Stochastic Evolution Algorithm For Technology Mapping

Ahmad S. Al-Mulhem Alaaeldin Amin Habib Youssef

Department of Computer Engineering
King Fahd University of Petroleum and Minerals
Dhahran-31261, Saudi Arabia
E-mail: {amin,youssef}@ccse.kfupm.edu.sa

Abstract

A new technology mapper (SELF-Map) for Look-Up Table (LUT) based Field Programmable Gate Arrays (FPGAs) is described. SELF-Map is based on the Stochastic Evolution (SE) algorithm. The state space model of the problem is defined and suitable cost function which allows optimization for area, delay, or area-delay combinations is proposed. Experimental results show that SELF-Map has an overall better performance compared to other algorithms reported in the literature.

1 Introduction

In this paper, we present SELF-MAP, a new iterative technology mapping algorithm for K-LUT FPGAs. The logic modules of the FPGA are K-input look-up tables (K-LUTs). Technology mapping of K-LUT FPGAs maps a combinational Boolean network (BN) into a functionally equivalent network of K-LUTs. The mapping algorithm is based on the Stochastic Evolution (SE) algorithm [8]. At each iteration, a possible partitioning of the BN into a forest of trees is performed, and each tree is then optimally technology mapped. The global solution is constructed from the obtained *tree-solutions*. When some stopping criteria are met, the iterative search process is stopped and the best solution reported. SELF-MAP is designed for mapping LUTs with an arbitrary number of inputs K (K-LUT) but only a single output.

2 Terminology and Background

A combinational logic circuit can be represented as a *directed acyclic graph* (DAG) $G(V, E)$ where each node $v \in V$ represents a Boolean function and each directed edge $(u, v) \in E$ represents a connection between the output of u and the input of v. Such DAG representation is referred to as a *Boolean Network* (BN). A *Primary input* (PI) is a node with no in-coming edges, while a *Primary output* (PO) is a node with no out-going edges.

For node $v \in V$, *input(v)* is the set of nodes that supply inputs to v. In general, given a subset V_1 of V, $input(V_1)$ is the set of nodes in $V - V_1$ that supply inputs to nodes in V_1. A node u is a *predecessor* of node v if there is a directed *path* from u to v in the BN.

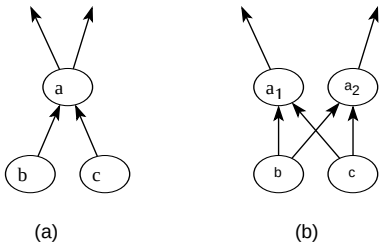

Figure 1: The effect of replicating a fanout node.

Definition 1 *A K-feasible cone at a node v, denoted by C_v, is defined as a subgraph consisting of v and a number of its predecessor nodes such that any path connecting v to any other node in C_v lies entirely in C_v, and $\mid input(C_v) \mid \leq K$.*

Since each K-LUT is a programmable block with K inputs and one output, a K-LUT can implement (or cover) any K-feasible cone in a BN.

Definition 2 *The depth of a node v is defined as the maximum number of LUTs (K-feasible cones) along any path from any primary input to v. The depth of a primary input is taken as zero and the* **depth** *of a BN is the largest node depth in the BN.*

A node v with one out-going edge is termed a *fanout-free* (FF) node. A node v with $n > 1$ out-going edges is termed a *fanout* node. Such node can be replaced by n FF nodes without affecting the BN functionality. This is accomplished by *replicating* the fanout node n times as illustrated in Figure 1.

In Figure 1.a, node a with 2 outgoing edges ($n = 2$) can be replaced (Figure 1.b) by two fanout-free nodes (a_1 and a_2) by replicating the original node functionality and connectivity twice. The replicated nodes (a_1 and a_2) are fanout free (Figure 1.b). It should be noted that if a fanout node is replicated n times, the out-degree of its immediate predecessor nodes will increase by $n - 1$. If any of its immediate nodes is FF, it will turn into a fanout node.

Definition 3 *A potential fanout node is a fanout-free node which feeds a fanout node.*

In Figure 1, node a is a fanout node while nodes b and c are potential fanout nodes. As shown in Figure 1.b, once node a is replicated into nodes a_1 and a_2, nodes b and c become fanout nodes themselves.

3 SELF-MAP

The work reported here adapts the SE algorithm [8] to the technology mapping problem of LUT-FPGAs. The mapper is called SELF-MAP, an acronym for **S**tochastic **E**volution **L**UT-**F**PGA **MAP**per. SELF-MAP can be used to optimize either for area, delay, or a combination of both area and delay.

The SE algorithm is a general strategy which can be used to solve a wide range of combinatorial optimization problems. The algorithm, however, has to be adapted to the type of problem under investigation. Specifically, the solution space has to be well defined and a suitable state representation must be adopted. Combinatorial optimization problems can be modeled in a number of ways. SE models the problem as a finite set of *movable objects* M and a finite set of *locations* L. A state (solution) is defined as a function $S : M \rightarrow L$ satisfying certain constraints.

The idea of the SE algorithm is to find a suitable location $S(m)$ for each movable element $m \in M$ which eventually leads to a lower cost of the whole state $S \in \Omega$, where Ω is the state space. Many of the combinatorial optimization problems can be formulated according to this model [8].

3.1 The Solution Space

In a general BN, nodes are either *fanout* nodes or *fanout-free* nodes. In the special case where all nodes of the BN are fanout-free, i.e. the BN is actually a tree, optimal technology mapping of such a network in linear time has been shown to be possible [2]. General networks, however, can be technology mapped using one of two possible approaches. In the first approach, the BN is decomposed into a forest of fanout-free trees by partitioning the network at every fanout node. In essence, breaking the BN at a fanout node implies that this node is going to be the output of a lookup table in the final mapping. The resulting trees are then individually mapped and the final solution is obtained by re-assembling the individually mapped trees. In the second approach, the BN is converted into a forest of fanout-free trees by *replicating* fanout nodes and the cones feeding them. This process is repeated till all nodes become fanout-free. In essence, a replicated node, implies that the logic represented by this node will be implemented more than once in different LUTs. Again, the resulting trees are individually mapped and the final solution is obtained by re-assembling the individually mapped trees. Figure 2 illustrates the two approaches. In this figure, the original BN (Figure 2.a) consists of one fanout node **a** and three cones C_1, C_2, and C_3. To map the BN using the first approach, the BN is partitioned by clipping the multiple fanout edges out-going from **a** as shown by the dotted curved line in Figure 2.b. As a result, the BN is transformed into a forest of three trees as shown by the dotted rectangles in Figure 2.b. To map the BN using the second approach, the fanout node a and the cone that feeds it (C_1) are replicated as shown in Figure 2.c. As a result, the BN becomes a forest of two trees as shown by the dotted rectangles in Figure 2.c. Following any of the above approaches results in a forest of trees where each tree is optimally mapped, and the overall solution is obtained by re-assembling the individually mapped trees.

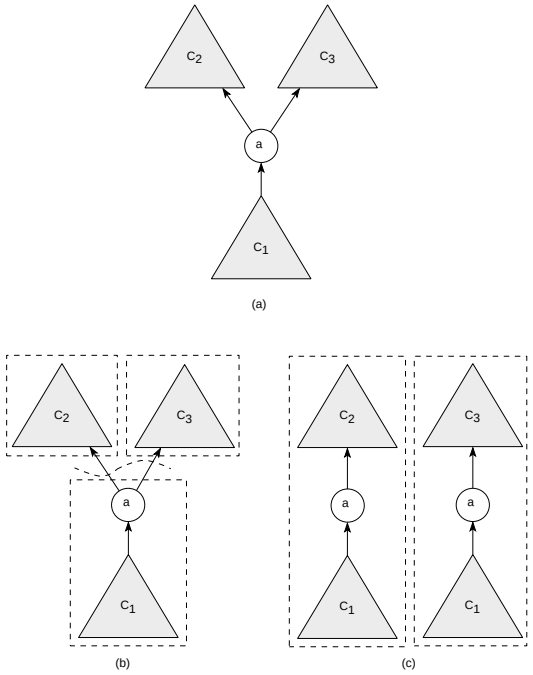

Figure 2: Mapping networks with fanout nodes.

The above two approaches represent two extremes which are unlikely to yield good solutions. The search capability of an iterative algorithm such as SE can be effectively utilized to obtain better solutions where only some fanout nodes and part of the cones feeding them are replicated. Thus, the replicated part of the BN may include some, all, or none of the fanout nodes and their fanin cones. The mapping algorithm will partition the resulting BN at all non-replicated fanout nodes. The resulting network is thus a forest of trees where each tree is mapped individually and the final solution is obtained by re-assembling the mapped trees. With the proper choice of the cost function and the state model (solution space), the search capability of the iterative algorithm can be utilized to select the set of nodes in the BN which should be replicated and the set of nodes which should be assigned to the outputs of LUTs in order to optimize some target criteria. Obviously, the two approaches mentioned earlier are only special cases of this more general one and both would miss the larger part of the search space. In this

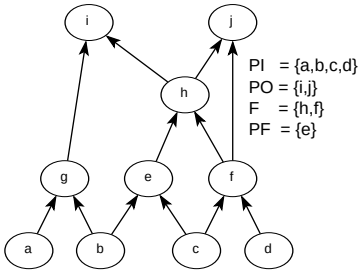

Figure 3: Example

work, we adopted this general approach which allows a more thorough investigation of the solution space.

3.1.1 Movable Objects and Locations

In mapping a BN, SELF-MAP assigns LUTs to some of the fanout nodes and replicates some others. A replicated fanout node implies that its immediate fanout-free predecessor nodes (*potential fanout nodes*) would turn into fanout nodes themselves and would thus be either replicated or assigned to the output of some LUT. Therefore, the set of movable objects M is chosen to be the set of fanout(F) and potential fanout (PF) nodes, i.e. $M = F \cup PF$, where F is the set of fanout nodes and PF is the set of potential fanout nodes (see Section 2).

In the final mapping, node $m \in M$ will either be a fanout-free node (FF), a fanout node which is not replicated (NRF), or a fanout node which is replicated (RF). Accordingly, the set of possible locations L consists of three possible locations: FF, RF, and NRF.

Fanout nodes can arbitrarily move between locations NRF and RF. A potential fanout node (PF) is initially assigned to location FF. If, however, its immediate successor node is replicated, i.e. moved from location NRF to RF, the PF node is moved to location NRF. From there, a potential fanout node can move back and forth between locations NRF and RF. If, at any time, its immediate successor is un-replicated, i.e. moved back from location RF to location NRF, the PF node is then moved back to location FF.

The above state model allows the SE algorithm to investigate full, partial, or no replication of the nodes in cones that feed any fanout node in the network. The following example illustrates these concepts.

Example: Consider the network shown in Figure 3. The network has four primary inputs $PI = \{a, b, c, d\}$, two primary outputs $PO = \{i, j\}$, two fanout nodes $F = \{h, f\}$, and one potential fanout node $PF = \{e\}$. Note that the primary inputs are not classified as fanout nodes, e.g. nodes b and c, or as potential fanout nodes, e.g. node d.

According to our state model, the set of movable objects M is chosen to be the set of fanout and potential fanout nodes. Therefore, $M = \{e, h, f\}$.

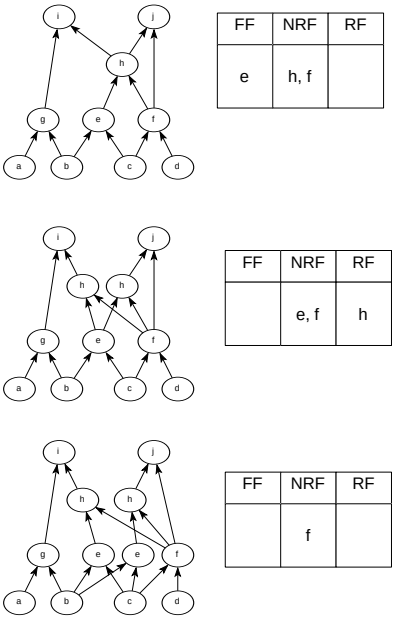

Figure 4: Example continued

Initially, h and f are in location NRF, and e is in location FF. As the algorithm proceeds in searching the solution space, those movable objects change their locations. Since h and f are fanout nodes, they can only move between locations NRF and RF. However, node e is a potential fanout node and will thus be in location FF unless h moves to location RF in which case it will be automatically moved to location NRF from which it may move to location RF. If h moves back to location NRF, e automatically moves to location FF. Figure 4 shows some possible moves and valid states in the solution space and the locations of the movable objects in each case.

3.1.2 Ordering of the Movable Objects

The **PERTURB** function of the SE algorithm [8] scans the set of movable elements M according to some apriori ordering and moves every $m \in M$ to a new location $l \in L$. In this work, the following two orderings of movable objects have been examined.

- Random ordering, where every movable object is randomly picked and perturbed. Each object is picked only once. This is a slight departure from the original SE algorithm, where a deterministic order is followed [8].
- Depth First Search (DFS) ordering, where nodes closer to primary outputs are perturbed first.

To illustrate, consider the previous example (Figure 3). The set of movable objects M is $\{e, h, f\}$.

ALGORITHM COST$(G(V,E), c_1, c_2)$;
Perform a topological sort on $G(V,E)$;
FOR every $v \in V$ **DO**
  compute d_v, z_v, and D_v ;
  IF $d_v > K$ **THEN**
    (i) Sort $ipred(v)$ in descending order of
      their weights, where $\forall p \in ipred(v)$,
      Weight(p) $= c_1 \times z_p - c_2 \times D_p \times \frac{K}{D_v}$
    (ii) Keep assigning LUTs to each $p \in ipred(v)$
      till $d_v \leq K$;
  ENDIF
  IF v is a PO or at location NRF **THEN**
    assign a LUT to v;
    set $z_v = 1$ and update D_v;
  ENDIF
ENDFOR
RETURN($\hat{c_1} \times$ No_of_LUTs + $\hat{c_2} \times$ Depth)
END (* of COST *);

Figure 5: The cost algorithm.

For the random ordering, the order of objects can be any one of the possible six combinations, i.e. $\{ehf, hef, hfe, feh, efh, fhe\}$. For the DFS ordering, the order of the objects can be one of two possible combinations, i.e. $\{hef, hfe\}$.

The ordering of objects affects the path taken by the algorithm to reach the desired solution. However, none of the above two orderings consistently performed better than the other. Furthermore, for the MCNC benchmark circuits [10], there were no significant difference between the cost of the best solutions reached by the two ordering techniques.

3.2 The Cost Function

To evaluate the fitness of a state (solution), a cost function is required. Obviously, the cost function should depend on the target optimization objective. In this work, the optimization objective can be *area*, *delay*, or both *area and delay*. The total number of K-LUTs in the final mapping is used as an estimate of the required implementation area. Likewise, the **depth** of the final mapping is used as a measure for the resulting circuit delay.

The adopted cost function is a mapping algorithm based on the Level-Map algorithm reported in [2]. It constructively maps an input BN into a functionally equivalent network whose nodes represent K-LUTs in linear time. Unlike Level-Map, the used cost function incorporates a measure for the input BN's depth. Upon completion, the returned cost is a weighted sum of the number of LUTs and the depth of the resulting mapping. The optimization target of the mapping is controlled by two user-defined weight factors c_1 and c_2 which determine the desired relative weights assigned to area optimization and delay optimization respectively. These optimization weight factors are chosen such that $c_1 + c_2 = 1$. Thus, if $c_1 = 1$ and $c_2 = 0$, the algorithm maps targeting area optimization only. Conversely, if $c_1 = 0$ and $c_2 = 1$, the algorithm maps targeting delay optimization only. Other values of c_1 and c_2 provide means for obtaining various area-delay trade-off solutions. Furthermore, the adopted cost function doesn't need the fanout factor used in Level-Map since, according to our state model, a fanout node in location NRF must be assigned to the output of a LUT.

The algorithm accepts a DAG representing the input BN and the parameter K which is the number of inputs to a LUT. It performs a topological sort of all nodes starting from the primary inputs. Let *ipred(v)* be the set of immediate predecessors of node v. The algorithm computes for each node v, its dependency d_v, its contribution z_v, and its depth D_v as follows (see Figure 5),

- Contribution z_v:
  a) $z_v = 1$ if v is a primary input or v is assigned a LUT,
  b) $z_v = \sum z_p \ \forall p \in ipred(v)$, otherwise.
- Dependency d_v:
  a) $d_v = 1$ if v is a primary input,
  b) $d_v = \sum z_p \ \forall p \in ipred(v)$, otherwise.
- Depth D_v:
  a) $D_v = 0$ if v is a primary input,
  b) $D_v = max(D_p) \ \forall p \in ipred(v)$, if v is not assigned a LUT,
  c) $D_v = max(D_p) + 1 \ \forall p \in ipred(v)$, if v is assigned a LUT.

If the dependency d_v of any node v is found to be greater than K, its immediate predecessor nodes (*ipred(v)*) are sorted in a descending order according to the following weight function,

$$Weight(p) = c_1 \times z_p - c_2 \times D_p \times \frac{K}{D_v}$$

LUTs are assigned to the predecessor nodes of v with larger weights till $d_v \leq K$. Once a node is assigned to be the output of a LUT, its z value is set to 1, and the d and D values of its successor nodes are accordingly updated.

The choice of the *Weight* function is justified by the following argument. If the mapping objective is area minimization, i.e. $c_1 = 1$ and $c_2 = 0$, then one should assign LUTs to nodes with larger contributions (z values) as this tends to reduce the number of LUTs. If the objective, however, is delay minimization, i.e. $c_1 = 0$ and $c_2 = 1$, then one should assign LUTs to nodes with smaller depths (D values) as this tends to reduce the overall depth of the mapped network. The scale factor $\frac{K}{D_v}$ is used to limit the coefficient of c_2 to a maximum value of K which is also the maximum value of the contribution term z_p.

The cost function algorithm assigns a LUT to every primary output node and every node at location NRF. The z values of these nodes are set to 1 and their D values are updated. Upon completion, the returned cost value is a weighted sum of the number of LUTs and the depth of the resulting mapping. This value is

used by the SE algorithm to determine the fitness or goodness of a given mapping. The control parameters $\hat{c_1}$ and $\hat{c_2}$ used in the weighted sum (Figure 5) are scaled versions of c_1 and c_2. The reason is that the number of LUTs and depth of the resulting mapping are not necessarily comparable in value. Therefore, there is a need to scale the control parameters c_1 and c_2 to reflect the target relative weights for area versus delay optimization. The scaled factors and the cost function are computed as follows,

- The cost function is invoked once with $c_1 = 1$ and $c_2 = 0$ (area optimization), the number of LUTs and depth of the resulting mapping are denoted N_1 and D_1 respectively.
- The cost function is invoked another time but with $c_1 = 0$ and $c_2 = 1$ (delay optimization) and the resulting number of LUTs and depth are denoted N_2 and D_2 respectively.
- Compute the scaled c_1, $\hat{c_1} = c_1/N_{ref}$ and the scaled c_2, $\hat{c_2} = c_2/D_{ref}$, where

$$N_{ref} = \frac{(N_1 + N_2)}{2} \quad ; \quad D_{ref} = \frac{(D_1 + D_2)}{2}$$

- Compute the total cost of a mapping as,

$$\begin{aligned} Cost &= \hat{c_1} \times N_{LUTs} + \hat{c_2} \times D \\ &= c_1 \times \frac{N_{LUTs}}{N_{ref}} + c_2 \times \frac{D}{D_{ref}} \end{aligned}$$

  where, N_{LUTs} and D are respectively the number of LUTs and depth of the resulting solution.

4 Results and Conclusion

SELF-Map has been tested on several MCNC benchmark circuits [10]. Prior to running SELF-MAP, the BN is first run through a SIS [9] technology-independent optimization script, followed by a node decomposition script which decomposes the BN into a 2-input BN where the number of inputs to each node does not exceed 2. This is because packing smaller nodes into K-LUTs generally results in a more efficient mapping.

The resulting BN is then fed to SELF-MAP. A LUT capacity of $K = 5$ has been used to allow comparison to solutions obtained by other reported technology mappers.

In case of area optimization, Table 1 compares the results obtained by Chortle-crf [3], GAFPGA [5], and mis-pga [6] to those obtained using SELF-Map. Out of 17 benchmark circuits, SELF-Map gives better results compared to all other techniques in a total of 6 circuits. SELF-Map results for the other circuits come somewhere in between. SELF-Map provides an overall saving in area of 4% over Chortle-crf, 4% over GAFPGA, and 10% over mis-pga. For two circuits (rd84 and 9sym), SELF-Map produced decisively much better mappings than any of the reported algorithms. We believe that this show of consistently good results is

Table 1: Results targeting area optimization

Circuit	Chortle-crf	GA-FPGA	mis-pga	SELF-MAP
z4ml	6	5	8	7
misex1	19	15	11	15
vg2	23	22	30	27
5xp1	28	22	31	34
count	31	31	31	39
9symml	55	57	56	52
9sym	64	56	72	42
apex7	64	61	64	67
rd84	45	43	40	25
e64	81	81	82	82
C880	86	86	103	98
alu2	112	112	129	94
duke2	123	119	128	114
C499	70	64	66	66
rot	203	200	200	214
apex6	213	194	243	200
des	955	1006	1016	908
Total	2178	2174	2310	2084

attributed to two main reasons: (1) The perturbation moves of SELF-Map cleverly allow the partial duplication of parts of any fanout cone, not necessarily entire fanout cones, thus exploring sub-search spaces that would have been otherwise missed; and (2) the fact that the SE algorithm occasionaly accepts higher cost solutions, thus allowing the search to escape from local minima.

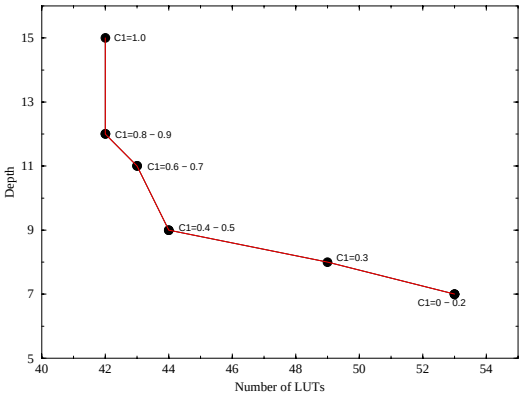

Figure 6: Depth versus Number of LUTS for various values of c_1 ($c_2 = 1 - c_1$). (*9sym* benchmark)

In case of delay optimization, Table 2 compares the LUTs depth produced by FlowMap [1], mis-pga(delay) [7], Chortle-d [4], and SELF-Map. As shown in the table, SELF-Map overall results are as good as Chortle-

Table 2: Results targeting delay optimization

Circuit	Flow-Map	mis-pga (delay)	Chortle-d	SELF-MAP
z4ml	3	2	3	3
misex1	2	2	2	3
vg2	4	4	4	4
5xp1	3	2	3	3
count	3	4	4	3
9symml	5	3	5	4
9sym	5	3	5	3
apex7	4	4	4	4
rd84	4	3	4	4
C880	8	9	8	8
alu2	8	6	9	7
duke2	4	6	4	5
C499	5	8	6	6
rot	6	7	6	6
apex6	4	5	4	4
alu4	10	11	10	11
des	5	11	6	9
Total	83	90	87	87

d, slightly better than mis-pga(delay), and slightly worse than FlowMap.

Figure 6 shows an example of area-delay trade-off for the '*9sym*' benchmark circuit, where the area weight factor c_1 is varied from 1 to 0, with a consequent variation of the delay weight factor ($c_2 = 1.0 - c_1$). Similar results were obtained for other benchmark circuits. The figure indicates that solutions with a small number of LUTs and a reasonable depth require that c_1 be in the range of 0.7 -to- 0.8. That is, one should not ignore the depth criterion even if the main objective is area optimization, as that would lead to a solution with unacceptable depth. A similar argument holds for the case when the main objective is delay (depth) minimization.

The run-time of SELF-MAP was found to be 25-70 times faster compared to that of the Genetic Algorithm approach (GAFPGA) [5]. This supports the low run-time claim of the SE algorithm [8].

Acknowledgement

The authors acknowledge the support of King Abdul-Aziz City of Science and Technology under research grant AR-11-67, and the support of King Fahd University of Petroleum & Minerals.

References

[1] J. Cong and Y. Ding. FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs. *IEEE Trans. CAD of Integrated and Systems*, 13(1):1–12, January 1994.

[2] Amir H. Farrahi and Majid Sarrafzadeh. Complexity of the Lookup-Table Minimization Problem for FPGA Technology Mapping. *IEEE Transaction of Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1319–1332, November 1994.

[3] R. Francis, J. Rose, and Z. Vranesic. Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs. *28th DAC*, pages 227–233, 1991.

[4] R. Francis, J. Rose, and Z. Vranesic. Technology Mapping of Lookup Table-Based FPGAs for Performance. *ICCAD*, pages 568–571, November 1991.

[5] V. Kommu and I. Pomeranz. GAFPGA: Genetic Algorithm for FPGA Technology Mapping. In *IEEE EURO-DAC*, pages 300–305, 1993.

[6] R. Murgai, Y. Nishizaki, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentlli. Logic Synthesis for Programmable Gate Arrays. *28th DAC*, pages 620–625, 1991.

[7] R. Murgai, N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentlli. Performance Directed Synthesis for Look Up Programmable Gate Arrays. *ICCAD*, pages 572–575, 1991.

[8] Youssef G. Saab and Vasant B. Rao. Combinatorial Optimization by Stochastic Evolution. *IEEE Transaction of Computer-Aided Design*, 10(4):525–535, April 1991.

[9] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. S. Vincentelli. SIS: A System for Sequential Circuit Synthesis. *Electronics Research Laboratory Memorandum*, (UCB/ERL M92/41), May 1992.

[10] Saeyang Yang. *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0.* Microelectronics Center of North Carolina, PO Box 12889, Research Triangle Park, NC 27709, January 15 1991.