Technology Mapping with Boolean Matching, Supergates and Choices

Alan Mishchenko Satrajit Chatterjee Robert Brayton

Department of EECS
University of California, Berkeley
{alanmi, satrajit, brayton}@eecs.berkeley.edu

Xinning Wang Timothy Kam

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

Abstract

As an important step of any design flow, technology mapping, expresses logic functions of a netlist using gates from a technology library, in the presence of various design constraints. This paper proposes a new approach to technology mapping, which relies and enhances upon several known techniques, integrated and fine tuned to work in a new way. The previous work on DAG mapping is extended, by proposing new methods for enumerating mapping choices and performing Boolean matching, which guarantees delay-optimum phase assignment at the gate boundaries. Two ways of capturing flexibility in technology mapping are explored and compared: supergates and choice nodes. 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 and Previous Work

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 so that some objective function, such as total area or delay, is optimized. In these general terms, technology mapping is intractable. However, 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 multi-level network of library gates by means of local substitutions. Unfortunately, this simplification means that the structure of the subject graph dictates to a large extent the structure of the mapped network. This is commonly known as the problem of structural bias. Sometimes such a mapper could negate benefits gained during the technology independent decomposition phase.

The problem of structural bias is tackled by on-going research. Research literature on technology mapping provides a spectrum of techniques that trade-off 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 constrainted by the given subject graph. Existing Boolean approaches (e.g. [10], [16]) 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. This limitation 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. This leads to less dependence on the original subject graph, but this method suffers from longer run-times. Wavefront mapping is a practical enhancement of this approach [19].

Our current effort was inspired observation that cut-based techniques found in technology mapping for FPGA look-up tables can be adapted to work for standard cell libraries using Boolean matching. Furthermore, this Boolean matching technique is faster than the classical structural matching based on subgraph-isomorphism. It is also superior in terms of mapping quality, since in order to have fast run-times, structural matchers often do not exhaustively try all applicable matches. This is especially true for industrial libraries with large and complex gates. This matching technique is described in Section 2 of this paper.

The runtime advantage afforded by our matching technique can be used to address the structural bias problem using two complementary approaches. In the first approach, which is applied as a preprocessing step before mapping, supergates are constructed out of library gates [16]. A supergate is a single-output combinational network of a few gates which is treated as a single library gate by our algorithms. This allows the matching process to be less local and look deeper into the circuit for matching alternatives, by matching larger portions of the circuit at one time. In contrast, traditional technology mappers accept a biased subject graph and try mapping many small gates onto it greedily following its biased multi-level structure. Supergate generation and usage is described in Section 3.

The second approach is based on explicitly storing decomposition choices in the subject graph [13]. Technology mapping over such a graph can be thought of as running multiple traditional mappings, one subject graph per choice combination, and then selecting the best overall mapping. These choices may be obtained in a number of ways, such as local rewriting of the subject graph or by combining intermediate networks. As we shall see in Section 4, choice nodes can be incorporated naturally in the cut-based Boolean matching methodology.

In summary, this paper makes three main contributions: The first is a new locally-Boolean matching framework for ASICs which combines exhaustive cut function enumeration with Boolean matching ensuring optimal phase assignment. The algorithm was developed independently, though related to ideas in [22] and [23]. Second, we show how this framework can naturally accomodate supergates without any change to the core algorithm using library pre-processing. Third, we extend the matching algorithm to include choices in the mapping graph, and present a new way of generating choices leveraging ideas from combinational equivalence checking.

A prototype of these ideas has been implemented in the MVSIS logic synthesis environment, and its experimental results (in Section 5) show significant improvements in both quality and run-time over other state-of-the-art mappers.

2 New Mapping Flow

In this section we present the basic working of the mapper first without considering supergates and choice nodes.

2.1 Overview

Although the techniques presented in this paper can be applied to a variety of mapping problems, for easy of exposition we focus on a specific mapping task. Our objective is to minimize the delay of the longest path in the mapped netlist. We assume a load-independent delay model for the gates in accordance with a gain-based logic synthesis flow. As shown by Kukimoto et al. [12], this problem can be optimally solved using dynamic programming for DAG-covering. The key difference of our method with the conventional approaches based on DAG-covering lies in the matching step: we use Boolean instead of structural matching.

Our algorithm begins with the subject graph represented as an And-Inverter Graph (AIG). An AIG is a DAG whose nodes represent either And gates or primary inputs. Its edges represent wires. Inverters are represented by bubbles on the edges.

Example. Figure 1 shows an AIG. We shall use this as a running example to illustrate the mapping process.

The mapping is broken in to 5 steps. First, for every node, we compute all of its *k-feasible cuts* (defined in Section 2.2). Second, for every cut, we assign a formal variable to each node in the cut and compute the function of the corresponding node in terms of these variables. (The function is computed as a bit-vector representing the truth-table.) Third, these functions are looked up in a hash table and matched with gates from the library. Fourth, in topological order, starting with arrival times of the primary inputs, the best arrival time for each node is computed by choosing the library gate with minimum delay. Finally, in reverse topological order, the best covering is chosen. (The last two steps are exactly the same as in traditional mapping.) As a post-processing step, area is recovered on the non-critical paths using the technique presented by Manohara-rajah et al. [14].

2.2 Computing 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 $\{N\}$ composed of node N alone is always a k-feasible cut of node N (for any k) and is called the *trivial cut*.

Example. In the AIG of Figure 1, $\{n_2\}$, $\{n_4, n_5\}$, $\{n_4, x_2, x_3\}$ $\{n_5, x_1, x_2\}$, $\{x_1, x_2, x_3\}$ are all the 3-feasible cuts of n_2.

We compute all k-feasible cuts of every node in the network by the simple bottom-up traversal algorithm shown in Figure 2. Although in general a graph may have exponentially many cuts, most test-cases have between 20 and 30 5-feasible cuts per node.

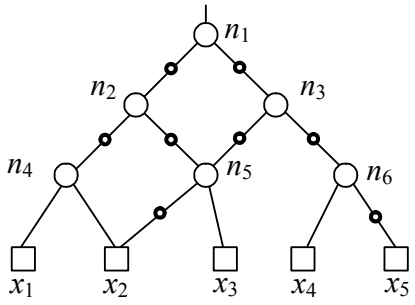


Figure 1: An example of an AIG.

Computation of all k-feasible cuts for all nodes in the network is performed in one pass over the nodes as shown in Figure 2. The cut set of the node is derived by merging the cut sets of the fanins. The trivial cut composed of the node itself, $\{n\}$, is added to the resulting set. Procedure *MergeCutSets* initializes the resulting cut set to be empty, and considers all pairs of cuts from the two sets. For each pair, the merged cut is found as the union of the nodes belonging to the generating cuts. If the size of the merged cut does not exceed k, and the cut is encountered for the first time, it is added to the resulting cut set.

Example. The cut set of node n_4 is $\{\{x_1,x_2\},\{n_4\}\}$. The cut set of node n_5 is $\{\{x_2,x_3\},\{n_5\}\}$. Suppose we compute 2-feasible cuts of node n_2 using procedure *MergeCutSets.* The four cut pairs yield the following cuts: $\{\{x_1,x_2,x_3\}$, $\{x_1,x_2,n_5\}$, $\{x_2,x_3,n_4\}$, $\{n_4, n_5\}\}$. Only one cut is two-feasible. So the resulting cut set is $\{\{n_4, n_5\}\}$.

```
void NetworkKFeasibleCuts( Graph g, int k ) {
    for each primary output node n of g
        NodeKFeasibleCuts( n, k )
}
cutset NodeKFeasibleCuts( Node n, int k ) {
    if ( n is primary input ) return { { n } }
    if ( n is visited ) return NodeReadCutSet( n )
    mark n as visited
    cutset Set1 = NodeKFeasibleCuts( NodeReadChild1( n ), k )
    cutset Set2 = NodeKFeasibleCuts( NodeReadChild2( n ), k )
    cutset Result = MergeCutSets( Set1, Set2, k ) ∪ { n }
    NodeWriteCutSet( n, Result )
    return Result
}
cutset MergeCutSets ( cutset Set1, cutset Set2, int k ) {
    cutset Result = { }
    for each cut Cut1 in Set1
        for each cut Cut2 in Set2
            if ( | Cut1 ∪ Cut2 | ≤ k ) then Result = Result ∪ { Cut1 ∪ Cut2 }
    return Result
}
```

Figure 2. Computation of all *k*-feasible cuts.

2.3 Computing Cut Functions

The next step is to compute the local function of a node in terms of its cut. This is done for every k-feasible cut, other than the trivial, 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. Using these variables, the functionality of the node is computed symbolically. Since usually only 5- or 6-feasible cuts are considered, this symbolic function computation can be performed efficiently using 32- or 64-bit integers to represent the truth-tables. In what follows we use the words 'function' and 'truth-table' interchangeably.

Example. In the AIG of Figure 1, consider the cut $\{n_4, n_5, n_3\}$ of node n_1 viewed as a 4-feasible cut. We assign formal variables to each node in the cut. Thus the functions at the leaves are the elementary functions: n_4 = 0101 0101 0101 0101, n_5 = 0011 0011 0011 0011 and n_3 = 0000 1111 0000 1111. (Since this is only a 3-cut, the fourth elementary function is not used.) Using these functions, by bit-parallel simulation we compute the function of node n_1 as 0111 0000 0111 0000 in terms of the leaf nodes.

2.4 Boolean Matching

In the next step, an appropriate library gate is chosen to implement the cuts. However, for this step the library has to be pre-processed. For each gate in the library, we compute its truth table in terms of the variables assigned to the inputs of the gate. Since, in general, a gate may be asymmetric in its inputs, it is necessary to compute the truth tables for all possible assignments of formal variables to the inputs. Similar to the cut functions,

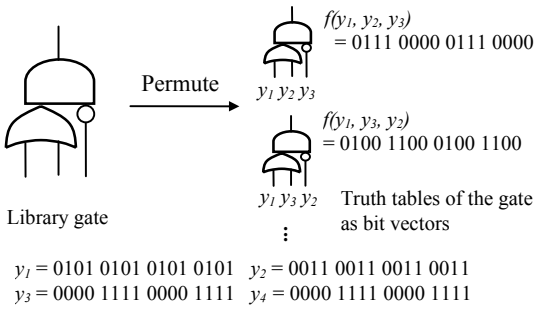


Figure 3: Example of library pre-processing.

these truth tables are computed as 32- or 64-bit integers. As the final step of the pre-processing, a hash-table is created to maps a truth table into the library gates implementing that function.

Example. See Figure 2. The different truth tables generated using different input permuations for the library gate Or-And are added to the hash table during pre-processing.

During the matching, the truth table for a cut is looked up in the hash-table. This gives a list of library gates that may be used to implement the cut. It is possible that no library gate implements the functionality of the cut. However, since *every k*-feasible cut of a node is computed, in particular, the 2-feasible cut of the node is also computed. Therefore, as long as library contains an AND-gate or a NAND-gate, at least one cut of every node can be implemented, and a feasible mapping can be found.

Example. In our example, for the cut $\{n_4, n_5, n_3\}$ of node n_1 the function is 0111 0000 0111 0000. Using this to index the hash table, we obtain the library gate Or-And (Figure 2) as a possible implementation for this cut.

Optimal Phase Assignment. The mapping quality can be improved by exploiting the additional flexibility of computing the mapping choices for each cut in both polarities: positive (as above) and negative. When the final mapping is selected, the appropriate polarity is choices to guarantee the shortest delay on each path. This can lead to area duplication because a node can be used in both negative and positive polarities on two different paths. The following example demonstrates this situation.

Example. Consider a subgraph of an AIG with three AND-nodes and two outputs: o_1 = AND(a, x), o_2 = AND(x', b), x = AND(c, d) . Node x is required in the positive polarity on the path to o_1 and in the negative polarity on the path to o_2. Suppose the library is composed of INV (delay 0.9), NAND2 (delay 1.0) and AND2 (delay 1.2). If x is mapped in only one polarity, the other polarity has to be produced using an inverter. However, adding an inverter with delay 0.9 creates longer delay than using the opposite polarity mapped directly into AND2 and NAND2.

Observe that this "dual-rail" mapping also means that the inputs of a cut are available in either polarity. Consequently, the function of a cut is not precisely defined: it belongs to a class of functions, which differ (only) by complementation of inputs. This class is called the *N-equivalence* class. This leads to greater flexibility since the cut can be implemented by any function in this class.

Boolean matching is extended to match *N*-equivalent classes. During library pre-processing, after computing the truth table of a function, the truth table of the representative of its *N*-canonical class is computed. Similarly, during matching, the *N*-representative of the cut is used to look up the hash table.

2.5 Computing Best Matches

In topological order, starting with the inputs, the best matches and their corresponding arrival times are selected for both phases of each cut. For every node, we do the following. For each cut of the node, we look at the list of library gates implementing that cut. From these, we pick the best gate to implement the node in positive and in negative polarity. (The best gate is the gate that leads to the earliest arrival time for the node.) Next across all the cuts of the node, the best gate for both polarities is selected. When this is done, the node is considered mapped.

The arrival times can be accurately computed since we are proceeding in the topological order: so all nodes appearing in the cuts of this node have already been mapped. The only complication is due to the fact that inverters may need to be inserted at the outputs of some cuts, if one of the phases of the node is implemented using another phase.

2.6 Choosing the Best Cover

This final step is done in the usual manner. In the reverse topological order, the best gate for each primary output (in the positive polarity) is chosen. Next, the best gates implementing the inputs of these gates are chosen and so on until PIs are reached.

3 Supergates

3.1 Definition and Motivation

A supergate is a single-output combinational network of a small number of library gates. This network is treated as a single gate by the mapping process described in Section 2. The use of these large gates addresses the structural bias problem as shown in the following example.

Example. Figure 4 shows a supergate *S* composed of an Or gate and an And gate. In Figure 1, consider the cut $\{n_4, n_5, n_6\}$ of node n_1. Observe that the function of n_1 in terms of the cut nodes is $(n_4 + n_5)(n_5 + n_6)$ i.e. $n_4n_6 + n_5$. Thus this cut can be directly implemented by the supergate *S*. However, if we restrict ourselves to the library gates, we would never consider this implementation, since there is no node with the function n_4n_6 in the subject graph.

As the above example shows, when supergates are used the mapping process looks deeper into the network and can find better matches because of overcoming the structural bias**.**

For a library of gates, the supergate library is generated as a preprocessing step before mapping. The supergate generation is guarded by constraints and resource limits, such as the limit on the number of inputs, the limit on the number of levels, the limit on the total area and delay, and the runtime limit. The generation process is described in the following subsection.

It is important that 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 the supergate generation has an additional advantage of reducing the total runtime of mapping by pre-computing and re-using the mapping information, which depends on the library but not on the netlist to be mapped.

A supergate library can be viewed as an elementary gate library, which contains many gates with diverse functionality. A library of elementary gates can be viewed as a supergate library with supergates composed of one level of the elementary gates. The proposed algorithms work uniformly for both types of libraries.

3.2 Generation

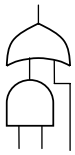
The supergate generation is performed recursively. One recursive step adds one level of gates on top of the available

supergates. To avoid confusion, we use the term *elementary gates* to mean the actual gates in the library.

At the beginning, the set of available supergates is the set of elementary variables. In the current implementation, their number may be up to six; it depends on the largest allowed support size of the generated supergate library. In each recursion step, all possible elementary gates are used as *root* gates, and the supergates already generated (at lower levels of recursion) are plugged into these root gates to create new supergate candidates.

Example. The process generation process is illustrated in Figure 5. The root gate AOI21 is added on top of three supergates: S_1 (the elementary variable a), S_2 (the supergate composed of two NAND2 gates), and S_3 (the OR2 gate).

The resulting candidates have an additional level of elementary gates, compared to the starting set of supergates. Before accepting a candidate supergate, its truth table is computed and the hash table is checked for a supergate with the same functionality but better delay-area parameters. If such gate exists, the new supergate is discarded. If it does not exist, the new supergate is added to the set of supergates used to generate the next level.



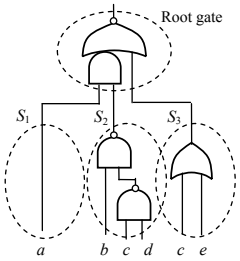


Figure 4: A supergate. Figure 5: Supergate generation.

Constraints. The runtime of supergate generation can be dramatically reduced by applying constraints on the candidates. The constraints on runtime of generation and on the number of logic levels of gates are obvious. Two other types of constraints are trickier to implement. These constraints include restrictions on the maximum area and on the maximum pin-to-pin delay of the resulting supergates.

To ensure that a candidate with a pin-to-pin delay exceeding the given limit is never created, the available supergates are sorted by their maximum pin-to-pin delay. Now, when we consider a root gate with some value of the maximum pin-to-pin delay, we only try the supergates, whose delay, when added to the delay of the root gate, does not exceed the delay limit.

Example. Consider the supergate candidate in Figure 5. Suppose AOI21 has delay 1.7, NAND2 has delay 1.0, and OR2 has delay 1.5. Suppose the global delay limit imposed on the generated supergates is 4.0. In this case, the largest allowed maximum delay for the component supergates is equal to the global delay limit minus the delay of the root gate (4.0 – 1.7 = 2.3). The supergate composed of two NANDs has delay 2.0. Both this supergate and OR2 have delay less than 2.3. Therefore, the supergate candidate in Figure 5 will be considered. Suppose the delay of NAND2 is 1.2. In this case, the supergate composed of two NANDs will have delay 2.4, which is larger than 2.3. Therefore, this supergate will not be allowed as a component when generating supergates with the root gate AOI21.

A similar restriction can be developed for area. For each root gate, the available supergates are sorted by area. Now, while adding available supergates to the supergate under construction, the maximum area limit is checked. If the limit is exceeded, the supergate under construction is dropped and another one is tried.

Storage. From the generation process it is seen that the supergates have a natural recursive structure: A supergate comprises an elementary gate and its fanins which are other (smaller) supergates. Therefore the set of supergates can be compactly represented as a DAG, whose leaves are the elementary variables. This representation is quite compact and can be used to quickly read and write large supergate libraries.

4 Choice Nodes and Lossless Logic Synthesis

Choice nodes provide an efficient way of encoding multiple decompositions of a network. We start with an arbitrary Boolean network, which contains multiple functionally equivalent points. When constructing the AIG from the network, we identify, on-the-fly, all functionally equivalent nodes in the network and collect them into equivalence classes of AIG nodes.

We note here that recent advances in combinational equivalence checking allow very efficient detection of functionally equivalent points by combining simulation and satisfiability testing. These techniques enable us to detect equivalent points in large industrial circuits, without constructing global BDDs. We refer the reader to [hidden] for details on the efficient implementation of these ideas for use in logic synthesis.

4.1 Local Rewriting of the AIG

One 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. If this process is done iteratively, it is equivalent to identifying the maximal multi-input And-gates in the AIG and to add all possible tree-decompositions. Similarly, the distributivity of And over Or provides another source of choices.

4.2 Lossless Logic Synthesis

In contrast to the re-writing procedure described above which generates local choices, lossless logic synthesis generates "global" choices. In conventional synthesis, the initial network is processed using a script which specifies a series of transformations. Only the network obtained at the end of this process is used for technology mapping; the intermediate networks are discarded. However, since the transformations are heuristic, there is no guarantee that the final network obtained in this process is the one with best delay.

In lossless synthesis, some subset of the networks seen during synthesis is collected into one AIG. This AIG therefore has large classes of functionally equivalent nodes, and during mapping the best representative of each class is automatically selected. Thus lossless logic synthesis allows the "best" parts of each network to be used to synthesize the mapped result.

4.3 Mapping with Choices

The Boolean matching algorithm can be naturally extended to handle choice nodes, by modifying the *k*-feasible cut computation and the cut function computation. The cut computation is lifted to work on the (functional) equivalence classes of And nodes. The cut set of an equivalence class is the union of the cut set of each member node. The algorithm in Figure 2 can be used to compute the cut set of each member node with the proviso that the equivalence classes of the children be used (instead of the children themselves) when computing *Set1* and *Set2*.

If only the above extension is implemented, there is inefficiency in the cut function computation, since a cut could originate from any member in the equivalence class of the node. This inefficiency can be avoided by storing the pointers to the two "parents" of a cut. (The two parents are the two cuts merged in

MergeCutSets procedure in Figure 2.) The resultant DAG of cuts can be traversed quickly for the cut function computation.

4.4 Comparison of Supergates and Choice Nodes

Both supergates and choice nodes help increase the search space and overcome the structural bias. Our experiments indicate that supergates and choice nodes are the sources of flexibility that is largely orthogonal. Supergates are limited, by construction, to several logic levels of gates. As a result, the extended search space due to supergates is relatively "shallow". However, the population of this space is "dense" because the generation process is exhaustive for the given number of logic levels.

On the other hand, the choice nodes derived from different versions of the netlist lead to a search space extension, which is "deep" and "sparse". This is because the structural differences between the netlists used to generate the choices may encompass several logic levels. Meanwhile, the number of the structural differences is relatively small because their enumeration is not exhaustive and is controlled by external resource limits.

5 Experimental results

The techniques described in this paper have been implemented in the MVSIS logic synthesis system. The implementation is freely available. We performed a large 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. In this paper, we present only four sets of data due to space limitations.

Table 1 shows the performance of the mapper on some large circuits from the publicly available benchmark suites using the MCNC library. The benchmarks are optimized with script.rugged followed by balancing during technology independent synthesis. The mapper has the fastest run-time in the *baseline* mode (no choices, no supergates) where it runs 5 times faster than the tree mapper in SIS and produces 33% better delay without degrading area. The best delay (29% over baseline, 54% over SIS) is seen when choices generated by lossless synthesis (using intermediate networks generated by script.rugged) are used with supergates.

Table 2: Comparison of various mapper modes.

Mode	Delay	Area	Run-time
B (Baseline)	1.00	1.00	1.00
L (Local rewriting)	0.91	0.96	39.45
S (Supergates)	0.84	1.13	20.26
C (Lossless)	0.79	0.97	12.02
L-S	0.80	1.07	> 200
L-C	0.75	1.09	90.08
S-C	0.71	1.11	87.16
L-S-C	0.69	1.22	> 200

Table 2 shows the results (averaged over a set of MCNC benchmarks) of an experiment to quantify the relative benefits of using supergates verses local re-writing and lossless synthesis. If only one of the techniques is used, lossless synthesis produces the best results (21% better delay than baseline) at a 12x increase in run-time. When two techniques are used, the combination of supergates and lossless choices produces the best results (29% improvement in delay) with about 87X increase in run-time. Note that since the baseline mapper is extremely fast (Table 1), even a 87X increase in run-time is still practical.

Table 4: Comparison on large industrial circuits.

Design	DAG Mapper			Baseline		
	area	delay	time	area	delay	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	6010	-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 3 shows a comparison of the mapper with two other state-of-the art mappers: the DAG mapper [12] and GraphMap which is an independent implementation of [13]. The comparison was done on a set of timing critical combinational blocks extracted from a high-performance microprocessor design using an industrial library. For both mappers, we first apply *script.rugged* followed by *speed_up* to obtain a good multi-level logic structure. It outperforms both Graph Map and DAG mapper in delay and area and has a significantly shorter runtime.

Finally, **Table 4** shows the performance of the mapper on some larger blocks from a microprocessor, in comparison with DAG mapper. Delay reduces by 12% while area reduces by 24%. Thus, with larger blocks, the improvement in area is greater. It was pointed in [12] that DAG mapper can produce significantly faster circuits compared to traditional tree mapping approach [18]. 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 power consumption is becoming an increasingly important consideration in high-performance designs.

6 Conclusions and Future Work

Our experiments demonstrate that the Boolean matching technique with optimal phase assignment is a better alternative to the graph 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.

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 [22] to handle choices would be useful.

The exhaustive nature of supergate generation (as presented in Section 3.2) 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 intriguing 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.

Acknowledgement

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, Fujitsu, Intel, Magma, and Synplicity.

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] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, “Multilevel logic synthesis”, Proc. IEEE, Vol. 78, Feb.1990.

[3] R. K. Brayton and C. McMullen, “The decomposition and factorization of Boolean expressions,” Proc. ISCAS ‘82, pp. 29-54.

[4] S. Hassoun and T. Sasao, eds., *Logic synthesis and verification*, Kluwer 2002, Chapter 5, “Technology mapping”, pp. 115-140.

[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, No. 1 (January 1994), pp. 1-12.

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

[7] D.-J. Jongeneel, R. Otten, Y. Watanabe, R. K. Brayton, “Area and search space control for technology mapping,” DAC ’00, pp. 86-91.

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

[9] V. N. Kravets. *Constructive multi-level synthesis by way of functional properties.* Ph. D. Thesis, University of Michigan, 2001.

[10] V. N. Kravets and K. A. Sakallah, “Constructive library-aware synthesis using symmetries,” Proc. DATE ’00, pp. 208-216.

[11] V. N. Kravets and P. Kudva, “Implicit enumeration of structural changes in circuit optimization”, Proc. DAC ’04, pp. ???.

[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] V. Manohararajah, S. D. Brown, Z. G. Vranesic, “Heuristics for area minimization in LUT-based FPGA technology mapping”, IWLS ’04.

[15] A. Mishchenko, B. Steinbach, M. Perkowski, "An algorithm for bi-decomposition of logic functions," Proc. DAC '01, pp. 103-108.

[16] A. Mishchenko, X. Wang, T. Kam, “A new enhanced constructive decomposition and mapping algorithm”, Proc. DAC 2003, Los Angeles, pp. 143-147.

[17] Hidden for blind review.

[18] E. Sentovich, et al. “SIS: A system for sequential circuit synthesis”, Tech. Rep. UCB/ERI, M92/41, ERL, Dept. of EECS, Univ. of California, Berkeley, 1992.

[19] L. Stok, M. A. Iyer, A. J. Sullivan, “Wavefront technology mapping”, Proc. DATE ’99. pp. 531-536.

[20] H. Touati, “Performance-oriented technology mapping”, Ph.D. dissertation, UC Berkeley, November 1990.

[21] C. Yang and M. Ciesielski, “BDS: A BDD-based logic optimization system”. IEEE Trans. CAD, Vol. 21 (7), July 2002, pp. 866-876.

[22] J. Cong, C. Wu and Y. Ding, “Cut Ranking and Pruning: Enabling A General And Efficient FPGA Mapping Solution,” In FPGA `99.

[23] U. Hinsberger and R. Kolla, “Boolean matching for large libraries,” Proc. DAC `98, pp 206-211.

Table 1: Comparison with SIS on public benchmarks.

Name	Delay			Area			Runtime		
	SIS	Baseline	C-S	SIS	Baseline	C-S	SIS	Baseline	C-S
C5315	1.51	26	0.76	1.02	3423	1.00	4.67	0.3	104.17
C7552	1.45	24.6	0.69	0.98	4668	1.00	3.16	0.57	62.77
dsip	1.39	9.2	0.75	1.26	5010	1.53	6.83	0.41	13.20
pj2	1.71	14.9	0.77	1.20	5151	1.15	4.44	0.63	55.52
bigkey	1.39	11.4	0.64	1.12	6149	1.26	8.11	0.37	88.32
s15850	1.48	32.9	0.80	1.12	6617	1.02	8.25	0.4	74.03
C6288	1.96	82.6	0.67	0.74	8590	1.21	2.92	0.96	79.60
b14	1.90	69.8	0.49	1.09	13775	0.83	3.64	1.62	109.57
b15	1.58	74.2	0.53	1.14	17646	1.01	4.97	1.57	110.13
s35932	1.49	9.2	0.83	1.12	17713	0.98	10.00	1.21	50.60
pj3	1.71	28.3	0.70	1.06	18700	1.15	3.40	2.56	134.39
s38417	1.58	22.2	0.71	1.09	20904	0.96	4.27	2.39	65.38
clmb	1.35	34.2	0.81	1.04	23886	1.15	4.51	2.53	135.92
clma	1.38	36.6	0.76	1.04	23937	1.11	6.65	1.67	337.02
pj1	1.68	41.1	0.62	1.10	29088	1.06	5.00	2.86	153.69
Ratio	1.57	1.00	0.70	1.07	1.00	1.10	5.39	1.00	104.95

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

Design	DAG Mapper		GraphMap		C-S	
	area	delay	area	delay	area	delay
ex1	42.00	-124.90	49.00	-115.18	40.00	-89.74
ex2	51.00	-92.64	59.00	-76.06	55.00	-75.03
ex3	53.00	-92.44	61.00	-78.03	54.00	-72.71
ex4	177.00	-177.89	208.00	-131.92	171.00	-123.45
ex5	118.00	-162.49	156.00	-132.92	102.00	-129.81
ex6	103.00	-123.02	103.00	-101.37	88.00	-93.16
ex7	41.00	-56.45	47.00	-53.42	47.00	-53.96
ex8	41.00	-56.45	47.00	-53.42	47.00	-53.96
ex9	96.00	-146.78	154.00	-133.96	98.00	-111.62
ex10	102.00	-48.11	92.00	-44.65	105.00	-44.55
ex11	91.00	-74.80	85.00	-60.16	72.00	-60.89
ex12	239.00	-225.11	323.00	-189.73	205.00	-209.11
Average	1.00	1.00	1.20	0.85	0.94	0.81