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

KUANG-CHIEN CHEN
Fujitsu America
JASON CONG
YUZHENG DING
ANDREW B. KAHNG
PETER TRAJMAR
University of California, Los Angeles

This article presents a graph-based technology-mapping package for delay optimization in lookup-table-based FPGA designs. In contrast to most previous techniques, DAG-Map carries out technology mapping and delay optimization on the entire Boolean network. The package consists of three main parts: transformation of an arbitrary network into a two-input network, technology mapping for delay minimization, and area optimization in the mapping solution. The authors show DAG-Map's effectiveness in comparative benchmark tests.

THE FIELD-PROGRAMMABLE gate array is a relatively new technology that allows 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 with either *K*-input RAM/ROM lookup tables (*K*-LUTs)[1] or programmable multiplexers.[2] 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 and standard cells. Starting from a high-level design specification, the 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 that sets the values of all the programmable elements in an FPGA chip.

Fast turnaround and low manufacturing cost have made FPGA technology popular for system prototyping and low- or medium-volume production. Moreover, lookup-table-based FPGAs (such as those developed by Xilinx[1]) allow dynamic reconfiguration of chip functions, leading to many interesting applications. But field-programmable components usually introduce longer delays than conventional devices, so performance is a major consideration in many applications.

In this article, we study the technology-mapping problem for delay optimization of lookup-table-based FPGAs. The technology-mapping problem is to implement a synthesized Boolean network with logic cells from a prescribed cell family. Researchers have done much work on the technology-mapping problem for conventional gate array or standard cell designs.[3,4] In particular, Keutzer showed that a Boolean network can be decomposed into a set of fan-out-free trees and that optimal technology mapping can be performed for each tree independently with a dynamic programming approach.[4] However, the conventional methods do not apply directly to the technology-mapping problem for FPGAs; 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, researchers have proposed a number of technology-mapping algorithms for area optimization in lookup-table-based FPGA designs, with the objective of minimizing the number of programmable logic blocks in the mapping solution. The MIS-pga program, developed by Murgai et al.,[5] first decomposes a Boolean net-

On average, DAG-Map reduces both network delay and number of lookup tables.

work into a feasible network, using Roth-Karp decomposition and kernel extraction to bound the number of inputs at each node. Then the program enumerates all the possible realizations of each network node and solves the binate covering problem to get a mapping solution using the fewest lookup tables. An improved MIS-pga program[6] incorporates more decomposition techniques, including bin packing, cofactoring, and AND-OR decomposition. It solves the covering problem more efficiently via certain preprocessing operations.

The Chortle program and its successor Chortle-crf, developed by Francis et al.,[7,8] decompose a Boolean network into a set of fan-out-free trees and then carry out technology mapping on each tree, using dynamic programming. Chortle-crf uses bin-packing heuristics for gate-level decomposition, achieving significant improvement over its predecessor in solution quality and running time.

The Xmap program, developed by Karplus,[9] transforms a Boolean network into an if-then-else DAG (directed acyclic graph) representation and then goes through a simple marking process to determine the final mapping.

Another algorithm, proposed by Woo,[10] introduces the notion of invisible edges to denote edges that do not appear in the resulting network after mapping. The algorithm first partitions a network into subgraphs of reasonable size and then exhaustively determines the invisible edges in each subgraph.

Previous work on FPGA mapping for delay optimization includes Chortle-d, developed by Francis et al.,[11] and an extension of MIS-pga, developed by Murgai et al.[12] Chortle-d's basic approach is similar to that of Chortle-crf. It decomposes the network into fan-out-free trees and then uses dynamic programming and bin-packing heuristics to map each tree independently, at each step minimizing the depth of the node being processed. The method indeed reduces the depths of mapping solutions considerably, but it has two drawbacks. First, it decomposes the network into a set of fan-out-free trees. Although it guarantees the optimal depth for each tree (when the input limit of each lookup table is no more than six), this *prior* decomposition usually results in suboptimal depth for the overall network. Second, Chortle-d uses many more lookup tables than area optimization algorithms (MIS-pga and Chortle-crf) use.

The MIS-pga extension contains two phases: mapping, and placement and routing. The mapping phase computes a delay-optimized two-input network and then traverses the network from the primary inputs, collapsing nodes in the longest paths into their fan-outs to reduce network depth. During this procedure, various decomposition techniques dynamically resynthesize the network, so the program uses a reduced number of lookup tables. The advantage of this approach is that it takes layout information into consideration at the technology-mapping stage. However, on average it yields a greater network depth than Chortle-d, especially for large networks, and requires much more computation time.

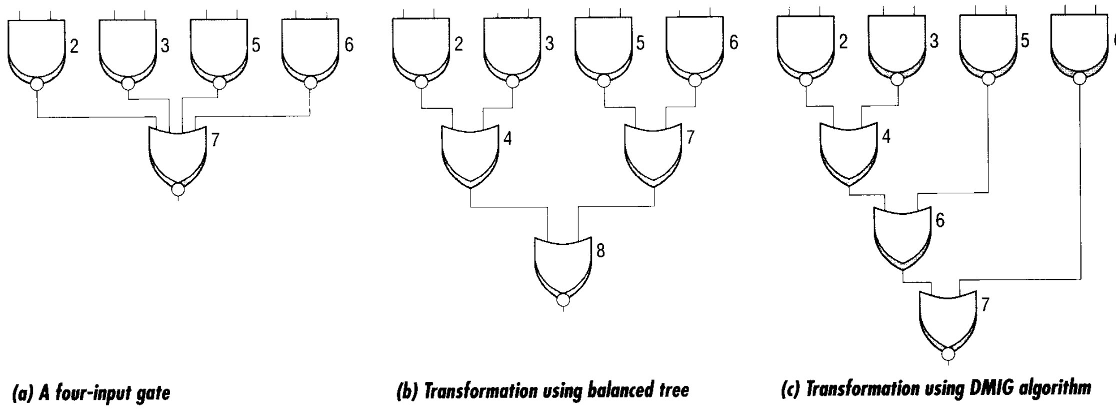
In this article, we present DAG-Map, a graph-based technology-mapping algorithm for delay optimization in lookup-table-based FPGA designs. Our algorithm carries out technology mapping and delay optimization on the entire Boolean network, instead of decomposing it into fan-out-free trees as Chortle-d does. DAG-Map is optimal for trees for any K-LUTs, while Chortle-d is optimal for trees only when K is no more than six.[11] As a preprocessing phase of DAG-Map, we introduce a general algorithm called DMIG, which transforms an arbitrary n-node network into a two-input network with only an $O(1)$ factor increase in network depth. In contrast, previous transformation procedures may result in an $\Omega(\log n)$ factor increase in network depth. Finally, we present a matching-based technique that minimizes area without increasing network delay, which we use in the postprocessing phase of DAG-Map.

We have compared DAG-Map with previous FPGA mapping algorithms on a set of MCNC (Microelectronics Center of North Carolina) logic synthesis benchmarks. Our experimental results show that on average, DAG-Map reduces both network delay and the number of lookup tables, compared with either Chortle-d or the mapping phase of the MIS-pga extension for delay optimization.

Problem formulation

We can represent a Boolean network as a directed acyclic graph in which each node represents a logic gate, and in which a directed edge (i, j) exists 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 that supply inputs to gate v. Given a subgraph H of the Boolean network, $input(H)$ denotes the set of distinct nodes that 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 of v such that any path connecting a node in C_v and v lies entirely in C_v, and $|input(C_v)| \leq K$. (Node u is a predecessor of node v if there is a directed path from u to v.) 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



2
3
5
6
7
2
3
5
6
4
7
8
2
3
5
6
4
6
7
(a) A four-input gate
(b) Transformation using balanced tree
(c) Transformation using DMIG algorithm

Figure 1. Transforming a multi-input network into a two-input network (numbers indicate node levels).

logic block in an FPGA is a K-LUT that can implement any K-input Boolean function (this is true of Xilinx and AT&T FPGA chips[1,10,13]). Thus, each K-LUT can implement any K-feasible cone of a Boolean network. The technology-mapping problem is to cover a given Boolean network with K-feasible cones. (Note that we do not require the covering to be disjoint, since we allow network nodes to be replicated if necessary, as long as the resulting network is logically equivalent to the original one.) A technology-mapping solution S is a DAG in which 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 a small circuit delay and, secondarily, uses a small chip area. Two factors determine the delay of an FPGA circuit: 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; that is, we minimize the number of lookup tables after we have obtained a mapping solution with minimum delay.

The DAG-Map package

The DAG-Map package consists of three major parts. First, a preprocessing procedure transforms an arbitrary Boolean network into a two-input network. Second, the DAG-Map algorithm maps the two-input network into a K-LUT FPGA network with minimum delay. Third, postprocessing performs area optimization of the FPGA network without increasing network delay.

Network transformation. As in the Chortle programs,[8,11] we assume that each node in the Boolean network is a simple gate (AND, OR, NAND, or NOR). (If the network has complex gates, we can represent each complex gate in the sum-of-products form and then replace it with two levels of simple gates. In particular, we use the MIS technology decomposition command *tech_decomp -o 1000 -a 1000*, which realizes such a transformation.[14]) Our first step is to transform the Boolean network of simple gates into a two-input network (a network in which each gate has at most two inputs). We carry out this transformation for two reasons. First, we want to limit each gate's inputs to 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 network gates into K-LUTs, intuitively we know that smaller gates fit more easily, 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 of the same gate type. (Such a transformation maintains logical equivalence as long as the gate function is associative.) Figure 1a shows a four-input gate v. Figure 1b shows the result of replacing it with 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 with the binary tree shown in Figure 1c, the level of v remains 7. Our goal is to replace each multi-input node with 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, shown in Figure 2, transforms G into a two-input network G' as follows. We process the nodes in G in topological order, starting from the PI

```
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.
```

***Figure 2.** The DMIG algorithm.*

nodes. For each multi-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.[15] We write $input(v) = \{u_1, u_2, \ldots, u_m\}$. Note that nodes $u_1, u_2, \ldots, u_m$ have already been processed by the time we process v; their levels $level'(u_i)$ $(1 \le i \le m)$ in the new network G' have been determined. Intuitively, we want to combine nodes with smaller levels first when we construct the binary tree $T(v)$.

If we apply the DMIG algorithm to the example in Figure 1a, we indeed obtain the binary tree shown in Figure 1c. Wang[16] proposed an algorithm similar to DMIG for timing-driven decomposition in the synthesis of multilevel Boolean networks. As detailed in the following theorem, we have shown that the DMIG algorithm increases the network depth after the two-input decomposition by at most a small constant factor. We present the proof in the box. (Hoover et al.[17] carried out a similar analysis in bounding the maximum degree of fan-out in a Boolean network.)

Proof of Theorem 1

First, we show the following lemma:

Lemma. Let $V = \{u_1, u_2, \ldots, u_m\}$ 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 $X_i = \{x_{i,1}, x_{i,2}, \ldots, x_{i,m-i}\}$ be the set of nodes left in V after the DMIG algorithm introduces the ith new node. Clearly, $X_0 = \{u_1, u_2, \ldots, u_m\}$. For convenience, we define $X_{m-1} = \{v\}$. Without loss of generality, assume that $level'(x_{i,1}) \le level'(x_{i,2}) \le \ldots \le level'(x_{i,m-i})$. 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

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

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

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

Note that

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

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 $2^{level'(x_{m-2,2})} = (1/2) \cdot 2^{level'(v)}$. 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,

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

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

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 fan-out y with $level(y) > i$. We will prove by induction that

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

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. From 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 $d - 1$ fan-outs in L_i. According to the lemma, we have

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

By the induction hypothesis, we have

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

We can conclude that the inequality $(*)$ holds for any $0 \le i \le H$. Let w be the 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,

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

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 fan-out in G and I is the number of PI nodes in G.

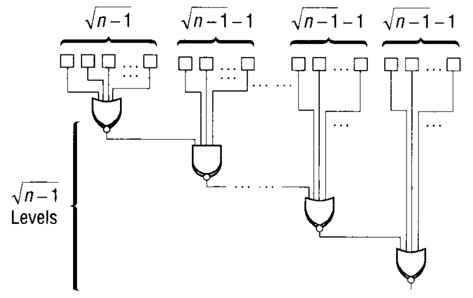
Note that any complex gate in the network can be decomposed into a two-level AND-OR subnetwork, so that the increase of network depth is bounded by a factor of two.

In practice, d is bounded by a constant (the fan-out limit of any output). Therefore, the depth of the two-input network G' is increased by only a constant factor $\log 2d$ away from *depth*(G), in contrast to the $\Omega(\log n)$ increase that may occur with the balanced binary tree transformation. (Here we assume that $depth(G) = \Omega(\log I)$, which is true for most networks in practice. This excludes the unrealistic case in which $\log I$ is the dominating term in the right-hand side of the inequality in Theorem 1.)

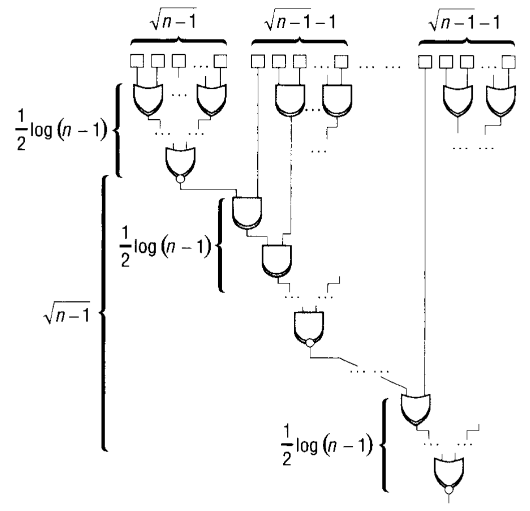
Figure 3 on the next page shows a pathological example of the balanced binary tree transformation. Figure 3a shows the initial network of size n, a fan-out-free circuit of depth $\sqrt{n-1}$ (assuming that the primary inputs are at level 0). Figure 3b shows the two-input network after the balanced binary tree transformation. The network in Figure 3b has depth $d_{BBT} = (1/2)\log(n-1) \cdot \sqrt{n-1}$, even though $d = 1$ in this case. Figure 3c shows the DMIG transformation result, which has depth $d_{DMIG} = \sqrt{n-1} + (1/2)\log(n-1) - 1$. Clearly d_{DMIG} is much smaller than d_{BBT} when n is large.

The experimental results presented later in this article show that the two-input networks obtained with our transformation procedure lead to smaller network depths and better mapping solutions than those obtained with the MIS transformation procedure.[14]

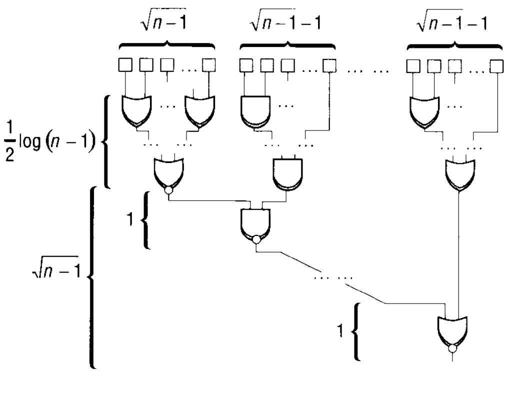
Technology mapping. After we ob-



(a) Original multi-input network



(b) Transformation using balanced binary tree: $d_{BBT}=\frac{1}{2}\log(n-1)\cdot\sqrt{n-1}$



(c) Transformation using DMIG: $d_{DMIG}=\sqrt{n-1}+\frac{1}{2}\log(n-1)-1$

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

tain 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.: module clustering to minimize delay in digital networks.[18] The DAG-Map mapping algorithm consists of two phases: labeling the network to determine each node's level in the final mapping solution and generating the logically equivalent network of K-LUTs. The algorithm is summarized in Figure 4.

The first phase 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 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 $input(N_p(v) \cup \{v\}) \leq K$, 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. (Note that $v \in N_{h(v)}(v)$ because v is a predecessor of itself.)

The second phase generates K-LUTs in the mapping solution. Let L represent the set of outputs to be implemented with 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 phase 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 DAG-Map mapping algorithm has several advantages:

- It works on the entire network without decomposing it into fan-out-free trees, usually leading to better mapping solutions. For example, decomposing the two-input network shown in Figure 5a into fan-out-free trees, as shown in Figure 5b, yields a two-level mapping solution with three lookup tables. However, the DAG-Map algorithm gives a one-level mapping solution with two lookup tables, as shown in Figure 5c.

- DAG-Map will replicate nodes, if necessary, to minimize network delay in the mapping solution. For the solution shown in Figure 5c, 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.
- The algorithm is optimal when the initial network is a tree, as stated in the following theorem.

Theorem 2. For any integer K, if the Boolean network is a tree with fan-in 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 fan-in limit of each node is K, the algorithm can successfully label all the nodes. 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 $h(u) = p$. $Level_M(LUT_v) \geq level_M(LUT_u) \geq h(u) = p$, and $h(v) \leq p + 1$ according to the DAG-Map labeling procedure. Therefore, 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, where C_v is the K-feasible cone at v 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 statement that C_v is K-feasible.

Francis et al. showed a result along similar lines for Chortle-d,[11] but it holds only for $K \leq 6$, since the Chortle-d bin-packing heuristics are no longer optimal for $K > 6$. (Chortle-d does not limit the fan-in of each node in the tree to no more than K, because it carries out node decomposition during the bin-packing process.)

Although the DAG-Map algorithm is optimal for trees, it may not be optimal for

```
algorithm DAG-Map
  /* phase 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 $|input(N_p(v) \cup \{v\})| \leq K$
    then $h(v) = p$
    else $h(v) = p + 1$
  end-while;
  /* phase 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.
```

Figure 4. *DAG-Map mapping algorithm.*

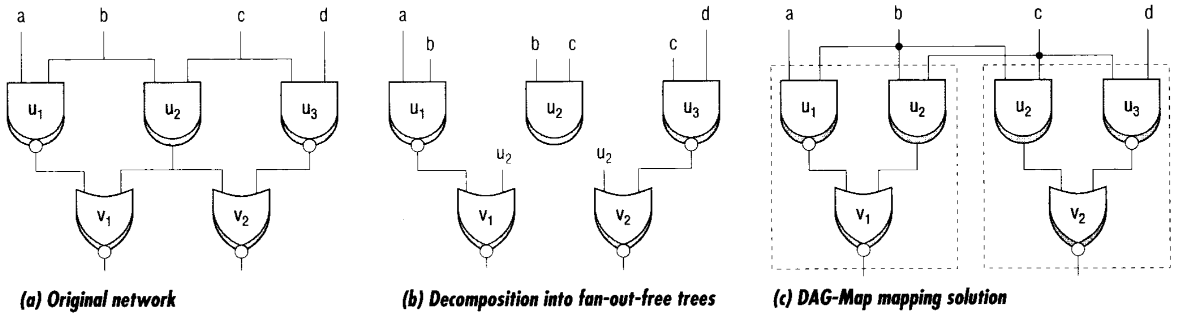
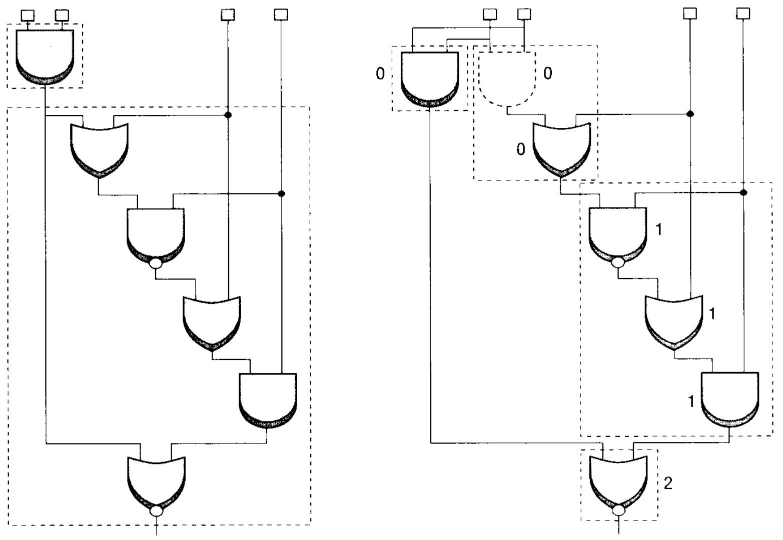


Figure 5. *A mapping example (assume $K = 3$).*



(a) Optimal mapping (two levels) **(b) DAG-Map mapping (three levels)**

Figure 6. A pathological example of DAG-Map mapping (assume K = 3. Numbers represent node levels in mapping solution).

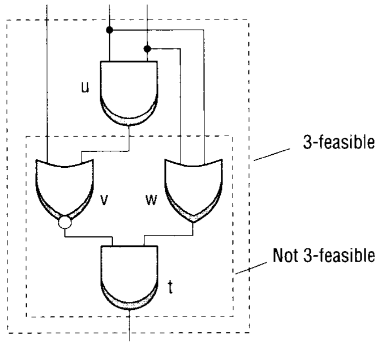


Figure 7. The constraint on the number of lookup-table inputs is not monotonic (assume K = 3).

general networks. Figure 6 shows an example in which DAG-Map produces a suboptimal mapping solution. However, DAG-Map would be optimal if the mapping constraint for each programmable logic block were monotonic. As defined by Lawler et al.,[18] a constraint X is monotonic if a network H satisfying X implies that any subgraph of H also satisfies X. For example, limiting the number of gates a programmable logic block can cover is a monotonic constraint.

Unfortunately, limiting the number of distinct inputs of each programmable logic block is not a monotonic constraint. In Figure 7 the whole network has three distinct inputs, but the subnetwork consisting of t, v, and w has four distinct inputs. However, our experimental results show that DAG-Map produces satisfactory mapping solutions with respect to delay optimization for all benchmark circuits.

Area optimization. Since the main objective of the DAG-Map mapping algorithm is to optimize the depth of the mapping solution, minimizing the number of K-LUTs is not a consideration. Therefore, we have developed two area optimization operations, which are used in postprocessing after we obtain a small-depth mapping solution. These operations reduce the number of K-LUTs in the mapping solution without increasing network depth. Note that in our discussion 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, gate decomposition, was inspired by the gate decomposition concept in Chortle-crf. The basic idea is as follows: Let us assume that 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 fan-out 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. Further, $input(v_{ij}) = \{u_i, u_j\}$ and $input(v') = input(v) \cup \{v_{ij}\} - \{u_i, u_j\}$. (In other words, we feed u_i and u_j into v_{ij} first, and then we use v_{ij} to replace 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 $|input(u_i) \cup input(u_j)| \le K$, 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 less input (a condition beneficial to subsequent gate decomposition and predecessor packing). Figure 8 illustrates the gate decomposition operation, which reduces the number of nodes, as well as the number of fan-ins of node v (v' after the operation), by one.

We can generalize this method to the case where the decomposed node v implements a complex function. We apply Roth-Karp decomposition[19] to determine if the node can be feasibly decomposed to v_{ij} and v' as just described. 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 functions if they exist. (This is a special case of Roth-Karp decomposition.) In this 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

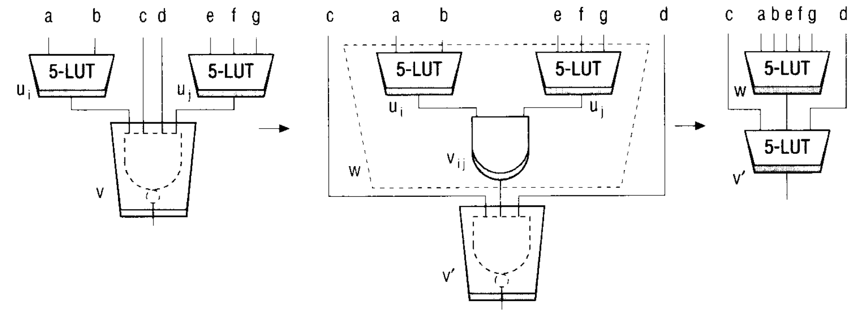


Figure 8. Gate decomposition for area optimization (assume K = 5).

u_j of node v, and $|input(u_i) \cup input(u_j)| \leq K$, 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 generally runs in exponential time, it takes only constant time in our algorithm, because the number of fan-ins of a K-LUT is bounded by a small constant K.

Another postprocessing operation for area optimization is predecessor packing. The concept behind this method is simple. For each node v, we examine all 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 fan-out, v and u_i are merged into a single K-LUT. In this case, we also say that nodes 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, which reduces the number of inputs to the current node v by one, this method actually increases the number of inputs 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. Figure 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 with one less K-LUT.

There are usually many pairs of mergeable nodes in a network, but not all the merge operations can be performed at the same time. Thus, to avoid merging nodes in arbitrary order, we use a graph-matching approach, which 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 reconstruct the graph G for the reduced network and repeat the procedure until we are unable to construct a nonempty E.

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,[20] our area optimization procedure can be implemented efficiently. (We used a standard procedure for maximum cardinality matching in undirected graphs, written by Ed Rothberg, which implements Gabow's algorithm.)

Note that in our discussion of these two operations, we assume that each node in a mergeable pair has only a single fan-out, unless it is also the base of the merge for predecessor packing. That 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 fan-outs, the application of the merge operation requires u to be replicated so that the copy involved in the merge operation is fan-out free. However, unless every fan-out 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 that is not merged to any of its fan-out nodes.

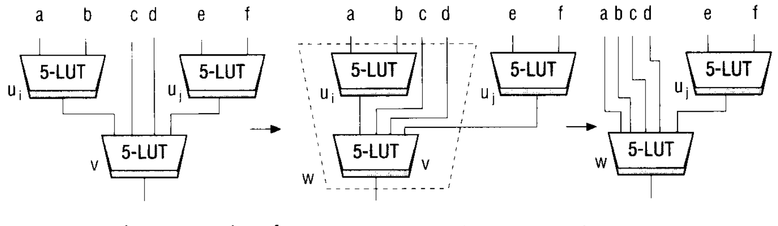


Figure 9. Predecessor packing for area optimization (assume K = 5).

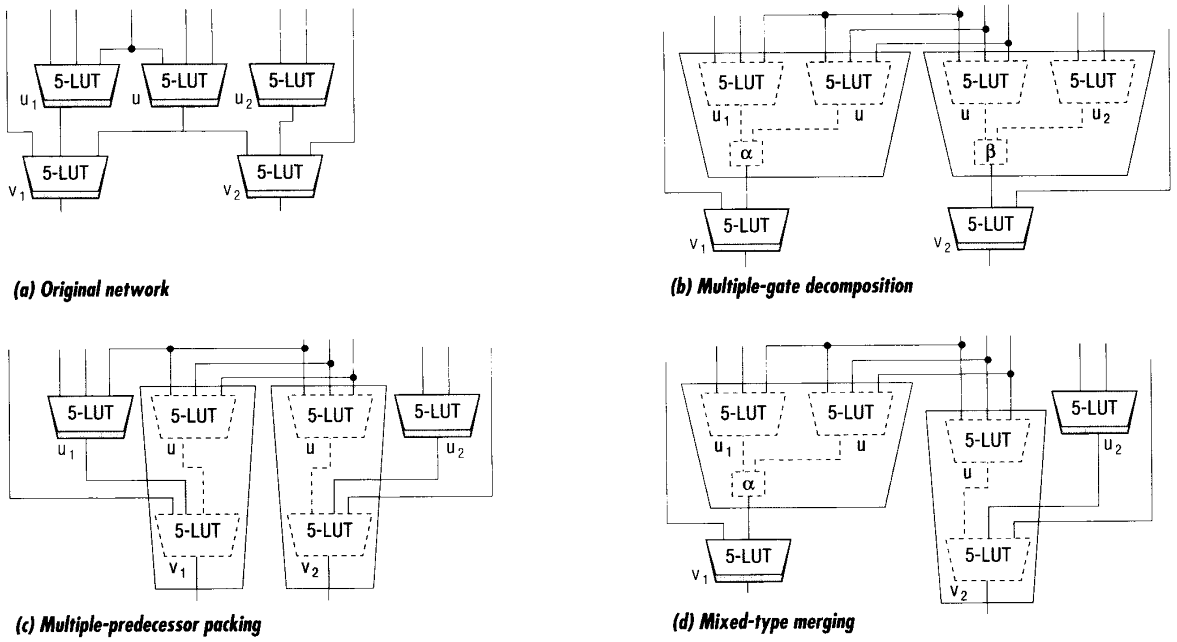

5-LUT
u1
u
u2
v1
v2
(a) Original network
(b) Multiple-gate decomposition
α
β
(c) Multiple-predecessor packing
(d) Mixed-type merging

Figure 10. *Merge operations on multi-fan-out node u (assume K = 5).*

We say a node u is removable if and only if for each of its fan-out nodes v_i, either u and v_i are mergeable via predecessor packing, or there is another fan-in node of v_i, say u_i, such that u and u_i are mergeable via gate decomposition. Figure 10 shows three different cases where node u is a removable node. For a removable node u, each of its fan-out 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 node u_i for each fan-out node v_i of u, which is selected in one of two ways: 1) If v_i is the base of a predecessor packing operation involving u, we can select v_i as u_i; or 2) if v_i is the base of a gate decomposition operation involving u, 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 Figure 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 fan-out free, a mergeable set of u is a mergeable pair as defined previously. Therefore, a mergeable pair is a special case of a mergeable set.

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 network nodes, and the hyperedges in E represent the mergeable sets. Note that H contains the simple graph $G = (V, E_2)$, representing mergeable pairs for fan-out-free nodes, as a subgraph. We call the edges in E_2 the simple edges, and we call the edges in $E - E_2$, each of which contains more than two vertices, the nonsimple edges. A matching in H is a set of disjoint edges in E. Clearly, a maximum matching in H yields the maximum number of network K-LUTs 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. 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. This matching will be included in the approximate solution for hypergraph matching. Let E_m be the set of nonsimple 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 let $M = M_2 \cup M_m$ be the matching computed with the preceding algorithm. Then $|M^*| \leq 2|M|$.

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 nonsimple edges in H that 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 nonsimple edges, we can obtain a better bound.

Corollary 1. If the number of nonsimple 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 $h(H) - |M_m|$ nonsimple 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 the fact that M_2 is a maximum matching of the simple edges in H). This implies that S contains at least $|S| - |M^+|$ nonsimple 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 we used in our experiments, we applied Corollary 1 and found that the error bound $h(H) - |M_m|$ is never greater than four.

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 I/O routines and other functions provided by MIS. We tested DAG-Map on a large number of MCNC benchmark examples and compared our results with those produced by Chortle-d[11] and by the mapping phase of the MIS-pga delay optimization algorithm.[12]

As in the Chortle-d and MIS-pga tests, we chose the size of the K-LUT to be $K = 5$, reflecting, for example, the Xilinx XC 3000 FPGA family.[1] 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 it into a 5-LUT network. Finally, we performed the matching-based postprocessing step. Table 1 on the next page compares the results of our algorithm with those of the other two algorithms.

We obtained the input networks to Chortle-d and DAG-Map from the original benchmarks, using the same standard MIS technology-independent optimization script used by Francis et al.,[11] except that Chortle-d goes through another speedup step for delay optimization. A direct comparison with MIS-pga is difficult because it combines logic optimization and technology mapping. Nevertheless, we include the mapping results produced by the MIS-pga delay optimization algorithm (quoted from Murgai et al.[12]) for reference. The running time of our algorithm, which includes transformation, mapping, and postprocessing, was recorded on a Sun Sparc IPC (15.8 MIPS). The running times of the other two algorithms are quoted from Murgai et al.[12]; the authors used a DEC5500 machine (28 MIPS). Overall, the Chortle-d solutions used 60% more lookup tables and had 2% larger network depth than the DAG-Map solutions; the MIS-pga delay optimization solutions 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.

To judge the effectiveness of our DMIG algorithm for transforming the initial network into a two-input network, we compared it with the MIS transformation procedure. We applied both the DMIG algorithm and the MIS decomposition command *tech_decomp -a 2 -o 2* to the same initial networks. (Again, we optimized the initial networks with the MIS minimization script as in the preceding experiment; in addition, we used *tech_decomp -a 1000 -o 1000* to transform them into simple gate networks.) 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; the last four columns compare the number of 5-LUTs and the depth of the 5-LUT networks after we applied DAG-Map to the two-input networks. In all cases, the DMIG procedure resulted in smaller or the same depths in 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

Table 1. Comparison of three algorithms for 5-LUT FPGA technology mapping.

Benchmark	Chortle-d			MIS-pga (d)			DAG-Map		
example	LUTs	Depth	Time (s)	LUTs	Depth	Time (s)	LUTs	Depth	Time (s)
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	2,086	6	9.2	1,397	11	937.8	1,423	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	4,906	87	16.3	3,182	90	1,277.4	3,062	85	175.6
Comparison	+60%	+2%	-	+4%	+6%	-	1	1	-

Table 2. Comparison of two-input network transformation algorithms.

	Before mapping				After 5-LUT mapping			
Benchmark	MIS tech-decomp		DMIG		MIS tech-decomp		DMIG	
example	Gates	Depth	Gates	Depth	LUTs	Depth	LUTs	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	3,049	19	3,049	16	1,461	6	1,423	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	7,440	337	7,440	292	3,119	91	3,062	85
Comparison	+0%	+15%	1	1	+2%	+7%	1	1

networks is always the same.)

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

Our results with DAG-Map show that the graph-based technology-mapping approach is more effective than the existing tree-based technology-mapping approaches in lookup-table-based FPGA designs. In subsequent work,[21] we have shown that the lookup-table-based FPGA technology-mapping problem for delay optimization can be solved optimally in polynomial time.

Table 3. Effectiveness of postprocessing for area optimization of depth-minimized 5-LUT mappings.

Benchmark example	Original LUTs	Original Depth	After postprocessing LUTs	After postprocessing 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	1,674	5	1,423	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	3,566	85	3,062	85
Comparison	+16%	+0%	1	1

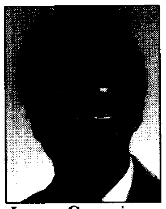
Acknowledgments

We thank Masakatsu Sugimoto for his support of this research. We also gratefully acknowledge support from National Science Foundation grants MIP-9110511 and MIP-9110696 and Army Research Office grant DAAK-70-92-K-0001. We thank Jonathan Rose and Robert Francis for providing the Chortle programs and necessary assistance for our comparative study.

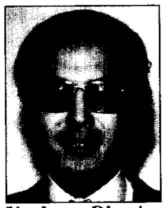
References

1. *Xilinx User Guide and Tutorials*, Xilinx, San Jose, Calif., 1991.
2. A. El Gamal et al., "An Architecture for Electrically Configurable Gate Arrays," *IEEE J. Solid-State Circuits*, Vol. 24, Apr. 1989, pp. 394-398.
3. E. Detjens et al., "Technology Mapping in MIS," *Proc. IEEE Int'l Conf. Computer-Aided Design*, IEEE Computer Society Press, Los Alamitos, Calif., 1987, pp. 116-119.
4. K. Keutzer, "DAGON: Technology Binding and Local Optimization by DAG Matching," *Proc. 24th Design Automation Conf.*, IEEE CS Press, 1987, pp. 341-347.
5. R. Murgai et al., "Logic Synthesis Algorithms for Programmable Gate Arrays," *Proc. 27th Design Automation Conf.*, IEEE CS Press, 1990, pp. 620-625.
6. R. Murgai et al., "Improved Logic Synthesis Algorithms for Table Look-Up Architectures," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1991, pp. 564-567.
7. R.J. Francis, J. Rose, and K. Chung, "Chortle: A Technology-Mapping Program for Lookup Table-Based Field Programmable Gate Arrays," *Proc. 27th Design Automation Conf.*, IEEE CS Press, 1990, pp. 613-619.
8. R.J. Francis, J. Rose, and Z. Vranesic, "Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs," *Proc. 28th Design Automation Conf.*, IEEE CS Press, 1991, pp. 613-619.
9. K. Karplus, "Xmap: A Technology Mapper for Table-Lookup Field-Programmable Gate Arrays," *Proc. 28th Design Automation Conf.*, 1991, pp. 240-243.
10. N.-S. Woo, "A Heuristic Method for FPGA Technology Mapping Based on the Edge Visibility," *Proc. 28th Design Automation Conf.*, IEEE CS Press, 1991, pp. 248-251.
11. R.J. Francis, J. Rose, and Z. Vranesic, "Technology Mapping of Lookup Table-Based FPGAs for Performance," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1991, pp. 568-571.
12. R. Murgai et al., "Performance-Directed Synthesis for Table Look-Up Programmable Gate Arrays," *Proc. Int'l Conf. Computer-Aided Design*, IEEE CS Press, 1991, pp. 572-575.
13. D. Hill, "A CAD System for the Design of Field-Programmable Gate Arrays," *Proc. 28th Design Automation Conf.*, IEEE CS Press, 1991, pp. 187-192.
14. R.K. Brayton, R. Rudell, and A.L. Sangiovanni-Vincentelli, "MIS: A Multiple-Level Logic Optimization," *IEEE Trans. CAD*, Nov. 1987, pp. 1062-1081.
15. D.A. Huffman, "A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE 40*, 1952, pp. 1098-1101.
16. A. Wang, "Algorithms for Multi-level Logic Optimization," Memo. UCB/ERL M89/50, Univ. of California, Berkeley, 1989.
17. H.J. Hoover, M.M. Klawe, and N.J. Pippenger, "Bounding Fan-out in Logic Networks," *J. ACM*, Vol. 31, Jan. 1984, pp. 13-18.
18. E.L. Lawler, K.N. Levitt, and J. Turner, "Module Clustering to Minimize Delay in Digital Networks," *IEEE Trans. Computers*, Vol. C-18, No. 1, Jan. 1969, pp. 47-57.
19. J.P. Roth and R.M. Karp, "Minimization Over Boolean Graphs," *IBM J. Research and Development*, Apr. 1962, pp. 227-238.
20. H. Gabow, "An Efficient Implementation of Edmonds' Algorithm for Maximum Matching on Graphs," *J. ACM*, Vol. 23, Apr. 1976, pp. 221-234.
21. J. Cong and Y. Ding, "An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs," *Proc. IEEE Int'l Conf. Computer-Aided Design*, IEEE CS Press, to appear Nov. 1992.

Kuang-Chien Chen works in research and development of advanced logic synthesis systems for Fujitsu America. His research interests include logic synthesis and optimization, high-level synthesis, and formal verification. He received the BS degree in electrical engineering from National Taiwan University and the MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign. Chen is a member of the ACM and the IEEE.

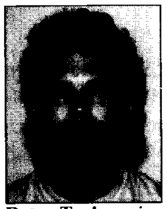


Jason Cong is an assistant professor in the Computer Science Department of the University of California, Los Angeles. Cong's research interests include computer-aided design of VLSI circuits and design and analysis of efficient combinatorial and geometric algorithms. He received his BS in computer science from Peking University and his MS and PhD in computer science from the University of Illinois at Urbana-Champaign. He is a member of the IEEE.



Yuzheng Ding is pursuing his PhD degree in computer science at UCLA. His research interests include the design and analysis of data structures and algorithms, and computer-aided design of VLSI circuits. He holds the BS in computer science from Peking University and the MS in computer science from Tsinghua University, both in Beijing, China.

Andrew B. Kahng is an assistant professor in the Computer Science Department at UCLA, where he received an NSF Young Investigator award. His research interests include computer-aided design of VLSI circuits, combinatorial and parallel algorithms, global optimization theory, and computational geometry. Kahng holds the AB degree in applied mathematics and physics from Harvard College and the MS and PhD degrees in computer science from the University of California, San Diego. He is a member of ACM, SIAM, and IEEE.



Peter Trajmar is with Zycad Corporation in Fremont, California, working on designing and implementing software systems for advanced simulation accelerator hardware. In the summer of 1991, he worked at Fujitsu America on FPGA technology-mapping algorithms. Trajmar received his BS in electrical engineering and computer science from the University of California, Berkeley, and his MS in computer science from UCLA. His academic work concentrated on computer architecture, VLSI, and CAD algorithms. He is a member of the IEEE.

Questions and comments should be addressed to Jason Cong, UCLA, Computer Science Dept., Los Angeles, CA 90024; e-mail: cong@cs.ucla.edu.