Logic Synthesis for Programmable Gate Arrays

Rajeev Murgai, Yoshihito Nishizaki*, Narendra Shenoy,
Robert K. Brayton and Alberto Sangiovanni-Vincentelli

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA-94720

Abstract

The problem of combinational logic synthesis is addressed for two interesting and popular classes of programmable gate array architectures: table-look-up and multiplexor-based. The constraints imposed by some of these architectures require new algorithms for minimization of the number of basic blocks of the target architecture, taking into account the wiring resources.

1 Introduction

Programmable devices (PD's) are devices that can be programmed by the user to implement a logic function. Because of short turnaround time, they are becoming increasingly important for rapid system prototyping. In addition, they have a low cost of manufacturing and are fully testable. For short turnaround time, it is necessary to have automatic tools that take a "high-level description" (like equations or a VHDL description) of a circuit and synthesize onto these architectures. PD's can be classified into two broad categories:

1. Programmable Logic Devices (PLD's) that are PLA-like. Typically these are interconnections of PLA's. Commonly used architectures are the ones manufactured by A.M.D. and Altera.
2. Programmable Gate Arrays (PGA's) that are gate-array-like. They are typically used to implement multi-level logic functions. Examples of such architectures are the Xilinx and Actel architectures.

Excellent tools (e.g. Espresso [9]) are available for optimizing the mapping of a high level description of the logic onto PLD's, since the differences vis-a-vis the standard PLA-minimization problem are small. However, very few tools are available for PGA's, since differences in basic building blocks as well as in the amount of wiring resources make the synthesis problem substantially more difficult than in the standard gate-array case.

The basic PGA architectures share a common feature: they consist of repeated arrays of identical logic blocks. A **logic block** (also called a **basic block**) is a versatile configuration of logic elements which can be programmed by the user. The interconnections to realize the circuit have to be programmed using scarce wiring resources. There are two main categories of block structures, namely **Table-Look-Up (TLU)** and **Multiplexor-Based (MB)**; the architectures resulting from them are called the TLU architectures and the MB architectures respectively. A basic block of a TLU implements any function with m inputs, ($m \geq 2$). For a given TLU architecture, m is a fixed number. In MB architectures, the basic block is a configuration of multiplexors.

Existing multi-level logic synthesis tools like misII [7] break the synthesis task into two separate phases: **technology-independent optimization** and **technology mapping**. This strategy is based on the possibility of measuring the cost of an implementation at a higher level of abstraction. However, this strategy is not well suited for PGA architectures since the constraints posed by such architectures are not easily captured at an abstract level. For PGA's, the two steps should be merged together as much as possible to provide a better overall optimization algorithm driven by the target technology/structure.

The paper is organized as follows: Section 2 describes the two classes PGA architectures, and states the problem to be solved for PGA's. New approaches for these two classes are presented in Sections 3 and 4. Results on a set of benchmark examples are presented in Section 5.

2 PGA Architectures

2.1 TLU Architectures

The basic unit of logic in these architectures is called a **configurable logic block (CLB)**. It can realize any logic function of up to m inputs. A typical example is the Xilinx architecture [1], in which $m = 5$. The interconnections between the logic blocks consist of metal segments joined by program-controlled pass transistors. The logic functions and the interconnections are determined by the configuration program data stored in the internal static memory cells. The main constraints from the synthesis point of view are:

1. A limited number of CLB's on a chip (e.g. the Xilinx chip typically has 64, 100 or 320 CLB's).
2. Maximum number of inputs a CLB can have.
3. Limited wiring resources.

2.2 MB Architectures

In this architecture, the basic block of logic has a multiplexor structure. An example of an MB architecture is the Actel architecture [2], in which the basic block has three 2-to-1 multiplexors and an OR gate as shown by the block STRUCT in Figure 3. The rows of logic blocks are separated by a routing channel consisting of routing tracks and a clock distribution network.

2.3 Problem Statement

Given a circuit description in terms of a set of Boolean equations, we are interested in its final realization using the basic blocks of the target PGA architecture. The objective could be to minimize the number of blocks used, the delay on the critical path or a combination thereof. In this paper, *we address the problem of minimizing the number of blocks used and of reducing the routing complexity.* The algorithms discussed below are implemented in the framework provided by misII. All the basic terminology used in this paper can be found in previous work ([7], [9], [4]).

3 An Approach for TLU Architectures

MisII uses the number of literals in the factored form of a node in the Boolean network as an estimate of the area or the gate count. This is a poor estimate in the minimization of the number of CLB's. For example, let $m = 5$, $f_1 = abcdeg$ and $f_2 = abc + b'de + a'e' + c'd'$. Functions f_1 and f_2 have 6 and 10 literals respectively. Function f_1 requires two CLB's in its optimum implementation, whereas f_2 needs just one (since f_2 is a function of five variables). Thus *the objective function has to do more with the number of inputs than with the actual logic that the function realizes.* Before we describe

* now with Kawasaki Steel Corporation, Tokyo, Japan.

in detail various parts of the algorithm, we need the following definitions.

Definition 4.1: *An expression is* **cube-free** *if no cube divides the expression evenly.* e.g., $ab + c$ is cube-free; $ab + ac$ is not. *The* **primary divisors** *of an expression f are the set of expressions $D(f) = \{fC \mid C$ is a cube$\}$. The* **kernels** *of an expression f are the set of expressions $K(f) = \{ g \mid g \in D(f)$ and g is cube-free $\}$. The* **residue** *associated with a kernel k_i of a function f is the expression for f with a new variable substituted for all occurrences of k_i in f.*

Definition 4.2: *The* **support** *of a function f, denoted as sup(f), is the set of variables f explicitly depends on.* $\mid sup(f) \mid$ represents the cardinality of *sup(f)*.

Definition 4.3: *A function f is a* **feasible function** *if* $\mid sup(f) \mid \leq m$.

Informally, a feasible function can be realized by one CLB.

Definition 4.4: *A Boolean network η is* **feasible** *if every intermediate node of η realizes a feasible function.*

The number of intermediate nodes in a feasible network gives an upper bound on the number of CLB's needed for the network. An outline of the algorithm developed for TLU architectures is as follows:

```
synthesis_for_TLU(η, m) /* m = max. inputs to CLB. */
{
    misII standard script /* minimize logic */
    η' = obtain_a_feasible_network(η, m);
    η'' = minimize_number_of_nodes(η', m);
}
```

To obtain a feasible network η' from the given network η, two routines *roth-karp_decompose* and *partition* have been developed. One of these two may be used to obtain η'. To reduce the number of nodes in η', *cover* and *merge* routines are used.

3.1 Roth-Karp decomposition

In the decomposition process *we are interested in decomposing non-feasible nodes into feasible nodes.* The network obtained after decomposition is a feasible network. A number of decomposition methods have been given (e.g. [5], [6]). We chose the Roth-Karp decomposition method [5] over others, because other methods need the construction of **decomposition charts** [6] whose size is always exponential in the number of inputs. In the Roth-Karp scheme, a cover representation of the on-set and the off-set of a function is used, which is more compact than decomposition charts.

Roth and Karp give necessary and sufficient conditions for a decomposition to exist. We have implemented their algorithm, but we do not search for the best possible decomposition in the present implementation, since this is computationally very expensive. Instead, we accept the first bound-set [5] that is found and do the decomposition.

3.2 Partition

Partition, like Roth-Karp decomposition, ensures that the function f_i associated with a node i is a feasible function. The difference is that *partition* relies on kernel extraction to do the decomposition. It operates in two phases. In the first phase, it examines nodes that do not have a feasible function associated with them. Kernels for the logic expression at a node (with logic function f) are enumerated. For every kernel k_i and residue r_i that are feasible functions, it associates a cost

$$cost(k_i) = \mid sup(k_i) \bigcap sup(r_i) \mid \qquad (1)$$

and chooses the kernel with the least cost. Else it recursively computes the cost for kernels or residues that are not feasible functions. If kernels cannot be extracted, an AND-OR decomposition is done at the node. The cost can be interpreted as a measure of routing complexity as follows: if we were to replace a node in the network by one of its kernels k_i and the corresponding residue r_i, the number of new edges created in the network will be equal to

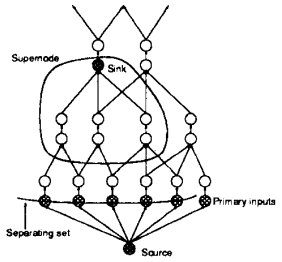

Figure 1: Separating sets using maxflow

$(\mid sup(k_i) \mid + \mid sup(r_i) \mid - \mid sup(f) \mid +1)$
$\approx (\mid sup(k_i) \bigcap sup(r_i) \mid)$.

The second phase of *partition* is a local operation: only the neighborhood of a node is searched for a solution. It uses a greedy heuristic to select nodes that may be collapsed into their fanouts. Every node which has a single fanout is collapsed into its fanout provided the function at the new node is a feasible function. Next the cost of collapsing each node into its fanouts is evaluated. The cost takes into account the duplication of logic and the additional routing cost incurred. Let $FO(i)$ be the fanouts and $FI(i)$ be the fanins of node i. Then

$$cost(i) = \sum_{j \in FO(i)} \begin{bmatrix} \mid FI(i) \mid + \mid FI(j) \mid \\ - \mid FI(i) \bigcap FI(j) \mid - 1 \\ - \left(\mid FO(j) \mid * \mid FI(j) \mid \right) \end{bmatrix} \qquad (2)$$

The first four numbers in every term of the summation represent the number of routing nets that would be created on collapsing. The remaining terms reflect the advantage of keeping the fanout of node i as the output of a CLB. The cheapest node is selected, collapsed into its fanouts, and costs are updated. This process is repeated until no further nodes can be collapsed. The *simplify* operation is used to minimize the logic at each node. This may reduce the number of fanins of some node and permit us to collapse more logic into it.

It should be noted that whereas *roth-karp_decompose* just decomposes nodes, *partition*, besides doing a decomposition, also tries to reduce the number of nodes in the feasible network and the wiring resources in the final implementation.

3.3 Covering

At this point, we have a feasible Boolean network. It may be possible to decrease the number of nodes by appropriate collapsing. The problem, then, is: *given a feasible Boolean network, collapse nodes so that the resulting network is feasible and the number of nodes is minimum.* We call this the **covering problem**. Unlike the second phase of *partition*, it uses a global view of the network to determine nodes to be collapsed. However, it is very time consuming.

Define a **supernode** corresponding to a node i of a Boolean network to be a cluster that contains i, and some nodes in the transitive fanin of i. A cluster cannot contain any primary inputs. Every node in the cluster has a path to node i which lies completely in the cluster. Note that the boundary of any cluster forms a separating set between node i and the primary inputs. The constraint associated with a supernode is that it should have a maximum of m inputs, the motivation being that all the nodes in a supernode can be implemented by one CLB and hence collapsed. There may be several supernodes for a node.

Covering consists of two stages. In the first stage, all supernodes corresponding to each node of the network are generated by repeatedly applying the maxflow algorithm (Figure 1). Each invocation of the maxflow algorithm generates a separating set. If the cardinality of this set is less than or equal to m, this indicates that a new supernode has been found. In the second stage, we choose a subset T of supernodes from the candidates obtained in the first stage such that T covers the entire network. The aim is to minimize the cardinality of the set T, since each supernode in T corresponds to a CLB. This problem is formulated as a **binate covering problem**. Our formulation of the binate covering problem is similar to

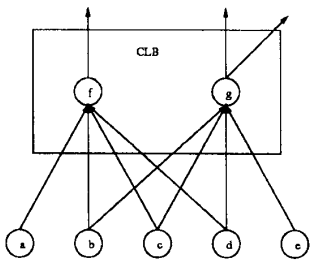

Figure 2: Example of mergeable functions

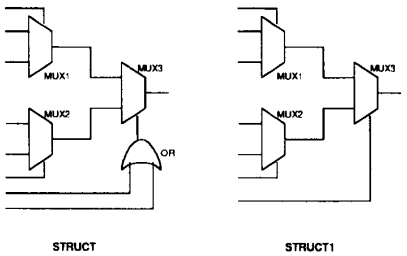

Figure 3: Two mux structures: STRUCT and STRUCT1

the one described in [10]. There are two kinds of constraints concerning the choice of candidate supernodes. The first is that every intermediate node in the Boolean network should be included in at least one supernode. The second is that if a supernode is chosen, then for each input to the supernode, some supernode whose output supplies the input must also be chosen. (In ordinary covering problem (unate covering problem), the second set of constraints is absent.) We want to obtain a minimum set of supernodes which satisfy these constraints. Note that the cost of each supernode is one, since it can be implemented by one CLB. An exact algorithm [12] as well as heuristic ones have been developed and/or implemented [11].

3.4 Merging

Special features of an architecture may be exploited at this point to improve the results. For example, in the Xilinx architecture, a CLB has the following properties:

- It can implement a feasible function ($m = 5$).
- It can implement two feasible functions f and g provided: a) each of them has at most four inputs, b) the number of their common inputs is at most three, and c) the total number of inputs is at most five. If f and g satisfy these three conditions, they are said to be **mergeable**. See Figure 2 for an example.

Given a feasible Boolean network η, we want to find a largest set of disjoint pairs of mergeable functions. Consider a graph G, in which each vertex v represents a feasible function f at a node n of the Boolean network η. There is an edge between vertices v_i and v_j of G if and only if the feasible functions f_i and f_j (at nodes n_i and n_j of η respectively) are mergeable. Then the problem of merging is equivalent to the problem of finding a **maximum cardinality matching** in G. This can be solved using standard optimization techniques.

4 An Approach for MB Architectures

MB architectures can be of many different forms. For our experiments we chose the Actel architecture as our primary target. In this architecture, the basic block, STRUCT, is a configuration of three 2-to-1 multiplexors, with an OR gate providing the select input to the output multiplexor. For the sake of simplicity, we explain our basic algorithm on a simplified, more uniform structure: STRUCT1 where the OR gate is ignored (Figure 3). However, the algorithm takes into account the OR gate when producing the final results.

In conventional technology mapping [13], each gate is chosen from a library and has a cost associated with it. The optimized network is decomposed into a **subject graph** in terms of the **base functions**. Typically the base functions are NAND gates and inverters. The logic function of every gate is similarly represented as **pattern graphs**, which are also in terms of the base functions. The subject graph is then covered by a set of pattern graphs such that the cost of this set is minimized. One can use this approach to solve the MB problem, by deriving a library of gates from the basic block. The cost, then, is the number of STRUCT blocks needed to realize the function.

The basic idea of our approach is *to select an appropriate base function, a library of cells and a set of pattern-graphs*. Given the MB architecture, we chose 2-to-1 multiplexor as the base function. The library consists of all the functions that can be realized by one basic block. Then, the complete set of pattern-graphs may be very large. However, if we have a *good* subject-graph, a very small subset of pattern-graphs generated from the cells in the library may suffice (Section 4.1).

Binary Decision Diagrams (BDD's) and Reduced Ordered Binary Decision Diagrams (ROBDD's) [4] are well known representations of Boolean functions in terms of multiplexors, and as such are used to build the subject-graphs and the pattern-graphs. First, we define a few terms. A function f can be represented by a matrix whose rows are the cubes of f. This is called a **cover** of the function f. A **unate cover** is a cover in which every variable appears in only one phase. A **binate cover** is one in which at least one variable occurs in both phases. Such a variable is called a **binate variable**. A cover is said to be a **tautology** if it covers the entire Boolean space. The **cofactor** of a function f with respect to a literal l is the function when l evaluates to 1. Let the function f be a function of variables $x_1, x_2, ..., x_n$. Then **Binary Decision Diagram** of f takes the form of a rooted directed graph with vertex set V containing two types of vertices. A **non-terminal vertex** v has as attributes an argument **index**$(v) \in \{1, 2, ...n\}$, and two children, **low**(v) and **high**$(v) \in V$. A **terminal vertex** v has a value, **value**$(v) \in \{0, 1\}$. A BDD is said to be **ordered** if the indices of the vertices in all root-to-terminal vertex paths follow a fixed order. A BDD is said to be **reduced** if there is no vertex u with $low(u) = high(u)$, and there are no two distinct vertices v and w such that the sub-graphs rooted at v and w are isomorphic. A reduced ordered BDD is called an **ROBDD**. We say that a vertex of the subject-graph (pattern-graph) is a **leaf** if it is either a terminal vertex or a non-terminal vertex whose corresponding function evaluates to an input. A vertex that is not a leaf is a **non-leaf** vertex. The **non-leaf portion** of the graph is the graph with its leaves deleted. Also, a **leaf-DAG** is a directed acyclic graph (DAG) whose non-leaf vertices have single parents.

4.1 Pattern-graphs

We construct four pattern-graphs for STRUCT1 as shown in Figure 4. If a function is realizable by one STRUCT1 block, it either uses all the multiplexors, or two, or just one[1]. These pattern-graphs are in one-to-one correspondence with these possibilities. So we need a very small set of patterns to capture all possible functions realizable by one STRUCT1 block. Note that all pattern-graphs are leaf-DAG's and only pattern-graph 1 uses all the multiplexors.

The introduction of the OR gate at the select input of MUX3 increases the number of functions realized by the block considerably. From an algorithmic point of view, it introduces some difficulties: the number of pattern-graphs increases, equivalence between the multiplexor usage and the pattern-graphs is destroyed, and some of the pattern-graphs may not be leaf-DAG's. The algorithm is modified so as to explore for a possible occurrence of the OR structure. For this, the pattern-set has to be expanded. Presently we have a set of 8 pattern-graphs for STRUCT.

The following propositions state two cases when optimum subject-graphs can be constructed for STRUCT and give a method to generate the subject graphs (proofs are omitted):

Proposition 4.1: *If a function $f = f(f1, f2, ... fk, g1, g2, ... , gl)$ is a single cube with input variables $f1, f2, ... fk$ occurring in the*

[1] If a function does not use any multiplexor, then it is a trivial function and can be realized without any basic block.

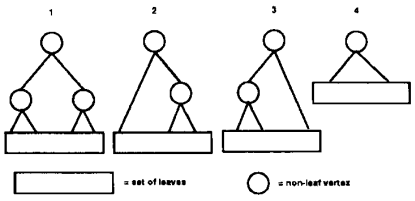

Figure 4: Patterns for STRUCT1

positive phase and g1, g2, g3,..... gl in the negative phase, then the ROBDD corresponding to the ordering f1, f2, g1, g2, f3, g3, g4, f4, g4, g5, ... of input variables results in the minimum number of STRUCT blocks after covering.

Proposition 4.2: *If a function f = f(f1, f2, .. fk, g1, g2, .. gl) consists of only single literal cubes with input variables f1, f2, ... fk occurring in the positive phase and g1, g2, ... gl in the negative phase, then the ROBDD corresponding to the ordering f1, g1, f2, f3, g2, f4, f5, g3, f6, f7, ... of input variables results in the minimum number of STRUCT blocks after covering.*

When inputs of a phase finish, the remaining inputs are concatenated to complete the ordering. Also, both the orderings are listed such that they start from the terminal vertices and end at the root of the ROBDD.

4.2 Construction of Subject-graphs

Experiments showed us that the construction of a subject-graph for the entire network (**global subject-graph** [14]) leads to poorer results than the construction of subject-graphs for each node in the network (**local subject-graph**) [11]. This is because of the following reasons:

- The global subject-graph requires all vertices to be indexed by primary inputs. This may be too restrictive; it may be possible to obtain a better representation if intermediate functions could be the indices of the vertices. The local subject-graph allows that, since fanins to a node could be the intermediate nodes.
- The ordering heuristic used to construct global subject-graphs does not generate a good enough subject-graph for covering by the small pattern-set. The basic assumption behind having a small set of pattern-graphs is that the subject-graph is close to optimum. If it is not, then the pattern-set may have to be expanded.

So, we construct subject-graphs for each node of the network separately. We now describe the two representations for the subject-graph and the heuristics used to build them.

1) ROBDD: We choose ROBDD as a representation for the subject-graph because it is compact and has no duplication of logic. The basic method is to construct an ROBDD for the function at a node (the inputs of the function being the fanins of the node) and then cover it with the pattern-graphs. The size of the ROBDD of a function is sensitive to the ordering of the input variables. No polynomial-time algorithm is known for finding the ordering that results in the smallest ROBDD. However, if the function has a small number of inputs, an optimum ordering[2] can be determined quickly by trying out all possible input permutations. This leads to the first heuristic for constructing the subject graph.

Heuristic 1: Transform the given network η into a network η' in which every node has at most N fanins. For each node, construct the ROBDD's corresponding to all the input orderings, evaluate their cost using covering algorithm, and pick the one which yields minimum cost. Since the problem of obtaining the network η' from η is similar to the problem for TLU architectures, the routines *partition, roth-karp_decompose and cover* are used with $m = N$. Values of N from 3 to 6 give good results [11].

A shortcoming of this representation is that the input ordering constraint imposed by the ROBDD may be too severe and may yield a poor result in terms of the basic blocks. Also, there may be a lot of vertices with multiple parents. Since the covering procedure is tree-based, it breaks the subject-graph at vertices with multiple parents (Section 4.3). If there are a lot of vertices with multiple-parents, the subject-graph is partitioned into too many sub-trees. This may not give a good result. This leads us to consider another representation which does not have these two restrictions.

2) BDD: We now give a heuristic which constructs a BDD (without any ordering restrictions) such that

- the number of vertices in the BDD is small, and
- the number of vertices with multiple parents is small.

Heuristic 2: For every node in the network, we construct a BDD trying to minimize the number of vertices. The logic cover of the node is cofactored with respect to its input variables until we reach a unate cover (also called a unate leaf). Cofactoring with respect to variable x_i corresponds to creating a vertex (with index i) in the BDD. The selection of the cofactoring-variable at each stage is done as follows. If there are variables that appear in all the cubes in the same phase, they are selected first (in any order). Next, at each step the most binate variable is chosen. Ties are broken by selecting those variables first that occur in both phases nearly the same number of times. Selecting variables this way tries to minimize the number of vertices in the BDD.

At the unate leaf, we construct a *0-1* matrix $B = (b_{ij})$ for the unate cover UC, where

$$b_{ij} = \begin{cases} 1 & \text{if } i^{th} \text{ cube of } UC \text{ contains variable } x_j, \\ 0 & \text{otherwise.} \end{cases} \quad (3)$$

A minimal column cover CC for B is then obtained [9]. It contains some variables occurring in the unate leaf. For each variable x_j in CC, a sub-cover SC_j is constructed using cubes of UC that depend on x_j. Each cube is put in exactly one such sub-cover (ties are broken arbitrarily). From the sub-cover SC_j, x_j is extracted out to yield modified sub-cover MSC_j. In MSC_j, x_j is a don't care. This corresponds to creating a vertex with index j in the BDD corresponding to SC_j. The column covers for MSC_j are recursively obtained. The recursion stops when either a tautology is reached or a single cube is left in MSC_j. A tautology corresponds to the terminal 1 vertex of the BDD. The BDD for a single cube is constructed using the ordering determined from Proposition 4.1 (Section 4.1). Hence we have the BDD of MSC_j after this operation. The BDD of SC_j is obtained by ANDing the BDD's of MSC_j and x_j, with x_j higher in the order. The BDD of UC is then obtained by ORing repeatedly pairs of BDD's corresponding to sub-covers SC_j. Constructing minimal column cover helps in reducing the number of vertices with multiple parents in the BDD. We make the following observation:

Observation 4.1: *If q is the number of cubes in the BDD of a unate cover UC, and p is the number of non-terminal vertices in the BDD with multiple parents, then* $p \leq (q - 1)$.

In the BDD of the function f, there are no non-terminal vertices with multiple parents until unate leaves are reached. By choosing the most binate variable at each stage, the depth of the BDD and the number of unate leaves is kept small. If there are few cubes in each unate leaf, the number of vertices with multiple parents in the BDD for f will be small.

A drawback of this heuristic is that there may be duplication in different branches of the BDD. It may not be possible to predict which one of ROBDD or BDD will result in a lower cost. So, we may wish to construct both types of subject-graphs for a node and select the one with the lower cost. This leads to Heuristic 3.

Heuristic 3: Construct both subject-graphs for the node - ROBDD and BDD; cover them using the covering algorithm and select the one with the lower cost.

[2] In our case, the optimum ordering is the one that results in an ROBDD having minimum cost.

4.3 Covering algorithm

We use the tree-covering heuristic proposed in [13]. However, the subject graph and the pattern graphs are in terms of 2-to-1 multiplexors. We now justify the use of the set of pattern-graphs by stating the following theorem (proof is omitted):

Theorem 4.1: *For a function f realizable by one STRUCT1 module, ∃ a subject-graph S whose non-leaf portion is isomorphic to the non-leaf portion of one of the four pattern-graphs of Figure 4, say p. The covering algorithm will map S onto p.*[3]

Since the pattern-set is small, the covering algorithm is fast.

4.4 Iterative improvement

By performing some local transformations on the network, it may be possible to improve the result. The basic idea is to apply these transformations in sequence - if a transformation yields a lower cost, it is accepted. If the subject-graph construction is fast, we can afford to recompute the cost. The outline of the *iterative_improvement* algorithm follows:

```
iterative_improvement()
{
    repeat {
        partial_collapse(network);
        decompose_nodes(network);
    } (until satisfied or no further improvement);
    quick_phase(network);
}
```

4.4.1 Partial_collapse

Partial_collapse collapses a node into each of its fanouts. The aim is to *select nodes for simultaneous partial_collapse such that the gain is maximized.* If by *partial_collapsing* a node *n*, the cost of the network decreases, the node is a candidate for being selected for final *partial_collapse*.

A cluster C corresponding to n is formed. It contains n and all its fanouts. This cluster represents the subnetwork affected by the *partial_collapse* operation on n. The process is repeated for all the nodes. We impose the condition that only disjoint clusters be simultaneously collapsed. This condition guarantees that the cost computed for a node of a cluster at the time of forming that cluster remains valid after all the selected nodes are collapsed into their fanouts. We formulate the problem as a 0-1 integer program. Let *cost(n)* be the cost of node n and C be the cluster that corresponds to node n. The cost for cluster C, *cost(C)*, is the sum of the costs of the nodes in the cluster before n is *partially_collapsed*. After n is collapsed into its fanout nodes, let the sum of the new costs of the fanouts be *newcost(C)*. Define

$$gain(C) = cost(C) - newcost(C). \tag{4}$$

If $gain(C) > 0$, then node n is a candidate for selection. We compute the gain for every cluster, and retain only those clusters which have a positive gain. Let us call these clusters **good clusters**.

Let x_i be the 0-1 variable corresponding to the good cluster C_i. Let $M = (m_{ij})$ be the node-cluster incidence matrix, where

$$x_i = \begin{cases} 1 & \text{if } C_i \text{ is selected for } \textit{partial\_collapse}, \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

$$m_{ij} = \begin{cases} 1 & \text{if node } n_i \text{ belongs to good cluster } C_j, \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

The problem, then, can be formulated as

$$\textbf{maximize} \sum_i (Gain(C_i) \times x_i) \tag{7}$$

s.t.

$$\sum_j m_{ij} x_j \leq 1 \quad \forall i \tag{8}$$

[3]This subject-graph S is an ROBDD and can be found by constructing ROBDD's for all possible input orderings for f. Note that f is a function of at most 7 inputs.

name	*n1*	*mis-pga*		*ind. result*
		CLB's	*time*	*CLB's*
10bitreg	20	10	6.9	11
10count	126	23	154.2	19
180degc	146	21	74.5	34
3to8dmux	101	30	81.1	25
4-16dec	70	12	57.5	18
4cnt	66	17	35	10
8bappreg	141	27	64.9	27
8count	138	20	57.7	29
9bcsac1	135	34	1028.9	45
9bcasc2	133	29	91.9	42
arbiter	146	21	73.3	13
C5315[1]	376	497	3467.4	-
C499	162	50	137.5	-
vg2	11	21	25.6	-
rd84	20	32	65.1	-
rot	169	153	844.8	-
apex6	149	191	1376.8	-
apex7	59	50	117.3	-
duke2	84	105	357.1	-
5xp1	31	23	45.5	-

[1] In *merge*, the program ran out of memory.

Table 1: Results: TLU architecture

4.4.2 Decompose_nodes

Decomposition is an inverse operation of *partial_collapse*. It breaks down a large node into smaller nodes. If the sum of the costs of the smaller nodes is less than the cost of the original node, the node is decomposed. Since decomposing a node has no affect on other nodes, this operation is independent of other nodes in the network.

4.4.3 Quick_phase

This operation decides if it would be beneficial to implement a node in its complemented form. Since this operation affects the fanout nodes, the cost computation is done as in *partial_collapse*. In this greedy heuristic, a node is picked; if it is beneficial to implement it in the complemented form, the same is done and the fanout nodes are appropriately modified. The process is repeated till a single pass is performed over all the nodes.

5 Results

The algorithms discussed above have been implemented in C and incorporated in misII [7] to form a system called **mis-pga**. It can be run in interactive or batch mode. In batch mode, a "script" is used to control the optimization. The input to our program is the result of a misII optimization with the standard script. All the examples were run on VAX 8800.

5.1 TLU Architectures

We set m to 5 for this set of results. Since there are no published results for TLU architectures, we compare our results with some results obtained from industry [8] in Table 1. However, we also present our results on some MCNC and ISCAS benchmarks. We use the following order of commands for the experiments: *partition, simplify, cover, merge.* For each example, the table shows the number of nodes in the initial Boolean network (column *n1*), the number of CLB's after synthesis by *mis-pga*, the run time (in seconds) and the corresponding industrial results (column *ind. result*). It was experimentally observed that *cover* is effective but expensive. For small examples, exact algorithm can be used, but for large ones, the heuristic methods should be used. The *merge* reduces the CLB count by 5-15%. This increases the CLB utilization on the chip.

5.2 MB Architectures

We compare the performance of *mis-pga* (using our direct approach), with misII (using the Actel library of cells [3]) on some MCNC and ISCAS benchmarks in Table 2. Each entry in the columns shows the block count and the run time in seconds (in parentheses)[4]. The *mis-pga* results are shown in the columns *h1*,

[4]For *h1*, the time does not include the time taken for converting the initial network into a network with nodes having inputs at most three.

name	misII	mis-pga							
		h1		h2		h3		h4	
	map(time)	no it.(time)	it.(time)	no it.(time)	it.(time)	no it.(time)	it.(time)	it.(time)
duke2	172 (83)	198 (3.6)	187 (264.6)	233 (1.6)	183 (42.2)	223(89.0)	163(898.3)	177(141.5)
f51m	52 (21)	56 (1.7)	51 (117.5)	60 (0.3)	50 (5.3)	57 (9.0)	47 (690.0)	50(59.2)
C1908	189 (104)	197 (7.8)	182 (803.9)	271 (2.1)	187 (59.8)	204(156.7)	172(2445.3)	175(244.4)
bw	80 (46)	80 (1.5)	71 (67.1)	75 (0.4)	68 (4.0)	68 (174.1)	61(478.0)	63(89.9)
clip	57 (19)	62 (1.9)	51 (182.1)	64 (0.4)	58 (9.3)	60 (7.8)	47(958.0)	52 (62.8)
vg2	45 (12.3)	46 (0.9)	41 (69.9)	37 (0.2)	37 (1.1)	37 (2.4)	37 (47.0)	37 (3.6)
rd84	62 (36.3)	72 (1.8)	64 (211.8)	188 (1.2)	71 (11.2)	188 (34.4)	58 (909.7)	71 (59.2)
5xp1	52 (19)	53 (1.5)	51 (103.6)	62 (0.4)	47 (5.6)	61 (39.1)	41(197.0)	46(40.8)
C499	174 (81)	173 (10.0)	171 (751.6)	166 (1.3)	166 (33.2)	166(6.1)	166(147.9)	166(39.5)
C5315	732 (353)	812 (29.7)	776 (3798.4)	737 (4.0)	663 (993.9)	674(1572.9)	610(9104.1)	630(2697.8)
rot	310 (108)	-	-	383 (2.8)	300 (236.9)	334(455.3)	271(4537.8)	289(625.3)

Table 2: Results: MB architecture

h2, h3, h4, corresponding to Heuristics 1 through 4 (for Heuristic 4, refer to the next paragraph). Heuristics are run with and without the iterative improvement phase (subcolumns *it.* and *no it.* respectively). The number of iterations for Heuristics 1 and 3 was set to 1, and for Heuristics 2 and 4, to 4. In Heuristic 1 the value of N was set to 3. Optimum ROBDD's are constructed for functions with up to six inputs. For the rest, an ordering heuristic is used [14]. Also, for Heuristic 3, in the *iterative_improvement* phase, a **last_gasp** phase was entered at the end. In this phase, a subnetwork is formed from each node, and the mapping procedure is applied to this subnetwork. If the cost of the subnetwork is less than the original node, it replaces the node in the original network. The *last_gasp* was used for this Heuristic, because our aim was to get the best possible overall result. A "-" indicates that the program ran out of memory[5].

Heuristic 1 does give good results. Without iterations, it is fast, since all nodes have at most three fanins and constructing an optimum ROBDD is computationally not expensive. However, with iterations, it takes a lot of time. This is expected because *partial_collapse* creates nodes with higher fanins and the heuristic constructs optimum ROBDD's for any node with fanin at most six. Heuristic 2 is quite fast; the time taken for mapping without iterations is nearly 15-20 times less than misII and the results obtained are sometimes better than the misII results. Therefore, *iterative_improvement* may be allowed to do a lot of restructuring. Then, in almost all the examples, Heuristic 2 gives lower block counts than misII in comparable run times. Since Heuristic 3 constructs two subject-graphs for each node, it is computationally expensive, but gives the best results as compared to other heuristics. It consistently outperforms misII on all the benchmark examples we ran and resulted in 4 - 24% improvement over misII results. While experimenting with all these heuristics, it seemed a good idea to use a fast heuristic for restructuring and then apply the best mapping heuristic. This led to the development of **Heuristic 4**, in which we use Heuristic 2 to do initial mapping and restructuring; then in the final phase of mapping, both the subject-graphs for a node of the resulting network are constructed; the one with the lower cost is picked. This heuristic outperforms misII (in almost all the examples) and Heuristic 2. Finally, it was observed that the iterative improvement phase often yields substantial (typically 20 - 40%) savings in the block count, though it can be computationally expensive.

6 Conclusions

We have proposed new techniques for minimization of combinational circuits for two generic PGA architectures that make optimization and technology-mapping tightly-coupled. These algorithms are general: they can be used for a TLU architecture having CLB's with m inputs ($m \geq 2$) and can be easily extended for any MB architecture. In future, we will address timing optimization and synthesis for sequential circuits.

7 Acknowledgement

The authors wish to thank E. Detjens, A. Wang, S. Malik, L. Lavagno and K.J. Singh for their assistance in this project. We also thank AT&T Bell Labs., Actel Corporation and Xilinx for their help. The financial support provided by NSF under contract number EMC-84-19744 and DARPA under contract number 44-26555 is gratefully acknowledged.

[5] In the only example (*rot*) where it happens, it is because the TLU script ran out of memory.

References

[1] Xilinx Programmable Gate Array User's Guide. 1988 Xilinx, Inc.

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

[3] Act 1 Family Gate Arrays, Design Reference Manual

[4] R.Bryant, "Graph based algorithms for Boolean function manipulation", *IEEE Transactions on Computers*, C-35(8) August 1986, pp 667-691.

[5] P. J. Roth and R. M. Karp, "Minimization over Boolean graphs", *IBM Journal of Research and Development* vol. 6/No. 2/April 1962.

[6] R. L. Ashenhurst, "The Decomposition of switching functions", *Proc. of International Symp theory of Switching Functions*, 1959.

[7] 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.

[8] E. Detjens, private communication.

[9] R. K. Brayton, C. McMullen, G. D. Hachtel and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI synthesis*, Kluwer Academic Publishers, 1984.

[10] R. L. Rudell, "Logic Synthesis for VLSI Design", UCB/ERL Memorandum M89/49, April 1989.

[11] Rajeev Murgai, Yoshihito Nishizaki, Narendra Shenoy, Robert K. Brayton, and Alberto S. Vincentelli, "Logic Synthesis for Programmable Gate Arrays", UCB/ERL Memorandum, 1990.

[12] H. -J. Mathony, "Universal logic design algorithm and its application to the synthesis of two-level switching circuits", *IEE Proc., Vol. 136, Pt. E, No. 3*, May 1989.

[13] K. Keutzer, "Dagon: Technology Binding and Local Optimization by DAG Matching", *Proc. 24th Design Automation Conference*, June 1987.

[14] S. Malik, A. Wang, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams", *IEEE Int. Conf. on CAD (ICCAD)*, 1988.