Improvements to Technology Mapping for LUT-Based FPGAs

Alan Mishchenko **Satrajit Chatterjee** **Robert Brayton**

Department of EECS, University of California, Berkeley

{alanmi, satrajit, brayton}@eecs.berkeley.edu

Abstract

The paper presents several improvements to state-of-the-art in FPGA technology mapping exemplified by a recent advanced technology mapper DAOmap [Chen and Cong, ICCAD `04]. Improved cut enumeration computes all K-feasible cuts without pruning for up to 7 inputs for the largest MCNC benchmarks. A new technique for on-the-fly cut dropping reduces by orders of magnitude memory needed to represent cuts for large designs. Improved area recovery leads to mappings with area on average 7% smaller than DAOmap, while preserving delay optimality when starting from the same optimized netlists. Applying mapping with structural choices derived by a synthesis flow on average reduces delay by 7% and area by 14%, compared to DAOmap.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Optimization*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate arrays*; J.6 [**Computer-Aided Engineering**]: *Computer-aided design (CAD)*

General Terms

Algorithms

Keywords

FPGA, Technology Mapping, Cut Enumeration, Area Recovery, Lossless Synthesis

1 Introduction

Field Programmable Gate Arrays (FPGAs) are an attractive hardware design option, making technology mapping for FPGAs an important EDA problem. For an excellent overview of the classical and recent work on FPGA technology mapping, focusing on area, delay, and power minimization, the reader is referred to [2].

The recent advanced algorithms for FPGA mapping, such as [2][12][16][23], focus on area minimization under delay constraints. If delay constraints are not given, first the optimum delay for the given logic structure is found and then area is minimized without changing delay.

In terms of the algorithms employed, the mappers are divided into structural and functional. Structural mappers consider the circuit graph as a given and find a covering of the graph with *K*-input subgraphs corresponding to LUTs. The functional approaches perform Boolean decomposition of the logic functions of the nodes into sub-functions of limited support size realizable by individual LUTs.

Since functional mappers explore a larger solution space, they tend to be time-consuming, which limits their use to small designs. In practice, FPGA mapping for large designs is done using structural mappers, whereas the functional mappers are used for resynthesis after technology mapping.

In this paper, we consider the recent work on DAOmap [2] as representative of the advanced structural technology mapping for LUT-based FPGAs and refer to it as "the previous work" and discuss several ways of improving it. Specifically, our contributions fall into three categories:

(1) *Improved cut computation*

Computation of all *K*-feasible cuts is typically a run-time and memory bottleneck of a structural mapper. We propose several enhancements to the standard cut enumeration procedure [7][22]. Specifically, we introduce cut filtering with signatures and show that it leads to a speed-up. This makes exhaustive cut enumeration for 6 and 7 inputs practical for many test-cases.

Since the number of *K*-feasible cuts per node, for large *K*, can exceed 100, storing all the computed cuts in memory is problematic for large benchmarks. We address this difficulty by allowing cut enumeration to "drop" the cuts at the nodes whose fanouts have already been processed. This allows the mapper to store only a small fraction of all *K*-feasible cuts at any time, thereby reducing memory usage for large benchmarks by an order of magnitude or more.

(2) *Better, simpler, and faster area recovery*

Area optimization after delay-optimum structural mapping proceeds in several passes over the network. Each pass assigns cuts with a better area among the ones that do not violate the required time. The previous work relied on several sophisticated heuristics for ranking the cuts, trying to estimate their potential to save area. The previous work concluded that not all the heuristics are equally useful but, to get good area, a number of them need to be applied.

In this paper, we show that the combination of two simple techniques is enough to ensure reasonable mapping quality and improve on the results of the previous work by

FPGA '06, February 22–24, 2006, Monterey, California, USA.

7% on average. The proposed combination of techniques works well since the first one attempts heuristically to find a global optimum, whereas, the second ensures that at least a local optimum is reached.

It should be noted that the first heuristic (known as *effective area* [7] or *area flow* [16]) is used in the previous work but it is applied in a reverse topological order, while we argue below that a direct topological order works better.

(3) *Lossless synthesis*

The main drawback of the structural approaches to technology mapping is their dependence on the initial circuit structure. If the structure is bad, neither heuristics nor iterative recovery will improve the results of mapping.

To obtain a good structure for the network several technology independent synthesis steps are usually performed. An example of this is *script.rugged* in SIS followed by a two-input gate decomposition. Each synthesis step in the script is heuristic, and the subject graph produced at the end is not necessarily optimum. Indeed, it is possible that the initial or an intermediate network is better in some respects than the final network.

In this paper, we explore the idea of combining these intermediate networks into a single subject graph with choices and using that to derive the mapped netlist. The mapper is not constrained to use any one network, but can pick and choose the best parts of each. We call this approach *lossless synthesis,* since no network seen during the synthesis process is ever lost. By including the initial network in the choice network, we can be sure that the heuristic logic synthesis operations never make things worse. We can also use multiple scripts and repeatedly go through each accumulating more choices. We defer discussion of related work to Section 5.3.

The rest of the paper is organized as follows. Section 2 describes the background. Sections 3-5 give details on the three contributions of the paper listed above. Section 6 shows experimental results. Section 7 concludes the paper and outlines future work.

2 Background

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms network, Boolean network, and circuit are used interchangeably in this paper.

A node has zero or more *fanins,* i.e. nodes that are driving this node, and zero or more *fanouts*, i.e. nodes driven by this node. The *primary inputs* (PIs) of the network are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. If the network is sequential, the flip-flop outputs/inputs are treated as additional PIs/POs. In the following, it is assumed that each node has a unique integer number called the *node ID.*

A network is *K-bounded* if the number of fanins of each node does not exceed *K*. An *subject graph* is a *K*-bounded network used for technology mapping. Any combinational network can be represented as an AND-INV graph (AIG), composed of two-input ANDs and inverters. Without limiting the generality, in this paper we assume subject graphs to be AIGs.

A *cut C* of node *n* is a set of nodes of the network, called *leaves*, such that each path from a PI to *n* passes through at least one leaf. A *trivial cut* of the node is the cut composed of the node itself. A cut is *K-feasible* if the number of nodes in it does not exceed *K*. A cut is said to be *dominated* it there is another cut of the same node, which is contained, set-theoretically, in the given cut.

A *fanin* (*fanout*) *cone* of node *n* is a subset of all nodes of the network reachable through the fanin (fanout) edges from the given node. A *maximum fanout free cone* (MFFC) of node *n* is a subset of the fanin cone, such that every path from a node in the subset to the POs passes through *n*. Informally, the MFFC of a node contains all the logic used only by the node. When a node is removed or substituted, the logic in its MFFC can also be removed.

The *level* of a node is the length of the longest path from any PI to the node. The node itself is counted towards the path lengths but the PIs are not. The network *depth* is the largest level of an internal node in the network. The delay and area of FPGA mapping is measured by the depth of the resulting LUT network and the number of LUTs in it.

A typical procedure for structural technology mapping performs the following steps:

1. Cut computation.
2. Delay-optimum mapping.
3. Area recovery using heuristics.
4. Writing out the resulting LUT network.

For a detailed description on these steps, we refer the reader to [2] and [16].

3 Improved cut computation

Structural technology mapping into FPGAs containing *K*-input LUTs starts by computing *K*-feasible cuts for each internal two-input node of the subject graph.

Of the two procedures for cut computation, the network flow [5] and the cut enumeration [7][22], the latter is faster. The advantage of the former is that it can be applied incrementally to compute cuts for individual nodes. However, at the beginning of mapping, computing cuts for all nodes is desirable.

3.1 Cut enumeration

The result of cut enumeration is a set of all *K*-feasible cuts assigned for each node. Cut enumeration starts at the PIs and proceeds in the topological order to the POs. Processing nodes in the topological order guarantees that cut computation is called for an internal node after it has completed for its fanins. For a PI, the set of cuts contains only the trivial cut. For an internal node *n* with two fanins,

a and *b*, the set of cuts $\Phi(n)$ is computed by *merging* the sets of cuts of *a* and *b* as follows:

$$\Phi(n) = \{\{n\}\} \cup \{u \cup v \mid u \in \Phi(a), u \in \Phi(b), |u \cup v| \leq k\}$$

Informally, merging two sets of cuts adds the trivial cut of the node to the set of pair-wise unions of cuts belonging to the fanins, while keeping only *K*-feasible cuts.

The resulting set of cuts, $\Phi(n)$, may contain duplicated and dominated cuts. Removing them before computing cuts for the next node in the order reduces the number of cut pairs considered, without impacting the quality of mapping. In practice, the total number of cut pairs tried greatly exceeds the number of *K*-feasible cuts found. This makes checking *K*-feasibility of the unions of cut pairs, and testing duplication and dominance of individual cuts, the performance bottle-neck of the cut computation.

3.2 Using signatures

In this paper, we propose to use signatures for testing cut properties, such as duplication, dominance, and *K*-feasibility. Conceptually, it is similar to the use of Bloom filters for encoding sets [1] and to the use of signatures for comparing clauses in [9]. Note that the use of signatures only speeds up the computation; no pruning is done.

A signature, $sign(C)$, of cut C is an M-bit integer whose bit-wise representation contains 1s in the positions corresponding to the node IDs. The signature is computed by the bitwise addition of integers as follows:

$$sign(C) = \sum_{n \in C} 2^{\mathrm{ID}(n) \bmod M} .$$

Testing cut properties with signatures is much faster than testing them by directly comparing leaves. The following propositions state the necessary conditions for duplication, dominance, and *K*-feasibility of cuts. The contrapositives of the propositions are the sufficient conditions for the cuts to be non-duplicated, non-dominated, and not *K*-feasible.

Proposition 1: If cuts C_1 and C_2 are equal, so are their signatures. (Thus, if the signatures of C_1 and C_2 are not equal, neither are the cuts.)

Proposition 2: If cut C_1 dominates cut C_2, the 1s of $sign(C_1)$ are contained in the 1s of $sign(C_2)$. (Thus, if 1s of $sign(C_1)$ are not contained in the 1s of $sign(C_2)$, then cut C_1 does not dominate cut C_2.)

Proposition 3: If $C_1 \cup C_2$ is a *K*-feasible cut, $|sign(C_1) + sign(C_2)| \leq K$. (Thus, if $|sign(C_1) + sign(C_2)| > K$, *then* $C_1 \cup C_2$ is not a *K*-feasible cut.) Here $|n|$ denotes the number of ones in the binary representation of *n,* and addition is done modulo *M*.

Our current implementation uses one machine word (composed of 32 bits on a 32-bit machine) to represent the signature of a cut i.e. $M = 32$. As a result, most of the checks are performed using several bit-wise machine operations, and only if the signatures fail to disprove a property, the actual comparison of leaves is performed.

3.3 Practical observations

In the literature on technology mapping, the 4-input and 5-input cuts are typically computed exhaustively, whereas computation of cuts with more inputs is considered time-consuming because of the large number of these cuts. Different heuristics have been investigated in the literature [7] to rank and prune cuts to reduce the run-time. We experimented with these heuristics and found that they work for area but lead to sub-optimal delay.

In order to preserve delay optimality, we focus on perfecting the cut computation and computing all cuts whenever possible. Pruning is done only if the number of cuts at a node exceeds a predefined limit set to 1000 in our experiments. When computing *K*-feasible cuts with $4 \leq K \leq 7$ for the largest MCNC benchmarks, the limit was never reached, and no pruning was performed, meaning that the cuts were computed exhaustively. Due to the use of signatures, the run-time for $4 \leq K \leq 7$ was also quite affordable, as evidenced by the experiments. However, for 8-input cuts, pruning was required for some benchmarks.

3.4 Reducing memory for cut representation

The number of *K*-feasible cuts for $K > 5$ can be large. The average number of exhaustively computed 7-input cuts in the largest MCNC benchmarks is around 95 cuts per node. In large industrial designs, the total number of cuts could be of the order of tens of millions. Therefore, once the speed of cut enumeration is improved, memory usage for the cut representation becomes the next pressing issue.

To address this issue, we modified the cut enumeration algorithm to free the cuts as soon as they are not needed for the subsequent enumeration steps. This idea is based on the observation that the cuts of the nodes, whose fanouts have already been processed, can be deallocated without impacting cut enumeration. It should be noted that if technology mapping is performed in several topological passes over the subject graph, the cuts are re-computed in each pass. However, given the speed of the improved cut computation, this does not seem to be a problem.

Experimental results (presented in Table 2) show that by enabling cut dropping, as explained above, the memory usage for the cut representation is reduced by an order of magnitude for MCNC benchmarks. We see that for larger benchmarks, the reduction in memory is even more substantial.

It is possible to reduce the run-time of the repeated cut computation by recording the "cut enumeration trace", which is saved during the first pass of cut enumeration and used in the subsequent passes. The idea is based on the observation that, even when signatures are used, the most time-consuming part of the cut enumeration is determining what cut pairs lead to non-duplicated, non-dominated, *K*-feasible cuts at each node. The number of such cut pairs is very small, compared to the total number of cut pairs at each node. The cut enumeration trace recorded in the first pass compactly stores information about all such pairs and

the order of merging them to produce all the *K*-feasible cuts at each node. The trace serves as an oracle for the subsequent cut enumeration passes, which can now skip checking all cut pairs and immediately derive useful cuts.

This option was implemented and tested in our cut enumeration package but it was not used in the experimental results because the benchmarks allowed for storing all the cuts in memory at the same time. We mention this option here because we expect it to be useful for industrial mappers working on very large designs.

4 Improved area recovery

Exact area minimization during technology mapping for DAGs is NP-hard [10] and hence not tractable for large circuits. Various heuristics for approximate area minimization during mapping have shown good results [2][12][16][23].

In this study, we use a combination of two heuristics, which work well in practice. The order of applying the heuristics is important since they are complementary. The first heuristic has a global view and selects logic cones with more shared logic. The second heuristic provides a missing local view by minimizing the area exactly at each node.

4.1 Global view heuristic

Area flow [16] (*effective area* [7]) is a useful extension of the notion of area. It can be computed in one pass over the network from the PIs to the POs. Area flow for the PIs is set to 0. Area flow at a node *n* is:

$$AF(n) = [Area(n) + \Sigma_i AF(Leaf_i(n))] / NumFanouts(n),$$

where Area(*n*) is the area of the LUT used to map the current best cut of node *n*, $Leaf_i(n)$ is the *i*-th leaf of the best cut at *n*, and NumFanouts(*n*) is the number of fanouts of node *n* in the currently selected mapping. If a node is not used in the current mapping, for the purposes of area flow computation, its fanout count is assumed to be 1.

If nodes are processed from the PIs to the POs, computing area flow is fast. The advantage of area flow over exact area is that area flow gives a global view of how useful is logic in the cone for the current mapping. Area flow estimates sharing between cones without the need to re-traverse them, which would be required if the exact area were computed.

In our mapper, as in the previous work [2] and in [16], area flow is the tie-breaker used in the first pass when a delay-optimum mapping is computed. In the first stage of area recovery, area flow is the primary cost function used to choose among the cuts, whose arrival times do not exceed the required times.

4.2 Local view heuristic

The second heuristic providing a local view for area recovery in our mapper is not used in the previous work. This heuristic looks at the exact area to be gained by locally updating the best cut at each node when nodes are processed in the topological order. The *exact area of a cut* is defined as the sum of areas of the LUTs in the MFFC of the cut, i.e. the LUTs to be added to the mapping if the cut is selected as the best one. Thus, minimizing exact area at each node is a helpful heuristic to minimize the total area of the mapping, which still remains NP hard.

The exact area of a cut is computed using a fast local DFS traversal of the subject graph starting from the root node. This traversal is similar to the recursive dereferencing of BDD nodes performed in a BDD package. The *reference counter of a node* in the subject graph is equal to the number of times it is used in the current mapping, i.e. the number of times it appears as a leaf of the best cut at some other node, or as a PO. Some internal nodes may have a zero reference counter, meaning that they are not used in the current mapping.

The exact area computation procedure is called for a cut. It adds the cut area to the local area being computed, dereferences the cut leaves, and recursively call itself for the best cuts of the leaves whose reference counters are zero. This procedure recurs as many times as there are LUTs in the MFFC of the cut, for which it is called. This number is typically small, which explains why computing the exact area is reasonably quick. Once the exact area is computed, a similar recursive referencing is performed to reset the reference counters to their initial values, before computing the exact area for other cuts.

We note here that MFFCs have been used in mapping previously [6]. Decomposition of the network into MFFCs was used for duplication-free mapping, which was alternated with depth relaxation for area minimization. Although both our method and [6] use MFFCs, the heuristics are different. In particular, our work employs reference counting for efficient computation and evaluation of MFFCs with duplication, which facilitates logic sharing.

Experimentally we found that, after computing a delay-optimum mapping, two passes of area recovery are enough to produce a good quality mapping. The first pass uses the area flow; the second one uses the exact area. Iterating area recovery using both of the heuristics additionally saves up to 2% of the total area of mapping, which may or may not justify the extra run-time.

It is interesting to observe that the previous work recovers area at each node in the *reverse* topological order, whereas our mapper works in the direct topological order. We argue that our approach works better for incremental area recovery since it allows most of the slack to be used on non-critical paths closer to the PIs where logic is denser and, therefore, optimization opportunities are more abundant. This argument is based on an observation that many circuits are wider on the PI side than on the PO side.

5 Lossless synthesis

The idea behind lossless logic synthesis is to “remember” some or all networks seen during a logic synthesis flow (or a set of flows) and to select the best parts of each network during technology mapping. This is useful for two reasons.

First, technology-independent synthesis algorithms are usually heuristic, and so there is no guarantee that the final network is optimum. When only the final network is used for mapping, the mapper may miss a better result that could be obtained from an intermediate network in the flow.

Second, synthesis operations usually apply to the network as a whole. So a flow to optimize delay may significantly increase area, since the whole network is optimized for delay. By combining such a delay-optimized network with another network that has been optimized for area, it is possible to get the best of both. On the critical path, the mapper can choose from the delay-optimized network, whereas off the critical path, the mapper chooses from the area-optimized network.

Section 5.1 gives an overview of constructing the choice network efficiently. Section 5.2 extends the cut computation to handle choices.

5.1 Constructing the choice network

The choice network is constructed from a collection of networks that are functionally equivalent. The key idea is to use recent advances in equivalence checking that are based on identifying functionally equivalent internal points in the networks being checked [13][15].

Conceptually the procedure is as follows: one can imagine each network to be decomposed into AND gates and inverters to form an AIG. Now for every node in the network the global function is computed, say, by building BDDs. All those nodes which have the same global function are collected in equivalence classes. Thus, the choice network is an AIG which has multiple functionally equivalent points collected in equivalence classes.

However, for large circuits computing global BDDs is not feasible. One can use random simulation to identify potentially equivalent nodes, and then use a SAT engine to verify equivalence and construct the equivalence classes. To this end, we implemented a package called FRAIG (Functionally Reduced And-Inverter Graphs) that exposes the APIs comparable to those of a BDD package but internally uses simulation and SAT. More details about FRAIGs may be found in the technical report [17].

Example. Figures 1 and 2 illustrate construction of a network with choices. Networks 1 and 2 in Figure 1 show the subject graphs obtained from two networks that are functionally equivalent but structurally different. The nodes x_1 and x_2 in the two subject graphs are functionally equivalent (up to complementation). They are combined in an equivalence class in the choice network, and an arbitrary member (x_1 in this case) is set as the class representative. Node p does not lead to a choice because p is structurally the same in both networks. Note also that there is no choice corresponding to the output node o since the procedure detects the maximal commonality of the two networks.

A different way of generating choices is by iteratively applying the Λ- and Δ-transformations [14]. Given an AIG, we use the associativity of the AND operation to locally re-write the graph (the Λ-transformation), i.e. whenever the

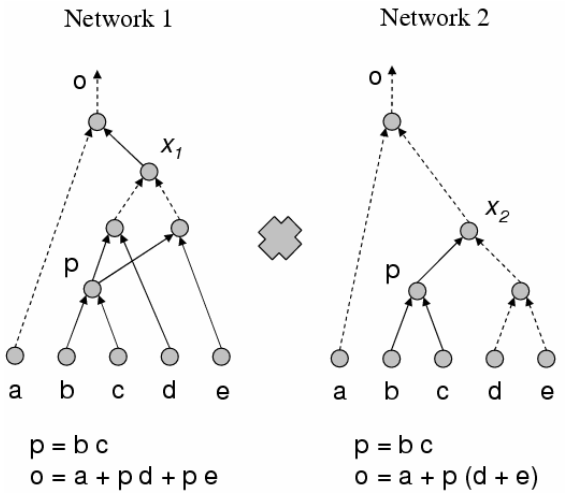

Figure 1. Equivalent networks before choicing.

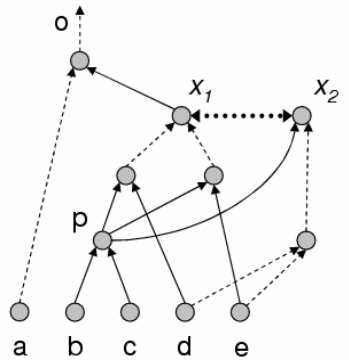

Figure 2. The choice network.

structure AND(AND(x_1, x_2), x_3) is seen in the AIG, it is replaced by the equivalent structures AND(AND(x_1, x_3), x_2) and AND(x_1, AND(x_2, x_3)). If this process is done until no new AND nodes are created, it is equivalent to identifying the maximal multi-input AND-gates in the AIG and adding all possible tree decompositions of these gates. Similarly, the distributivity of AND over OR (the Δ-transformation) provides another source of choices.

Using structural choices leads to a new way of thinking about logic synthesis: rather than trying to come up with a good final netlist used as an input to mapping, one can accumulate choices by applying arbitrary transformations, which lead to improvement in some sense. The best combination of these choices is selected during mapping.

5.2 Cut enumeration with choices

The cut-based structural FPGA mapping procedure can be extended naturally to handle equivalence classes of nodes. It is remarkable that only the cut enumeration step needs modification.

Given a node n, let N denote the equivalence class it belongs to. Let $\Phi(N)$ denote the set of cuts of the

equivalence class N. Then, $\Phi(N) = \bigcup_{n \in N} \Phi(n)$, where, if a and b are the two inputs of n belonging to equivalence classes A and B, respectively,

$\Phi(n) = \{\{n\}\} \cup \{u \cup v \mid u \in \Phi(A), u \in \Phi(B), |u \cup v| \leq k\}$.

This expression for $\Phi(n)$ is a slight modification of the one used in Section 3 to compute the cuts without choices. The cuts of n are obtained from the cuts of the *equivalence classes* of its fanins (instead of the cuts of its fanins). In the absence of choices (which corresponds to the situation when each equivalence class has only one node) this computation is the same as the one presented in Section 3. As before, the cut enumeration is done in one topological pass from the PIs to the POs.

Example. Consider the computation of the 3-feasible cuts of the equivalence class $\{o\}$ in Figure 2. Let X represent the equivalence class $\{x_1, x_2\}$. Now, $\Phi(X) = \Phi(x_1) \cup \Phi(x_2) = \{\{x_1\}, \{x_2\}, \{q, r\}, \{p, s\}, \{q, p, e\}, \{p, d, r\}, \{p, d, e\}, \{b, c, s\}\}$. We have $\Phi(\{o\}) = \Phi(o) = \{\{o\}\} \cup \{u \cup v \mid u \in \Phi(\{a\}), u \in \Phi(\{x_1\}), |u \cup v| \leq 3\}$.

Since $\Phi(\{a\}) = \Phi(a) = \{a\}$ and $\Phi(\{x_1\}) = \Phi(X)$, we get $\Phi(\{o\}) = \{\{o\}, \{a, x_1\}, \{a, x_2\}, \{a, q, r\}, \{a, p, s\}\}$. Observe that the set of cuts of o involves nodes from the two choices x_1 and x_2, i.e. o may be implemented using either of the two structures.

The subsequent steps of the mapping process (computing delay-optimum mapping and performing area recovery) remain unchanged, except that now the additional cuts can be used for mapping at each node.

5.3 Related Work

Technology mapping over a network that encodes different decompositions originated in the context of standard cell mapping in the work of Lehman et al. [14]. Chen and Cong adapted some aspects of this method for FPGAs in their work on SLDMap [4]. To be specific, they identified large (5- to 8-input) AND gates in the subject graph, and added choices corresponding to the different decompositions of the large AND gates into 2-input AND gates. They used BDDs to find globally equivalent points. This limited the scalability of the approach.

The present work is an extension of our work in standard cells [3] to FPGA mapping. This approach differs from SLDMap in two ways. First, the use of structural equivalence checking instead of BDDs makes the choice detection scalable and robust. Second, instead of adding a dense set of algebraic choices by brute-force, we add a sparse set of (possibly Boolean) choices obtained from synthesis. The expectation is that most of the algebraic choices that are added are not useful, but increase run-time. In contrast the choices added from synthesis are expected to be better, since they are a result of optimization. This is supported by experiments on standard cells [3] and we expect similar results to hold for FPGAs.

6 Experimental results

The proposed improvements to FPGA technology mapping are currently implemented in MVSIS [20] as command *fpga*. The cut enumeration is implemented in ABC [21] as command *cut*. (Since the first version of this paper, command *fpga* was improved and ported to ABC, making ABC our main tool for future experiments.)

6.1 Improved cut computation (run-time)

Table 1 shows the results of cut computation for the largest MCNC benchmarks. To derive AIGs required for cut enumeration in ABC, the benchmarks were structurally hashed and balanced using command *balance* in ABC.

The experiment was performed for computing K-feasible cuts for $4 \leq K \leq 8$. Column N gives the number of AND nodes in the AIG for each benchmark. Columns *C/N* give the average number of cuts per node, including trivial cuts composed of the nodes themselves. Columns T give the run-time in seconds on an IBM ThinkPad laptop with 1.6GHz CPU and 1GB of RAM. The final column *L/N* lists the percentage of nodes, for which the number of 8-input cuts exceeded the predefined limit, set to 1000 for benchmarks. In computing cuts for $4 \leq K \leq 7$, the number of cuts never exceeded the limit and, as a result, the cuts are computed exhaustively.

In summary, although the number of cuts and their computation time are exponential in the number of cut inputs (K), all the cuts can be computed for up to 7 inputs for most benchmarks in reasonable run-time, resulting in over 100 cuts per node.

6.2 Improved cut computation (memory)

The second experiment presented in **Table 2** addresses the issue of memory requirements for the cut representation, by showing the reduction in the peak memory with and without cut dropping. The amount of memory used for a K-feasible cut in the ABC data structure is $(12+4*K)$ bytes.

Columns labeled *Total* list memory usage (in megabytes) for all the non-dominated, K-feasible cuts at all nodes. Columns labeled *Drop* list the peak memory usage (in megabytes) for the cuts at any moment in the process of cut enumeration, when the nodes are visited in the topological order and the cuts at a node are dropped as soon as the cuts at all the fanouts are computed.

In summary, dropping cuts at the internal nodes after they are computed and used reduces memory requirements for the mapper by an order of magnitude on the largest MCNC benchmarks, and by more then two orders of magnitude on the large industrial benchmarks, such as [11].

6.3 Improved area recovery

Sections *DAOmap* and *MVSIS-baseline* of **Table 3** compare the FPGA mapping results for 5-input LUTs using DAOmap [2][1] and our mapper with improved area recovery. Both DAOmap and MVSIS were run on a 4 CPU

3.00GHz computer with 510Mb RAM under Linux. The benchmarks are pre-optimized using *script.algebraic* in SIS followed by decomposition into two-input gates using command *dmig* in the RASP package [8]. To ensure identical starting logic structures, the pre-optimized circuits from [2][1] were used in this experiment. All the resulting netlists have been verified by a SAT-based equivalence checker in MVSIS.

Columns 2 and 5 give the number of logic levels of LUT networks after technology mapping. The values in these columns are equal in all but two cases. This supports the claim that both mappers perform delay-optimum mapping for the given logic structure. Differences may be explained by minor variations in the manipulation of the subject graph, such as AIG balancing performed by MVSIS.

Columns 3 and 6 show the number of LUTs after technology mapping. The difference between the results produced by the two mappers reflects the fact that they use different area recovery heuristics and, possibly, that MVSIS-baseline performs area recovery in a topological order, whereas DAOmap uses a reverse topological order.

Columns 4 and 7 report the run-times in seconds. These include the time for constructing the subject graph and perform technology mapping with area recovery but not the time for reading the input BLIF file. For smaller benchmarks, the differences in run-times might be explained by the differences in the basic data structures. The increased run-time advantages of MVSIS on larger benchmarks may be due to better scalability and filtering heuristics employed by the MVSIS mapper.

In summary, Table 3 demonstrates that the mapper in MVSIS designed using the proposed heuristics for area recovery outperforms DAOmap in area and run-time.

The run-time of FPGA mapping is dominated by the *K*-feasible cut computation. The results for MVSIS reported in Table 3 use an old implementation of cut enumeration, which is several times slower than that reported in Table 1. We expect the run-time of the proposed mapper to improve after integrating the new cut computation.

6.4 Lossless synthesis

Section *MVSIS-choices* of **Table 3** gives mapping results for the same benchmarks when lossless synthesis is applied. The alternative logic structures were generated in MVSIS by applying *choice.script* given in [3]. This script is similar to *script.rugged* in SIS. The difference is that the original network and five intermediate networks are combined into one choice network while detecting functionally equivalent nodes, as shown in Section 5. The mapping run-time listed in Table 3 does not include the run-time of choicing. This run-time was smaller than the run-time of intermediate transformations of technology independent synthesis (such as *eliminate*, *fast_extract*, *sweep* etc).

Section *MVSIS-choices 2x* shows the results of repeated application of mapping with choices. For this, the netlist mapped into LUTs by the first mapping with choices was decomposed into an AIG by factoring logic functions of the LUTs, and subjected again to lossless synthesis followed by mapping with choices. The last column shows the run-time, in seconds, taken by the second iteration of mapping with choices. As before, this run-time does not include the run-time of logic optimization and choice generation resulting from applying *choice.script*.

In summary, the above experiments demonstrate that lossless synthesis has a potential for substantially reducing delay and area of the mapped netlists, both as a stand-alone mapping procedure and as a post-processing step applied to the already computed FPGA mapping.

7 Conclusions

The paper presented several improvements to the state-of-the-art in technology mapping for LUT-based FPGAs. The improvements are: (1) reduction in run-time and memory requirements for cut enumeration; (2) improved area recovery through combined use of global-view and local-view heuristics; and (3) improved delay and area through the use of multiple circuit structures to mitigate structural bias during technology mapping.

The experimental results confirm that the improved area recovery procedure leads, on average, to a 3x improvement in run-time and a 7% smaller area, compared to DAOmap, while preserving the optimum delay when starting from the same logic structure. When multiple logic structures are used in lossless synthesis, the proposed mapper leads to 7% improvement in delay along with a 14% reduction in area with a slight increase in run-time, compared to DAOmap.

The next step is integrating the efficient cut enumeration package into the FPGA mapper (currently the mapper uses simple cut enumeration without cut dropping). The future work will also extend the FPGA mapping to perform integrated sequential optimization, which consists of logic restructuring, mapping, and retiming, as presented in [19].

Acknowledgment

This research was supported in part by NSF contract, CCR-0312676, by the MARCO Focus Center for Circuit System Solution under contract 2003-CT-888 and by the California Micro program with our industrial sponsors, Intel, Magma, and Synplicity.

The authors are grateful to Jason Cong and Deming Chen for providing the set of pre-optimized benchmarks from [2], which allowed for a comparison with DAOmap in Table 3.

References

[1] B. Bloom. “Space/time tradeoffs in hash coding with allowable errors,” *Comm. of the ACM* 13:7 (1970), pp. 422-426.

[2] D. Chen, J. Cong. “DAOmap: A depth-optimal area optimization mapping algorithm for FPGA designs,” *Proc. ICCAD ’04*, pp. 752-757.

[3] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD '05*,

pp. 519-526. http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf

[4] G. Chen and J. Cong, “Simultaneous logic decomposition with technology mapping in FPGA designs,” *Proc. FPGA `01*, pp 48-55.

[5] J. Cong and Y. Ding, “FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs”, *IEEE Trans. CAD*, Vol.13(1), Jan. 1994, pp. 1-12.

[6] J. Cong and Y. Ding, “On area/depth trade-off in LUT-based FPGA technology mapping,” *IEEE Trans. VLSI, Vol 2(2),* Jun. 1994, pp 137-148.

[7] J. Cong, C. Wu and Y. Ding, “Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution,” *Proc. FPGA `99*, pp. 29-36.

[8] J. Cong et al, *RASP: FPGA/CPLD Technology Mapping and Synthesis Package*. http://ballade.cs.ucla.edu/software_release/rasp/htdocs/

[9] N. Eén, A. Biere “Effective preprocessing in SAT through variable and clause elimination,” *Proc. SAT'05*.

[10] A. Farrahi and M. Sarrafzadeh, “Complexity of lookup-table minimization problem for FPGA technology mapping,” *IEEE Trans. CAD*, vol. 13 (11), 1994, pp. 1319-1332.

[11] IWLS 2005 Benchmarks. http://iwls.org/iwls2005/benchmarks.html

[12] C.-C. Kao, Y.-T. Lai, “An efficient algorithm for finding minimum-area FPGA technology mapping". *ACM TODAES*, vol. 10(1), Jan. 2005, pp. 168-186.

[13] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. CAD*, Vol. 21(12), 2002, pp. 1377-1394.

[14] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, “Logic decomposition during technology mapping,” *IEEE Trans. CAD*, vol. 16(8), 1997, pp. 813-833.

[15] F. Lu, L. Wang, K. Cheng, J. Moondanos and Z. Hanna, “A signal correlation guided ATPG solver and its applications for solving difficult industrial cases," *Proc. DAC `03*, pp. 668-673.

[16] V. Manohara-rajah, S. D. Brown, Z. G. Vranesic, “Heuristics for area minimization in LUT-based FPGA technology mapping,” *Proc. IWLS '04,* pp. 14-21.

[17] A. Mishchenko, S. Chatterjee, R. Jiang, R. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," *ERL Technical Report*, EECS Dept., UC Berkeley, March 2005.

[18] A. Mishchenko, S. Chatterjee, R. Brayton, and M. Ciesielski, "An integrated technology mapping environment," *Proc. IWLS '05*, pp. 383-390. http://www.eecs.berkeley.edu/~alanmi/publications/2005/iwls05_env.pdf

[19] A. Mishchenko, S. Chatterjee, R. Brayton, and P. Pan, "Integrating logic synthesis, technology mapping, and retiming", *Proc. IWLS '05*, pp. 383-390. Also, submitted to *DAC '06*. http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_int.pdf

[20] MVSIS Group. *MVSIS: Multi-Valued Logic Synthesis System.* UC Berkeley. http://www-cad.eecs.berkeley.edu/mvsis/

[21] Berkeley Logic Synthesis and Verification Group, ABC: A System for Sequential Synthesis and Verification, Release 50905. http://www.eecs.berkeley.edu/~alanmi/abc/

[22] P. Pan and C.-C. Lin, “A new retiming-based technology mapping algorithm for LUT-based FPGAs,” *Proc. FPGA '98*, pp. 35-42.

[23] M. Teslenko and E. Dubrova, “Hermes: LUT FPGA technology mapping algorithm for area minimization with optimum depth,” *Proc. ICCAD '04*, pp. 748-751.

Table 1. Performance of improved *K*-feasible cut computation (see Section 6.1).

		K = 4		*K* = 5		*K* = 6		*K* = 7		*K* = 8		
Name	N	C/N	T, s	C/N	T, s	C/N	T, s	C/N	T, s	C/N	T, s	L/N, %
alu4	2642	6.7	0.00	12.3	0.01	23.1	0.04	45.5	0.18	94.7	1.02	0.00
apex2	2940	7.2	0.01	14.2	0.02	29.2	0.07	62.6	0.32	139.7	1.90	0.00
apex4	2017	8.5	0.00	19.5	0.03	47.0	0.10	116.3	0.62	293.5	4.49	0.10
bigkey	3080	6.6	0.01	12.1	0.02	24.2	0.05	50.1	0.20	99.7	0.84	0.00
clma	11869	8.1	0.04	18.2	0.11	44.4	0.51	114.9	3.01	306.3	20.99	1.64
des	3020	8.0	0.01	17.0	0.03	38.7	0.12	92.0	0.69	218.0	4.80	4.37
diffeq	2566	6.5	0.01	12.3	0.01	26.6	0.07	65.0	0.50	155.9	2.80	3.66
dsip	2521	6.2	0.01	10.7	0.01	20.7	0.03	42.0	0.10	86.7	0.44	0.00
elliptic	5502	6.4	0.01	10.6	0.03	18.5	0.07	36.9	0.33	83.4	2.12	0.20
ex1010	7652	9.2	0.02	23.3	0.11	61.8	0.61	165.8	4.01	438.2	30.43	1.99
ex5p	1719	9.4	0.01	24.1	0.02	66.2	0.17	188.2	1.30	514.8	10.50	14.14
frisc	5905	7.1	0.01	14.4	0.04	32.3	0.16	79.8	0.88	209.0	6.30	1.24
misex3	2441	7.7	0.01	15.7	0.02	33.3	0.08	73.7	0.38	170.7	2.48	0.00
pdc	7527	9.4	0.03	24.8	0.12	67.4	0.68	183.7	4.41	489.4	31.71	4.40
s298	2514	7.9	0.00	17.5	0.02	44.0	0.13	121.9	0.94	346.5	7.10	7.56
s38417	12867	6.6	0.03	13.5	0.10	32.0	0.46	83.1	3.24	225.9	23.72	3.38
s38584.1	11074	6.1	0.03	11.4	0.06	22.4	0.20	46.7	0.98	101.5	5.81	0.86
seq	2761	7.5	0.00	15.2	0.02	31.7	0.08	68.6	0.37	153.3	2.25	0.04
spla	6556	9.6	0.03	25.8	0.11	73.9	0.69	215.5	4.98	561.4	31.14	13.83
tseng	1920	6.5	0.01	11.8	0.01	23.5	0.04	50.6	0.21	112.7	1.32	1.35
Average	4954.65	7.56	0.01	16.22	0.05	38.05	0.22	95.15	1.38	240.07	9.61	2.94

Table 2. Peak memory requirements, in megabytes, for the cuts with and without dropping (see Section 6.2).

	K = 4		*K* = 5		*K* = 6		*K* = 7		*K* = 8	
Name	Total	Drop	Total	Drop	Total	Drop	Total	Drop	Total	Drop
clma	2.56	0.10	6.60	0.22	18.09	0.54	52.03	1.47	152.55	4.07
ex1010	1.87	0.37	5.45	0.97	16.25	2.27	48.40	4.68	140.70	8.38
pdc	1.90	0.27	5.69	0.75	17.42	2.00	52.75	4.98	154.56	11.83
s38417	2.28	0.15	5.28	0.37	14.12	1.10	40.80	3.55	121.98	10.25
s38584.1	1.80	0.11	3.86	0.20	8.52	0.40	19.72	0.86	47.15	1.94
spla	1.68	0.21	5.15	0.59	16.63	1.65	53.88	4.34	154.44	10.04
Ratio	1.00	0.11	1.00	0.10	1.00	0.08	1.00	0.07	1.00	0.06

Table 3. Comparing FPGA mapper with improvements with DAOmap [2] (see Section 6.3).

Example	DAOmap			MVSIS-baseline			MVSIS-choices			MVSIS-choices 2x		
	Depth	LUTs	T, s	Depth	LUTs	T, s	Depth	LUTs	T, s	Depth	LUTs	T, s
alu4	6	1065	0.5	6	992	0.34	6	972	0.64	6	949	+0.84
apex2	7	1352	0.6	7	1200	0.36	7	1249	0.95	7	1191	+1.34
apex4	6	931	0.7	6	891	0.24	6	895	0.74	6	894	+1.47
bigkey	3	1245	0.6	3	797	0.34	3	797	0.75	3	684	+1.07
clma	13	5425	5.9	13	4426	1.50	11	3883	4.30	11	3453	+5.20
des	5	965	0.8	5	1024	0.36	5	947	0.93	5	1104	+1.87
diffeq	10	817	0.6	10	844	0.30	9	745	0.46	9	736	+0.43
dsip	3	686	0.5	3	686	0.23	3	685	0.19	3	684	+0.36
elliptic	12	1965	2.0	12	2017	0.61	12	2005	0.72	12	2022	+1.25
ex1010	7	3564	4.0	7	3258	1.15	7	3305	3.39	7	3302	+5.80
ex5p	6	778	1.0	6	744	0.36	5	724	1.17	5	675	+1.40
frisc	16	1999	1.9	15	2009	0.76	14	1875	1.54	13	1867	+1.58
misex3	6	980	0.8	6	957	0.26	6	926	0.73	6	861	+0.94
pdc	7	3222	4.6	8	2920	1.13	7	2738	4.73	7	2692	+5.59
s298	13	1258	2.4	13	826	0.30	12	863	4.07	11	826	+1.49
s38417	9	3815	3.8	9	3864	1.46	8	2989	4.04	7	2729	+2.76
s38584	7	2987	27.0	7	2844	1.11	7	2497	2.58	6	2470	+1.69
seq	6	1188	0.8	6	1109	0.30	5	1136	0.79	6	1016	+1.38
spla	7	2734	4.0	7	2535	1.03	7	2319	4.68	7	2224	+4.79
tseng	10	706	0.6	10	752	0.25	8	719	0.39	8	705	+0.31
Ratio	1.00	1.00	1.00	1.00	0.93	0.37	0.95	0.89	0.95	0.93	0.86	1.46