DAG-Map: Graph Based FPGA Technology Mapping For Delay Optimization

Kuang-Chien Chen

Fujitsu America, Inc.
3055 Orchard Drive, San Jose, CA 95134

Jason Cong, Yuzheng Ding, Andrew Kahng, Peter Trajmar[1]

Department of Computer Science
University of California, Los Angeles, CA 90024

Abstract

In this paper, we present a graph based technology mapping algorithm, called DAG-Map, for delay optimization in lookup-table based FPGA designs. Our algorithm carries out technology mapping and delay optimization on the entire Boolean network, instead of decomposing the network into fanout-free trees and mapping each tree separately as in most previous FPGA technology mapping algorithms. As a preprocessing step of DAG-Map, we introduce a general algorithm for transforming an arbitrary n-input network into a two-input network with only $O(1)$ factor increase in the network depth; previous transformation procedures may result in an $\Omega(\log n)$ factor increase in the network depth. Finally, we present a graph matching based technique which performs area optimization without increasing the network delay; this is used as a postprocessing step for DAG-Map. We implemented the DAG-Map algorithm and tested it on the MCNC logic synthesis benchmarks. Compared with previous FPGA technology mapping algorithms for delay optimization (Chortle-d and MIS-pga), DAG-Map reduces both the network depth and the number of lookup-tables.

[1] Current address: Zycad Corporation, 47100 Bayside Parkway, Fremont, CA 94538.

1. Introduction

The Field Programmable Gate Array (FPGA) is a relatively new technology which allows the circuit designers to produce ASIC chips without going through the fabrication process. An FPGA chip usually consists of three components: programmable logic blocks, programmable interconnections, and programmable I/O blocks. Current technology implements programmable logic blocks using either K-input RAM/ROM lookup-tables (K-LUTs) [21] or programmable multiplexors [4]. Programmable interconnections consist of one-dimensional segmented channels or two-dimensional routing grids with switch-matrices. Programmable I/O blocks provide a user configurable interface between internal logic blocks and I/O pads. The design process for FPGAs is similar to that for conventional gate arrays or standard cells. Given a high-level design specification, the design process includes logic synthesis, technology mapping, placement, and routing. However, the end result is not a set of masks for fabrication, but rather a configuration matrix which sets the values of all the programmable elements in a FPGA chip. The fast turn-around time and low manufacturing cost have made the FPGA technology popular for system prototyping and low- or medium-volume production. Moreover, the lookup-table based FPGAs (such as those developed by Xilinx [21]) allow dynamic reconfiguration of the chip functionality, which leads to many interesting applications. However, the field programmable components usually introduce larger delay than conventional devices. Therefore, performance is a major consideration in many applications that use FPGA technology. In this paper, we study the technology mapping problem for delay optimization of lookup-table based FPGAs.

Given a synthesized Boolean network, the technology mapping problem is to implement the network using logic cells from a prescribed cell family. Much work has been done on the technology mapping problem for conventional gate array or standard cell designs (e.g. [13, 3]). In particular, it was shown that a Boolean network can be decomposed into a set of fanout-free trees and that the technology mapping problem can be solved optimally for each tree independently using a dynamic programming approach [13]. However, these methods do not apply immediately to the technology mapping problem for FPGAs since a K-LUT can implement any one of 2^{2^K} K-input logic gates, and consequently the equivalent cell family is too large to be manipulated efficiently.

Recently, a number of technology mapping algorithms have been proposed for area optimization in lookup-table based FPGA designs, where the objective is to minimize the number of programmable logic blocks in the mapping solution. The MIS-pga program developed by Murgai *et al.* [15] first decomposes a given Boolean network into a feasible network using Roth-Karp decomposition and kernel extraction so that the number of inputs at each node is bounded. MIS-pga then enumerates all the possible realizations of each network node and solves the binate covering problem to get a mapping solution using the least number of lookup-tables. In the improved MIS-pga (new) [17] more decomposition techniques are incorporated, including bin-packing, cofactoring, and AND-OR decomposition. The covering problem is solved more efficiently via certain preprocessing operations. The Chortle program and its successor Chortle-crf, developed by Francis *et al.* [5, 6], decomposes a given Boolean network into a set of fanout-free trees and then carries out technology mapping on each tree using the dynamic programming approach. Bin-packing heuristics are used in Chortle-crf for gate-level decomposition, yielding significant improvement over its predecessor in the quality of solutions and the running time. The Xmap program developed by Karplus [12] transforms a given Boolean network into an if-then-else DAG representation and then goes through a simple marking process to determine the final mapping. Another FPGA technology mapping algorithm was proposed by Woo [20], who introduces the notion of invisible edges to denote the edges which do not appear in the resulting network after mapping. A given network is first partitioned into subgraphs of reasonable size, and then an exhaustive procedure is used to determine the invisible edges in each subgraph.

Previous work on FPGA mapping for delay optimization consists of Chortle-d, developed by Francis *et al.* [7], and an extension of MIS-pga, developed by Murgai *et al.* [16]. The basic approach used in Chortle-d is similar to that in Chortle-crf, i.e., decompose the network into fanout-free trees and then use dynamic programming and bin-packing heuristics to map each tree independently, minimizing at each step the depth of the node being processed. Their method indeed reduces the depths of mapping solutions considerably. However, the method has two drawbacks. First, it decomposes the network into a set of fanout-free trees. Although it guarantees the optimal depth for each tree (when the input limit of each lookup-table is no more than 6), this *prior* decomposition usually results in sub-optimal depth for the overall network. Second, Chortle-d uses many more lookup-tables than are used by area optimization algorithms (MIS-pga and Chortle-crf).

The MIS-pga extension of [16] contains two phases, mapping and placement/routing. The mapping phase first computes a delay-optimized two-input network, then traverses the network from the primary inputs, collapsing the nodes in the longest paths into their fanouts to reduce the network depth. During this procedure various decomposition techniques are used to dynamically resynthesize the network, so this method uses a reduced number of look-up tables. The advantage of this approach is that it takes layout information into consideration at the technology mapping stage. However, on average it yields larger network depth than Chortle-d, especially for large networks, and requires much longer computation time.

In this paper, we present a graph based technology mapping algorithm, called DAG-Map, for delay optimization in lookup-table based FPGA design. Our algorithm carries out technology mapping and delay optimization on the entire Boolean network, instead of decomposing it into fanout-free trees as in Chortle-d. Our algorithm is optimal for trees for any K-LUTs while Chortle-d is optimal for trees only when K is no more than six [7]. For the preprocessing phase of DAG-Map, we introduce a general algorithm which transforms an arbitrary n-node network into a two-input network with only $O(1)$ factor increase in the network depth, while the previous transformation procedure may result in $\Omega(\log n)$ factor increase in the network depth. Finally, we present a matching based technique which minimizes area without increasing the network delay, and this is used in the postprocessing phase of DAG-Map. We have compared DAG-Map with previous FPGA mapping algorithms on a set of MCNC logic synthesis benchmarks. Our experimental results show that on average, DAG-Map reduces both the network delay and the number of lookup-tables when compared with either Chortle-d or the mapping phase of the MIS-pga extension for delay optimization.

The remainder of this paper is organized as follows. Section 2 gives the precise problem formulation. Section 3 describes our DAG-Map algorithm in detail. Experimental and comparative results are presented in Section 4. The proof of Theorem 1 is presented in the Appendix.

2. Problem Formulation

A Boolean network can be represented as a directed acyclic graph (DAG) where each node represents a logic gate and there is a directed edge (i, j) if the output of gate i is an input of gate j. A primary input (PI) node has no incoming edge and a primary output (PO) node has no outgoing edge. We use $input(v)$ to denote the set of nodes which supply inputs to gate v. Given a subgraph H of the Boolean network, $input(H)$ denotes the set of *distinct* nodes which supply inputs to the gates in H. For a node v in the network, a *K-feasible cone at v*, denoted C_v, is a subgraph consisting of v and predecessors[2] of v such that any path connecting a node in C_v and v lies entirely in C_v, and $|input(C_v)| \leq K$. The *level* of a node v is the length of the longest path from any PI node to v. The level of a PI node is zero. The *depth* of a network is the largest node level in the network.

We assume that each programmable logic block in an FPGA is a K-input lookup-table (K-LUT) that can implement any K-input Boolean function (this is true for the FPGA chips produced by Xilinx and AT&T [21, 9, 20]). Thus, each K-LUT can implement any K-feasible cone of a Boolean network. The technology mapping problem is then to cover a given Boolean network with K-feasible cones[3]. A technology mapping solution S is a DAG where each node is a K-feasible cone (equivalently, a K-LUT) and the edge (C_u, C_v) exists if u is in $input(C_v)$. Our goal is to compute a mapping solution that results in small circuit delay and, secondarily, uses small chip area. The delay of a FPGA circuit is determined by two parts: delay in K-LUTs and delay in the interconnection paths. Because layout information is not available at this stage, we simply approximate the circuit delay by the depth of S, since the access time of each K-LUT is independent of the function implemented. Therefore, the main objective of our algorithm is to determine a mapping solution S with minimum depth. Our secondary objective is area optimization, i.e., we minimize the number of lookup-tables after we have obtained a mapping solution with minimum delay.

3. The DAG-Map Algorithm

The DAG-Map algorithm consists of three major steps. The first step transforms an arbitrary Boolean network into a two-input network. The second step maps the two-input network into a K-LUT FPGA network with minimum delay. The third step performs a postprocessing area optimization of the FPGA network without increasing the network delay. This section describes these three steps in detail.

[2] u is a predecessor of v if there is a directed path from u to v.

[3] Note that we do not require the covering to be disjoint since we allow nodes in the network to be replicated, if necessary, as long as the resulting network is logically equivalent to the original one.

3.1. Transforming Arbitrary Networks into Two-Input Networks

As in [6, 7], we assume that each node in the given Boolean network is a simple gate (i.e. AND, OR, NAND, or NOR gate).[4] The first step of our algorithm is to transform the given Boolean network of simple gates into a two-input network (i.e. each gate in the network has at most two inputs). There are two reasons for carrying out such a transformation. First, we want to limit the number of inputs of each gate to be no more than K so that we do not have to decompose gates during technology mapping. Second, if we think of FPGA technology mapping as a process of packing gates in a given network into K-LUTs, then intuitively smaller gates will be more easily packed, with less wasted space in each K-LUT.

A straightforward way to transform an n-node arbitrary network into a two-input network is to replace each m-input gate ($m \geq 3$) by a balanced binary tree[5]. Fig. 1(a) shows a 4-input gate v (where the

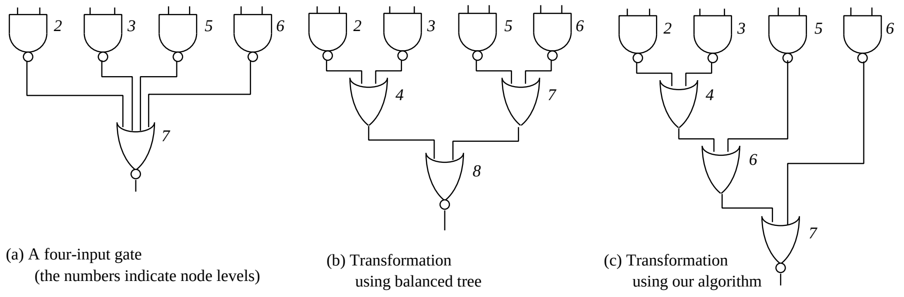


Fig. 1 Transforming a multi-input network into a two-input network.

```
algorithm decompose-multi-input-gate (DMIG)
    let $V = input(v) = \{u_1, u_2, ..., u_m\}$;
    while $|V| > 2$ do
        let $u_i$ and $u_j$ be the two nodes of $V$ with smallest levels;
        introduce a new node $x$;
        $input(x) = \{u_i, u_j\}$;
        $level'(x) = \max(level'(u_i), level'(u_j)) + 1$;
        $V = (V - \{u_i, u_j\}) \cup \{x\}$
    end-while;
    Connect the only two nodes left in $V$ to $v$ as its inputs;
    Return the binary tree $T(v)$ rooted at $v$;
end-algorithm.
```

Fig. 2 Algorithm DMIG.

[4] If the network has complex gates, we can represent each complex gate in the sum-of-products form and then replace it with by two levels of simple gates. In particular, we use the technology decomposition command `tech_decomp -o 1000 -a 1000` in MIS [1], which realizes such a transformation.

[5] The gate type of each node in the binary tree is the same as the gate type of the original multiple-input node. Such a transformation maintains logical equivalence as long as the gate function is associative.

numbers beside the nodes indicate their levels) and Fig. 1(b) shows the result of replacing it by a balanced binary tree. We see that the level of v increases from 7 to 8. In general, this straightforward transformation may increase the network depth by as much as an $\Omega(log\ n)$ factor. However, if we replace v by the binary tree shown in Fig. 1(c), the level of v remains 7. Our goal is to replace each multiple-input node by a binary tree so that the overall depth of the resulting network is as small as possible.

Given an arbitrary Boolean network G, our DMIG (Decompose-Multi-Input-Gate) algorithm transforms G into a two-input network G' as follows. We process the nodes in G in topological order starting from the PI nodes. At each multiple-input node v, we construct a binary tree $T(v)$ rooted at v using an algorithm similar to Huffman's algorithm for constructing a prefix code of minimum average length [11]. Writing $input(v) = \{u_1, u_2, ..., u_m\}$, note that nodes $u_1, u_2, ..., u_m$ have already been processed by the time we process v: their levels $level'(u_i)$ $(1 \leq i \leq m)$ in the new network G' have been determined. Intuitively, we want to combine nodes with smaller levels first when we construct the binary tree $T(v)$. The DMIG algorithm is shown in Fig. 2.

If we apply the DMIG algorithm to the example in Fig. 1(a), we indeed obtain the binary tree shown in Fig. 1(c). An algorithm similar to DMIG was proposed by Wang [19] for timing-driven decomposition in the synthesis of multi-level Boolean network. We have shown that the DMIG algorithm increases the network depth after the two-input decomposition by at most a small constant factor. The proof will be presented in the Appendix; similar analysis was carried out by Hoover, Klawe, and Pippenger in bounding the maximum degree of fanout in a Boolean network [10].

Theorem 1 For an arbitrary Boolean network G of simple gates[6], let G' be the network obtained by applying the DMIG algorithm to each multi-input gate in topological order starting from the PI nodes. Then $depth(G') \leq \log 2d \cdot depth(G) + \log I$, where d is the maximum degree of fanout in G and I is the number of PI nodes in G. □

Since in practice d is bounded by a constant (fanout limit of any output), the depth of the two-input network G' is increased by a only a constant factor $\log 2d$ away from $depth(G)$,[7] in contrast to the

[6] Any complex gate in the network can be decomposed into a two-level AND-OR subnetwork so that the increase of network depth is bounded by a factor of two.

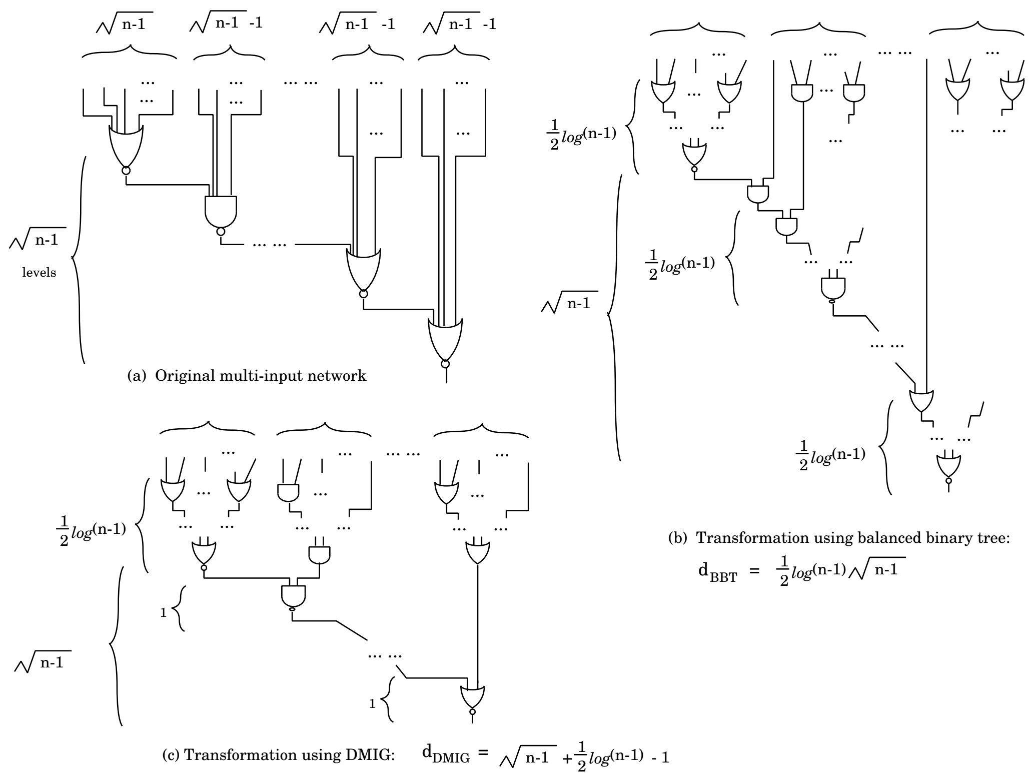

$\sqrt{n-1}$
$\sqrt{n-1}$ -1
$\sqrt{n-1}$ levels
(a) Original multi-input network
$\frac{1}{2}log(n-1)$
$\sqrt{n-1}$
(b) Transformation using balanced binary tree:
$d_{BBT} = \frac{1}{2}log(n-1)\sqrt{n-1}$
1
(c) Transformation using DMIG: $d_{DMIG} = \sqrt{n-1} + \frac{1}{2}log(n-1) - 1$

Fig. 3 A pathological example for the balanced binary tree transformation (assume $n = 2^{2m}+1$ for some m).

$\Omega(log\ n)$ increase which may occur with the balanced binary tree transformation. Fig. 3 shows a pathological example for the balanced binary tree transformation. The initial network of size n is shown in Fig. 3(a), which is a fanout free circuit of depth $\sqrt{n-1}$ (assuming that the primary inputs are at level 0). The two-input network after the balanced binary tree transformation (Fig. 3(b)) has depth d_{BBT}=½$log\,(n-1)\cdot\sqrt{n-1}$, even with d=1 in this case. Fig. 3(c) shows the DMIG transformation result, which has depth d_{DMIG}= $\sqrt{n-1}$+½$log\,(n-1)-1$. Clearly d_{DMIG} is much smaller than d_{BBT} when n is large.

The experimental results in Section 4 show that the two-input networks obtained using our transformation procedure lead to smaller network depths and better mapping solutions than those obtained using the transformation procedure in MIS [1].

[7] Here we assume that $depth(G) = \Omega(\log I)$, which is true for most networks in practice. This excludes the unrealistic case where log I is the dominating term in the right-hand side of the inequality in Theorem 1.

3.2. Technology Mapping for Delay Minimization

After we obtain a two-input Boolean network, we carry out technology mapping directly on the entire network. We use a method similar to that of Lawler *et al.* for module clustering to minimize delay in digital networks [14]. Our algorithm consists of two steps. We first label the network to determine the level of each node in the final mapping solution. We then generate the logically equivalent network of K-LUTs.

The first step assigns a label $h(v)$ to each node v of the two-input network, with $h(v)$ equal to the level of the K-LUT containing v in the final mapping solution. Clearly, we want $h(v)$ to be as small as possible in order to achieve delay minimization. We label the nodes in a topological order starting from the PI nodes. The label of each PI node is zero. If node v is not a PI node, let p be the maximum label of the nodes in $input(v)$. We use $N_p(v)$ to denote the set of predecessors of v with label p. Then, if $input(N_p(v) \cup \{v\}) \le K$, we assign $h(v) = p$; otherwise, we assign $h(v) = p + 1$. With this labeling, it is evident that $N_{h(v)}(v)$ forms a K-feasible cone at v for each node v in the network[8].

The second step generates K-LUTs in the mapping solution. Let L represent the set of outputs which are to be implemented using K-LUTs. Initially, L contains all the PO nodes. We process the nodes in L one by one. For each node v in L, we remove v from L and generate a K-LUT v' to implement the function of gate v such that $input(v') = input(N_{h(v)}(v))$. Then, we update the set L to be $L \cup input(v')$. The second step ends when L consists of only PI nodes in the original network. Clearly, we obtain a network of K-LUTs that is logically equivalent to the original network. The algorithm is summarized in Fig. 4.

The DAG-Map algorithm has several advantages:

(1) It works on the entire network without decomposing it into fanout-free trees; this usually leads to better mapping solutions. For example, decomposing the two-input network shown in Fig. 5(a) into fanout-free trees (as shown in Fig. 5(b)) yields a two-level mapping solution with three lookup-tables. However, the DAG-Map algorithm gives an one-level mapping solution with two lookup-tables (as shown in Fig. 5(c)).

---

[8] Note that $v \in N_{h(v)}(v)$ since v is a predecessor of itself.

```
**algorithm** DAG-Map
    /* step 1: labeling the network */
    for each PI node $v$ do
        $h(v) = 0$;
    $T$ = list of non-PI nodes in topological order;
    while $T$ is not empty do
        remove the first node $v$ from $T$;
        let $p = \max\{h(u) \mid u \in input(v)\}$;
        if $|input(N_p(v) \cup \{v\})| \le K$
        then $h(v) = p$
        else $h(v) = p + 1$
    end-while;
    /* step 2 : generate K-LUTs */
    $L$ = list of PO nodes;
    while $L$ contains non-PI nodes do
        remove a non-PI node $v$ from $L$, i.e. $L = L - \{v\}$;
        introduce a K-LUT $v'$ to implement the function of $v$ such that
                $input(v') = input(N_{h(v)}(v))$;
        $L = L \cup input(v')$
    end-while;
end-algorithm.
```

Fig. 4 Algorithm DAG-Map.

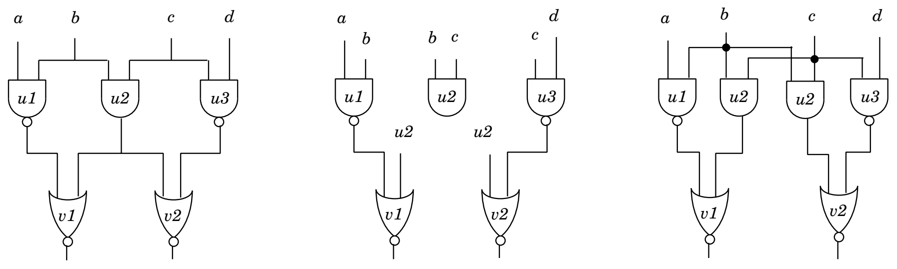

a
b
c
d
u1
u2
u3
v1
v2
a
b
b
c
c
d
u1
u2
u3
u2
u2
v1
v2
a
b
c
d
u1
u2
u2
u3
v1
v2
(a) Original network
(b) Decomposition into fanout-free trees
(c) Mapping solution by DAG-Map

Fig. 5 A mapping example for K=3.

(2) The DAG-Map algorithm will replicate nodes, if necessary, to minimize the network delay in the mapping solution. For the solution shown in Fig. 5(c), node u_2 is replicated to get a one-level mapping solution. Note that if node u_2 is not replicated, the depth of the mapping solution is at least two.

(3) The DAG-Map algorithm is optimal when the initial network is a tree.

Theorem 2 For any integer K, if the given Boolean network is a tree with fanin no more than K at each node, the DAG-Map algorithm produces a minimum depth mapping solution for K-LUT based FPGAs.

Proof It is easy to see that given a tree T, if the fanin limit of each node is K, the DAG-Map algorithm can successfully label all the nodes T. Moreover, for any node v in T, the label $h(v)$ is the level of LUT_v in the mapping solution produced by DAG-Map, where LUT_v is the K-LUT containing v. We will show that for any mapping solution M, the level of any node v satisfies $level_M(LUT_v) \geq h(v)$, where $level_M(LUT_v)$ is the level of the K-LUT LUT_v in M.

Assume toward a contradiction that M is a mapping solution such that $level_M(LUT_v) < h(v)$ for some node v. Furthermore, let v be the node with the lowest level in T such that $level_M(LUT_v) < h(v)$. Then, for any predecessor w of v, we have $level_M(LUT_w) \geq h(w)$. Let u be the predecessor of v with the maximum label $h(u) = p$. Since $level_M(LUT_v) \geq level_M(LUT_u) \geq h(u) = p$, and $h(v) \leq p + 1$ according to the labeling procedure of DAG-Map, we conclude that $level_M(LUT_v) = p$ and $h(v) = p + 1$. Note that $level_M(LUT_v) = p$ implies that $C_v \supseteq N_p(v) \cup \{v\}$, and $h(v) = p + 1$ implies that $|input(N_p(v) \cup \{v\})| > K$ according to the labeling procedure in the algorithm, where C_v is the K-feasible cone at v which is contained in LUT_v in M. However, since T is a tree, we have

$$|input(C_v)| \geq |input(N_p(v) \cup \{v\})| > K,$$

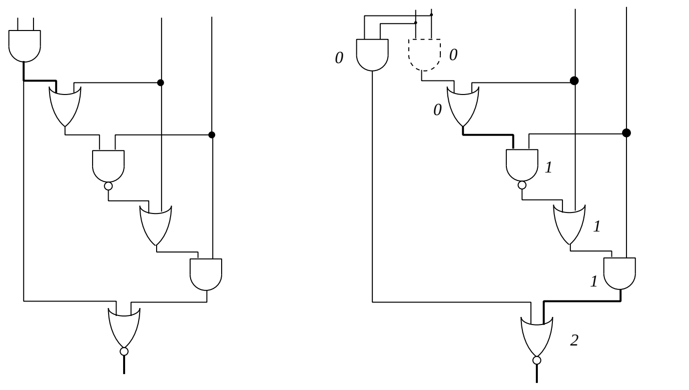
which contradicts the fact that C_v is K-feasible. $\square$

A result along similar lines was shown for Chortle-d in [7], but holds only for $K \leq 6$ since the Chortle-d bin-packing heuristics are no longer optimal for $K > 6$.[9]

Although the DAG-Map algorithm is optimal for trees, it may not be optimal for general networks. Fig. 6 shows an example where DAG-Map produces a sub-optimal mapping solution. However, DAG-Map would be optimal if the mapping constraint for each programmable logic block is monotone.

As defined in [14], a constraint X is *monotone* if a network H satisfying X implies that any subgraph of H also satisfies X. For example, if we limit the number of gates a programmable logic block may cover, it will be a monotone constraint. Unfortunately, limiting the number of distinct inputs of each programmable logic block is not a monotone constraint. In Fig. 7 the whole network has three distinct inputs, but the subnetwork consisting of t, v and w has four distinct inputs. However, the experimental results in Section 4 show that DAG-Map produces very satisfactory mapping solutions with respect to

[9] Chortle-d does not require the fanin limit of each node in the tree to be no more than K, since it carries out node decomposition during the bin-packing process.



(a) Optimal mapping (2 levels)

(b) Mapping by DAG-Map (3 levels)

Fig. 6 A pathological example for the DAG-Map algorithm.
(Assume K=3. Numbers are labels corresponding to node levels in mapping solution.)

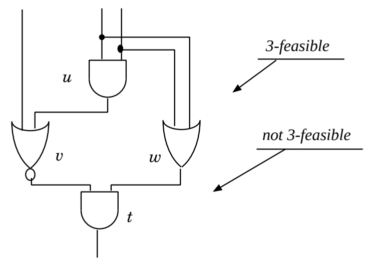


Fig. 7 Constraint on number of inputs of LUT is not monotone (Assume K=3).

delay optimization for all benchmark circuits.

3.3. Area Optimization Without Increasing Delay

Since the main objective of the DAG-Map algorithm is to optimize the depth of the mapping solution, minimizing the number of K-LUTs is not a consideration. For this reason, we have developed two operations for area optimization which are used in postprocessing steps after we obtain a mapping solution of small depth. These operations reduce the number of K-LUTs in the mapping solution without increasing the network depth. Note that in this subsection, each node in the network is a K-LUT instead of a simple gate as in the preceding subsections.

The first operation is called *gate decomposition*, which is inspired by the gate decomposition concept used in Chortle-crf [6]. The basic idea is as follows. If node v is a simple gate of multiple inputs in the mapping solution, for any two of its inputs u_i and u_j, if u_i and u_j are single fanout nodes, we can decompose v into two nodes v_{ij} and v' such that v' is of the same type as v and v_{ij} is of the same type as v in *non-negated form*, and $input(v_{ij}) = \{u_i,\ u_j\}$ and $input(v') = input(v) \cup \{v_{ij}\} - \{u_i,\ u_j\}$ (intuitively, u_i and u_j are fed into v_{ij} first and then v_{ij} replaces u_i and u_j as an input to v'). Such a decomposition produces a logically equivalent network because of the associativity of the simple functions. In this case, if $|input(u_i) \cup input(u_j)| \le K$, then we can implement u_i, u_j and v_{ij} using one K-LUT. The result is that the number of K-LUTs is reduced by one and the decomposed node v has one fewer inputs (which is beneficial to subsequent gate decomposition and predecessor packing). Figure 8 illustrates the gate decomposition operation, where the number of nodes, as well as the number of fanins of node v (v' after the operation), is reduced by one.

This method can be generalized to the case where the decomposed node v implements a complex function. In this case, we apply Roth-Karp decomposition [18] to determine if the node can be feasibly decomposed to v_{ij} and v' as in the preceding paragraph. Given a Boolean function $F(X,Y)$, where X and Y are Boolean vectors, the Roth-Karp decomposition determines if there is a pair of Boolean functions G and H such that $F(X,Y)=G(H(X),Y)$, and generates such G and H if they exist.[10]. In our case, F is the function implemented by v, $X=(u_i,u_j)$, and Y consists of the remaining inputs of v. If the Roth-Karp decomposition succeeds on a pair of inputs u_i and u_j of node v, and $|input(u_i) \cup input(u_j)| \le K$, then the gate-decomposition operation is applicable. In this case, we say u_i and u_j are *mergeable* and we call v the *base* of the merge. Although Roth-Karp decomposition may run in exponential time in general, it takes only constant time in our algorithm, since the number of fanins of a K-LUT is bounded by a small constant K.

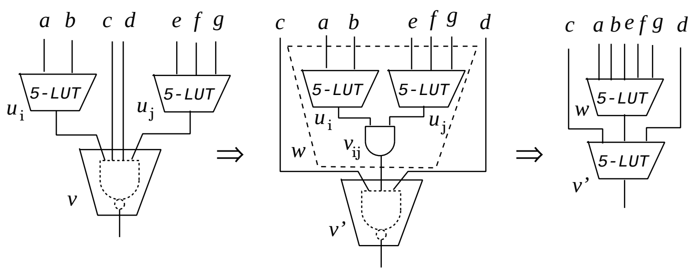


Fig. 8 Gate decomposition for area optimization (assume K=5).

[10] This actually is a special case of Roth-Karp decomposition. The details can be found in [18].

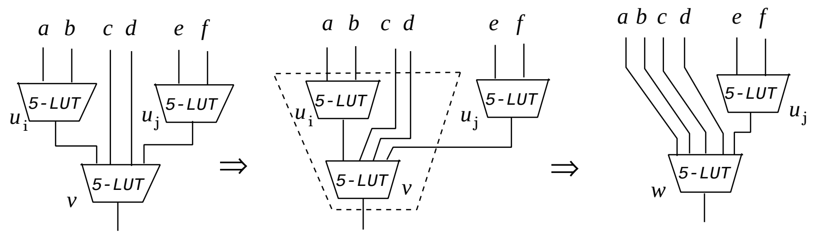


Fig. 9 Predecessor packing for area optimization (assume K=5).

Another postprocessing operation for area optimization is called *predecessor packing*. The concept behind this method is simple. For each node v, we examine all of its input nodes. If $|input(v) \cup input(u_i)| \leq K$ for some input node u_i, and u_i has only a single fanout, then v and u_i are merged into a single K-LUT. In this case we also say that node u_i and v are *mergeable*, and call v the *base* of the merge. This operation reduces the number of K-LUTs by one. Unlike the gate decomposition method where the number of inputs to the current node v is reduced by one, with this method the number of inputs is actually increased by $|input(u_i) - input(v)|$. Although it is less conducive to subsequent gate-decomposition or predecessor packing, the number of instances to which this operation applies is large. Fig. 9 shows an example of the predecessor packing operation. In this example predecessor packing leads to a solution with the same depth as the original network but one fewer K-LUT.

There are usually many pairs of mergeable nodes in a network, but not all of these merge operations can be performed at the same time. We thus use a graph matching approach to avoid merging nodes in arbitrary order; this achieves a globally good result. We construct an undirected graph $G=(V,E)$, where the vertex set V represents the nodes of the K-LUT network, and an edge (v_i, v_j) is in the edge set E if and only if v_i and v_j are mergeable. Clearly a maximum cardinality matching in G corresponds to a maximum set of merge operations that can be applied simultaneously. Therefore we find a maximum matching in G and apply the merge operations corresponding to the matched edges. We then re-construct the graph G for the reduced network and repeat the above procedure until we are unable to construct a non-empty E. The experimental results show that this matching based merge algorithm usually converges after only one or two iterations. Since the maximum graph matching problem can be solved in $O(n^3)$ time [8], our area optimization procedure can be implemented efficiently.[11]

Note that in the above discussion concerning these two operations we assume that each node in a mergeable pair has only a single fanout, unless it is also the base of the merge for predecessor packing. This is because the resulting K-LUT must have only one output. If a node u in a mergeable pair is not the base of the merge and has multiple fanouts, the application of the merge operation requires u to be replicated so that the copy involved in the merge operation is fanout free. However, unless every fanout node of u is a base of some merge operation that involves u, we cannot reduce the number of nodes in the network, since there will always be a remaining copy of u which is not merged to any of its fanout nodes.

We say a node u is *removable* if and only if for *each* of its fanout nodes v_i, either u and v_i are mergeable via predecessor packing, or there is another fanin node of v_i, say u_j, such that u and u_j are mergeable via gate decomposition. Fig. 10 shows three different cases where node u is a removable node. For a removable node u, each of its fanout nodes is a base of a merge operation involving u, and u is removed if all these merge operations are applied simultaneously. Therefore, for a removable node u, we define a *mergeable set* of u, denoted as R_u, to be a set of nodes involved in removing u. More precisely, R_u contains u itself and *exactly one* nodes for each fanout node v_i of u, which is selected in the following ways: (1) if v_i is the base of a predecessor packing operation involving u, then we can select v_i as u_i; or (2) if v_i is the base of a gate decomposition operation involving u, then we can select u_i to be the node other than u involved in this gate decomposition. Note that a removable node may have more than one mergeable set. For example, in Fig. 10 node u has mergeable sets $\{u,u_1,u_2\}$, $\{u,v_1,v_2\}$, $\{u,u_1,v_2\}$, and $\{u,v_1,u_2\}$ (the last one is not shown in the figure). If u is fanout free, then a mergeable set of u is a mergeable pair defined previously. Therefore, a mergeable pair is a special case of a mergeable set.

In order to reduce the number of K-LUTs as much as possible, we want to determine a maximum collection of mergeable sets for which merge operations can be performed independently. This becomes the matching problem in a *hypergraph*. We construct a hypergraph $H=(V,E)$ for the K-LUT network, where the vertices in V represent the nodes of the network, and the *hyperedges* in E represent the mergeable sets. Note that H contains the simple graph $G=(V,E_2)$, representing mergeable pairs for fanout free nodes, as a subgraph. We call the edges in E_2 the *simple* edges, and call the edges in $E - E_2$, each of

[11] We used a standard procedure for maximum cardinality matching in undirected graphs, written by Ed Rothberg, that implements Gabow's algorithm [8].

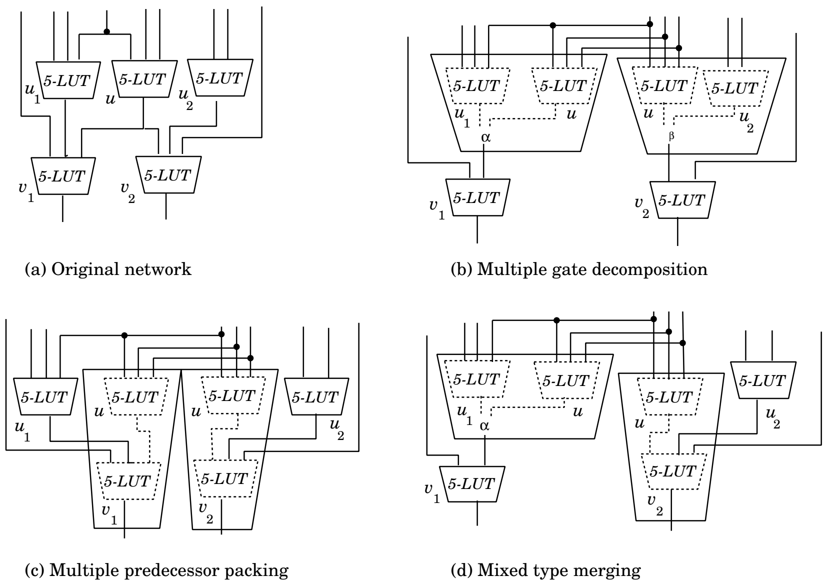


Fig. 10. Merge operations on multi-fanout node *u* (assume K=5).

which contains more than two vertices, the *non-simple* edges. A *matching* in H is a set of disjoint edges in E. It is easily seen that a maximum matching in H yields the maximum number of K-LUTs in the network that can be removed. However, the maximum matching problem in a hypergraph is NP-complete. Instead of solving this problem optimally, we compute an approximate solution as follows. First, we construct the hypergraph $H=(V,E)$ for the K-LUT network as described above. Then, we identify the subgraph $G=(V,E_2)$ of H, consisting of all the simple edges in H. Next, we find a maximum cardinality matching $M_2 \subseteq E_2$ in G using Gabow's algorithm [8]. This matching will be included in the approximate solution for hypergraph matching. Let E_m be the set of non-simple edges in H that are disjoint from the edges in M_2. We use an exhaustive search procedure to find a maximum matching $M_m \subseteq E_m$ and return $M_2 \cup M_m$ as the approximate maximum matching solution in H.

In practice, $|E_m|$ is quite small. For example, for all the benchmark circuits we used in our experiments, $|E_m|$ never exceeds 10. Therefore, M_m can be computed efficiently.

It is obvious that this algorithm finds a *maximal* hypergraph matching. Although it may not be maximum, we have the following bound.

Theorem 3 Given a hypergraph H, let M^* be a maximum matching of H, and $M = M_2 \cup M_m$ be the matching computed using the above algorithm, then $|M^*| \le 2|M|$.

Proof Let $M^+=M_2-M^*$, which is the set of simple edges that are in M but not in M^*, and $M^-=M^*-M_2$. If M^+ is not empty, $M^*\cup M^+$ is not a matching since M^* is maximum. But because M_2 is also a matching, and $M^+\subseteq M_2$, we can always find a set $S\subseteq M^-$ such that $M'=(M^*\cup M^+)-S$ is a maximal matching. It is easy to see that $M_2\subseteq M'$, and $|M^*|-|M'|=|S|-|M^+|$. Since adding a simple edge to any matching results in the removal of at most two edges for maintaining the matching property, we have $|S|\leq 2|M^+|$. Therefore, $|M^*|-|M'|\leq |M^+|$.

Since M_2 is a maximum matching among all the simple edges in H, M' cannot contain any simple edges other than those in M_2. So $M'-M_2$ is a maximal matching among the non-simple edges in H which are disjoint with the edges in M_2. According to the construction of M_m, we have $|M'-M_2|\leq |M_m|$, which leads to $|M'|\leq |M|$.

Therefore, we have $|M^*|-|M|\leq |M^+|$. Since $|M^+|\leq |M|$, we conclude that $|M^*|\leq 2|M|$. □

For general hypergraphs, this bound is *tight*. However, for hypergraphs that contain only a few non-simple edges, we can obtain a better bound.

Corollary 1 If the number of non-simple edges in a hypergraph H is $h(H)$, then $|M^*|-|M|\leq h(H)-|M_m|$.

Proof Since $M'\cap S=\emptyset$, and M' is a maximal matching, we have $M_m\cap S=\emptyset$ (otherwise M' can be augmented by the edges in $M_m\cap S$). Therefore, S contains no more than $h(H)-|M_m|$ non-simple edges. On the other hand, S contains no more than $|M^+|$ simple edges (otherwise M^* would contain more simple edges than M_2, which contradicts that fact that M_2 is a maximum matching of the simple edges in H). This implies that S contains at least $|S|-|M^+|$ non-simple edges.

Moreover, from the proof of Theorem 3, we know that $|M^*|-|M|\leq |M^*|-|M'|=|S|-|M^+|$. Therefore, we have $|M^*|-|M|\leq |S|-|M^+|\leq h(H)-|M_m|$. □

For all the benchmark circuits that we used in our experiments, we applied Corollary 1 and found out that the error bound $h(H)-|M_m|$ is never greater than 4.

4. Experimental Results

We implemented the DAG-Map algorithm using the C language on Sun SPARC workstations. We integrated our program as an extension of the MIS system so that we could exploit input/output routines and other functions provided by MIS. DAG-Map was tested on a large number of MCNC benchmark examples and results were compared with both those produced by Chortle-d [7], and those produced by the mapping phase of the MIS-pga delay optimization algorithm [16].

As in [7, 16], we chose the size of the K-LUT to be $K = 5$, reflecting, e.g. the XC 3000 FPGA family produced by Xilinx [21]. For each input network, we first applied the DMIG algorithm to transform it into a two-input network. We then used DAG-Map to map into a 5-LUT network. Finally, the matching based postprocessing step was performed. Table 1 shows the comparison of the results of our algorithm with those of the Chortle-d algorithm and those of the mapping phase of the MIS-pga delay optimization algorithm.

Technology Mapping for 5-LUT FPGAs									
	chortle-d			**MIS-pga (d)**			**DAG-Map**		
	LUTs	**depth**	**time**	**LUTs**	**depth**	**time**	**LUTs**	**depth**	**time**
5xp1	26	3	0.1	21	2	3.5	22	3	1.1
9sym	63	5	0.2	7	3	15.2	60	5	2.3
9symml	59	5	0.1	7	3	9.9	55	5	2.5
C499	382	6	1.8	199	8	58.8	68	4	12.2
C880	329	8	0.9	259	9	39.0	128	8	6.3
alu2	227	9	0.7	122	6	42.6	156	9	7.8
alu4	500	10	0.3	155	11	15.4	272	10	16.5
apex6	308	4	0.8	274	5	60.0	246	5	10.9
apex7	108	4	0.2	95	4	8.4	81	4	3.0
count	91	4	0.1	81	4	5.1	31	5	1.4
des	2086	6	9.2	1397	11	937.8	1423	5	91.2
duke2	241	4	0.4	164	6	16.4	177	4	4.9
misex1	19	2	0.1	17	2	1.7	16	2	0.7
rd84	61	4	0.2	13	3	9.8	46	4	2.5
rot	326	6	1.0	322	7	50.0	246	7	11.1
vg2	55	4	0.1	39	4	1.7	29	3	0.9
z4ml	25	3	0.1	10	2	2.1	5	2	0.3
total	4906	87	16.3	3182	90	1277.4	3062	85	175.6
comparison	+60%	+2%	-	+4%	+6%	-	1	1	-

Table 1 Comparison with Chortle-d and MIS-pga (delay optimization) programs.

The input networks to Chortle-d and DAG-Map are obtained from the original benchmarks using the same standard MIS technology independent optimization script which was used by Francis *et al.* [7], except that Chortle-d also goes through another speed-up step for delay optimization. A direct comparison with MIS-pga is difficult since it combines logic optimization and technology mapping. Nevertheless, we include the mapping results produced by the MIS-pga delay optimization algorithm (quoted from [16]) for reference. The running time (sec) of our algorithm, which includes transformation, mapping, and postprocessing, was recorded on a Sun SPARC IPC (15.8 MIPS); the running time of the other two algorithms is quoted from [16], where the authors used a DEC5500 machine (28 MIPS). Overall, the solutions of Chortle-d used 60% more lookup-tables and had 2% larger network depth, and the solutions of MIS-pga with delay optimization used 4% more lookup-tables and had 6% larger network depth. In all cases, the running time of our algorithm is no more than 100 seconds.

Comparison of 2-Input Network Transformation Methods								
	Before Mappings				**After 5-LUT Mappings**			
	mis tech_decomp		**DMIG algo**		**mis tech_decomp**		**DMIG algo**	
	gates	**depth**	**gates**	**depth**	**gates**	**depth**	**gates**	**depth**
5xp1	88	9	88	9	22	3	22	3
9sym	201	16	201	13	65	5	60	5
9symml	199	17	199	13	61	5	55	5
C499	392	25	392	25	66	4	68	4
C880	347	37	347	35	131	8	128	8
alu2	371	36	371	31	159	10	156	9
alu4	664	40	664	34	263	11	272	10
apex6	651	16	651	15	250	6	246	5
apex7	201	14	201	13	80	4	82	4
count	112	20	112	19	31	5	31	5
des	3049	19	3049	16	1461	6	1423	5
duke2	325	16	325	11	177	5	177	4
misex1	49	6	49	6	19	2	16	2
rd84	153	14	153	11	44	4	46	4
rot	539	27	539	21	256	7	246	7
vg2	72	15	72	10	29	4	29	3
z4ml	27	10	27	10	5	2	5	2
total	7440	337	7440	292	3119	91	3062	85
comparison	+0%	+15%	1	1	+2%	+7%	1	1

Table 2 Comparison of two-input network transformation algorithms.

Effectiveness of Post-Processing Step for Depth Minimized 5-Input Lookup Table Mappings				
	original		**after post-processing**	
	LUTs	**depth**	**LUTs**	**depth**
5xp1	25	3	22	3
9sym	76	5	60	5
9symml	68	5	55	5
C499	80	4	68	4
C880	137	8	128	8
alu2	169	9	156	9
alu4	301	10	272	10
apex6	313	5	246	5
apex7	101	4	82	4
count	43	5	31	5
des	1674	5	1423	5
duke2	196	4	177	4
misex1	20	2	16	2
rd84	51	4	46	4
rot	275	7	246	7
vg2	32	3	29	3
z4ml	5	2	5	2
total	3566	85	3062	85
comparison	+16%	+0%	1	1

Table 3 Effect of the postprocessing steps for area minimization.

In order to judge the effectiveness of our DMIG algorithm for transforming the initial network into a two-input network, we compared it with the transformation procedure in MIS. Both our DMIG algorithm and the MIS decomposition command `tech_decomp -a 2 -o 2` were applied to the same initial networks[12]. We also ran the DAG-Map algorithm on each set of the resulting two-input networks. In Table 2, the first four columns compare the number of gates and the depth of the two-input networks produced by the two algorithms, while the last four columns compare the number of 5-LUTs and the depth of the 5-LUT network after DAG-Map is applied to the different two-input networks produced by the two algorithms. In all cases, the DMIG procedure resulted in smaller or the same depths in both the two-input networks after decomposition and the 5-LUT networks after mapping, and on average it used fewer lookup-tables. (Since both algorithms decompose a network into a binary tree, the number of gates in the resulting two-input networks is always the same.)

[12] Again, the initial networks were optimized using the MIS minimization script as in the preceding experiment, and in addition to this we also used `tech_decomp -a 1000 -o 1000` to transform it into simple gate network.

Finally, we also tested the effectiveness of the postprocessing procedure for area optimization used in our algorithm, and the results are shown in Table 3. The first two columns show the statistics for the mapping solutions produced by DAG-Map without any postpreprocessing for area optimization. The last two columns describe the same solutions after the postprocessing. The total number of lookup-tables is reduced by 16%.

5. Conclusions

In this paper, we have presented a graph based technology mapping algorithm for delay optimization in lookup-table based FPGA design. It carries out technology mapping and delay optimization on the entire Boolean network. Our algorithm consists of three main steps: transformation of an arbitrary network into a two-input network, technology mapping on the entire two-input network for delay minimization, and area optimization in the mapping solution. We have also presented several theoretical results which show the effectiveness of our algorithm. The algorithm has been tested on a large set of benchmark examples and gives satisfactory results.

In conclusion, this work shows that the graph-based technology-mapping approach is more effective than the existing tree-based technology-mapping approaches in LUT-based FPGA designs. After this work was submitted, Cong and Ding recently showed that the LUT-based FPGA technology mapping-problem for delay optimization can be solved optimally in polynomial time [2].

6. Acknowledgments

We thank Dr. Masakatsu Sugimoto for his support of this research. Support from the NSF via grants MIP-9110511 and MIP-9110696 is also gratefully acknowledged. We thank Professor Jonathan Rose and Robert Francis for providing the Chortle programs and necessary assistance for our comparative study.

References

[1] Brayton, R. K., R. Rudell, and A. L. Sangiovanni-Vincentelli, ‘‘MIS: A Multiple-Level Logic Optimization,’’ *IEEE Transactions on CAD*, pp. 1062-1081, Nov. 1987.

[2] Cong, J. and Y. Ding, ‘‘An Optimal Technology Mapping Algorithm fo Delay Optimization in Lookup-Table Based FPGA Designs,’’ *Proc. IEEE Int’l Conf. on Computer-Aided Design*, pp. 48-53, Nov. 1992.

[3] Detjens, E., G. Gannot, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, ‘‘Technology Mapping in MIS,’’ *Proc. IEEE Int’l Conf. on Computer-Aided Design*, pp. 116-119, Nov. 1987.

[4] El Gamal, A. et al, ‘‘An Architecture for Electrically Configurable Gate Arrays,’’ *IEEE J. Solid-State Circuits*, Vol. **24**, pp. 394-398, April 1989.

[5] Francis, R. J., J. Rose, and K. Chung, ‘‘Chortle: A Technology Mapping Program for Lookup Table-Based Field Programmable Gate Arrays,’’ *Proc. 27th ACM/IEEE Design Automation Conference*, pp. 613-619, June 1990.

[6] Francis, R. J., J. Rose, and Z. Vranesic, ‘‘Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs,’’ *Proc. 28th ACM/IEEE Design Automation Conference,* pp. 613-619, June 1991.

[7] Francis, R. J., J. Rose, and Z. Vranesic, ‘‘Technology Mapping of Lookup Table-Based FPGAs for Performance,’’ *Proc. IEEE Int’l Conf. on Computer-Aided Design*, pp. 568-571, Nov. 1991.

[8] Gabow, H., ‘‘An Efficient Implementation of Edmonds’ Algorithm for Maximum Matching on Graphs,’’ *Journal of the ACM*, Vol. **23**, pp. 221-234, Apr. 1976.

[9] Hill, D., ‘‘A CAD System for the Design of Field Programmable Gate Arrays,’’ *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 187-192, June 1991.

[10] Hoover, H. J., M. M. Klawe, and N. J. Pippenger, ‘‘Bounding Fan-out in Logic Networks,’’ *Journal of Association for Computing Machinery*, Vol. **31**, pp. 13-18, Jan. 1984.

[11] Huffman, D. A., ‘‘A method for the construction of minimum redundancy codes,’’ *Proc. IRE 40*, pp. 1098-1101, 1952.

[12] Karplus, K., ‘‘Xmap: A Technology Mapper for Table-lookup Field-Programmable Gate Arrays,’’ *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 240-243, June 1991.

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

[14] Lawler, E. L., K. N. Levitt, and J. Turner, ‘‘Module Clustering to Minimize Delay in Digital Networks,’’ *IEEE Transactions on Computers*, Vol. **C-18**(1) pp. 47-57, January 1969.

[15] Murgai, R., Y. Nishizaki, N. Shenay, R. Brayton, and A. Sangiovanni-Vincentelli, ‘‘Logic Synthesis Algorithms for Programmable Gate Arrays,’’ *Proc. 27th ACM/IEEE Design Automation Conf.*, pp. 620-625, 1990.

[16] Murgai, R., N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, ‘‘Performance Directed Synthesis for Table Look Up Programmable Gate Arrays,’’ *Proc. IEEE Int’l Conf. on Computer-Aided Design*, pp. 572-575, Nov. 1991.

[17] Murgai, R., N. Shenoy, R. K. Brayton, and A. Sangiovanni-Vincentelli, ‘‘Improved Logic Synthesis Algorithms for Table Look Up Architectures ,’’ *Proc. IEEE Int’l Conf. on Computer-Aided Design*, pp. 564-567, Nov. 1991.

[18] Roth, J. P. and R. M. Karp, ‘‘Minimization Over Boolean Graphs,’’ *IBM Journal of Research and Development*, pp. 227-238, April 1962.

[19] Wang, A., ‘‘Algorithms for Multi-level Logic Optimization,’’ *U.C.Berkeley Memorandum No. UCB/ERL M89/50,* April 1989.

[20] Woo, N.-S., ‘‘A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility,’’ *Proc. 28th ACM/IEEE Design Automation Conference*, pp. 248-251, June 1991.

[21] Xilinx, *User Guide and Tutorials,* Xilinx, San Jose (1991).

Appendix

We present the proof of Theorem 1 in this Appendix. First, we show the following lemma.

Lemma Let $V = \{u_1, u_2, ..., u_m\}$ be the set of inputs of a multi-input node v in the initial network G. Then, after applying the DMIG algorithm we have

$$2^{level'(v)} \le \sum_{i=1}^{m} 2^{level'(u_i)+1}$$

where $level'(x)$ is the level of node x in the two-input network G'.

Proof It is easy to see that the DMIG algorithm will introduce $m-2$ new nodes in processing v. (For any binary tree, the number of leaves equals the number of internal nodes (including the root) plus one.) Let $X_i = \{x_{i,1}, x_{i,2}, ... x_{i,m-i}\}$ be the set of nodes left in V after the DMIG algorithm introduces the i-th new node. Clearly, $X_0 = \{u_1, u_2, ..., u_m\}$. For convenience, we define $X_{m-1} = \{v\}$. Without loss of generality, assume that $level'(x_{i,1}) \le level'(x_{i,2}) \le \cdots \le level'(x_{i,m-i})$. Then, the $(i+1)$-th new node has $x_{i,1}$ and $x_{i,2}$ as its inputs, and its level is given by $level'(x_{i,2}) + 1$. Therefore, we have

$$\begin{aligned}\sum_{x \in X_{i+1}} 2^{level'(x)} &= 2^{level'(x_{i,2})+1} - 2^{level'(x_{i,1})} - 2^{level'(x_{i,2})} + \sum_{x \in X_i} 2^{level'(x)} \\ &= 2^{level'(x_{i,2})} - 2^{level'(x_{i,1})} + \sum_{x \in X_i} 2^{level'(x)}\end{aligned}$$

Taking the sum of both sides of the last equation from $i = 0$ to $m-2$, we have

$$\sum_{i=0}^{m-2} \sum_{x \in X_{i+1}} 2^{level'(x)} = \sum_{i=0}^{m-2} (2^{level'(x_{i,2})} - 2^{level'(x_{i,1})}) + \sum_{i=0}^{m-2} \sum_{x \in X_i} 2^{level'(x)}$$

Subtracting $\sum_{i=1}^{m-2} \sum_{x \in X_i} 2^{level'(x)}$ from both sides, we get

$$2^{level'(v)} = \sum_{i=0}^{m-2} (2^{level'(x_{i,2})} - 2^{level'(x_{i,1})}) + \sum_{x \in X_0} 2^{level'(x)}$$

Note that

$$\sum_{i=0}^{m-2} (2^{level'(x_{i,2})} - 2^{level'(x_{i,1})}) = \sum_{i=0}^{m-2} (2^{level'(x_{i,2})} - 2^{level'(x_{i+1,1})}) + 2^{level'(x_{m-2,2})} - 2^{level'(x_{0,1})}$$

Moreover, $2^{level'(x_{i,2})} - 2^{level'(x_{i+1,1})} \le 0$ for any $0 \le i \le m-2$ (since $x_{i+1,1}$ is either the $(i+1)$-th new node or a node in $X_i - \{x_{i,1}, x_{i,2}\}$; in either case we have $level'(x_{i+1,1}) \ge level'(x_{i,2})$) and $2^{level'(x_{m-2,2})} = \frac{1}{2} \cdot 2^{level'(v)}$. It follows that

$$2^{level'(v)} \le \frac{1}{2} \cdot 2^{level'(v)} - 2^{level'(x_{0,1})} + \sum_{x \in X_0} 2^{level'(x)}$$

Therefore,

$$2^{level'(v)} \le 2\cdot(\sum_{x\in X_0} 2^{level'(x)} - 2^{level'(x_{0,1})}) < \sum_{x\in X_0} 2^{level'(x)+1} = \sum_{i=1}^{m} 2^{level'(u_i)+1} \qquad \square$$

Based on this lemma, we present the proof of Theorem 1 as follows.

Theorem 1 For an arbitrary Boolean network G of simple gates, let G' be the network obtained by applying the DMIG algorithm to each multi-input gate in topological order starting from the PI nodes. Then $depth(G') \le \log 2d \cdot depth(G) + \log I$, where d is the maximum degree of fanout in G and I is the number of PI nodes in G.

Proof Let H denote $depth(G)$. Let L_i denote the set of nodes $\{x \mid x\in G,\ level(x) = i\}$. (Note that $level(x)$ is the level of node x in the initial network G.) Let A_i denote the set of nodes x in G such that $level(x) \le i$ and x has at least a fanout y with $level(y) > i$. We will prove by induction that

$$\sum_{v\in A_i} 2^{level'(v)} \le (2d)^i \cdot I. \tag{*}$$

Since A_0 is the set of PI nodes in G, the inequality (*) holds for $i = 0$. Suppose that the inequality holds for $i - 1$, we want to show that it also holds for i. According to the definition of A_i, it is not difficult to see that $A_i \subseteq (A_i \cap A_{i-1}) \cup L_i$. Moreover, each node v in $A_i \cap A_{i-1}$ has at most $d - 1$ fanouts in L_i. According to the lemma, we have

$$\sum_{v\in A_i} 2^{level'(v)} \le \sum_{v\in A_{i-1}\cap A_i} 2^{level'(v)} + \sum_{v\in L_i} 2^{level'(v)}$$

$$\le \sum_{v\in A_{i-1}\cap A_i} 2^{level'(v)} + (d-1)\cdot \sum_{v\in A_{i-1}\cap A_i} 2^{level'(v)+1} + d\cdot \sum_{v\in A_{i-1}-A_i} 2^{level'(v)+1}$$

$$\le 2d\cdot \sum_{v\in A_{i-1}} 2^{level'(v)}$$

By induction hypothesis, we have

$$\sum_{v\in A_i} 2^{level'(v)} \le 2d\cdot[(2d)^{i-1}\cdot I] = (2d)^i \cdot I$$

It concludes that the inequality (*) holds for any $0 \le i \le H$. Let w be node in G that achieves the maximum level in G', then all the inputs of w are in A_{H-1}. According to the lemma and the inequality (*), we have

$$2^{level'(w)} \le \sum_{v\in input(w)} 2^{level'(v)+1} \le \sum_{v\in A_{H-1}} 2^{level'(v)+1} \le 2\cdot(2d)^{H-1}\cdot I \le (2d)^H \cdot I.$$

Therefore,

$$depth(G') = level'(w) \le \log[(2d)^H \cdot I] \le \log 2d \cdot H + \log I. \qquad \square$$