Reducing Structural Bias in Technology Mapping

S. Chatterjee A. Mishchenko R. Brayton
Department of EECS
U. C. Berkeley
{satrajit, alanmi, brayton}@eecs.berkeley.edu

X. Wang T. Kam
Strategic CAD Labs
Intel Corporation
{xinning.wang, timothy.kam}@intel.com

ABSTRACT

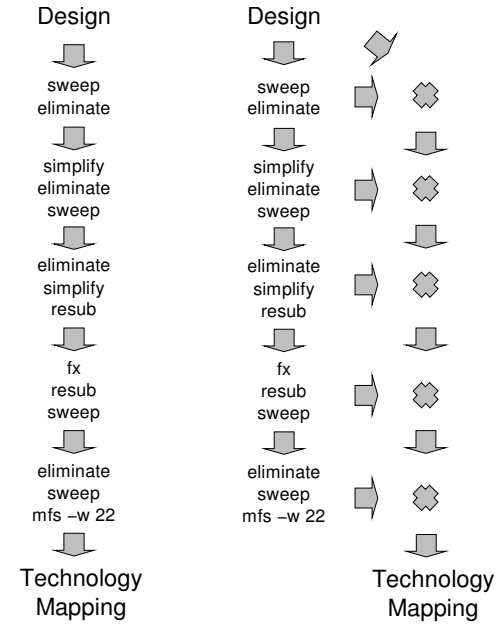
Technology mapping based on DAG-covering suffers from the problem of structural bias: the structure of the mapped netlist depends strongly on the subject graph. In this paper we present a new mapper aimed at mitigating structural bias. It is based on a simplified cut-based Boolean matching algorithm, and using the speed afforded by this simplification we explore two ideas to reduce structural bias. The first, called lossless synthesis, leverages recent advances in structure-based combinational equivalence checking to combine the different networks seen during technology independent synthesis into a single network with choices in a scalable manner. We show how cut-based mapping extends naturally to handle such networks with choices. The second idea is to combine several library gates into a single gate (called a supergate) in order to make the matching process less local. We show how supergates help address the structural bias problem, and how they fit naturally into the cut-based Boolean matching scheme. An implementation based on these ideas significantly outperforms state-of-the-art mappers in terms of delay, area and run-time on academic and industrial benchmarks.

1. INTRODUCTION

The task of technology mapping in standard-cell logic synthesis is to express a given Boolean function as a network of gates chosen from a given standard-cell library to optimize some objective function such as total area or delay. In these general terms, technology mapping is intractable. The problem is usually simplified by first representing the Boolean function as a good initial multi-level network of simple gates called the subject graph. The subject graph is then transformed into a multilevel network of library gates by means of local substitutions. This simplification means that the structure of the subject graph dictates to a large extent the structure of the mapped network; this is known as structural bias.

In this work we present a new Boolean technology mapper aimed at mitigating the effects of structural bias. At the core of the mapper is a simplified Boolean matching algorithm that is faster than structural matching and produces better results. With the speed afforded by this matching technique, we propose two complementary techniques to reduce structural bias: lossless synthesis and supergates.

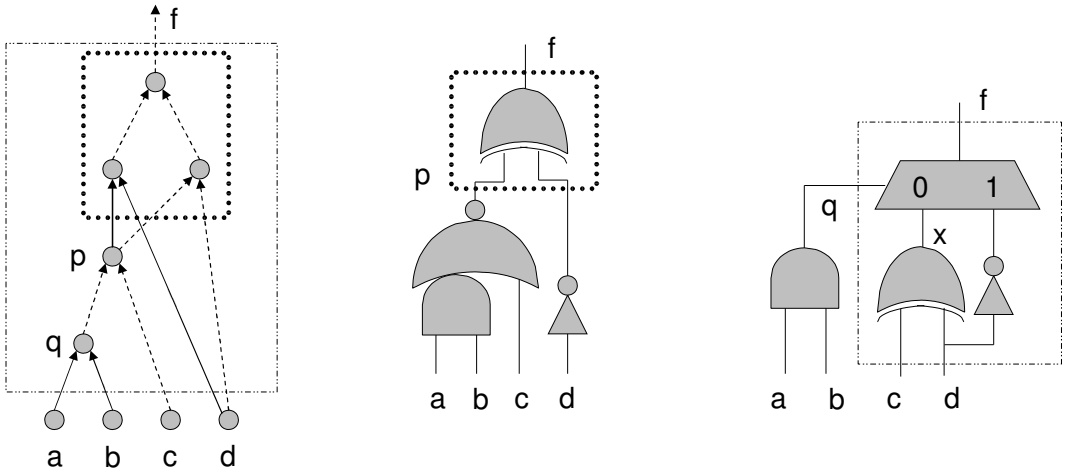
Lossless Synthesis. To obtain a good structure for the subject graph a number of technology independent synthesis steps are usually performed. An example of this is the SIS rugged script shown in Figure 1(a). Each step in the script is heuristic, and the subject graph produced at the end of the script is not necessarily optimal; indeed it is possible that an intermediate network is better in some respects than the final network.



(a) Conventional Synthesis (b) Lossless Synthesis

Figure 1: In lossless synthesis intermediate networks are combined to create a choice network which is then used for mapping.

We explore the idea of combining these intermediate networks into a single subject graph with choices and using that to derive the mapped netlist. This is shown schematically in Figure 1(b). Following the work of Lehman et al. [13], the mapping is done optimally over all the networks encoded in the subject graph. The mapper is not constrained to use any one network, but can pick and choose the best parts of each network. We call this *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. Furthermore, as in [13], multiple scripts (with possibly different objectives) could be used to accumulate more choices.



(a) The subject graph. (b) An easy match to find. (c) A hard match to find.

Figure 2: Simple example illustrating the utility of supergates. Consider the subject graph shown in (a) where each node represents a 2-input AND gate and a dashed edge indicates the presence of an inverter. The match in (b) is easy to find since both inputs of the XOR are present in the subject graph. In contrast the match in (c) is hard to find, since the MUX input labeled x is not present in the subject graph. The match in (c) can be found by combining some gates into a supergate. Note that all the inputs of the supergate are still required to be present in the subject graph.

The challenging part of this proposal is the efficient creation of the choice network. Towards this end we propose to leverage recent advances in combinational equivalence. State-of-the-art equivalence checkers depend on finding functionally equivalent internal points in the networks being checked in order to reduce the complexity of the decision procedure. By using a combination of random simulation and SAT it is possible to quickly determine equivalent points in two circuits. These equivalent points provide the choices for mapping in our proposal.

Supergates. A supergate is a single output combinational network of a few library gates which is treated as a single library gate by our algorithms. The advantage of doing this may not be immediately obvious: one might expect that if a supergate matches at a node, the conventional matching algorithm would return the same result, except it would match library gate by library gate rather than all the gates at once. This is not true and Figure 2 provides a simple counter-example.

The subject graph is shown in Figure 2(a). Conventional mapping would find the mapping consisting of the XOR and AOI gates as shown in Figure 2(b), but would fail to find the mapping with the MUX as shown in Figure 2(c). To see this, observe that one of the inputs of the MUX (labeled x in Figure 2(c)) is not present in the subject graph. Consequently, the MUX by itself is not a valid match for f. In contrast, the inputs of the XOR (nets p and $\neg$d) in Figure 2(b) are both present in the subject graph; thus making the XOR a valid match at f.

Now if we connect the MUX, the XOR, and the inverter together to form a new gate (in the manner shown in Figure 2(c)), it is easily verified that this new gate is a match at f in the subject graph. This new gate is a supergate built from the library gates expressly for the purpose of finding better matches.

This example illustrates the main idea behind supergates: Using bigger gates allows the matching procedure to be less local, and thus less affected by the structure of the subject graph. Furthermore, as this example illustrates, supergates are useful even with standard cell libraries that are functionally rich.

The three ideas presented in this paper – the faster matching algorithm, lossless synthesis, and supergates – are logically independent. However, one of the contributions of this paper is to show how these ideas come together naturally in the context of cut-based Boolean mapping.

In Section 2 we present related work. Section 3 has an overview of the mapper and details of the matching algorithm. In Section 4 we present practical techniques to construct the choice network for lossless synthesis and extend the basic mapping to handle choices. In Section 5 we present details of supergate construction. In Section 6 we present the results of experiments designed to determine the runtime–quality trade-off of the techniques presented in this paper. We also present comparisons with other industrial and academic mappers. We conclude in Section 7 by pointing out some limitations of these techniques and suggesting future work.

Finally, we note that the focus of this paper is on mitigating structural bias, that is, on the logical aspects of the technology mapping problem. Therefore to simplify exposition we present the algorithms in the context of a simple gain-based delay model which does not take capacitive loading into account. Such a model is useful in practice in a flow that uses technology mapping for topology selection, and does sizing and buffering iteratively with placement. Furthermore, the techniques in this paper continue to be applicable in a more traditional load-based flow, since essentially

they are techniques to increase the number of topologies considered during mapping, and are orthogonal to the usual techniques of considering loads during technology mapping.

2. RELATED WORK

The literature on technology mapping provides a spectrum of techniques that trade structural bias for computational complexity. The classical structural approaches, such as tree- and DAG-covering [8, 12], lie at one end of this spectrum. They have relatively short run-times but provide sub-optimal results since their mapping choices are completely constrained by the given subject graph. Constructive Boolean approaches [9, 19] lie closer to the other end of the spectrum. Although they do not depend as much on the structure of the subject graph, they are limited by the choice of their (heuristic) decomposition schemes and local view since they decompose one node at a time. These limitations combined with long run-time makes them useful mostly in a re-synthesis flow after a mapped network has been obtained by some other means.

The approach described by Lehman et al. [13] lies between these two extremes: a number of different local algebraic decompositions are encoded into the given subject graph as choices. The subject graph itself is constructed by combining the results of different technology mapping scripts. The ideas relating to lossless synthesis presented in this paper are variations on this basic approach. Beyond the obvious difference of using intermediate networks from one synthesis script, there are significant algorithmic differences. First, the use of structural equivalence checking to detect choices (as opposed to BDDs) allows for lossless synthesis on large circuits. Second, the extension of Boolean matching to handle choices presented in this paper overcomes the run-time limitations of the structural matching methods used by Lehman et al. Furthermore, there is an improvement in quality since matching is done directly on general DAGs and complex multi-fanout gates are handled more naturally.

Wavefront mapping [21] is a practical enhancement of [13] that maps the circuit in stages, thus reducing the amount of match data that is stored at one time, though at the cost of optimality. An extension of this algorithm allows for dynamic decomposition based on a partially mapped circuit. Compared to the approach of Lehman et al. it reduces the number of decompositions that need to be explored. The idea of wavefront can be used in conjunction with the techniques in this paper to reduce the amount of match data that needs to be stored in memory at one time.

In addition, this paper puts forth the idea of combining library gates into supergates to obtain a more Boolean matching. By squinting a little, one can view supergates as being an alternate source of choices. The choices induced by supergates come from the library rather than the network. In the literature similar combinations of gates have been used for other purposes such as constructive decomposition [19] and rewiring [2]. We note here that supergates allows library-aware *Boolean* decompositions to be explored as opposed to algebraic decompositions in [13], and we defer the discussion to Section 5.5.

The work related to the simplified Boolean matching is presented in Section 3.2.

3. BOOLEAN MATCHING

We introduce the simplified, Boolean matching algorithm in the context of the overall mapping flow. The mapping procedure is a Boolean, cut-based one [4, 6, 7] on a DAG using dynamic programming [3, 12] to guarantee delay optimality.

3.1 Overview of Boolean Mapping

The input to the mapping procedure is an AND-Inverter graph (AIG) [11]. An AIG is a DAG whose nodes represent either AND gates or primary inputs(PIs). Its edges represent wires. Inverters are represented by bubbles on the edges. Given an AIG, the mapping is done in 5 steps.

Step 1. Compute k-feasible cuts. A *feasible cut* of a node N in the AIG is a set of nodes $\{X_i\}$ in the transitive fan-in cone of N such that an arbitrary assignment of values to X_i completely determines the value of N. A feasible cut is *redundant* if the value of a node in the cut is completely determined by an assignment of values to the other nodes in the cut. A *k-feasible cut* is a feasible cut of size at most k that is not redundant. The cut $\{N\}$ is always a k-feasible cut of node N (for any k) and is called the *trivial cut*.

Let $\Phi(N)$ denote the set of k-feasible cuts of node N. If N is a PI, then $\Phi(N) = \{\{N\}\}$. If N is a AND node with inputs A and B, then $\Phi(N) =$

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

We compute all 5-feasible cuts of every node in the network by the simple bottom-up traversal based on the above recursion. Although in general a graph may have $O(n^5)$ 5-feasible cuts, we found that most test-cases have between 20 and 30 5-feasible cuts per node. We restrict our attention to 5-feasible cuts since our experiments show that the total number of cuts increases very quickly with k. Pruning techniques have to be applied, and the mapping results are not significantly better (since the pruning is quite arbitrary).

Step 2. Compute truth-tables of cuts. The next step is to compute the local function of a node in terms of its cut. This is done for every non-trivial k-feasible cut of every node in the network. Given a node N, and a cut $\{X_i\}$ of that node, formal variables are assigned to the each cut node (in no particular order). Using these variables, the functionality of the node is computed symbolically. We note here that BDDs are not necessary for this computation: since usually only 5-feasible cuts are considered the symbolic function computation can be performed efficiently using 32-bit machine words to represent truth-tables. In what follows we use the words "function" and "truth-table" interchangeably.

Step 3. Boolean Matching. For each node in the network, for every cut, an appropriate gate (if one exists) is chosen from the library. Each gate thus chosen is called a match for the node. Our matching procedure differs from the traditional approach and we present this in greater detail in the following section.

Step 4. Compute best arrival time at each node. Starting from the PIs and working in topological order towards the outputs, the best arrival time is computed for each node from amongst all its matches.

Step 5. Choose the best cover. In the reverse topological order, the best gate for each primary output is cho-

sen. Next, the best gates implementing the inputs of these gates are chosen and so on until all primary inputs have been reached.

3.2 Simplified Boolean Matching

In the matching step we wish to choose the appropriate library gate to implement a given function. The traditional solution is to use NPN-equivalence classes. (Two Boolean functions are NPN-equivalent if one can be transformed into the other by permuting or negating the inputs or negating the output.) First the NPN-canonical representatives of the library gates are computed. The gates are stored in a hash table indexed by the representatives. During matching the NPN-canonical representative is computed for every cut function. It is used to look up the hash table to find the appropriate gate.

However this approach is slow since it requires the computation of the NPN-representative for every cut function. Furthermore, once the appropriate gate is found, the appropriate variable correspondence must be found between the library gate and the cut function. Since these computations are done in the inner body of the mapper, there has been a lot of research on speeding them up [1, 5, 7, 22].

Our simplified matching procedure is motivated by the fact that in the mapping procedure described above, the functions have only 5 variables. It is therefore advantageous to precompute all functions obtained by permuting the inputs to library gates and to add those to the hash table. Thus during the actual matching, there is no need to compute a canonical representative. The cut function can be used directly to look up the hash table. In addition to avoiding the NPN-equivalence checking, it also avoids the need to establish input correspondence after the gate is found.

Example. For an AOI gate having the function $\neg(x_1 \cdot x_2 + x_3)$, we precompute all permutations i.e. $\neg(x_1 \cdot x_3 + x_2)$, $\neg(x_2 \cdot x_1 + x_3)$, $\neg(x_2 \cdot x_3 + x_1)$ etc. Note that both $\neg(x_2 \cdot x_1 + x_3)$ and $\neg(x_1 \cdot x_2 + x_3)$ are kept although they are the same function. This is because in industrial libraries, the input-to-output delays are often different even for functionally symmetric pins.

Once again, all functional computations are done with truth tables. In Section 5.2 on supergate generation we describe this precomputation procedure in more detail. (The precomputation can be thought of as generation of supergates consisting of single library gates.)

So far we have not considered the phase assignment at the inputs. This is because the mapping is done "dual rail" as explained in the following section on optimal phase selection.

The problem with this simplified matching procedure is that it is not scalable. Indeed, this would not work if the functions we were interested in had, say, 10 variables. But in the framework of Boolean mapping it suffices since we deal with functions having only a few variables.[1] This means that in the worst case (a library gate with 5 inputs, and no symmetries) 120 functions are added to the hash table (instead of 1 as in the NPN-representative case). However since each match is now only a hash table lookup (versus an NPN-representative computation, lookup, and input correspondence), the matching procedure runs faster.

[1]In Boolean mapping, as more variables are used, the cut computation becomes a bottleneck before the matching does.

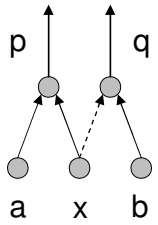


Figure 3: In order to map p and q optimally for delay, the node x must be mapped in both polarities.

3.3 Optimal Phase Selection

The mapping quality can be improved by exploiting the additional flexibility of mapping each node in both polarities: positive (as above) and negative. When the final mapping is selected (in Step 5 of Section 3.1) the appropriate polarity is chosen to guarantee the shortest delay on each path. This may lead to an increase in area since the same node may be required in both polarities, and may have to be duplicated.

Example. Consider the AIG fragment in Figure 3. Node x is required in both polarities. If it is mapped in only one polarity then the arrival time at either p or q increases by an additional inverter delay.

Observe that this "dual rail" mapping means that the inputs of a cut are available in both polarities. Consequently the function of a cut is not precisely determined: it belongs to a class of functions which differ only by complementation of inputs. This class is called the N-equivalence class. There is greater flexibility since any gate belonging to the N-equivalence class may be used to implement the cut.

The simplified matching procedure above is extended to work with N-equivalence classes. During library preprocessing, for every permutation of a gate, the N-equivalence representative is computed. This is defined as the function with the lexicographically smallest truth table. Similarly during matching, for every node function the N-equivalence representative is computed and used for looking up in the hash table.

The alert reader would have noticed that this extension negates some speed benefit of the simplified matching procedure presented in Section 3.2. However this may be justified with the following arguments. First, computing the N-equivalence representative is less expensive than computing the NPN-equivalence representative since the class is smaller (2^n versus $2^{n+1}n!$). Second, this is a necessary price to pay for exploring this larger search space of decompositions (c.f. the inverter transform in Lehman et al. [13]). In structural mappers this search space is explored by adding a pair of inverters between two nodes, and by adding a wire (with zero cost) to the library. That technique is not well suited for Boolean mapping since it significantly increases the number of cuts in the network.

3.4 Using Don't Cares

An advantage of the Boolean matching technique outlined above is the ability to handle local satisfiability don't cares during the matching process. If the matching is done with $k-$feasible cuts, then for every node, a cut larger than k is constructed. This can be done by a simple depth-first search starting from the node. (For example if matching is done

with 5-feasible cuts, a larger cut of size, say, 10 is considered.) For every $k-$feasible cut of the node, it is possible to compute the satisfiability don't cares symbolically. This can be done either with exhaustive simulation using truth-tables or with BDDs. Once the satisfiability don't cares of a $k-$feasible cut are identified, each don't care minterm can be set to one or zero to obtain a completely specified cut function. This completely specified function is used to look-up the hash table as before. Thus each $k-$feasible cut gives rise to a set of cut functions when satisfiability don't cares are considered.

The main drawback of this method is the exponential blow-up in the number of cut functions since each don't care gives rise to two possible functions. Thus in practice only an arbitrary subset of the cut functions can be used for matching. Our preliminary experiments confirmed that using don't cares in this manner improved the quality of mapping. However, in our application the increase in quality was not justified by the increase in run-time.

4. LOSSLESS LOGIC SYNTHESIS

The idea behind lossless logic synthesis is to "remember" every network seen during a 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 optimal. By mapping using only the final network, we might miss out on a better result that could be obtained from an intermediate network in the optimization flow.

Second, synthesis operations usually apply to the network as a whole. So a flow to optimize delay might 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.

The main problem is constructing the choice network efficiently. In Section 4.1 we give an overview of how this is done. In Section 4.2 we extend the Boolean mapping procedure of Section 3.1 to handle choices.

4.1 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.

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. Note that in the procedure outlined above, it is not necessary to actually compute BDDs. One can use random simulation to identify potentially equivalent nodes, and then use a SAT engine to verify equivalence and construct the equivalence classes. We have implemented a package called FRAIG (functionally reduced and-inverter graphs) that exposes the same API as a BDD package but internally uses simulation and SAT. (The details of this package are in the technical report [18]. A discussion of the issues involved can also be found in recent work on structure-based equivalence checking [10, 11, 14, 15].)

Example. Figure 4 illustrates the creation of a network with choices. Networks 1 and 2 show the subject graphs obtained from two networks that are functionally equivalent, but structurally different. The nodes x_1 and x_2 are functionally equivalent (up to complementation) in the two subject graphs. They are collected in an equivalence class in the choice network, and an arbitrary member (x_1 in this case) is used as a representative for the class in the choice network. Note that there is no choice corresponding to node o since the procedure described above detects the maximal commonality between the two networks.

Algebraic re-writing. A different way to generate choices is by iteratively applying the Λ- and Δ-transformations described by Lehman et al. [13] Given an AIG, we use the associativity of AND to locally re-write the graph (the Λ-transformation), i.e. whenever the structure AND(AND(x1, x2), x3) is seen in the AIG, it is replaced by the equivalent structures AND(AND(x1, x3), x2) and AND(x1, AND(x2, x3)). 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. Similarly, the distributivity of AND over OR (the Δ-transformation) provides another source of choices. In Section 6 we present experimental results that compare the flexibility provided by these choices with those from lossless synthesis.

Note that this leads to a new way of thinking about logic synthesis: one can use arbitrary transformations to re-write the network and create choices. The best combination of these choices is selected during mapping.

4.2 Mapping with choices

The cut-based Boolean mapping procedure of Section 3.1 can be extended naturally to handle equivalence classes of nodes. Only the cut computation step needs modification. Given a node N, let $N_\simeq$ denote the equivalence class it belongs to. Let $\Phi_\simeq(\mathcal{N})$ denote the set of cuts of the equivalence class $\mathcal{N}$. Then,

$$\Phi_\simeq(\mathcal{N}) = \bigcup_{N \in \mathcal{N}} \Phi(N)$$

where, if A and B are the two inputs of N, $\Phi(N)$ is given by

$$\{\{N\}\} \cup \{u \cup v \mid u \in \Phi_\simeq(A_\simeq),\ v \in \Phi_\simeq(B_\simeq),\ |u \cup v| \leq k\}$$

This expression for $\Phi(N)$ is a slight modification of the one in Section 3.1: The cuts of N are obtained from the cuts of the *equivalence classes* of its inputs (instead of the cuts of just its inputs). The reader should verify that in the absence of choices (which corresponds to the situation when all equivalence classes have only one node) this computation is essentially the same as the one presented in Section 3.1.

As before, the cut computation can be done in a bottom-up manner from PIs to outputs in a single pass.

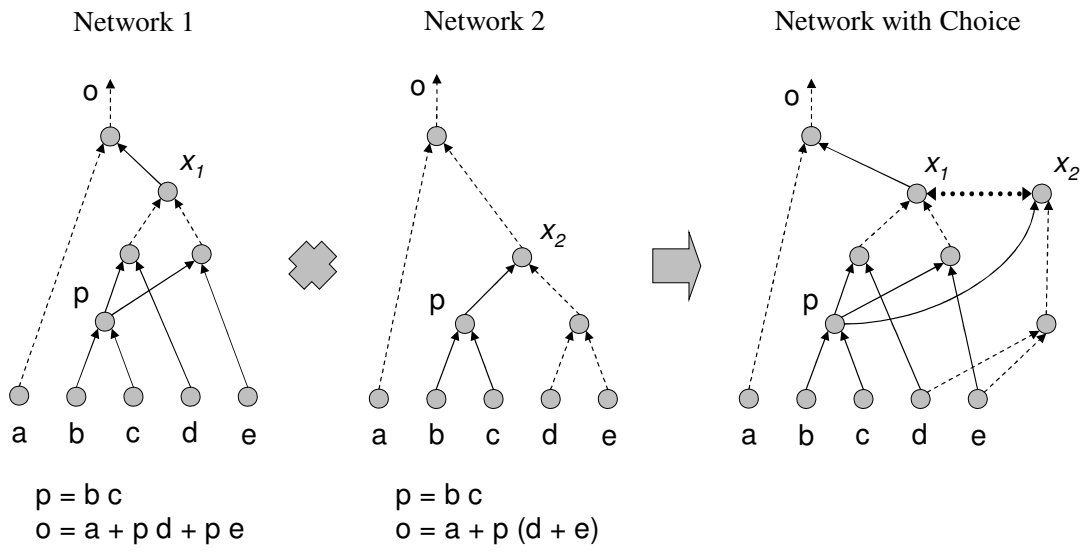


Figure 4: Example illustrating the creation of a network with choices. A choice is created when there are two nodes with the same global function (up to complementation), but different structures. Thus nodes x_1 and x_2 lead to a choice, but node p does not (p is structurally the same in both networks).

Example. Consider the computation of the 3-feasible cuts of the equivalence class $\{o\}$ in Figure 4. Let x represent the equivalence class $\{x_1, x_2\}$. Now,

$$\begin{aligned}\Phi_{\simeq}(x) &= \Phi(x_1) \cup \Phi(x_2) \\ &= \{\{x_1\},\{x_2\},\{q,r\},\{p,s\}, \\ &\quad \{q,p,e\},\{p,d,r\},\{p,d,e\},\{b,c,s\}\}\end{aligned}$$

and $\Phi_{\simeq}(\{a\}) = \Phi(a) = \{a\}$. We have $\Phi_{\simeq}(\{o\}) = \Phi(\{o\}) =$

$$\{\{o\}\} \cup \{u \cup v \mid u \in \Phi_{\simeq}(a_{\simeq}),\ v \in \Phi_{\simeq}(x_{1\simeq}),\ |u \cup v| \leq 3\}$$

Now since $a_{\simeq} = \{a\}$, and $x_{1\simeq} = x$, we get

$$\Phi_{\simeq}(\{o\}) = \{\{o\},\{a,x_1\},\{a,x_2\},\{a,q,r\},\{a,p,s\}\}$$

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 (steps 2–5 of Section 3.1) remain unchanged, except now the operations are done for equivalence classes of nodes, rather than for individual nodes.

5. SUPERGATES

A supergate is a single output combinational network of a few library gates which is treated as a single library gate by the mapping procedure. As the example in the introduction shows (see Figure 2), supergates match larger portions of the subject graph than library gates. This makes the matching more Boolean and less dependent on the structure of the subject graph. This greatly increases the number of matches seen by the mapper, and leads to better results.

In what follows we use the term *simple gates* to mean the gates in the original library.

5.1 Use in Mapping

Supergates require no change to the mapping procedure since they are no different from simple gates for the mapper. Supergates are generated in a pre-processing step as described in Section 5.2. The supergate library is generated once, stored compactly in a file, and used when technology mapping is invoked. The supergates are recomputed only if changes are made to the original library. This is why supergate generation has an additional advantage of reducing the total run-time of mapping by pre-computing and re-using the mapping information, which depends on the library but not on the network to be mapped.

After the network is mapped using supergates, each supergate in the mapped netlist is replaced by its constituent simple gates; thus the final netlist consists of only the library gates.

5.2 Supergate Generation

Supergates are generated recursively in a number of rounds. In each round we generate a new set of supergates and compute their functions. These supergates are used in subsequent rounds to generate new supergates. As usual we represent functions by truth tables.

Let k be the maximum number of inputs in the support of the supergates we consider. For example when mapping with 5-feasible cuts, k would be 5.

Let G_i be the set of supergates at round i. Let L be the set of simple library gates. Let J_k denote $\{1\,..\,k\}$, $\pi(X)$ the set of permutations of a set X, and $|g|$ the number of inputs of a simple gate $g \in L$.

The supergate generation is as follows. Initially, $G_0 = \{x_i | i \in J_k\}$ where x_i are elementary functions. Each G_{i+1} is the union of G_i and an additional set of functions of the form $g(x_{i_1}, x_{i_2}, ..\, x_{i_n})$ where $g \in G_i$ and $(i_1, i_2, ..\, i_n) \in \pi(Y)$, Y being a subset of J_k of cardinality $n = |g|$.

The total number of rounds corresponds to the maximum number of logic levels in a supergate, and is an user-specified

parameter. If the generation is stopped after the first round (i.e. at G_1), then the set of supergates contains only the library gates with all permutations of input variables (c.f. Section 3.2).

Example. Let $k = 3$, i.e. we are interested in supergates with at most 3 inputs. Let $L = \{\text{AND, OR}\}$.

$G_0 = \{x_1, x_2, x_3\}$ where the truth table of x_1 is 0101 0101, x_2 is 0011 0011 and x_3 is 0000 1111.

Each G_{i+1} contains in addition to the functions in G_i, functions of the form AND(y_1, y_2) and OR(y_1, y_2) where y_1, y_2 are functions in G_i.

Thus x_1, AND(x_1, x_2) (whose truth table is 0001 0001), AND(x_2, x_1), AND(x_1, x_3), etc. are some functions in G_1. Similarly, AND(AND(x_1, x_2), x_3), AND(OR(x_2, x_1), AND(x_1, x_3)), AND(AND(x_1, x_2), AND(x_1, x_3)), are some functions in G_2.

5.3 Pruning by Dominance

Since the above procedure is exhaustive, a large number of supergates is generated. However, some of the supergates generated above are sub-optimal, i.e. they are dominated by other gates.

Example. Consider the gate AND(AND(x_1, x_2), AND(x_1, x_3)) from the example above. It has worse delay and area than the functionally equivalent supergate AND(AND(x_1, x_2), x_3).

Whenever a new supergate is created, it is checked against existing supergates that implement the same function (by means of a hash table). If the new supergate is worse in terms of area and delay than an existing one, then it is not added to the set of supergates.

Also note that usually in industrial libraries functional symmetry of the underlying gate is not very useful. For example both AND(x_1, x_2) and AND(x_2, x_1) have to be retained since the pin-to-pin delays are different in the two cases.

5.4 Pruning by Resource Limits

Another technique to reduce the number of supergates is by pruning based on resource limits. This is important because of a combinatorial explosion inherent in the above formulation.

Example. If at one point there are 1000 supergates, and a 4-input NAND is used as a root gate, there would be 1000^4 supergates to consider. Many of these would be added (since the pin-to-pin delays are different) and the number of supergates would increase significantly, making the next round impossible to complete.

We experimented with a number of heuristics to reduce the combinatorial explosion. The simplest heuristic is to use only small support gates (say ≤ 3) as root gates. A second heuristic to set area and delay limits on the supergates. These limits can be handled efficiently by sorting the supergates and using only those supergates as inputs to the root gate such that the resulting supergate would be within the area and delay limits.

The supergate generation technique presented above is rather basic and inefficient because of the bottom-up nature of the generation process. Improving the generation process is an interesting research problem. One idea is to study the cut functions actually encountered during mapping, and then to employ constructive decomposition techniques to generate good supergates for the commonly occurring functions. This leads to the exciting possibility of a mapper that learns from the circuits it processes!

Mode	Delay	Area	Run-time
B	1.00	1.00	1.00
B+L	0.79	1.03	3.79
B+S	0.75	1.11	7.00
B+L+S	0.68	1.13	28.91

Table 1: Comparison of various mapper modes. B is Baseline, L is Lossless synthesis, S is Supergates. Run-time for B+L and B+L+S does not include the choice generation time. The time required for choice generation is roughly a factor of three of the baseline run-time.

5.5 Comparison with Algebraic Re-Writing

Recall from Section 4.1 that algebraic re-writing includes the addition of choices by adding alternative decompositions based on associative and distributive transforms in the manner of Lehman et al. [13].

If supergates are generated exhaustively without resource limits, it is possible to do exact logic synthesis (the entire circuit would map to a single supergate). In this theoretical setting, mapping with supergates is the most general form of logic synthesis. In practice, as discussed above, only a limited set of supergates can be used.

Practically, the search space explored by supergates is different from the search space explored by re-writing in two ways. First, since supergates are built by combining gates, supergates capture different *Boolean* decompositions of a function. Therefore they are more general than algebraic re-writing which by definition is limited to *algebraic* decompositions. Second, since the set of decompositions explored depends on the library, and the pruning techniques described above prune away sub-optimal decompositions, supergates explore a space of "good" decompositions relative to the gates in the library.

In summary, the decompositions due to supergates are a targeted – though sparse – sampling of the large space of Boolean decompositions. In contrast, algebraic re-writing is a dense sampling of the smaller space of algebraic decompositions.

6. EXPERIMENTAL RESULTS

The techniques described in this paper have been implemented in the MVSIS logic synthesis system [17]. The implementation is freely available. We performed a number of experiments to characterize the performance of the mapper in its various modes, and to benchmark it against other state-of-the-art mappers.

Area recovery. In the experiments reported here, delay is the parameter of interest, and area is not directly controlled during mapping. However, unnecessary duplication of logic during DAG-mapping can lead to poor area. The area of the mapped netlist is improved by making two passes after the mapping procedure described in Section 3.1. Before each pass, the required time is computed at every node. The nodes are then processed in topological order (from inputs to outputs). For nodes having positive slacks, matches that

minimize area without violating required times are chosen. The two passes differ in the metric used to measure area. The first pass uses area-flow [16], a global metric that accounts for fanout. The second pass uses exact area, a local metric that captures the area contribution due to gates in the maximal fanout-free cone of a match. (In exact area, the area cost of a match is the sum of the areas of all gates used *exclusively* by the match.) Our experiments show that the combination of these heuristics (in this order) is very effective: post sizing area is reduced by about 30% without increasing delay.

6.1 Basic Evaluation

The first two experiments were done in an academic setting, using *mcnc.genlib* and simple load independent delay model. The benchmarks chosen were the 15 largest publicly available ones (listed in Table 2), and they were preprocessed with the script shown in Figure 1(a) followed by balancing for delay. For lossless synthesis, the choices were generated using the scheme shown in Figure 1(b).

Characterizing the quality–run-time trade-off. Table 1 summarizes the relative delay, area and run-times of the mapper in its various modes. As might be expected, the fastest run-time is obtained when neither supergates nor lossless synthesis is used (this mode is called baseline), and the best quality (32% improvement in delay over baseline) is obtained when both techniques are used (the mode is called B+L+S).

In addition to these extreme cases, Table 1 also shows the intermediate situations when either lossless synthesis or supergates is used alone giving a range of quality–run-time trade-offs. We note that since the absolute run-time of the baseline mapper is very small (less than 4 seconds for the largest public benchmarks), the order of magnitude increase in run-time when using both lossless synthesis and supergates is acceptable for better quality.

Experimental comparison with algebraic re-writing. Since a direct comparison with the implementation described by Lehman et al. [13] was not possible, we performed a simple experiment to estimate the effect of algebraic re-writing. As per Section 4.1, a number of associative and distributive decompositions were added through local re-writing. Decompositions were iteratively added until the number of nodes in the AIG tripled.

Compared to baseline, algebraic re-writing led to a 9% reduction in delay (cf. the 32% reduction with B+L+S). When used in conjunction with either lossless synthesis or supergates, re-writing led to a smaller improvement in delay (about 4% in both cases). When used in conjunction with both supergates and lossless synthesis, there was an improvement of only 2% in delay. This confirms the analysis in Section 5.5 that in practice supergates and algebraic re-writing explore different search spaces.

In the subsequent experiments we use the baseline and the B+L+S modes (supergates and lossless synthesis) for comparisons with other mappers.

Comparison with SIS. Table 2 shows the performance of the mapper on the benchmark circuits in comparison with the mapper in SIS (used in delay optimal mode). In the baseline mode the mapper runs 5 times faster than the tree mapper in SIS [20] and produces 33% better delay without degrading area. In the B+L+S mode, the mapper produces the best results with 30% reduction in delay over baseline, and 54% over SIS.

6.2 Comparison with Industrial Mappers

The next set of experiments are conducted in an industrial setting. The examples are timing-critical combinational blocks extracted from a high-performance microprocessor design which were optimized for delay during technology independent synthesis. After technology mapping, buffering and sizing is done separately in accordance with a gain-based flow. As part of the mapping, an attempt is made to prefer those gates that can drive the estimated fanout loads. (This is done iteratively like area-recovery using fanout-estimation techniques similar to those used in the area-flow algorithm [16]).

Table 3 shows a comparison of the mapper with two other state-of-the art mappers: DAG mapper [12] and GraphMap which is an independent implementation of the Lehman-Watanabe mapper [13] that uses Boolean matching. Both mappers do not have area recovery. Using supergates and choices, the mapper outperforms both GraphMap and DAG mapper in delay and area and has a significantly shorter run-time.

Table 4 shows the performance of the mapper on some larger blocks from the microprocessor, in comparison with DAG mapper. Delay reduces by 12% while area (measured after sizing) reduces by 24%. Thus, with larger blocks, the improvement in area is greater. It was pointed out in [12] that DAG mapper can produce significantly faster circuits compared to the traditional tree mapping approach [8]. However, the area increase for DAG mapper sometimes can be quite significant. The significant area reduction by the new mapper makes DAG mapping approach much more practical, especially when leakage power consumption is becoming an increasingly important consideration in high-performance designs.

7. CONCLUSIONS AND FUTURE WORK

Our experiments demonstrate that Boolean mapping based on the simplified matching algorithm and optimal phase selection is a better alternative to structural matching since it produces superior results with shorter run-time. Supergates and choices fit nicely into this framework and greatly improve the quality of mapping by mitigating structural bias. Furthermore, the intermediate networks seen during technology independent synthesis are a useful source of choices for the final mapping. Supergates, though generated by brute-force enumeration, improve the quality of mapping, even with industrial libraries.

To give a balanced view of the techniques presented, we should point out their limitations. The exhaustive cut computation which works very well in baseline mode (when no choices are used) becomes a computational bottleneck when many choices are added. We have developed pruning heuristics to restrict the number of cuts considered for each node, but extensions of the techniques proposed in [4] to handle choices would be useful.

A general limitation of cut-based matching methods is that library gates with many inputs cannot be handled. In practice this is solved by using a structural matcher just for those gates during the matching phase (as is done in the IBM system [21]). We are currently exploring more seamless techniques to extend Boolean matching to gates with many inputs.

Name	Delay			Area			Run-time		
	SIS	Baseline	B+L+S	SIS	Baseline	B+L+S	SIS	Baseline	B+L+S
b14	1.90	69.70	0.45	1.44	10427.00	0.97	3.12	1.89	16.68
b15	1.57	74.50	0.51	1.38	14579.00	1.09	3.75	2.08	19.22
bigkey	1.39	11.40	0.61	1.30	5284.00	1.45	7.50	0.40	39.18
C5315	1.44	27.20	0.71	1.28	2733.00	1.05	4.00	0.35	22.69
C6288	1.96	82.60	0.66	0.98	6455.00	1.24	2.28	1.23	8.42
C7552	1.52	23.50	0.72	1.35	3393.00	1.14	3.11	0.58	17.09
clma	1.47	34.30	0.73	1.25	19968.00	1.17	5.29	2.10	55.70
clmb	1.28	36.20	0.70	1.26	19686.00	1.17	5.43	2.10	55.84
dsip	1.47	8.70	0.75	1.21	5205.00	1.26	9.03	0.31	9.68
pj1	1.68	41.00	0.58	1.31	24345.00	1.11	4.54	3.15	33.15
pj2	1.72	14.80	0.77	1.25	4957.00	1.16	7.36	0.38	36.63
pj3	1.70	28.50	0.67	1.28	15461.00	1.26	4.25	2.05	36.63
s15850	1.47	33.10	0.76	1.24	5963.00	0.97	7.33	0.45	32.76
s35932	1.52	9.00	0.84	1.30	15242.00	1.02	8.77	1.38	19.93
s38417	1.59	22.00	0.71	1.26	18057.00	0.96	5.87	1.74	30.06
Ratio	1.58	1.00	0.68	1.27	1.00	1.13	5.44	1.00	28.91

Table 2: Comparison with SIS on public benchmarks. The numbers for Baseline are absolute; those for SIS and C-S are relative to Baseline. Run-time is in seconds on a 1.6GHz Intel laptop.

Name	DAG Mapper		GraphMap		B+L+S	
	Area	Delay	Area	Delay	Area	Delay
ex1	42	124.90	49	115.18	40	89.74
ex2	51	92.64	59	76.06	55	75.03
ex3	53	92.44	61	78.03	54	72.71
ex4	177	177.89	208	131.92	171	123.45
ex5	118	162.49	156	132.92	102	129.81
ex6	103	123.02	103	101.37	88	93.16
ex7	41	56.45	47	53.42	47	53.96
ex8	41	56.45	47	53.42	47	53.96
ex9	96	146.78	154	133.96	98	111.62
ex10	102	48.11	92	44.65	105	44.55
ex11	91	74.80	85	60.16	72	60.89
ex12	239	225.11	323	189.73	205	209.11
avg	1.00	1.00	1.20	0.85	0.94	0.81

Table 3: Comparison with the other mappers on industrial benchmarks.

Name	DAG Mapper			Baseline		
	Area	Delay	Run-time	Area	Delay	Run-time
ex1	25412	171.11	406.30	18440	162.29	5.99
ex2	28550	167.27	600.10	23284	159.33	7.29
ex3	22576	89.70	283.30	17868	90.92	5.69
ex4	8500	296.64	26.80	6159	272.78	3.14
ex5	1148	252.15	99.40	601	203.52	2.66
ex6	4530	344.63	105.70	2294	272.17	3.80
avg	1.00	1.00	1.00	0.76	0.88	0.04

Table 4: Comparison on large industrial circuits.

The exhaustive nature of supergate generation (as presented in Section 5.2) is inefficient since (i) the generated functions may not correlate well with the actual cut functions in the circuits, and (ii) the same function may be generated multiple times. It would be interesting to explore methods for guided supergate generation where more computational effort is invested in finding the supergates for the frequently occurring cut functions. This suggests the possibility of a mapping procedure that learns from the previous runs how to guide supergate generation.

For our current prototype within a gain-based methodology, sizing and buffering are performed after mapping during physical synthesis. We plan to extend our mapper for use in a flow that combines logical and physical synthesis.

8. ACKNOWLEDGMENTS

The first three authors were supported in part by C2S2, the MARCO Focus Center for Circuit and System Solution, under MARCO contract 2003-CT-888; and in part by California Micro Program along with our industrial sponsors Cadence, Synplicity, Intel, Magma, and Fujitsu.

9. REFERENCES

[1] L. Benini, G. DeMicheli, "A survey of Boolean matching techniques for library binding," ACM TODAES, Vol. 2, No. 3, July 1997, pp. 193-226.

[2] C.-W. Chang and M. Marek-Sadowska, "Who are the alternative wires in your neighborhood? (Alternative wire identification without search)," Proc. GLSVLSI '01, pp. 103-108.

[3] 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), 1994, pp. 1-12.

[4] J. Cong, C. Wu and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," Proc. FPGA '99.

[5] D. Debnath and T. Sasao, "Fast Boolean matching under variable permutation using representative," Proc. ASP-DAC '99, pp. 359-362.

[6] S. Hassoun and T. Sasao, eds., Logic synthesis and verification, Kluwer 2002, Chapter 5, "Technology mapping," pp. 115-140.

[7] U. Hinsberger and R. Kolla, "Boolean matching for large libraries," Proc. DAC '98, pp 206-211.

[8] K. Keutzer, "DAGON: Technology binding and local optimizations by DAG matching," Proc. DAC '87, pp.617-623.

[9] V. N. Kravets and K. A. Sakallah, "Constructive library-aware synthesis using symmetries," Proc. DATE '00, pp. 208-216.

[10] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," Proc. ICCAD '04.

[11] 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.

[12] Y. Kukimoto, R. K. Brayton, P. Sawkar, "Delay-optimal technology mapping by DAG covering," Proc. DAC '98, pp. 348-351.

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

[14] F. Lu, L. Wang, K. Cheng, R. Huang. "A circuit SAT solver with signal correlation guided learning," Proc. DATE '03, pp. 892-897.

[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. Manohararajah, S. D. Brown, Z. G. Vranesic, "Heuristics for area minimization in LUT-based FPGA technology mapping," Proc. IWLS 04.

[17] MVSIS Group. MVSIS: Multi-Valued Logic Synthesis System. U. C. Berkeley. http://www-cad.eecs.berkeley.edu/Research/mvsis/

[18] A. Mishchenko, S. Chatterjee and R. Brayton, "Fraigs: Integrated verification and synthesis," Tech. Report., Dept. of EECS, U. C. Berkeley, 2004. http://www-cad.eecs.berkeley.edu/Research/mvsis/

[19] A. Mishchenko, X. Wang, T. Kam, "A new enhanced constructive decomposition and mapping algorithm," Proc. DAC '03, pp. 143-147.

[20] E. Sentovich, et al. "SIS: A system for sequential circuit synthesis," Tech. Rep. UCB/ERI, M92/41, ERL, Dept. of EECS, U. C. Berkeley, 1992.

[21] L. Stok, M. A. Iyer, A. J. Sullivan, "Wavefront technology mapping," Proc. DATE '99. pp. 531-536.

[22] Wu et al., "Efficient Boolean matching algorithm for cell libraries," Proc. ICCD '94, pp 36-39.