Technology Mapping for Electrically Programmable Gate Arrays

Silvia Ercolani and Giovanni De Micheli

Center for Integrated Systems
Stanford University
Stanford, CA 94305

Abstract

We describe a new approach for technology mapping of electrically programmable gate arrays (EPGAs). These are arrays of uncommitted modules, where the personalization is achieved by fuse/antifuse technology and can be modeled by stuck-at and/or bridging inputs. We present a matching algorithm that determines whether a portion of a combinational logic circuit can be implemented by personalizing a module. The algorithm has the advantage of considering the entire library of functions that can be implemented by the module without resorting to an explicit enumeration. The benefits are an increased efficiency in technology mapping, as well as portability to different types of electrically programmable gate arrays. Experimental results on standard benchmarks are reported.

1 Introduction

There has been an increasing interest in digital-system prototyping using *Electrically Programmable Gate Arrays* (EPGAs) due to their fast turn-around time and low manufacturing costs. One class of EPGAs uses anti-fuse technology, where logic gates and their interconnections are programmed by shorting wire segments in prescribed locations [1].

An EPGA consists of repeated arrays of identical logic modules. Each module is a multiple-input single-output combinational logic gate. A module can be configured to implement a logic function by forcing any input to logic low or logic high or by bridging inputs [1].

System design with EPGAs requires specific logic design tools. In particular, *technology mapping* is crucial for achieving an efficient implementation. Technology mapping is the process of transforming a set of logic equations into an interconnection of parts that are instances of the elements in a given library. In the case of EPGAs, the "library" consists of the set of combinational logic gates that can be derived from the uncommitted module.

Existing approaches to technology mapping include algorithms and tools that support an explicit arbitrary library definition, such as *MisII* [3] and *Ceres* [4]. In this case, the library of cells that can be derived from the uncommitted module needs to be derived explicitly. Since the enumeration of the library cells may be long, a subset of the library may be used to increase the mapping speed at the expense of the quality.

The *Mis_pga* program [2] is a specialized technology mapper for the EPGAs fabricated by Actel Inc. This program exploits the particular structure of the uncommitted module, based on multiplexers, in the mapping process. As a result, the program can model the entire library based on a few primitive building blocks. Unfortunately, the algorithms of *Mis_pga* support only multiplexer-based EPGAs.

This paper describes a new approach to technology mapping for EPGAs that:

- Does not require an explicit library enumeration.
- Supports generic EPGAs based on fuse/anti-fuse technology. The uncommitted module is assumed to be an arbitrary single-output combinational function.

The full library description is replaced by using the description of the uncommitted module only. The technology mapping algorithms check whether a given logic function can be implemented by programming the uncommitted module. Indeed, the process of personalizing an uncommitted module can be modeled by creating some *input stuck-at 0*, *input stuck-at 1* or *input bridging* faults on the uncommitted module. Therefore, the EPGA library can be represented by the set of equivalent gates that these faults induce on the uncommitted module.

As a result, any EPGA can be used as a target architecture by just describing the logic function of the uncommitted module. In this paper we use two EPGA modules developed at Actel Inc. as examples. The modules are called $act1$ and $act2$, and their logic diagram is shown in Fig. 1.

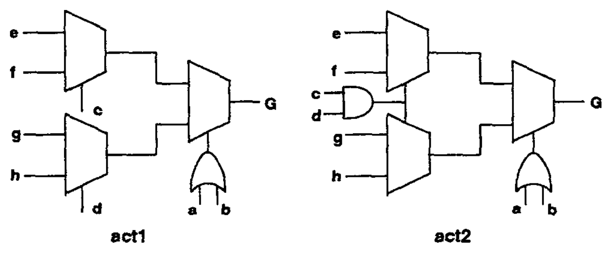

Figure 1: EPGAs uncommitted modules developed at Actel Inc.

The ease of retargeting the implementation to different EPGAs with different uncommitted modules is extremely important in validating the choice of modules. So far, uncommitted modules have been chosen by looking at the circuit configurations and performance. No analysis of the ease of mapping has been done, as in the case of Field Programmable Gate Arrays (FPGAs) [5]. An efficient technology mapper can be useful in researching module primitives that support efficient implementations.

This paper is organized as follows. First we describe the overall organization of the technology mapping program in Section 2. We then focus on the crucial problem in EPGA mapping, called *matching*, that consists of recognizing whether a logic function can be implemented by programming the uncommitted module. This is the major contribution of the paper. We show matching algorithms that use a generalization of Binary Decision Diagrams (BDDs) in Section 3. We then describe our implementation and report on experimental results and comparisons (Section 4). Finally, extensions and future directions are presented in Section 5.

2 Technology mapping

Algorithms for technology mapping were pioneered by Keutzer [6], Rudell [7] and Detjens [8]. Similarly to their approach, we use a heuristic method based on three different tasks:

- **Partitioning.** Partition a network into a collection of multiple-input single-output combinational sub-networks.
- **Decomposition.** Decompose each sub-network into two-input functions, to increase the network granularity.
- **Covering.** Cover each decomposed network by committed modules so that either area or delay is optimized.

Standard techniques are used for the first two tasks [6, 7, 8]. The covering algorithm is borrowed from Mailhot [4]. It uses the notion of *cluster* and *cluster function*. After partitioning and decomposition, the sub-network to be mapped is represented by a directed acyclic graph. The nodes of this graph correspond to the sub-network's inputs, output and two-input functions, and its edges represent the dependencies. A *cluster* is a connected subgraph having only one node with zero out-degree. The associated *cluster function* is the Boolean function obtained by collapsing the Boolean expressions associated with the nodes into a single Boolean function [3].

As an example [4], consider the Boolean sub-network of Fig. 2 that, after decomposition, is described by:

$$\begin{aligned} f &= j + t \\ j &= xy \\ x &= e + z \\ y &= a + c \\ z &= \bar{c} + d \end{aligned}$$

With each of the variables is associated one or more clusters, *e.g.* there are six possible *cluster functions* associated with variable j, due to the increased depth in the collapsing step:

$$\begin{aligned} \kappa_{j,1} &= xy \\ \kappa_{j,2} &= x(a+c) \\ \kappa_{j,3} &= (e+z)y \\ \kappa_{j,4} &= (e+z)(a+c) \\ \kappa_{j,5} &= (e+\bar{c}+d)y \\ \kappa_{j,6} &= (e+\bar{c}+d)(a+c) \end{aligned}$$

The covering algorithm selects clusters repeatedly and attempts to match them to modules. A cover of a sub-network is a set

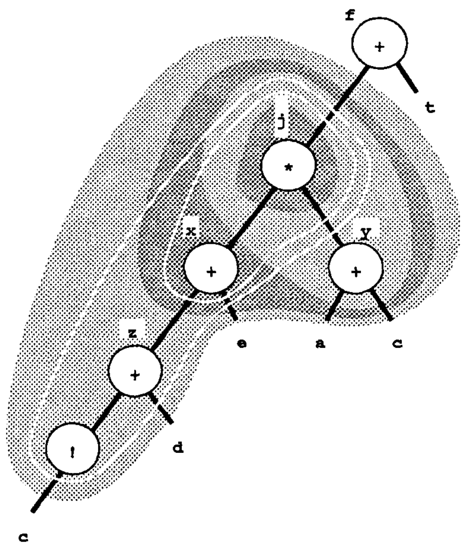

Figure 2: Clusters and cluster functions of variable j.

of clusters that are matched to modules and that cover the sub-network. A covering may optimize the overall area or timing cost. Each module has unit cost, because committed modules have the same area and similar delay.

The area cost of a cover is computed by adding one to the cost of the covers of the sub-graphs corresponding to the support variables in the Boolean cluster function, while the timing cost is obtained by adding one to the maximum of the cost. When matchings exist for multiple clusters, for any given decomposition with tree-structure, then the choice of the matching of minimal area (timing) cost guarantees minimality of the total area (time) cost of the matched sub-network [6, 8].

The technology mapping algorithm described here, which is dedicated to EPGAs, differs from library-based mapping in the *matching* step. The general framework is based on program *Ceres* [4]. This paper focuses on the matching step and describes it in detail.

3 The Matching Algorithm

The algorithm presented here performs a run-time customization of the EPGA module: by comparing the cluster and the module functions, it determines which module inputs should be set to 0/1, which should be bridged together, and which input ordering, if any, makes the module implement the cluster function. If no match is found, the algorithm returns that no matching exists and another cluster function is tried.

Both the uncommitted module of an EPGA and the cluster function are represented by their Boolean functions, denoted by $G(x_1, x_2, \ldots, x_n)$ and $F(x_1, x_2, \ldots x_m)$ respectively. The set of corresponding input variables is $sup(G)$ and $sup(F)$.

We call ***stuck-at set*** of the module the set of input variables that are set to 0/1 and we denote it by: $S = S0 \cup S1$. Note that $S0 \cap S1 = \emptyset$. We call ***bridge set*** the set of input variable subsets that are bridged together and we denote it by $B = \cup_j B_j$. Since bridging is transitive, $B_j \cap B_k = \emptyset$, $\forall j \neq k$.

We define G_S the function obtained from G by setting each variable $x_i \in S$ to 0/1. Similarly we define G_{SB} to be the function obtained from G_S by by bridging the inputs corresponding to variables $x_i \in B_j$, $\forall B_j$. Thus, given S and B, the function G_{SB}

is uniquely identified. The cardinality of the set of independent input variables R of G_{SB} representing the committed module is: $|R| = |sup(G_{SB})| = n - |S| - \sum_j (|B_j| - 1)$, where $n = |sup(G)|$.

As an example, in the case of the uncommitted module $act1$ of Fig. 1 $G = (a+b)(ce + \overline{c}f) + \overline{(a+b)}(dg + \overline{d}h)$ and $sup(G) = \{a, b, c, d, e, f, g, h\}$. Suppose that $S = S0 = \{e, f = 0\}$ and $B = B_1 \cup B_2 = \{a, b\} \cup \{d, g\}$, by substitution we have $G_{SB} = \overline{a}(d+h)$ and $R = \{a, d, h\}$.

We define the matching problem as follows:

Given a cluster function $F(y_1, \ldots, y_m)$, *and the module function* $G(x_1, \ldots, x_n)$ $m \leq n$ *find a stuck-at set* S, *a bridge set* B *and an ordering* $\Omega(R)$ *such that:* $F = G_{SB}(\Omega(R))$ *is a tautology.*

Note that a necessary condition for matching is that the cluster function and the committed module must have the same number of inputs, i.e. $|R| = m$ or, equivalently, $m = n - |S| - \sum_j (|B_j| - 1)$.

3.1 The simplified matching problem

For the sake of explaining the matching algorithm's steps, we consider first a simplified matching problem for EPGAs, in which bridging is not allowed.

The simplified matching problem can be stated as follows:

Given a cluster function $F(y_1, \ldots, y_m)$, $m \leq n$ *and the module function* $G(x_1, \ldots, x_n)$ *find a stuck-at set* S, *and an ordering* $\Omega(R)$ *such that:* $F = G_S(\Omega(R))$ *is a tautology.*

In this case, a necessary condition for matching is that the inputs to the cluster function equal in number the unstuck inputs of the committed module, i.e. $m = n - |S|$.

This problem can be solved by using Binary Decision Diagrams (BDDs) to represent Boolean functions [9, 10, 11]. A BDD is a directed acyclic graph that is leveled, each level being associated with a variable. Every node is associated with a function and has two outgoing edges, labeled 1 and 0. The root is associated with the Boolean function that the BDD represents. An internal node is associated with the sum of the co-factors with respect to the variables on the paths to the root, with the phase defined by the edge weights.

Unfortunately, a function does not have a unique BDD, since the structure of the BDD depends on the ordering of the input variables used to levelize the graph. However, given an input ordering, the reduced BDD [10] (*i.e.* a BDD such that no two sub_BDDs are isomorphic to each other) is a canonical form [10].

Sub-isomorphism between the cluster function and the module function BDDs can be used to detect matching. Consider for example the module with $G = a(bc + b'd) + a'(fe + f'g)$ and $n = 7$. Fig. 3 shows its BDD for the alphabetical input ordering (a, b, c, d, e, f, g). Suppose that the cluster function is the 2-input multiplexer $F = xy + x'z$, whose BDD is shown for the alphabetical ordering (x, y, z) in Fig. 3. Since the latter BDD is isomorphic to a sub-graph of the former one, the module can be committed to perform the cluster function, by selecting as set $S = S1 = \{a = 1\}$.

Unfortunately, matching is more complicated than performing a sub-isomorphism check. Indeed, in our case we do not question the equivalence between two functions, but their equivalence after a pin assignment (or variable ordering), which we call *P-equivalence.* Therefore we must be able to recognize matching corresponding to different variable orderings, and therefore to different BDD structures.

Consider for example the same module function G of Fig. 3, with a different input ordering (a, d, b, c, e, f, g) whose corresponding BDD is shown in Fig. 4. There is no sub-isomorphism between this BDD and the BDD of the cluster function, even though a matching exists for the previous input ordering of G.

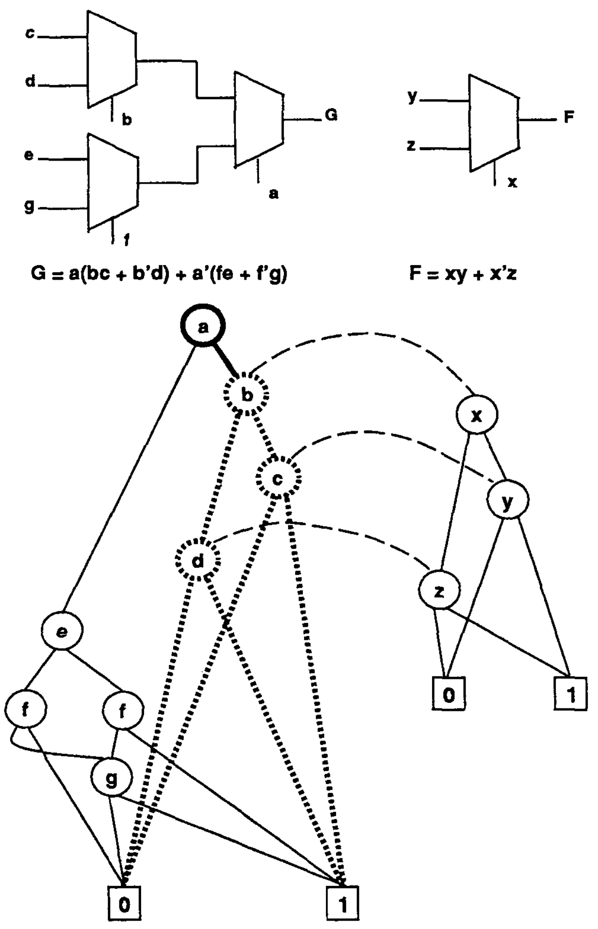

Figure 3: For the chosen orderings, the BDD of G has one subgraph (for $a = 1$) which is isomorphic to the BDD of F.

Therefore, all the BDDs corresponding to different variable orderings in the module function should be computed and checked for sub-isomorphism against the cluster function BDD, until a matching is found.

An algorithm for the simplified matching problem using BDDs is described here:

```
Matching_0( $G$, $F$ )
{
    /* generate the BDD for a given function and input ordering */
    BDD_$F$ = GenerateBdd( $F$, $\psi$ );    /* $\psi$ is the alphabetical ordering */
    for (each permutation $\varphi_i$ of $sup(G)$ ) {
        BDD_$G$ = GenerateBdd( $G$, $\varphi_i$ );
        if (SubIsomorph_0(BDD_$G$, BDD_$F$, $m$)) {    /* $m = |sup(F)|$ */
            $\Omega$ = DetermineOmega();
            $S$ = DetermineStuckatSet();
            return(TRUE);
        }
    }
    return(FALSE);
}
```

Procedure *SubIsomorph_0* inspects all the subgraphs of BDD_G of level m, until an isomorphism with BDD_F is found.

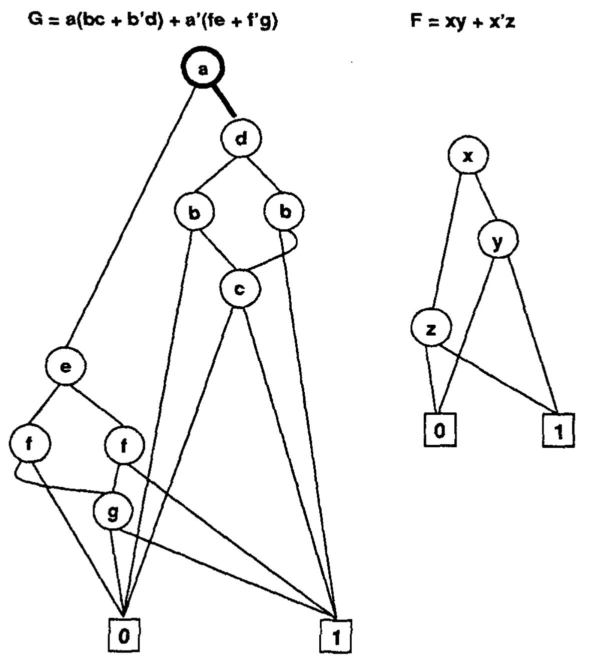

Figure 4: For this particular ordering, the BDD of G has no subgraph isomorphic to the BDD of F.

```
SubIsomorph_0( BDD_G, BDD_F, level )
{
        node_f = Root(BDD_F);
        for (each of the subgraphBDD_G of height m ) {
                node_g = Root(subgraphBDD_G);
                if (Isomorph(node_g, node_f)) return(TRUE);
        }
        return(FALSE);
}
```

Procedure *Isomorph* has been derived from Bryant's [10] BDD traverse procedure. Starting from the roots it inspects the branches until a terminal node is reached.

```
Isomorph( node_x, node_y)
{
        /* Level_Table and Node_Table keep track of the 1-to-1 */
        /* correspondence between the levels and the nodes of the two BDDs */
        Level_Table = InitializeLevelTable();
        Node_Table = InitializeNodeTable();

        if ((node_x == node_y == ZERO) || (node_x == node_y == ONE))
                return(TRUE);
        if (IsTerminal(node_x) || IsTerminal(node_y))
                return(FALSE);
        if (node_x and node_y are already mapped on each other)
                return(TRUE);

        if (level(node_x) and level(node_y) are unmapped)
                MapLevelsOntoEachOther(node_x, node_y, Level_Table);
        else if (level(node_x) and level(node_y) are not mapped on each other)
                return(FALSE);
        if (node_x and node_y are unmapped )
                MapNodesOntoEachOther(node_x, node_y, Node_Table);
        else if (node_x and node_y are not mapped on each other)
                return(FALSE);

        /* Inspect subtree */
        return(Isomorph(node_x->low, node_y->low) &&
                        Isomorph(node_x->high, node_y->high));
}
```

The *matching_0* algorithm compares the BDD of the cluster function (with an arbitrary input ordering) with the BDDs of the module function (using all possible orders). It is therefore guaranteed to find a match - if one exists - by setting to 1 or 0 the first $|S| = n - m$ variables in the order corresponding to the BDD in which a sub-isomorphism is detected. Procedures *DetermineOmega* and *DetermineStuckatSet* are straight-forward.

It is important to remark that comparing all the BDDs related to the cluster function against the subgraphs of the module BDD is not guaranteed to detect a matching, because the set of variables of G that need to be stuck may not be adjacent in the input ordering chosen for the BDD of the module cell, as it is shown in Fig. 5.

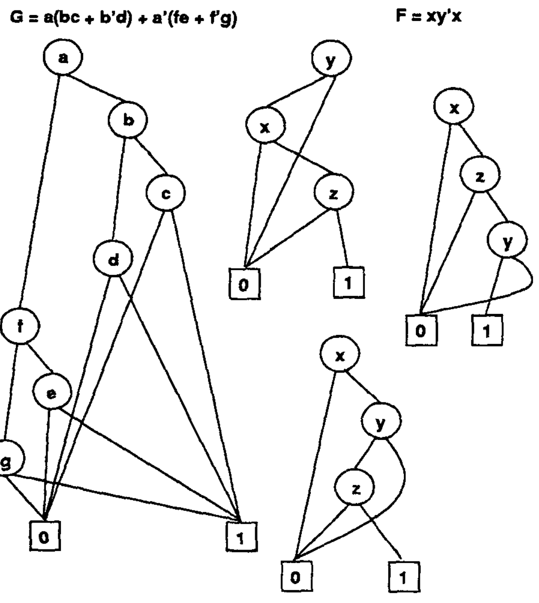

Figure 5: In this example, although F is functionally equivalent to G when $S = S0 = \{c, e, g = 0\}$, comparing all the different BDDs of F against that of G doesn't detect a matching.

3.2 Global Boolean Decision Diagrams

The complexity of algorithm *matching_0* is due to the inspection of all the BDDs of the module function, one for each input permutation. Actually, it is not necessary to explore all the BDDs, but only those BDDs that are different from one another. For a cell of n inputs there are $n!$ input permutations, but far fewer significant BDDs. Indeed, every symmetry between the input variables reduces by a factor 2 the number of non-isomorphic BDDs and then, due to some structural symmetry their number is further reduced.

Nevertheless, even if the input and structural symmetries of the module function decrease the number of BDDs to be inspected, their number is still very high. Table 1 shows the number of different BDDs for the module functions used in the Actel EPGAs $act1$ and $act2$ (Fig. 1).

EPGA module	Input permut.	Different BDDs	# of nodes for different BDDs	# of nodes for global BDD
act1	40320	11855	295361	39976
act2	40320	7470	217948	33996

Table 1: BDD statistics for modules $act1$ and $act2$.

To reduce the complexity of the algorithm we propose a new structure, called global BDD (GBDD) that gathers in a compact form the information of the different BDDs corresponding to the variable orderings of a function.

We define a GBDD as follows:

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

A GBDD is constructed as follows. Starting from the observation that different BDDs have subgraphs isomorphic to each other, we performed a *reduce* [10] on all the BDDs. The structure we obtain has as many roots as there are different BDDs. The number of nodes is much lower because there are no two subgraphs isomorph to each other.

```
GenerateGlobalBdd( $G$ )
{
    /* create all the different BDDs */
    for (each permutation $\varphi_i$ of $sup(G)$ ) {
        BDD_$G$ = GenerateBdd( $G$, $\varphi_i$ );
        if (BDD_$G$ is isomorph to any other previously generated BDD)
            Free(BDD_$G$);
    }
    GBDD_$G$ = Reduce(LinkedList_of_BDDs_of_$G$);
}
```

In this way, although we do not reduce the number of BDDs, we decrease the computational effort and the memory used by the program. Table 1 shows the total number of nodes for all the different BDDs against that of the GBDD for $act1$ and $act2$. Since the GBDD shares common subgraphs, we do not repeat the inspection of subgraphs that are isomorphic to each other, in the *SubIsomorph* procedure, thus improving the efficiency of the matching algorithm.

We show now the algorithm for the simplified matching problem using the GBDD.

```
Matching_1( $G$, $F$ )
{
    GBDD_$G$ = GenerateGlobalBdd( $G$ );
    BDD_$F$ = GenerateBdd( $F$, $\psi$ );
    if (SubIsomorph_1(GBDD_$G$, BDD_$F$, $m$)) {
        $\Omega$ = DetermineOmega();
        $S$ = DetermineStuckatSet();
        return(TRUE);
    }
    return(FALSE);
}
```

Procedure *SubIsomorph_1* inspects all the subgraphs of GBDD_G of level m, until an isomrphism with BDD_F is found.

```
SubIsomorph_1( GBDD_$G$, BDD_$F$, level )
{
    node_f = Root(BDD_$F$);
    for (each of the subgraphGBDD_$G$ of height $m$ ) {
        node_g = Root(subgraphGBDD_$G$);
        if (Isomorph(node_g, node_f)) return(TRUE);
        return(FALSE);
    }
}
```

3.3 An algorithm for the full matching problem using GBDDs

We now consider the complete matching problem, by extending the previous considerations. Given a function $H(x_1, \ldots, x_n)$, bridging k of its n inputs together produces a new function H_B, whose support cardinality is $n-(k-1)$. Consider the bridging of two input variables, say x_i and x_j. The Shannon decomposition with respect to variables x_i, x_j gives:

$$H(x_1, \ldots, x_n) = x_i(x_j H_{ij} + x_j' H_{ij'}) + x_i'(x_j H_{i'j} + x_j' H_{i'j'}),$$

where the terms $H_{ij}, H_{i'j}, H_{ij'}$ and $H_{i'j'}$ are the cofactors of H with respect to $x_i x_j, x_i' x_j, x_i x_j'$ and $x_i' x_j'$ respectively. If we bridge x_i with x_j the above equation becomes:

$$H_B(x_1, \ldots, x_i, \ldots, x_n) = x_i H_{ij} + x_i' H_{i'j'},$$

where $B = \{x_i, x_j\}$.

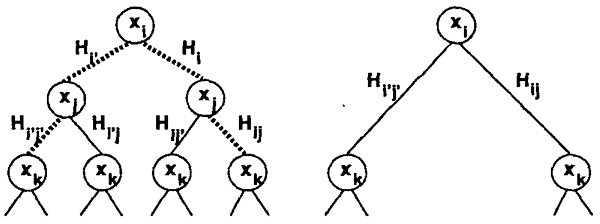

Figure 6: Bridging two variables x_i, x_j, in ordered adjacent position.

When the two variables x_i and x_j occupy adjacent positions of a given ordering, the effect of variable bridging on a BDD diagram is shown as in Fig. 6. Bridging the two variables x_i, x_j, corresponds to merging the nodes related to x_j into the nodes corresponding to x_i, and removing the dead branches.

This argument can be extended to bridging any $k < n$ successively ordered variables. Note that by considering all the input permutations of the module function, we implicitly consider all possible sub-sets of input variables whose bridging could be used to personalize the module to the given cluster function. This guarantees that the algorithm finds a match - if one exists - that can be obtained by bridging inputs of the module function (in addition to forcing them to 1 or 0).

The introduction of input bridging increases the possible ways to implement the cluster function F with m inputs. Let suppose $m = n-1$, we have two possible solutions: one input of G stuck or two inputs bridged together. If $m = n-2$, the possible choices increase to 4: two inputs stuck, one input stuck and two bridged, two sets of two input bridged together, and three inputs bridged together. In looking for a possible matching solution, it follows that we have to explore all possible combinations of bridgings and stucks, whose number increases with the difference $n-m$.

For the above reasons, and since the bridging procedure adds run-time cost, our algorithm first attempts to find a matching only using stuck-at inputs, as already described; if it fails, it starts to perform all the possible combinations and searches for a sub-isomorphism between the modified (*i.e.* bridged) GBDD of G and the BDD of F. Thus, by introducing the input bridging to reduce the degree of freedom of the module cell, as well as using the GBDD, the mapping algorithm can be rewritten:

```
Matching_2( $G$, $F$ )
{
    /* no bridging at first attempt */
    if (Matching_1( $G$, $F$)) return(TRUE);
    for (each set of feasible $B_k$ of $G$ ) {
        GBDD_$G$ = BridgeBdd(GBDD_$G$, $B_k$ );
        if (SubIsomorph_1(GBDD_$G$, BDD_$F$, $m$)) {
            $\Omega$ = DetermineOmega();
            $S$ = DetermineStuckatSet();
            return(TRUE);
        };
    }
    return(FALSE);
}
```

The procedure *BridgeBdd* modifies a BDD performing the bridging on a set of variables B_k in adjacent positions.

```
BridgeBdd( BDD_G , B_k)
{
    bridge_size = size(B_k) - 1;
    bridge_level = FirstInput(B_k);   /* level of the first element of B_k */
    for (each node_j of bridge_level) {
        /* bypass as many levels as the bridge_size */
        next = node_j->low;
        while (levelof(next) ≥ (bridge_level - bridge_size)) {
            node_j->low = next->low;
            next = next->low;
        }
        next = node_j->high;
        while (levelof(next) ≥ (bridge_level - bridge_size)) {
            node_j->high = next->high;
            next = next->high;
        }
    }
    DisposeUnreachableNodes();
}
```

4 Implementation and Results

The algorithms presented here have been incorporated in *Ceres* [4] to form an option called *Proserpine*. It reads the logic description of the module and creates the global BDD data structure. The partitioning, decomposition and covering tasks are those of *Ceres*, while the matching algorithm is based on the BDD sub-isomorphism described in this paper. *Proserpine* has been implemented in C and has been tested on the MCNC and ISCAS benchmarks. Two different EPGA modules (Actel *act*1 and *act*2) have been used.

Circuit	Proserpine		Mis_pga
	act1	*act2*	*act1*
duke2	177/178	164	198
f51m	63/65	52	56
bw	67	64	80
clip	73/74	62	62
vg2	46	41	46
rd84	70/75	63	72
5xp1	53/54	48	53
C499	170/274	170	173
misex1	25	23/24	-
misex2	45/46	40/42	-
alu4	350	310	-
apex6	396/411	293/302	-
apex7	121/122	107/108	-
rot	465/472	422/427	-

Table 2: Implementation cost for some circuits using EPGA *act*1 and *act*2. For *act*1 a comparison with *Mis_pga* is shown.

Table 2 summarizes our results, giving the number of modules needed to implement a logic circuit (*cost*). When two entries are reported, the first refers to solving the full matching problem, while the second refers to the simplified one. When only one data is reported the full and simplified problems achieve the same cost.

Comparisons to *Mis_pga* have been done considering heuristic 1, with no iteration. They are provided only for the cell *act*1, since there are no published data for *act*2. The total cost for the compared set of circuits is favorable to our solution being 619 for *Proserpine* against 640 for *Mis_pga*.

5 Conclusions and Future Work

We have presented a new matching algorithm for technology mapping of electrically programmable gate arrays. The algorithm personalizes an uncommitted module to perform a desired logic function - if it is possible - by determining the set of input variables that need to be stuck at 0/1 or bridged together. This matching algorithm allows a mapping program to capture the entire family of functions that can be implemented by a module by describing only one logic function, thus avoiding the enumeration of the entire library. In addition, the algorithm is not specific to a type of module, but can be applied to any type that can be represented by a single-output combinational logic function.

The matching algorithm has been implemented as a part of a technology mapping program called *Proserpine*. We have tested the program on a set of benchmarks and have concluded that it compares favorably to other approaches.

An outcome of this research has been the development of a tool to evaluate the effectiveness of different modules, for use in future EPGAs. Future research will address both the perfectioning of the algorithm and the comparative study of EPGA architectures, for implementing a given benchmark suite.

6 Acknowledgment

The authors would like to thank Alessandro Bedarida for developing and implementing part of the program and Frederic Mailhot for several stimulating discussions. This research was sponsored by an AEI/IEEE scholarship and by DEC and AT&T, jointly with NSF, under a PYI award. We acknowledge also support from ARPA, under contract No. J-FBI-89-101.

References

[1] A. El Gamal, J. Greene, J. Reynery, E. Rogoyski, K.A. El Ayat, and A. Mohsen, "Architecture for Electrically Configurable Gate Arrays", *IEEE Journal of Solid-State Circuits*, April 1989, pp. 394-398

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

[3] R.K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang, "MIS: A Multiple-Level Logic Optimization System", *IEEE Transactions on CAD*, November 1987, pp. 1062-1081

[4] F. Mailhot and G. De Micheli, " Technology Mapping with Boolean Matching ", *European Design Automation Conference*, Glasgow, Scotland, March 1990, pp. 212-216

[5] J. Rose, R. Francis, D. Lewis and P. Chow, "Architecture of Field Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency", *IEEE Journal of Solid State Circuits*, Vol. 25, No.5, October 1990, pp. 1217-1225

[6] K. Keutzer, "Dagon: Technology binding and local optimization by dag matching", in *24th ACM/IEEE Design Automation Conference*, 1987, pp. 341-347

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

[8] E. Detjens, G. Gannot, R.L. Rudell, A. Sangiovanni-Vincentelli, and A.R. Wang. "Technology mapping in mis", *International Conference on Computer-Aided Design*, November 1987, pp. 116-119

[9] S.B. Akers, "Binary Decision Diagrams", *IEEE Transactions on Computer*, June 1978, pp. 509-516

[10] R. Bryant, "Graph Based Algorithms for Boolean Function Manipulation", *IEEE Transactions on Computer*, August 1986, pp. 677-691

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