition of covering phase is to cover the DAG with supernodes, where each supernode can be implemented by one logic module. In our approach, we cover the DAG bottom-up as follows: Whenever we process a node $t$, denoted as *target node*, all nodes on the fanin paths of $t$ should have been processed. Since a target node $t$ may have many fanins of supernodes, we need to decompose this $t$ before covering it.

Two types of fanins occur when processing a node $t$: (1) *filled* supernodes which are not allowed to collapse any more gate to $t$ and (2) *nonfilled* supernodes that are directly connected to $t$ and have the possibility of being collapsed into this node $t$. Both type of supernodes are matchable, i.e. can be matched by logic modules, but just the filled supernodes should be mapped, i.e. they should be assigned by logic modules. These filled supernodes are seen as leaf nodes, because after they were mapped by logic modules, they become the inputs of other logic modules. Since a supernode may have more than one fanout, a supernode which collapses itself into more than one fanout node implies that we must duplicate it in the Boolean network. Owing to duplication will reduce the ability of further collapses, it is not beneficial to duplicate a node with a large number of fanout nodes. Besides that, it is also disadvantageous to duplicate a node with a large number of fanins. A threshold, 12, is exploited to determine whether a supernode should be duplicated. If the product of the number of fanouts and the number of fanins is larger than the above threshold, this supernode becomes a filled type. Otherwise, this supernode must be duplicated and collapsed into its fanout nodes respectively.

The main task of our technology mapping is on how to decompose node in a DAG and to select nonfilled supernodes to be collapsed into this node. We define two cost functions for the above purpose. Let $v$ be a nonfilled supernode and a fanin of the target node $t$.

$$d\_cost(v) = (1-\alpha)\cdot\frac{m-N_{min}+1}{N_{max}-N_{min}+1} + \alpha\cdot\frac{l-L_{min}+1}{L_{max}-L_{min}+1}$$

$$c\_cost(v) = (1-\alpha)\cdot m - \alpha\cdot l$$

where $\alpha$ is a factor weighting between area and depth, $m$ means number of fanins at supernode $v$, $N_{max}$ is the largest cardinality of fanins among all supernodes which fanout to the target node $t$, $N_{min}$ is the smallest cardinality, $l$ means the height at $v$, and $L_{max}$ and $L_{min}$ are the maximum height and the minimum height respectively at all fanin supernodes of $t$. Two cases of target nodes would be processed: (1) For each target node $t$ with more than two inputs, the decomposing cost, $d\_cost(v)$, of each input supernode $v$ is calculated and, then, all input supernodes are sorted according to their decomposing costs. A new two-input gate is split from $t$ and two supernodes with the least decomposing costs are chosen to be fed into this new gate. (2) For each target node $t$ with two inputs, the collapsing cost, $c\_cost(v)$, of each input supernode $v$ is computed and, then, the nonfilled supernode with lower collapsing cost is first tried to be collapsed.

The first term of each cost function is used to minimize the number of logic blocks required. The less the cardinality a supernode $v$ has, the more easily it is to be collapsed. The second term is used to minimize the total depth of a final circuit. During decomposition, in order to increase the height of $t$ as few as possible, the supernode with lower height is preferred. On the contrary, the nonfilled supernode with higher level has higher priority to be selected during collapsing. The cases of $\alpha = 1$ and $\alpha = 0$ are equivalent to simply perform the depth minimization and the area optimization, respectively. The tradeoff between area and depth is done by selecting an appropriate $\alpha$. Figure 6 shows an example of our approach. In Fig. 6(a),

the lowest two nonfilled supernodes, $x$ and $y$, are selected and removed from the input list of $v$. They are connected to inputs of a new target node $t$ as shown in Fig. 6(b). The selected nonfilled supernodes, $x$ and $y$ in Fig. 6(b), are tried to be collapsed themselves into a target supernode $t$ according to their individual *c_cost*. Figure 6(c) shows the result after processing node $t$.

## 4. Experimental Results.

Our approach has been implemented in C language and has been tested on the MCNC and ISCAS benchmarks. The ACT-1 was used as the EPGA module. We compared the performance of our mapper with MIS-pga1 [5] incorporated in SIS version 1.2. All benchmarks were optimized with "script.algebraic" first. The option used for MIS-pga1 is "act_map -h3 -n1 -q -d4 -f3 -M4 -l -g0.001". The results are shown in Table 1, where BC is the number of modules, LC is the depth, and T is the run time. We also listed the number of modules (BC) generated by Proserpine [10] for reference. The average number of modules obtained by MIS-pga1 is 4.7% more than ours and its average level is 21% more than ours. Our mapper is very fast compared with MIS-pga1, its running time is 155 times faster than MIS-pga1.

## 5. Conclusions and Future Reseach.

We have presented an efficient one-phase technology mapping approach for electrically programmable gate arrays. The algorithm uses extended GBDD hash table to do matching by determining how the set of input variables are assigned with stuck at 0/1 faults or are bridged together. The matching algorithm is very fast and has the ability to capture the entire family of functions that a logic module (ACT-1 or ACT-2) can generate. Also, this approach can be applied to any other type of modules with single-output. Since our matching operation is fast, we can develop a more sophisticated technology mapper without taking too much time.

### References


[1] A. El Gamal, J. Reynery, E. Rogoyski, K. A. El Ayat, and A. Mohsen, "Architecture for Electrically Configurable Gate Arrays", *IEEE Journal of Solid-State Circuits*, Vol. 24, No. 2, April 1989, pp. 394–398.

[2] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

[3] M. Mehendale, "MIM: Logic Module Independent Technology Mapping for Design and Evaluation of Antifuse-Based FPGAs", *Proc. 30th ACM/IEEE Design Automation Conference*, 1993, pp. 219–223.

[4] M. Mehendale, C. H. Shaw, and D. Wilmoth, "ALFA: Automatic library generation for logic module based FPGA's", *Proc. 1st Int. ACM/SIGDA Workshop on FPGAs*, 1992.

[5] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays", *Proc. 27th ACM/IEEE Design Automation Conference*, 1990, pp. 620–625.

[6] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli, "An Improved Synthesis Algorithm for Multiplexer-Based FPGA's", *Proc. 29th ACM/IEEE Design Automation Conference*, 1992, pp. 380–386.

[7] F. Mailhot and G. De Micheli, "Technology Mapping Using Boolean Matching and Don't Care Sets", *Proc. IEEE European Conference on Design Automation*, 1990, pp.212–216.

[8] F. Mailhot and G. De Micheli, "Algorithms for Technology Mapping Based on Binary Decision Diagrams and on Boolean Operations", *IEEE Trans. Computer-Aided Design*, Vol. 12, No. 5, May 1993, pp. 599–620.

[9] K. Karplus, "Amap: A Technology Mapper for Selector-Based Field-Programmable Gate Arrays", *Proc. 28th ACM/IEEE Design Automation Conference*, 1991, pp. 244–247.

[10] S. Ercolani and G. De Micheli, "Technology Mapping for Electrically Programmable Gate Arrays", *Proc. 28th ACM/IEEE Design Automation Conference*, 1991, pp. 234–239.

[11] J. R. Burch and D. E. Long, "Efficient Boolean Function Matching", *Proc. IEEE International Conference on Computer-Aided Design*, 1992, pp. 408–411.

[12] A. Sangiovanni-Vincentelli, A. El Gamal, and J. Rose, "Synthesis Methods for Field Programmable Gate Arrays", *Proceedings of IEEE*, Vol. 81, No. 7, July 1993, pp. 1057–1083.

[13] K. Keutzer, "Dagon: Technology Binding and Local Optimization by DAG Matching", *Proc. 24th ACM/IEEE Design Automation Conference*, 1987, pp. 341–347.

[14] R. Rudell, *Logic Synthesis for VLSI Design*, PhD thesis, U.C. Berkeley, April 1989, and Memorandum UCB/ERL M89/49.

[15] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "Technology Mapping in MIS", *Proc. IEEE International Conference on Computer-Aided Design*, 1987, pp. 116–119.

[16] S. B. Akers, "Binary Decision Diagrams", *IEEE Trans. Computers*, June 1978, pp. 509–516.

[17] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. Computers*, Vol. C-35, No. 8, August 1986, pp. 677–691.

[18] K. Brace, R. Rudell, and R. Bryant, "Efficient Implementation of a BDD package", *Proc. 27th ACM/IEEE Design Automation Conference*, Orlando, June 1990, pp.40–45.

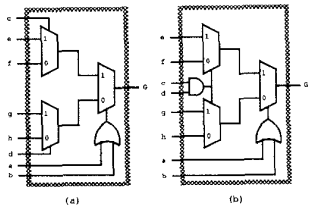

Figure 1: Architectures of Actel logic modules. (a) ACT-1. (b) ACT-2.

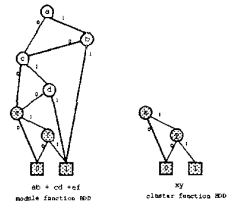

Figure 2: For the chosen ordering, the BDD of a module function has one subgraph which is isomorphic to the BDD of a cluster function.

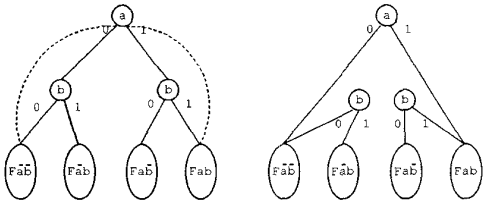

Figure 3: Edges to be adjusted in a BDD when $a$ and $b$ are bridged together (dashed curves represent new edges).

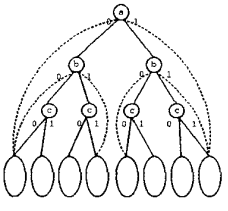

Figure 4: Edges to be adjusted in a BDD when $a$, $b$, and $c$ are bridged together (dashed curves represent new edges).

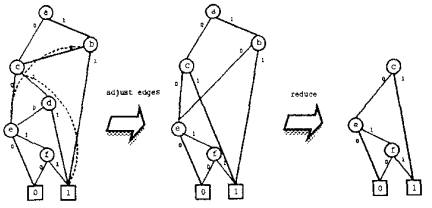

Figure 5: Bridging $b$, $c$, and $d$ together from the BDD of Figure 2.

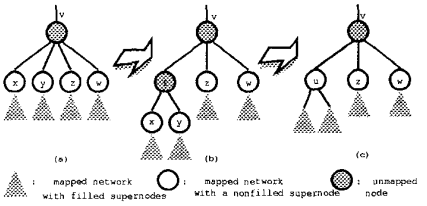

Figure 6: Combine decomposition and covering into a single phase.

Table 1: Experimental results of area optimization using *Act*-1 logic modules.

| Example | MIS-pga1 | | | one-phase | | | Proser-pine |
|---|---|---|---|---|---|---|---|
| | BC | LC | T | BC | LC | T | BC |
| misex1 | 21 | 9 | 20.8 | 16 | 9 | 0.4 | 25 |
| vg2 | 37 | 9 | 25.5 | 32 | 7 | 0.6 | 46 |
| 5xp1 | 53 | 16 | 63.3 | 45 | 14 | 0.7 | 53 |
| apex7 | 85 | 10 | 117.5 | 95 | 9 | 0.9 | 121 |
| duke2 | 166 | 16 | 532.1 | 178 | 13 | 1.5 | 177 |
| C499 | 168 | 18 | 109.0 | 176 | 15 | 1.5 | 170 |
| rot | 289 | 22 | 267.6 | 283 | 21 | 2.0 | 465 |
| apex6 | 321 | 12 | 443.8 | 302 | 12 | 2.3 | 465 |
| f51m | 60 | 25 | 42.7 | 46 | 19 | 0.7 | 63 |
| clip | 70 | 17 | 107.4 | 58 | 15 | 0.8 | 73 |
| bw | 98 | 9 | 101.2 | 101 | 8 | 1.0 | 67 |
| z4ml | 15 | 8 | 24.5 | 16 | 8 | 0.5 | - |
| count | 43 | 8 | 64.5 | 41 | 7 | 0.7 | - |
| 9symml | 96 | 12 | 109.5 | 78 | 14 | 1.0 | - |
| C880 | 158 | 18 | 179.8 | 160 | 19 | 1.3 | - |
| alu2 | 203 | 34 | 387.3 | 181 | 25 | 1.6 | - |
| sao2 | 65 | 20 | 92.8 | 48 | 16 | 0.7 | - |
| rd73 | 29 | 12 | 40.3 | 30 | 11 | 0.6 | - |
| b9 | 69 | 9 | 59.3 | 60 | 8 | 0.7 | - |
| C5315 | 650 | 20 | 1047.7 | 629 | 17 | 5.1 | - |
| Ratio | +4.7% | +21% | 154 | 1 | 1 | 1 | - |