On Algorithms for Technology Mapping

Satrajit Chatterjee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-100
http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-100.html

August 16, 2007

On Algorithms for Technology Mapping

by

Satrajit Chatterjee

B.Tech. (Indian Institute of Technology Bombay) 2001
M.Tech. (Indian Institute of Technology Bombay) 2001

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Robert Brayton, Chair
Professor Andreas Kuehlmann
Professor Kurt Keutzer
Professor Zuo-Jun Shen

Fall 2007

The dissertation of Satrajit Chatterjee is approved:

Professor Robert Brayton, Chair — Date

Professor Andreas Kuehlmann — Date

Professor Kurt Keutzer — Date

Professor Zuo-Jun Shen — Date

University of California, Berkeley

Fall 2007

On Algorithms for Technology Mapping

by

Satrajit Chatterjee

Abstract

On Algorithms for Technology Mapping

by

Satrajit Chatterjee

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Robert Brayton, Chair

The task of technology mapping in logic synthesis is to express a given Boolean network as a network of gates chosen from a given library with the goal of optimizing 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 network 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 enumerating the different library gates that match at each node in the subject graph (the *matching* step) and selecting the best subset of matches (the *selection* step). However, this simplification means that the structure of the initial network dictates to a large extent the structure of the mapped network; this is known as structural bias. In this work we improve the quality and run-time of technology mapping by introducing new methods and improving existing methods for mitigating structural bias. To this end we propose some new algorithms addressing both matching and selection. Our methods are useful for mapping to standard cell libraries and to lookup table-based FPGAs.

We start with matching. We present a scalable and robust algorithm (based on recent advances in combinational equivalence checking) to combine multiple networks

into a single subject graph with choices. This enables *lossless synthesis* where different snapshots taken during synthesis can be combined into one subject graph on which mapping is done. We improve on existing Boolean matching methods with a simpler and faster algorithm (based on improved cut computation and avoiding canonization) that finds matches across all networks encoded in the subject graph. We also introduce a new construct called a *supergate* that increases the number of matches found for standard cell mapping.

Having increased the set of matches found in the matching step, we turn to selection. For some cost functions such as delay under the constant-delay model, optimal selection is easy, but for an area cost function, optimal selection is NP-hard. We present a simple heuristic procedure that works well in practice even with the larger search space due to the increased set of matches with choices, and it outperforms the methods proposed in the literature. Although we focus on area for concreteness, the same procedure can be used to optimize for power since the two objectives are very similar.

Professor Robert Brayton, Chair Date

Acknowledgments

I thank Prof. Robert Brayton for always leaving open the door to his office. On numerous occasions he listened to my inchoate thoughts with (near) infinite patience, always willing to entertain even the most wild ideas. I hope that some of Bob's open-mindedness and clear thinking has rubbed off on me. I am grateful for his unwavering support and encouragement over the last five years.

I thank Alan Mishchenko for the long discussions we've had over the years on topics ranging from the mundane (saving a pointer per node in an And-Inverter Graph) to the sublime (the meaning of life). Alan's creativity and ingenuity played no small part in the work leading to this thesis, and to him must go a lion's share of the credit.

I thank Prof. Andreas Kuehlmann for his spirited teaching of EECS 219B in Spring '02 which was my introduction to logic synthesis and verification. In the years that followed he has always been an important sounding board for our ideas, and I am grateful for his constructive criticism.

I am grateful to Prof. Kurt Keutzer for reading this thesis, but perhaps even more, for bravely teaching "The Business of Software" to a motley group of engineers (who didn't "know the difference between sales and marketing") and MBAs (who presumably did).

I thank Prof. Max Shen for surrendering some of his time to read this thesis and for his interest in my research.

During the course of this research, I had the opportunity to inflict new technology mapping algorithms on real-life designs, a.k.a. industrial benchmarks. This was made possible in part by three internships, two of which were at Intel's Strategic CAD Labs in Hillsboro, Oregon and one at Magma Design Automation in San Jose, California.

At Intel, I thank Xinning Wang, Steve Burns, Mike Kishinevsky, Timothy Kam, Balaji Velikandanathan, and Brian Moore. I am grateful to Xinning and Steve in particular for educating me in the challenges of high performance (and "low" power) microprocessor design, and to Brian for navigating the Intel bureaucracy on my behalf. At Magma, I thank Mandar Waghmode, William Jiang, Samit Chaudhuri, Arvind Srinivasan, Premal Buch and Peichen Pan. I am particulary grateful to Mandar for taking time off to show me the ropes, and I wish him luck maintaining my code!

I thank our other industrial collaborators who supported this research: Actel (Eric Sather), Altera (Mike Hutton), IBM (Victor Kravets, Prabhakar Kudva and Jason Baumgartner), Synplicity (Ken McElvain) and Xilinx (Stephen Jang). Although I have mentioned only a few by name, I am grateful for the feedback we received from the many talented engineers and researchers at these companies.

I thank the past and present students of Bob's research group for the wonderful discussions during our group meetings: Philip Chong, Fan Mo, Roland Jiang, Yinghua Li, Aaron Hurst, Shauki Elassaad, Mike Case and Sungmin Cho. Collectively, they ensured that there was never a dull moment in the Friday evening proceedings.

I am grateful to La Shana Porlaris, Ruth Gjerde, Jennifer Stone, Delia Umel, Carol Zalon, Cindy Kennon, Jontae Gray, Gladys Khoury and Dan MacLeod for their help with administrative matters.

I have been lucky to make some great friends at Berkeley (in no particular order): Arkadeb Ghosal and Mohan Dunga, my roommates for three years, who bore my lack of culinary abilities with stoicism; Kaushik Ravindran who explained to me the finer points of normality and separability hours before the 250A homework was due; Abhishek Ghose and Anshuman Sharma who chose wisely; Steve Sinha who talked me into jumping off a cliff; Donald Chai who let me beat him at ping pong; Arindam Chakrabarty who shared his principles; Krishnendu Chatterjee who let me in on two and half secrets; Rahul Tandra who never called my bluffs; Alessandro Pinto,

Abhijit Davare, Farhana Sheikh, Animesh Kumar and Doug Densmore; and my most excellent office-mates Trevor Meyerowitz and Yanmei Li who listened patiently to all my writing troubles.

This thesis would not be possible without the support and encouragement of my family. I am grateful to my grandmother Shanta Banerjee for her unconditional love; to my parents Sumitra and Anjan Chatterjee for (mostly) good advice over the years; and to my wife Shuchi but for whom no doubt this thesis would have been finished sooner!

Contents

1 Introduction **1**

1.1 The Technology Mapping Problem 1

1.2 The Structural Approach to Mapping 2

1.3 Outline of this Chapter 4

1.4 The Problem of Structural Bias 5

1.5 Previous Work on Mitigating Structural Bias 7

1.6 Our Contributions to Mitigating Structural Bias 10

1.7 Previous Work on the Selection Problem 14

1.8 Our Contributions to the Selection Problem 16

1.9 Outline of this Thesis 19

2 And Inverter Graphs with Choice **22**

2.1 Overview 22

2.2 And Inverter Graphs 23

2.3 Constructing an And Inverter Graph 27

2.4 And Inverter Graphs with Choice 28

2.5 Detecting Choices in an And Inverter Graph 31

2.6 Choice Operator 39

2.7 Constructing And Inverter Graphs with Choice 40

2.8 Related Work 45

3 Cut Computation **47**

3.1 Overview 47

3.2 k-Feasible cut 48

3.3 Over-approximate k-Feasible Cut Enumeration 49

3.4 Implementation Details 51

3.5 Cut Function Computation 54

3.6 Cut Computation with Choices 61

3.7 Related Work 65

4 Boolean Matching and Supergates **68**

4.1 Overview 68

4.2 Preliminary Remarks 69

4.3 The Function of a Gate 70

4.4 Matching Under Permutations 73

4.5 Matching Under Permutations and Negations 78

4.6 Supergates 87

4.7 Supergates and Choices 96

4.8 Related Work 99

5 Match Selection for FPGA Mapping **104**

5.1 Overview 104

5.2 The Cost Model 105

5.3 Cover 106

5.4 Overview of the Selection Procedure 107

5.5 Depth-Oriented Selection 109

5.6 Area-Oriented Selection 111

5.7 Breaking Ties during Selection . . . 116
5.8 Minimizing Area Under Depth Constraints . . . 117
5.9 Related Work . . . 119

6 Match Selection for Standard Cells **123**
6.1 Overview . . . 123
6.2 Cost Model . . . 124
6.3 Electrical Edge . . . 126
6.4 Cover . . . 128
6.5 Overview of the Selection Procedure . . . 129
6.6 Delay-Oriented Selection . . . 132
6.7 Area-Oriented Selection . . . 134
6.8 Breaking Ties during Selection . . . 140
6.9 Minimizing Area Under Delay Constraints . . . 140
6.10 Related Work . . . 144

7 Experimental Results **147**
7.1 FPGA Mapping . . . 147
7.2 Standard Cell Mapping . . . 153

8 Conclusion **166**
8.1 Local Structural Bias . . . 166
8.2 Global Structural Bias . . . 168
8.3 Match Selection . . . 169

Bibliography **171**

Chapter 1

Introduction

Where choice begins, Paradise ends, innocence ends, for what is Paradise but the absence of any need to choose this action?

— Arthur Miller

1.1 The Technology Mapping Problem

We study the problem of *combinational* technology mapping: Given a set of gates $\mathcal{L}$, called the *library*, and a Boolean network G, let $\mathcal{M}$ be the set of Boolean networks constructed using gates from $\mathcal{L}$ that are functionally equivalent to G. These are called *mapped* networks. The goal of mapping is to find a mapped network $M \in \mathcal{M}$ that minimizes some objective such as area subject to certain constraints such as timing.

In this thesis we look at combinational technology mapping for both standard cells and look-up table (LUT) based FPGAs. (A look-up table with k inputs, called a k-LUT is a configurable gate that can implement any Boolean function of k variables.) Unless specified otherwise, we use the term "gate" to mean both gates in the conventional sense of standard cell design and also look-up tables in the context of FPGAs.

1.2 The Structural Approach to Mapping

In the general terms stated above, the mapping problem is intractable since it is very hard to enumerate either implicitly or explicitly the elements of $\mathcal{M}$. Keutzer obtained a significant simplification of the mapping problem by restricting the set of mapped networks considered during mapping to be those networks that are *structurally* similar to G [Keu87]. This restricted setting permits an algorithmic approach to technology mapping which we call the *structural approach*.

The main idea behind the structural mapping algorithm is a simple one. We assume that the original Boolean network G already has a "good" structure. Mapping is then done by a process of local re-writing. In this process, we identify a single output subnetwork N of G that is functionally equivalent to a gate g in $\mathcal{L}$ and replace N by g. As expected, there are many ways to do this re-writing, and Keutzer proved that for certain classes of networks (trees) and for certain cost functions (delay in the constant-delay timing model and area) it is possible to compute the optimal[1] mapped network by a dynamic programming algorithm.

For this to work, we need to assume that G has a "good" structure for the final network. This is ensured by applying *technology independent* logic synthesis algorithms to the initial Boolean network entered by the circuit designer to obtain G. Structural mapping is not expected to significantly alter the network structure of G, but merely to convert it in to a similar network built from gates in $\mathcal{L}$.

We now briefly review the main steps of structural mapping. The Boolean network G that is to be mapped is assumed to be in the form of a directed acyclic graph (DAG) where every node is either a two input AND gate or an inverter. In the context of mapping, G is called the *subject graph*. As noted before, G is obtained from technology independent synthesis.

[1] among structurally similar mapped networks

1. **Matching.** First, we visit each node n of G. For each library gate g, we check if there is a single-output sub-graph H of G rooted at n such that by replacing H by g, the functionality of G is not altered. If this is the case, then g is a possible *match* for n. The nodes of G that feed in to H are called the *inputs* of the match. At the end of the matching step, we have a list of all possible matches for every node in G. Note that in FPGA mapping, any subnetwork with k inputs or less is a match for a k-LUT.

2. **Match Selection.** Next, we select a subset of matches of the nodes in G, called a *cover*, that corresponds to a valid mapped network M that is functionally equivalent to G. The goal is to select a cover that optimizes the cost function subject to the given constraints. Match selection is usually divided into two steps:

   (a) **Evaluation.** We traverse the nodes of G in topological order from inputs to outputs. For each node, we pick the best match according to the cost function. The cost of a match includes the cost of the best matches of its inputs.

   (b) **Covering.** In the final step, for each output node n of G, we pick the best match m. Next, we pick the best match for each node m' that is an input of m, and so on until we reach the inputs.

   In this thesis we will often simply say *selection* to mean match selection.

From the above discussion, we see that the quality of results obtained by structural mapping depend on how well the matching and match selection steps are carried out.

Remark 1 The matching and selection steps are not independent. The result of matching defines the set $\mathcal{M}'$ of mapped networks considered during structural mapping. (In structural mapping, as we shall see later, $\mathcal{M}'$ is a proper subset of $\mathcal{M}$.)

However, the set of matches for every node computed as a result of matching is only an *implicit* description of the set $\mathcal{M}'$. The selection step is challenging because it must pick the best mapped network $M \in \mathcal{M}'$ without explicitly generating each mapped network in $\mathcal{M}'$.

The evaluation and covering steps of match selection allow us to select the best best cover in certain special cases. If we could explicitly enumerate the members of $\mathcal{M}'$, then finding the best mapped network would not be hard and we would not need the evaluation and covering steps. We could simply evaluate each member of $\mathcal{M}'$ and choose the best one according to our cost function. However, explicit enumeration is impractical since the size of $\mathcal{M}'$ is astronomical.

1.3 Outline of this Chapter

In this thesis we improve the quality of structural mapping by considering improvements to both matching and selection. In this chapter, we review the shortcomings of the methods proposed in the literature for these problems and present an overview of our contributions.

We start with matching. In Section 1.4 we review the problem of structural bias which limits the set of matches considered during matching. In Section 1.5 we review the literature on mitigating structural bias and in Section 1.6 we present an overview of our contributions in this area which are primarily aimed at increasing the set of matches.

Next, we look at selection. In Section 1.7 we review the prior work in the literature on the selection problem and identify a selection problem that has not been adequately addressed in the literature. We outline our contributions to the selection problem in Section 1.8.

Section 1.9 details the organization of the rest of this thesis.

1.4 The Problem of Structural Bias

The local nature of the matching process leads to structural bias — the structure of the mapped netlist corresponds very closely to the structure of the subject graph. This can be made precise as follows: Consider a gate g in the mapped network. Look at the gates that are inputs of g (in the mapped network). Each gate g' that is an input of g is a match for some node n' in the subject graph. This property of a mapped network – call it *structural correspondence* – follows from our definition of a match.

Thus the set of possible mapped networks considered during structural mapping is limited to those that obey the property of structural correspondence. However, there could be other mapped networks (i.e. elements of $\mathcal{M}$) that do *not* obey this property. To make this concrete, we present the following example:

Example 1.1 Figure 1.1 illustrates the phenomenon of structural bias. The Boolean network shown in (I) is a subject graph consisting of AND nodes and inverters. The mapped network shown in (II) can be obtained by conventional structural mapping. Note that the inputs of every match in (II) correspond to matches for nodes in the subject graph. For example, the inputs of the XNOR gate are the AOI gate and the inverter. They are matches for nodes u and w in the subject graph. However, the mapped network shown in (III) cannot be obtained by conventional structural mapping. To see this, consider the XOR at the input of the MUX. It is not a match for any node in the subject graph.

Although the above example is for standard cells, we want to emphasize that the structural bias is seen even in FPGA mapping.

But why is structural bias a problem? By limiting the mapping process to evaluate only a certain class of mapped networks, we may miss out on a better mapped network

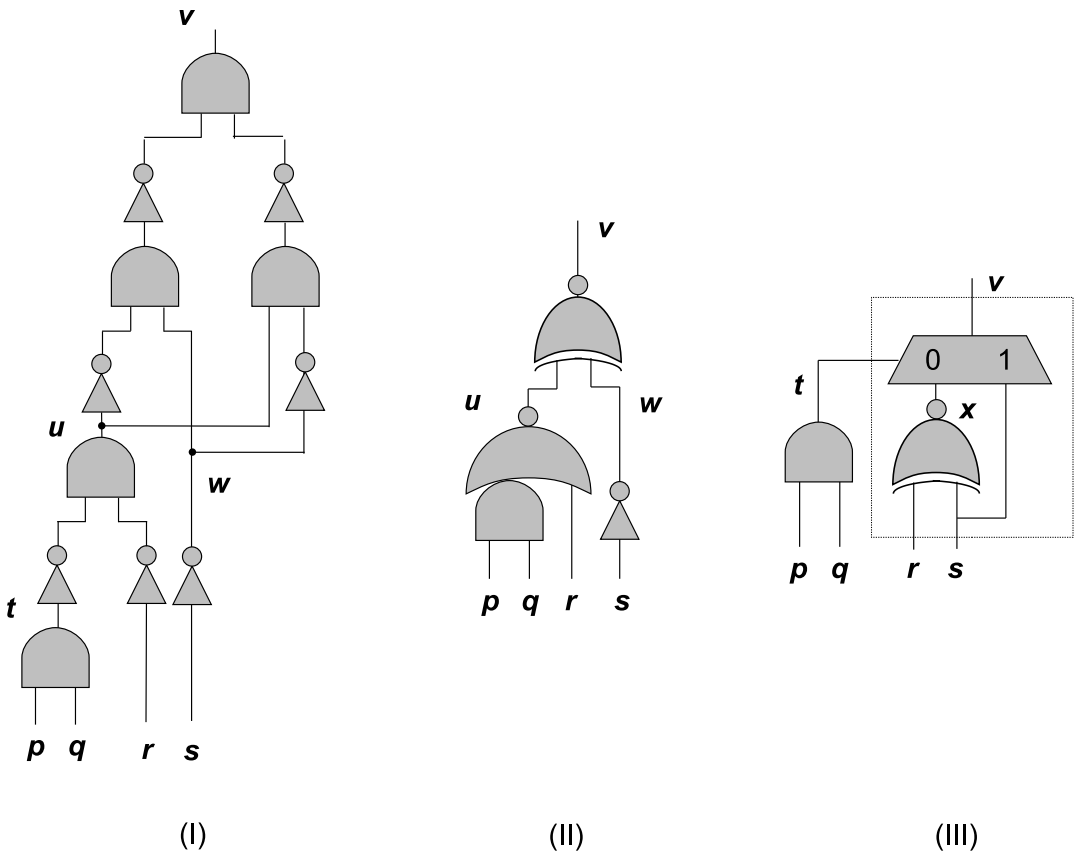


Figure 1.1: Illustration of structural bias. The Boolean network shown in (I) is the subject graph. The mapped network shown in (II) can be obtained with conventional structural mapping. However, the network shown in (III) cannot be obtained by conventional structural mapping since there is no net corresponding to x in the subject graph.

that is not in this class. Since the subject graph G itself is derived by heuristic technology independent optimization, it is possible (in fact quite likely) that a mapped network that is *not* in structural correspondence with G is better than all networks that are in correspondence.

In Example 1.1, depending on the library one of the mapped networks (II) or (III) is better than the other. Suppose that (III) is actually better than (II). Technology independent optimization, by definition does not have access to the library. Therefore,

it may be difficult to re-structure the subject graph so that mapped network (III) would be found by structural mapping.

1.5 Previous Work on Mitigating Structural Bias

The literature on technology mapping provides a number of extensions to Keutzer's original formulation that mitigate structural bias. These extensions are based on increasing the number of matches considered at a node. We begin by surveying the extensions proposed for standard cell matching, and later look at those for FPGAs.

In Keutzer's original work, the subject graph is first partitioned in to a forest of trees [Keu87]. Each library gate is decomposed in to a tree of two input AND gates and inverters[2] called a *pattern graph*. (In fact a gate may lead to multiple pattern graphs since the decomposition is not unique.) A pattern graph (and thereby a library gate) is matched at a node n by checking for structural isomorphism using a *tree matching* algorithm.

Rudell extended tree matching to the case where the pattern graphs could be leaf-DAGs [Rud89, Chapter 4]. This allowed non-tree gates such as multiplexers and XORs to be matched. He also observed that by replacing every wire in the subject graph by a pair of inverters, and by adding a "wire" gate to the library (whose pattern consists of a pair of inverters in series), the set of matches is larger.

A radically new method to reduce structural bias was proposed by Lehman *et al.* [LWGH95]. They observed that there are many different ways in which a subject graph can be derived from a Boolean network. For example, a multi-input AND can be decomposed in to a tree of two input AND gates in a variety of ways. However, in the mapping algorithm described above, a decision has to be made *a priori* as to which decomposition is to be used to generate the subject graph. Thus, certain

[2] Actually, Keutzer used NAND gates and inverters but the difference is not important here.

matches that would have been detected with a different decomposition are no longer detected.

Their insight was that the tree matching algorithm could be extended to work with a new kind of subject graph that compactly encodes multiple ordinary subject graphs by means of Choice nodes. They called this the *mapping graph*. (As in Keutzer, they partition the subject graph in to a forest of trees before matching.) The modified tree matching algorithm working on a mapping graph finds matches across all subject graphs encoded in the mapping graph.

Pursuing a different approach inspired by FPGA mapping (which will be reviewed shortly), Kukimoto extended the set of matches explored at a node by using a general DAG matching procedure instead of tree matching [KBS98]. Thus both the subject graph and the pattern graphs are allowed to be general DAGs in Kukimoto's extension. This significantly increases the number of matches found at a node, especially when partitioning the subject graph into a forest of trees would lead to many small trees.

A different line of research was initiated by Mailhot and de Micheli with their proposal for *Boolean matching* [MM90]. Recall that in Keutzer's work (and later in Lehman *et al.* and Kukimoto) the actual process of matching a gate (or more precisely, its pattern graph) with a sub-graph H rooted at a node n is through graph isomorphism. Since a library gate, especially a complex one, has many possible decompositions into pattern graphs, only a subset of all possible decompositions is used in practice. Now if the sub-graph of the subject graph is not structurally identical to any one of the decompositions of the library gate, a match may not be found, even though replacing the H by the gate is valid. One can think of this as being a more *local* structural bias as opposed to the *global* bias that we were so far concerned with. Boolean matching addresses local structural bias by directly matching a gate with the sub-graph H by comparing their Boolean functions. Thus, with Boolean matching there is no need for pattern graphs.

In the light of Boolean matching, we use the term *structural matching* to refer to the graph isomorphism-based matching. To summarize, structural matching leads to local structural bias (i.e. at the granularity of a gate) in addition to global structural bias. Boolean matching eliminates local structural bias, but still suffers from global structural bias.

To complete our discussion on methods to mitigate structural bias for standard cells, we must mention the *constructive synthesis* algorithm proposed by Kravets and Sakallah [KS00] and subsequently improved by Mishchenko *et al.* [MWK03]. In these algorithms, the structure of the subject graph is not used at all — instead the subject graph is used to construct a representation of the Boolean functions (using BDDs or truth-tables), and then a decomposition algorithm is applied to the functions to obtain the mapped netlist. At first, this may seem like the answer to our original technology mapping formulation. However, the chief conceptual drawback of these algorithms is that they are committed to a *specific* decomposition scheme. Thus they are not able to explore the full space of mapped solutions, i.e. $\mathcal{M}$. Therefore, they just introduce a structural bias of a different kind.

In practical terms, constructive algorithms suffer from long run-times since constructing a representation of Boolean functions suitable for decomposition is very expensive. Therefore, these algorithms are limited to small circuits and mostly used for greedily improving an already mapped circuit by resynthesis.

In FPGA mapping, local structural bias is not a problem since a k-LUT can implement any function of k variables or less; the main challenge is to enumerate the different subnetworks rooted at a node n in the subject graph that can be implemented by a k-LUT. Cong and Ding presented a network flow based algorithm *Flowmap* that can identify a *single* subnetwork rooted at a node n that minimizes depth[3] [CD92].

[3]In the FPGA mapping community it is common to use the term "depth" when talking about the delay due to LUTs.

This is done by identifying a cut of size k or less in the subject graph (without splitting it in to trees). It was this work that inspired Kukimoto to explore DAG matching in the context of standard cells.

The main limitation of Flowmap is that it produces only one cut (or equivalently, only one subnetwork) that minimizes depth. However, the network flow based method that it uses to find the cut cannot be extended easily to handle other cost functions. This motivated Cong and Ding to explore techniques to enumerate all cuts (i.e. all subnetworks that can be realized by a k-LUT) [CD93]. This work was improved by Pan and Lin who presented an elegant algorithm to enumerate all cuts [PL98]. We complete our review of attempts to reduce structural bias in FPGA mapping, by noting that Chen and Cong adapted the algorithm proposed by Lehman *et al.* for FPGA mapping [CC01].

1.6 Our Contributions to Mitigating Structural Bias

1.6.1 Improved Boolean Matching

The starting point of our approach is an improved Boolean matching procedure that has the following features:

1. It extends classical Boolean matching to work directly on a graph that encodes multiple subject graphs in a manner similar to Lehman *et al.*. We call this structure an And Inverter Graph (AIG) with choice. We want to emphasize that an AIG with choice encodes multiple DAGs in a natural manner without breaking them *a priori* in to trees as was done in Lehman. This feature combined with the use of Boolean matching ensures that more matches are found and structural bias is reduced further.

2. It is a simplification of the traditional Boolean matching procedure that leads both to a simpler implementation, and, a faster run-time. We postpone the detailed comparison of our simplified procedure with the state-of-the-art methods in the literature to Section 4.8. (For the impatient reader, the main idea is that we do *not* do $\mathcal{NPN}$-canonization.)

3. It naturally handles matching of inverters, buffers and incorporates a technique similar to Rudell's inverter-pair heuristic to consider matches in both polarities. Somewhat surprisingly, this aspect of Boolean matching has not been discussed in the literature. In this connection, an algorithmic simplification is obtained compared to Lehman *et al.*: They incorporate the inverter pair heuristic by introducing cycles in the matching graph which complicates their algorithm. The AIG with choice structure does not have cycles, and this leads to simpler implementation and simpler theoretical analysis.

1.6.2 Supergates

Next, we introduce a new construct called a *supergate* to further reduce structural bias. This method of reducing structural bias is complementary to the other methods proposed in the literature.

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 1.1 (III) provides a simple counter-example.

Example 1.2 As was discussed in Example 1.1, conventional mapping would find the mapping shown in (II) but would fail to find the one shown in (III). However,

if we connect the MUX and the XOR gates together in the manner shown in (III) to form a new gate (indicated in (III) by the box), it is easily verified that this new gate is a match for node v in the subject graph. (All inputs of this supergate correspond to nodes present in the subject graph.) This new gate is a supergate built from the library gates expressly for the purpose of finding more 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. Supergates may be seen as an extension of the Boolean matching idea to further decrease local structural bias beyond what regular Boolean matching can accomplish.

1.6.3 Lossless Synthesis

The ability to find matches in a mapping graph that encodes many different subject graphs leads to a new perspective on technology independent (TI) synthesis. Instead of seeing TI synthesis as a process that produces a *single* subject graph for mapping, we see it as a process that explores different network structures and produces *multiple* subject graphs. At the end of TI synthesis these subject graphs are collected together and encoded in an AIG with choice which is then used for mapping. As discussed earlier, the mapper is not constrained to use any one subject graph encoded in the AIG with choice, but can pick and choose the best parts of each subject graph. We call this *lossless synthesis* since conceptually, no network seen during the synthesis process is lost.

We note here that a similar idea already appears in Lehman *et al.* though it is only mentioned in passing: their focus was on adding different algebraic decompositions to construct the mapping graph [LWGH95]. In this thesis, we address two questions

relating to lossless synthesis that have not been addressed in Lehman.

Question 1. How do we *efficiently* combine multiple subject graphs to construct an AIG with choice?

We show that it is possible to leverage recent advances in combinational equivalence checking for this purpose. 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.

Lehman *et al.* do not provide any details on how multiple networks were combined. Since their benchmarks are small we believe that they use BDDs to detect functionally equivalent points. However, as is well known in the verification community, BDDs are not robust and do not scale in contrast to our proposal which is robust and scales well.

Question 2. Is lossless synthesis better than conventional synthesis?

This question is hard to answer in a comprehensive manner since it depends on how the subject graphs are generated during TI synthesis. In this thesis we answer this question by focusing on two specific scenarios for lossless synthesis and show that lossless synthesis is useful in these scenarios. (We do *not* study the efficacy of different ways of lossless synthesis and leave to future work to find a set of lossless synthesis scripts that lead to the best results.)

In the first scenario, we take intermediate snapshots of the Boolean network as it is being processed through `script.rugged`. (`script.rugged` is a standard technology independent optimization script that is used in the publicly available synthesis system SIS [SSL+92].) This is shown schematically in Figure 7.1(b). By combining these networks together in to an AIG with choice, and mapping over this, we obtain better

results than mapping on just the final network alone. This is expected since 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.

In the second scenario, we explore iterative mapping. Starting with a subject graph G_0, we map it. Next, we treat the mapped network as a regular Boolean network and subject it to TI synthesis. The resultant network is combined with the original subject graph G_0 to obtain an AIG with choice G_1 which is then used to obtain a new mapped network. This new mapped network is again subjected to TI synthesis and combined with G_1 to get G_2 which is again used for mapping. As this process is iterated, we find the resultant mapped networks improving in quality.

Lehman *et al.* do not provide details on how they obtained the different networks that they combine to get the mapping graph. They also do not separate the results of combining multiple networks from those obtained by adding algebraic decompositions exhaustively. Our experiments show that it is more effective to merge different networks than exhaustively adding algebraic decompositions. The use of different networks leads to more *global* choices in contrast to the more *local* choices introduced by the addition of different algebraic decompositions.

1.7 Previous Work on the Selection Problem

As noted in Remark 1 of Section 1.2, the matching and selection steps are not independent. Therefore, for a matching algorithm, given a cost function, one has to answer the following question: How can we select matches so that the mapped network (obtained by covering) is optimal among all mapped networks in $\mathcal{M}'$? In this section we go through the matching algorithms surveyed in Section 1.5 and try to answer the above question for area and delay cost functions.

In Keutzer's original work and in Rudell's extension, the subject graph is partitioned in to trees and matches are found by tree matching. In this case, they show that it is possible to find the optimal mapped network for *each tree* for (1) minimum area, and, (2) minimum delay in the constant delay model. For example, in the case of area, the best match for each node is determined as follows. The area of a match is defined as the area of the gate itself plus the area of the best matches at the inputs of the match that are not primary inputs. The match with the least area is chosen as the best match for the node.

There are two things to note about this. First, nothing is claimed about the optimality for the entire subject graph; only that each tree is optimally mapped. Second, when mapping for delay in the constant delay model, optimality can be claimed only if single pin-to-pin delays are used: Murgai shows that the problem for trees is NP-hard if the pin-to-pin rise and fall delays are different [Mur99].

Rudell also proposed a method to handle the load-dependent delay model during tree mapping [Rud89, Chapter 4]. In his method, we do not store one best match for each node, but store the best match for every possible load. Thus several "best" matches are stored at a node. Touati extended this work in his thesis and showed how this could be done optimally by using piece-wise linear curves to represent the optimal arrival times as a function of load [Tou90, Chapter 2].

In the case of Lehman *et al.*, once again, mapping for minimum area and minumum delay can be done optimally as long as the mapping graph is broken into trees. However, nothing can be said about the optimality of the mapping graph as a whole.

The first paper to present an optimality result for DAGs was the Flowmap paper on FPGA mapping where Cong and Ding presented an algorithm to optimally map for minimum delay [CD92]. As noted before, Kukimoto *et al.* adapted this algorithm for standard cells, and showed that optimality holds if the constant delay model is used with single pin-to-pin delays.

Farrahi and Sarrafzadeh showed that the problem of area optimal mapping on a DAG is NP-hard by a direct reduction from the 3-SAT problem and suggested some heuristics to solve the problem [FS94]. Although their proof is for FPGA mapping, the same argument holds for standard cells given a sufficiently rich library.

On the practical side, there have been a number of papers proposing heuristics for area-oriented mapping on DAGs for FPGAs. The most recent of these are DAOmap [CC04] and IMAP [MBV04; MBV06] which enumerate cuts using Pan and Lin's procedure and then use a variety of heuristics to select the best set of cuts for minimum area. (A more detailed review of the literature appears in Section 5.9.)

1.8 Our Contributions to the Selection Problem

Our contribution to the selection problem is an efficient technique for area-oriented selection that works for both standard cell mapping and FPGA mapping. The technique is based on two heuristics:

1. **Area-Flow.** This heuristic, taken from the literature [CWD99; MBV04; CC04], is used to obtain an *initial* area-oriented cover. The main idea behind area-flow is to divide the area cost of a match of a multi-fanout node in the subject graph among the different fanouts of the node.

2. **Exact Area.** A second heuristic is then used to *greedily* and *incrementally* optimize the initial cover. In this heuristic, each node in the subject graph is visited in topological order. At a given node, the best match (in terms of area) is selected, assuming that the rest of the cover is unchanged. This encourages the selection of matches that effectively exploit existing logic in the network which leads to greater sharing and smaller area.

Our experimental results show that the combination of these heuristics is very effective in practice and compares favorably with the larger set of heuristics used by DAOmap. Furthermore, in addition to improving quality, the proposed technique has the following advantages:

1. **Simplicity of implementation.** There are no magic constants to fine-tune in the proposed technique. This is in contrast to the techniques presented in the literature which typically involve some constants to be tuned. For example, the empirically determined constants α, β, etc. used by DAOmap [CC04, Section 4.2].

2. **Handling Choices.** Previous work on using choices to reduce the structural bias has focused on minimizing delay since that can be done optimally. The proposed technique is effective in optimizing for area even on subject graphs that encode multiple networks: our experiments show that, like delay, area can also be significantly reduced using lossless synthesis and choices.

   Exact area is particularly useful in this context since it is greedy and incremental, and does not rely on heuristic fanout estimates based on the subject graph. Estimating fanouts in such subject graphs is typically harder since many fanouts of a node are due to choices and are not "true" fanouts. Extending the heuristics used by IMAP and DAOmap to such subject graphs is likely to be difficult.

3. **Extension to Standard Cells.** The proposed technique can be adapted to work for the standard cell mapping problem. The selection problem in standard cell mapping is more complicated than in FPGAs due to (i) the presence of single input gates such as inverters, and (ii) the need to choose the correct polarity for each node in the subject graph. In this thesis, we show how the

proposed technique can be naturally extended to handle these complications. (The method also works for lossless synthesis for standard cells.)

Experiments with standard cell mapping show that the proposed area-oriented mapping technique can significantly reduce area, especially compared to Kukimoto *et al.*

Minimizing area under delay constraints. In practice, the selection problem typically involves minimizing area under delay constraints. Given an area-oriented selection method, and a delay-oriented selection method, there are different ways to combine them to minimize are under delay constraints. For example, one could start with a good area solution (ignoring delay constraints) and then successively remap critical paths. An alternative method involves maintaining area-delay trade-off curves during selection [CP92].

In this work our focus is on the underlying area-oriented selection. Therefore, we simply adopt a simple slack-based approach to minimizing area under delay constraints; and note that the proposed technique could be used with other schemes to minimize area under delay constraints.

Power and other cost functions. Finally, we would like to point out that the proposed technique could also be used for power-oriented mapping where the goal is to minimize power consumption. In general, we expect the proposed area-oriented selection method to work well for any "bulk" cost function that depends on all the gates in the mapped network. However, for concreteness, in this thesis, we focus only on area.

1.9 Outline of this Thesis

In this thesis we present a detailed and fairly self-contained description of the ideas outlined in Sections 1.6 and 1.8 by developing a technology mapper based on these ideas. (In fact, we develop two technology mappers: one for FPGA mapping, and another for standard cell mapping.)

Although the new ideas presented in this thesis improve upon existing mappers in terms of the quality of results, we believe that some of the ideas presented here also lead to *simpler* implementations of technology mappers. We hope that the description in this thesis is sufficiently detailed that the reader sees the essential simplicity of these ideas and is motivated to implement a technology mapper himself!

We also wish to point out that the simplicity of implementation does not come at the cost of efficiency. Indeed, we are concerned with run-times and scalability and we believe that the proposed ideas admit very efficient implementations.

The organization of Chapters 2–6 of this thesis roughly reflects the logical structure of a technology mapping program:

- **Chapter 2.** *And Inverter Graphs with Choice.*

  We formally introduce And Inverter Graphs (AIG) with choice as the basic data structure for representing subject graphs for lossless synthesis. This is an extension of the AIG data structure that is a popular circuit representation for equivalence checking. The main technical contribution here is to show how a modern equivalence checking algorithm (SAT sweeping) can be modified to combine multiple AIGs into a single AIG with choice that encodes them.

- **Chapter 3.** *Cut Computation.*

  We start by reviewing Pan and Lin's cut computation procedure to compute all k-feasible cuts for every node in an AIG. For FPGA mapping a k-feasible

cut directly matches a k-LUT. For standard cells, a k-feasible cut is a possible match for a gate with k or fewer inputs. (Details on the matching is covered in Chapter 4.) We then extend Pan and Lin's procedure in two ways: (i) we show how redundant cuts can be detected efficiently during cut computation, and (ii) we show how the function of a cut can be computed efficiently during cut computation. Finally, we extend the algorithm to compute cuts on an AIG with choice.

- **Chapter 4.** *Boolean Matching and Supergates.*

  We formally define the notion of a match and present the simplified Boolean matching algorithm to match k-feasible cuts of an AIG with choice to library gates for standard cell mapping. This algorithm handles matching of inverters and buffers and incorporates a technique similar to Rudell's inverter pair heuristic to consider matches in both polarities. Next, we introduce supergates and show how they can improve the quality of Boolean matching by reducing local structural bias. Finally, we describe how supergates are generated from a given library.

- **Chapter 5.** *Match Selection for* FPGA*s.*

  We start by defining the notion of a cover (for FPGA mapping) and review the cost model for FPGA mapping. Next, we present a generic selection procedure where the cost function is abstracted and then specialize it for depth-oriented mapping. Then we present the main contribution of this chapter which is the new area-oriented mapping algorithm based on area-flow and exact area. Finally, we show how the depth-oriented mapping and area-oriented mapping may be combined to map for area under depth constraints.

- **Chapter 6.** *Match Selection for Standard Cells.*

> We cover the same topics as Chapter 5, except this time the treatment is tailored for for standard cells. The selection problem for standard cells is more complicated than that for FPGAs due to the presence of inverters and buffers, and matching in both polarities. The main contribution here is to show how area-flow and exact area can be easily extended to handle these complications.

In short, Chapters 2, 3 and 4 can roughly be thought of as addressing the structural bias problem; and Chapters 5 and 6 as addressing the selection problem.

Finally, experimental results for both FPGAs and standard cells are presented in Chapter 7 where we evaluate the efficacy of the different techniques for reducing structural bias; and also compare our mapper against other mappers.

The reader interested only in FPGA mapping may skip Chapters 4 and 6 which deal exclusively with standard cell mapping. The reader interested only in standard cell mapping may skip Chapter 5 on match selection for FPGAs but is encouraged *not* to do so: It may be easier to first understand area-oriented selection in the simpler setting of FPGA mapping.

Chapter 2

And Inverter Graphs with Choice

2.1 Overview

The subject graph for our mapper is a data structure called an *And Inverter Graph with Choice* that enables lossless synthesis by encoding different Boolean networks efficiently into a single subject graph. In this chapter, we define this data structure and present efficient algorithms to construct it from other, more commonly used, representations of Boolean networks.

The And Inverter Graph with Choice structure is an extension of the And Inverter Graph (AIG) data structure which is commonly employed in combinational equivalence checking. We begin with a review of AIGs (Section 2.2) and briefly discuss how they are constructed from other network representations (Section 2.3). Next, we formally introduce AIGs with choice (Section 2.4). Although every AIG is trivially an AIG with choice, more interesting AIGs with choice can be constructed by detecting equivalent nodes in a (redundant) AIG. To this end we present a scalable and robust algorithm based on a combinational equivalence checking technique (SAT sweeping) (Section 2.5). Finally, we leverage this algorithm to define a *Choice* operator that can

efficiently combine multiple AIGs into a single AIG with choice.

We postpone the detailed discussion of related work in the literature to the end of the chapter (Section 2.8).

2.2 And Inverter Graphs

And Inverter Graphs (AIGs) were introduced by Kuehlmann and Krohm for combinational equivalence checking [KK97].

2.2.1 Definition

Let (X, A), $A \subseteq X \times X$, be a directed acyclic graph where each *node* $n \in X$ has either no incoming arcs[1] or exactly two incoming arcs. Among nodes with no incoming arcs, one is special and is called the *Zero* node; and the rest are called *Inputs*. Nodes with two incoming arcs are called *And* nodes. Let inv be a function from A to the set $\{0, 1\}$. The tuple $G = (X, A, \text{inv})$ is called an And Inverter Graph (AIG).

If $\text{inv}(a) = 1$ then arc a is said to be inverted. When drawing an AIG the Zero node is usually omitted; inverted arcs indicated with a dashed line, and uninverted arcs with a solid line.

Example 2.1 Figure 2.1 shows an AIG.

Members of the set $X \times \{0, 1\}$ are called *edges* of G. If $a = (u, v)$ is an arc, the edge $(u, \text{inv}(a))$ is called an input edge of node v, and u is called an input node of v. Edges $(u, 0)$ and $(u, 1)$ of G are said to be *complements* of each other. If z is the Zero node, the edge $(z, 0)$ is called the Zero edge. The complement of the Zero edge is called the One edge. The set A viewed as a relation on X is called the *fanin* relation.

[1] In the context of AIGs we distinguish between an arc (members of A) and an edge (to be defined).

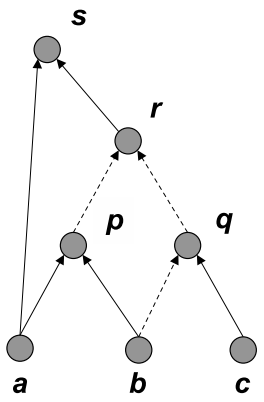


Figure 2.1: Example of an AIG. The Zero node is omitted. Nodes a, b, and c are Input nodes, and the rest are And nodes. Dashed lines indicate inverted arcs.

2.2.2 Semantics

Every node and edge in an AIG corresponds to a Boolean function, called the *function* of the node or edge. Every Input node n of an AIG is associated with a formal Boolean variable $\mathbb{X}_n$. The functions of the other nodes and edges are defined in terms of these Boolean variables.

Formally, the semantics of an AIG G is specified by defining a valuation function f that maps every node and edge of G to a Boolean function. If e is the edge (n, i), we define

$$f(e) = \begin{cases} f(n) & \text{if } i = 0 \\ \neg f(n) & \text{if } i = 1 \end{cases}$$

If n is a node, we define

$$f(n) = \begin{cases} 0 & \text{if } n \text{ is the Zero node} \\ \mathbb{X}_n & \text{if } n \text{ is an Input node} \\ f(e_l) \cdot f(e_r) & \text{otherwise } (n \text{ is an And node with input edges } e_l \text{ and } e_r) \end{cases}$$

This recursive definition of f is well-formed since the nodes in an AIG can be topo-

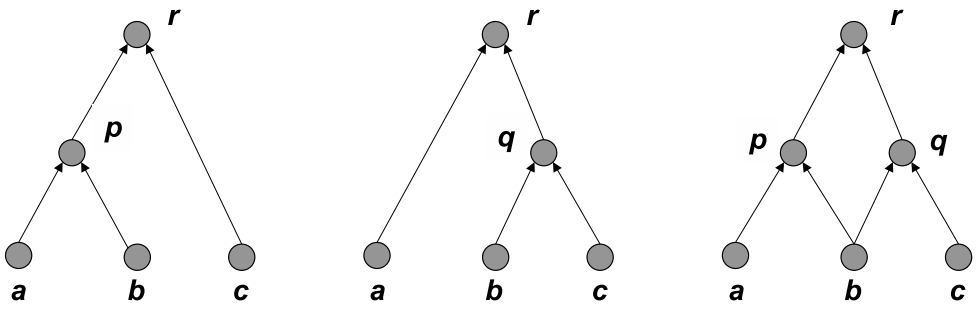


Figure 2.2: AIGs are not canonical representations of Boolean functions. The three different AIGs shown here correspond to the function $\mathbb{X}_a \cdot \mathbb{X}_b \cdot \mathbb{X}_c$.

logically ordered (as it is a directed acyclic graph).

An AIG G that has exactly one node n with no out-going arcs is called a *single-output* AIG. In this case we often abuse terminology and talk of the function of G. This should be understood to be the function of n.

Example 2.2 The functions of the nodes of the AIG in Figure 2.1 are as follows ($\mathbb{X}_n$ for $n \in \{a, b, c\}$ are Boolean variables associated with the inputs):

$$\begin{array}{lcllcl} f(a) & = & \mathbb{X}_a & f(q) & = & \neg\mathbb{X}_b \cdot \mathbb{X}_c \\ f(b) & = & \mathbb{X}_b & f(r) & = & \neg(\mathbb{X}_a \cdot \mathbb{X}_b) \cdot \neg(\neg\mathbb{X}_b \cdot \mathbb{X}_c) \\ f(c) & = & \mathbb{X}_c & f(s) & = & \mathbb{X}_a \cdot \neg(\mathbb{X}_a \cdot \mathbb{X}_b) \cdot \neg(\neg\mathbb{X}_b \cdot \mathbb{X}_c) \\ f(p) & = & \mathbb{X}_a \cdot \mathbb{X}_b & & & \end{array}$$

2.2.3 Canonicity

A Boolean function may be represented by different[2] single-output AIGs. In this sense AIGs are not canonical representations of Boolean functions.

Example 2.3 Figure 2.2 shows 3 different AIGs that correspond to the function $\mathbb{X}_a \cdot \mathbb{X}_b \cdot \mathbb{X}_c$.

[2]non-isomorphic when viewed as graphs with labels on PI nodes

A consequence of this is that an AIG may be redundant i.e. two different nodes in an AIG may compute the same Boolean function. Although in general it is computationally expensive to remove redundancies in an AIG (detecting if two nodes compute the same function is Co-NP Complete), in practical implementations some easily detected redundancies are prohibited by enforcing the following conditions:

- **Structural Hashing.** There is at most only one And node with a given pair of edges as inputs (since two nodes with same inputs compute the same function).
- **Redundant And Elimination.** There is no And node with both inputs the same.
- **Constant Zero Elimination.** There is no And node with an edge and its complement (since it computes the same function as the Zero node).
- **Constant Zero Propagation.** There is no And node with the Zero edge as an input (since it computes the same function as the Zero node).
- **Constant One Propagation.** There is no And node with the One edge as an input (since it computes the same function as the other input).

We abuse terminology and refer to these conditions collectively as *structural hashing* conditions. An AIG that satisfies these conditions is said to be *structurally hashed*. We generally assume that an AIG is structurally hashed.

2.2.4 Representation and Interface

An AIG is represented in programs as an array of nodes. Every And node stores its two input edges. An uncomplemented edge $(n, 0)$ is represented by the index of node n in the array, and a complemented edge $(n, 1)$ is represented by the negation of the index of node n. (For this representation to work it is usually convenient to not use

the zeroth location of the array, and start storing nodes from the first location. The Zero node is usually stored at the first location.)

An AIG package usually provides four functions to construct an AIG:

- GET-ZERO() which returns the Zero edge.
- COMPLEMENT(e) which returns the complement of edge e.
- CREATE-AND(e_l, e_r) which returns the positive edge of the And node with inputs e_l and e_r. The structural hashing rules are enforced by this function. In particular, it uses a hash table to check if there is an existing And node with inputs e_l and e_r. It only creates a new node if one doesn't exist.
- CREATE-VAR() which creates a new Input node and returns the positive edge to that node.

Note that the order in which the nodes are added to an AIG using this interface is a topological order for the AIG. Furthermore, every node n is given an unique integer id, written as id(n). This is usually the position of the node in the vector of nodes.

Kuehlmann *et al.* provide a nice overview of how these operations can be implemented [KPKG02, Section III.A]. (See however the discussion in Section 2.8.)

2.3 Constructing an And Inverter Graph

An AIG is constructed using the interface described in the previous section from a technology independent (TI) network in the last step of TI synthesis. If the representation of the TI network allows complex nodes (for e.g. the SIS network data structure allows PLAs), then some care must be taken while decomposing them during AIG construction. In particular, complex nodes should be converted in to factored form (for e.g. see [Bra87] or Chapter 5 in [HS02]) and large multi-input And and

Or gates should be decomposed in a balanced manner (e.g. using the Huffman-tree construction heuristic [CD96, Section 3.1.2]).

This is less of an issue with the fast re-writing based approach to TI synthesis since one can start with some decomposition, and apply AIG re-writing to obtain a good starting AIG for technology mapping [MCB06a].

Output Edges When an AIG G is constructed from a TI network N, the output ports of N will correspond to a set O of edges of G. This set is called the set of *output edges*. Although the definition of the output edges depends on the TI network N, in what follows we shall simply talk of the output edges of an AIG, trusting that the TI network is unambiguous. (It is the network used to construct the AIG.)

2.4 And Inverter Graphs with Choice

2.4.1 Definition

Let X be the set of nodes in an AIG G. An equivalence relation $\simeq$ on X is a *functional equivalence relation* if for $n_1, n_2 \in X$,

$$n_1 \simeq n_2 \implies f(n_1) = f(n_2) \text{ or } f(n_1) = \neg f(n_2)$$

Let $X_\simeq$ denote the set of equivalence classes of X under the functional equivalence relation $\simeq$ i.e. $X_\simeq = X/\simeq$. A subset $\mathcal{X}$ of X is called a *set of functional representatives* iff it contains exactly one node n_C from each equivalence class $C \in X_\simeq$. $n_C \in \mathcal{X}$ is called the *functional representative* of C.

Let G be an AIG, $\simeq$ be a functional equivalence relation, and $\mathcal{X}$ be a set of functional representatives. $\widehat{G} = (G, \simeq, \mathcal{X})$ is an AIG *with choice* if every node in G with an out-going arc is a functional representative of some equivalence class. In other

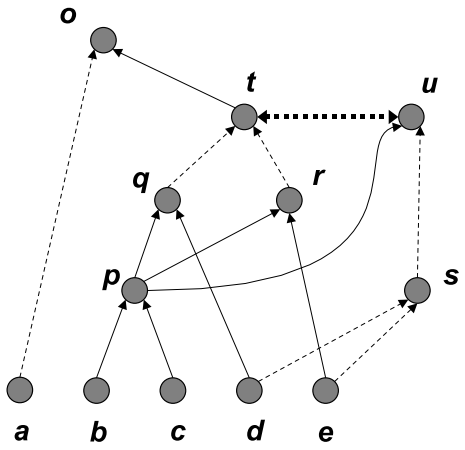


Figure 2.3: An AIG with choice. Nodes u and t are functionally equivalent up to complementation and belong to the same equivalence class.

words, if (u, v) is an arc of G, then $u \in \mathcal{X}$. An edge (r, i) is called a *representative edge* if $r \in \mathcal{X}$. The set of representative edges is denoted by $\mathcal{E}$.

For convenience, we assume that if a Input node belongs to an equivalence class, then it is the functional representative for that class. (This is well defined since an equivalence class cannot contain more than one Input node.)

Example 2.4 Every AIG is trivially an AIG with choice, since we can define the following relation:

$$n_1 \simeq_T n_2 \quad \text{iff } n_1 = n_2$$

It is easy to check that $\simeq_T$ is a functional equivalence relation. Furthermore, set $\mathcal{X} = X$ to obtain an AIG with choice.

In diagrams we use bi-directional lines to join nodes in the AIG which belong to the same equivalence classes. The line joining two equivalent nodes n_1 and n_2 is drawn dotted if $f(n_1) = \neg f(n_2)$, and solid if $f(n_1) = f(n_2)$.

Example 2.5 Figure 2.3 is an example of an AIG with choice. The only non-trivial equivalence is that of nodes t and u. (Every other node is equivalent only to itself.) Node t is the functional representative of the class containing $\{t, u\}$. Observe that node u has no fanouts (since it is not a functional representative).

In formal terms, the set of nodes X is $\{a, b, c, d, e, p, q, r, s, t, u, o\}$. Let eq be the equality relation on X, i.e. $\text{eq} = \{(n, n) \mid n \in X\}$. The relation $\simeq$ is given by

$$\simeq = \text{ eq} \cup \{(t, u), (u, t)\}$$

Since

$$f(t) = \neg(\mathbb{X}_b \cdot \mathbb{X}_c \cdot \mathbb{X}_d) \cdot \neg(\mathbb{X}_b \cdot \mathbb{X}_c \cdot \mathbb{X}_e) = \neg(\mathbb{X}_b \cdot \mathbb{X}_c \cdot (\mathbb{X}_d + \mathbb{X}_e))$$

and

$$f(u) = \mathbb{X}_b \cdot \mathbb{X}_c \cdot (\mathbb{X}_d + \mathbb{X}_e)$$

we have $f(t) = \neg f(u)$. This establishes that $\simeq$ is indeed a functional equivalence relation. In this example,

$$X_{\simeq} = \{\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{p\}, \{q\}, \{r\}, \{s\}, \{t, u\}, \{o\}\}$$

and $\mathcal{X} = X \backslash \{u\}$.

2.4.2 Representation and Interface

An AIG with choice is represented in programs as a regular AIG with some additional data. We store $X_{\simeq}$ rather than the functional equivalence relation $\simeq$ in the following manner:

- Every node has an attribute called *repr*. For a node n, repr$[n]$ indicates the

functionally equivalent edge of the representative of n.[3] Note that if r is a representative, then $\text{repr}[r] = (r, 0)$.

- All nodes in the same functional equivalence class are put into a linked list, and the representative is the head of the linked list. Thus starting from the representative, all members of an equivalence class can be easily determined.

The AIG construction interface (Section 2.2.4) is augmented with a new procedure called SET-CHOICE. It takes a node n and an edge $e = (r, i)$, and adds n to the equivalence class of r, and sets $\text{repr}[n]$ equal to e. We assume that r is a functional representative, i.e. $\text{repr}[r] = (r, 0)$ holds and that n has no fanouts.[4] Furthermore, when a new And node n is created (in CREATE-AND) $\text{repr}[n]$ is set to $(n, 0)$.

2.5 Detecting Choices in an And Inverter Graph

2.5.1 Overview

As was noted in Section 2.2.3, we may build an AIG with functionally equivalent nodes. In this section we outline a procedure for detecting these equivalent nodes and grouping them into equivalence classes to construct an AIG with choice. This procedure is a modification of the combinational equivalence checking procedure, SAT-sweeping, proposed by Kuehlmann [Kue04]. This will be used as a subroutine in a procedure to combine multiple AIGs into a single AIG with choice in Section 2.6.

Formally, we are given an AIG G with nodes X. To obtain an AIG with choice, we need to

1. construct a functional equivalence relation $\simeq$ on X,

[3] Recall that the representative may either have the same function as the n or the complement.
[4] Otherwise a non-representative would have fanout.

2. determine the set of representatives $\mathcal{X}$, and

3. modify G so that only nodes in $\mathcal{X}$ have fanouts.

Computationally, we do not explicitly construct the relation $\simeq$, but instead compute directly the equivalence classes $X_{\simeq}$. Furthermore, the three steps outlined above are not done separately, but in an integrated manner. This is done by building a new AIG with choice from scratch, using G as a template. Throughout the construction, the newly created AIG is a valid AIG with choice.

We call the procedure BUILD-CHOICE-AIG. The input to BUILD-CHOICE-AIG is the AIG G. The procedure constructs a new AIG with choice $\widehat{G}$ in topological order by processing G (also in topological order). For every node n in G, the procedure tries to create a corresponding node n' in $\widehat{G}$. When two nodes in G are functionally equivalent, there are two possibilities in $\widehat{G}$:

- **Case (a)** There are two corresponding nodes in $\widehat{G}$. In this case, the node created earlier – call it r' – is chosen as the representative. The other node is added to the equivalence class of r'. Only r' is allowed to be used an input for other nodes in $\widehat{G}$.

- **Case (b)** There is only one node corresponding to both nodes in $\widehat{G}$. This happens due to additional structural hashing in $\widehat{G}$ as a result of allowing only representatives to have fanout.

Example 2.6 Consider AIGs G and $\widehat{G}$ shown in Figure 2.4; $\widehat{G}$ has been constructed from G. ($\widehat{G}$ is the same AIG as in Figure 2.3.) Observe that t and u are functionally equivalent nodes in G, and they correspond to two different nodes t' and u' in $\widehat{G}$ (case (a) above). On the other hand, although o and n are functionally equivalent nodes in G, both correspond to node o' in $\widehat{G}$ (case (b) above).

Listing 2.1 BUILD-CHOICE-AIG(G)

Input: AIG G
Output: AIG with choice $\widehat{G}$

Fix a topological order on G.

Perform random simulation on G.

Compute simrep[n] for each node n in G (see Section 2.5.2)

$\widehat{G} \leftarrow$ NEW-AIG-WITH-CHOICE()
for each Input node v in G **do**
  $v' \leftarrow$ CREATE-VAR($\widehat{G}$)
  final[v] $\leftarrow v'$
end for

for each And node n in G in topological order **do**

  Let (n_l, i_l) and (n_r, i_r) be input edges of n
  $e'_l \leftarrow$ if $i_l = 0$ then final[n_l] else COMPLEMENT(final[n_l])
  $e'_r \leftarrow$ if $i_r = 0$ then final[n_r] else COMPLEMENT(final[n_r])
  $n' \leftarrow$ CREATE-AND($\widehat{G}, e'_l, e'_r$)

  $r \leftarrow$ simrep[n]

  if $r \neq n$ **then**

    if n and r have identical simulation vectors **then**
      $r' \leftarrow$ final[r]
    else assert n and r have complementary simulation vectors
      $r' =$ COMPLEMENT(final[r])
    end if

    if $r' \neq n'$ **then**
      if SAT-PROVE(r', n') **then**
        SET-CHOICE($\widehat{G}, n', r'$) — make r' the representative of n' in $\widehat{G}$
      else
        Resimulate G with counter-example from the SAT solver
        Update simrep[x] for each node x in G (see Section 2.5.2)
      end if
    end if
  end if

  final[n] $\leftarrow$ repr[n']

end for

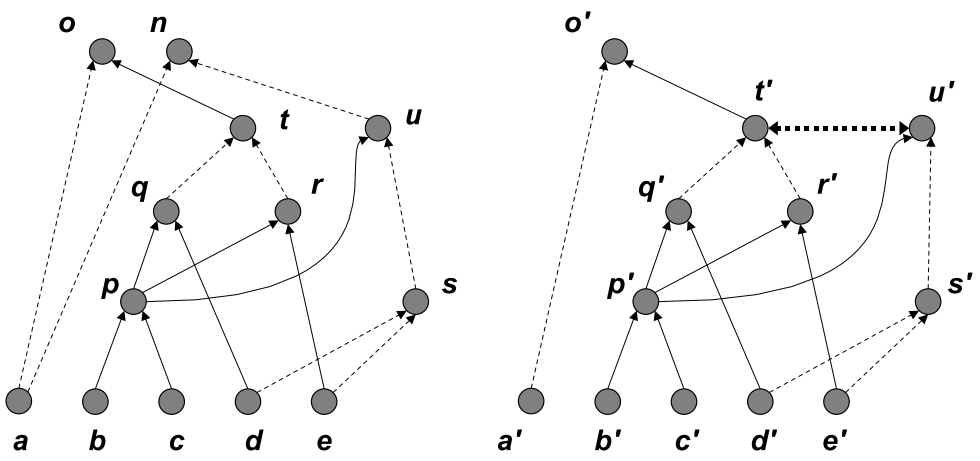


Figure 2.4: Construction of an AIG with choices $\widehat{G}$ (on the right) from an AIG with functionally equivalent nodes G (left). Note that nodes n and o in G are both represented by node o' in $\widehat{G}$ due to additional structural hashing in $\widehat{G}$.

2.5.2 Algorithmic Details

Listing 2.1 shows the pseudocode of BUILD-CHOICE-AIG. First, we fix a topological order on G. Next, we identify *possible* equivalences among nodes in G using random simulation: We assign random input vectors to inputs of G and compute corresponding simulation vectors for each And node.

We then collect nodes with identical simulation vectors (up to complementation) into *simulation equivalence* classes $\mathcal{S}_i$. For each $\mathcal{S}_i$, we select the lowest node (according to the topological order on G) as the representative. Every node n gets marked with its representative simrep[n]. This can be done in one topological pass over G with the help of a hash table.

In the rest of the procedure, we use a map called *final* to keep track of the correspondence between G and $\widehat{G}$. Specifically, for every node n in G that has been processed, final[n] stores an edge of $\widehat{G}$ that computes the same Boolean function in $\widehat{G}$ as n does in G.

We start constructing $\widehat{G}$ by adding input nodes corresponding to those in G. Next, in the main loop of the procedure, we process all And nodes in G in topological order. For an And node n in G, we get the corresponding node n' in $\widehat{G}$ by calling CREATE-AND. (This is always possible since we have already created the corresponding inputs in $\widehat{G}$ as we process G in topological order.) Recall that when CREATE-AND creates a new node n', it sets $\text{repr}[n'] = n'$.

Let $r = \text{simrep}[n]$. If $n = r$ then we know that n is its own simulation representative in G, and there is no earlier[5] functionally equivalent node in G. Therefore, n' becomes the representative (and sole member) of a new functional equivalence class in $\widehat{G}$.

Otherwise, n' and the node corresponding to r in $\widehat{G}$ i.e. $\text{final}[r]$ may be functionally equivalent (up to complementation). Let r' be the appropriate edge of r. We pick r' by seeing if the simulation vectors of n and r are exactly equal, or are complemented. There are two possibilities:

1. $r' = n'$. The two nodes r and n are identified in $\widehat{G}$ by structural equivalence.[6] This corresponds to case (b) in the previous section.

2. $r' \neq n'$. We need to ascertain functional equivalence by invoking the SAT solver. This corresponds to case (a) in the previous section. There are further two possibilities now:

   (a) The SAT solver proves r' functionally equivalent to n'. In this case we designate r' as the representative of n' in $\widehat{G}$ by calling the SET-CHOICE procedure. Recall that it sets $\text{repr}[n']$ equal to r'.

   (b) The SAT solver disproves the equivalence of r' and n'. In this case the SAT solver provides a simulation vector that distinguishes r' and n' in $\widehat{G}$,

[5] according to the topological order

[6] This proves the *functional* equivalence of r and n in G by structural hashing.

and therefore r and n in G. We resimulate G with this vector. This *splits* some simulation classes of G — in particular the class containing nodes r and n. Note that r becomes the simulation representative of the new class formed. (We could resimulate G with some additional vectors to split G even further; for e.g. using the intelligent simulation heuristic [MCBE06].) Since by construction there are no earlier nodes in $\widehat{G}$ with this functionality n' becomes the representative (and sole member) of a new functional equivalence class in $\widehat{G}$.

To complete the processing of node n, we set final$[n]$ equal to the repr$[n']$. This ensures that trivial equivalences in the fanout of n will be detected by structural hashing (as in the example of the previous section).

Observe that an Input node in $\widehat{G}$ is the representative of the equivalence class containing it (in accordance with the assumption made in Section 2.4.1).

Example 2.7 Consider AIG G on the left in Figure 2.4. Suppose we fix a topological order on G such that t is before u. Then applying procedure BUILD-CHOICE-AIG to G leads to the construction of AIG $\widehat{G}$ with choices shown on the right of the same figure.

2.5.3 Proving Equivalence with a SAT Solver

For completeness, we describe how equivalence checking is done with a SAT solver. We assume access to an incremental solver such as MiniSAT [ES03] that:

1. allows addition of clauses without re-starting the solver (to reuse learnt clauses across multiple runs) and,

2. allows unit assumptions (to restrict the search space of satisfying solutions on a run-by-run basis).

The equivalence checking problem is converted to a satisfiability problem in the standard manner. Suppose we are given an AIG with choice $\widehat{G}$, and we want to check the equivalence of two edges x and y in $\widehat{G}$. Every node n in $\widehat{G}$ is assigned a variable $\nu(n)$ in the SAT formulation. An edge $e = (n, i)$ in $\widehat{G}$ is assigned a literal $\nu(e)$ in the natural manner:

$$\nu(e) = \begin{cases} \nu(n) & \text{if } i = 0 \\ \neg\nu(n) & \text{if } i = 1 \end{cases}$$

For every And node n in $\widehat{G}$ with inputs e_l and e_r, three clauses are added to the SAT problem:

$$(\neg\nu(e) + \nu(e_l)) \qquad (\neg\nu(e) + \nu(e_r)) \qquad (\nu(e) + \neg\nu(e_l) + \neg\nu(e_r))$$

where e is the edge $(n, 0)$. If z is the Zero edge in $\widehat{G}$, the clause $(\neg\nu(z))$ is also added to the SAT problem. Call the set of clauses $\mathcal{C}$. The conjunction of clauses in $\mathcal{C}$ is a Boolean function which is true exactly for all valid assignments of values to the nodes of the AIG.

To relate this discussion with the procedure BUILD-CHOICE-AIG, we note that each time SAT-PROVE is called, it does not construct the SAT problem from scratch. It adds new clauses to $\mathcal{C}$ for those And nodes which were added to the AIG since the last call.

To check equivalence of edges x and y, SAT-PROVE runs the solver twice. First, under the unit assumptions $\nu(x)$ and $\neg\nu(y)$, and second, under the assumptions $\neg\nu(x)$ and $\nu(y)$.[7] If either problem is satisfiable, then the solver returns an assignment that distinguishes x and y, and they are not equivalent. The values of the variables corresponding to the Input nodes in the assignment provide an input vector to distinguish

[7] We solve two SAT instances instead of the one instance that asserts the equivalence of x and y in order to adhere to the MiniSAT-like incremental interface outlined at the beginning of this section.

x and y by simulation. On the other hand, if both problems are unsatisfiable, then x and y always take on same values, and are functionally equivalent.

2.5.4 Preventing Cycles in Choices

Given two nodes n and m, $n \neq m$ in an AIG with choice $\widehat{G}$, n *depends* on m, denoted as $m \rightsquigarrow n$, if either

- there is an arc from m to n in $\widehat{G}$, or
- the representative of m is n.

Let $\overset{\mathrm{T}}{\rightsquigarrow}$ be the transitive closure of $\rightsquigarrow$. If $m \overset{\mathrm{T}}{\rightsquigarrow} n$, then m is said to be a pre-requisite of n. Note that the notion of a pre-requisite generalizes the notion of transitive fanin by taking into account choices.

If the AIG with choice $\widehat{G}$ is redundant, it is possible that the functional representative r of a node n is a pre-requisite of n i.e. there is a cycle in the $\overset{\mathrm{T}}{\rightsquigarrow}$ relation (and therefore in the $\rightsquigarrow$ relation as well).

Example 2.8 Consider the AIG G shown in Figure 2.5. Nodes r and t both compute $\mathbb{X}_a \cdot \mathbb{X}_b \cdot \mathbb{X}_c$ and are functionally equivalent. When we run procedure BUILD-CHOICE-AIG, we obtain AIG $\widehat{G}$ where node r' is the representative of node t' (since r is before t in topological order), and therefore $t' \overset{\mathrm{T}}{\rightsquigarrow} r'$. Since r' is also in the transitive fanin of t', we also have $r' \overset{\mathrm{T}}{\rightsquigarrow} t'$. Therefore, there is a directed cycle in the $\overset{\mathrm{T}}{\rightsquigarrow}$ relation.

For the subsequent steps of technology mapping (especially cut computation), it is preferable to not have cycles in the $\overset{\mathrm{T}}{\rightsquigarrow}$ relation. We modify SET-CHOICE to prevent such cycles. If SET-CHOICE is called with node n and an edge $e = (r, i)$, it needs to check if $r \overset{\mathrm{T}}{\rightsquigarrow} n$. Since the representation of $\widehat{G}$ directly provides the $\rightsquigarrow$ relation,[8] we can easily check if $r \overset{\mathrm{T}}{\rightsquigarrow} n$ by a depth first search on the graph of $\rightsquigarrow$.

[8] Recall that a representative node points to the list of nodes in its functional equivalence class.

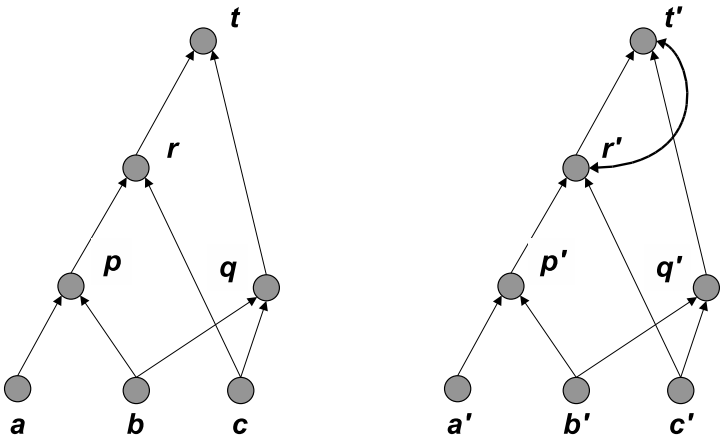


Figure 2.5: When procedure BUILD-CHOICE-AIG is run on the AIG G on the left, it produces the AIG with choice $\widehat{G}$ on the right. The depends relation in $\widehat{G}$ has a cycle since representative r' of t' is also a fanin.

If $r \overset{\mathrm{T}}{\rightsquigarrow} n$, we do *not* add n to the equivalence class of r since otherwise a cycle would result. We still set repr$[n]$ to e for subsequent structural hashing detect equivalences in BUILD-CHOICE-AIG.

2.6 Choice Operator

The BUILD-CHOICE-AIG procedure in Section 2.5.2 builds an AIG with choice from a *single* AIG. In practice, however, we would like a procedure that combines *two* different AIGs to create an AIG with choice. Furthermore, to make this compositional,[9] we would like a binary *choice* operator that takes an AIG with choice $\widehat{G}$, and a second AIG G (without choices) and combines them into an AIG with choice $\widehat{G}'$. (Figure 7.1(b) shows an example of how such a choice operator may be used.)

The choice operator can be implemented trivially using the BUILD-CHOICE-AIG procedure. We assume that we are given a correspondence between the Inputs of $\widehat{G}$

[9]We wish to combine multiple AIGs into a single AIG with choice.

and G. Next, we iterate over the And nodes of G in topological order, and construct corresponding And nodes in $\widehat{G}$ using CREATE-AND. Finally, we call BUILD-CHOICE-AIG on $\widehat{G}$ (interpreting it as a ordinary AIG) to construct $\widehat{G}'$.

The thrifty reader may be concerned BUILD-CHOICE-AIG computes more than necessary in this problem: It re-proves all equivalences originally present in $\widehat{G}$ instead of just detecting additional equivalences due to the nodes from G. The inefficiency can be fixed by modifying BUILD-CHOICE-AIG to keep track of known equivalences between nodes in $\widehat{G}$.

Example 2.9 Figure 2.6 shows an example of applying the choice operator on two AIGs. We start with AIG (I) and then add corresponding nodes of AIG (II) by traversing (II) in topological order starting from inputs. This process leads to AIG (III). Observe that new nodes are created for nodes s, u, and n of (II) in AIG (III) since there are no structurally similar nodes in (I). In contrast, no new node corresponding to m is created in (III) since such a node would structurally hash to p. Finally, as seen in Example 2.7, applying BUILD-CHOICE-AIG to (III) produces the AIG with choice (IV).

2.7 Constructing And Inverter Graphs with Choice

2.7.1 Lossless Synthesis

The choice operator of Section 2.6 provides a simple technique to create AIGs with choice from different technology independent (TI) networks. An example of this is the lossless synthesis flow where snapshots are taken during the TI synthesis operations. Each TI network snapshot is converted into an AIG and the choice operator is used to combine these networks. An example of such a flow is shown in Figure 7.1 (b).

An useful alternative technique to generate choices is to take a mapped network,

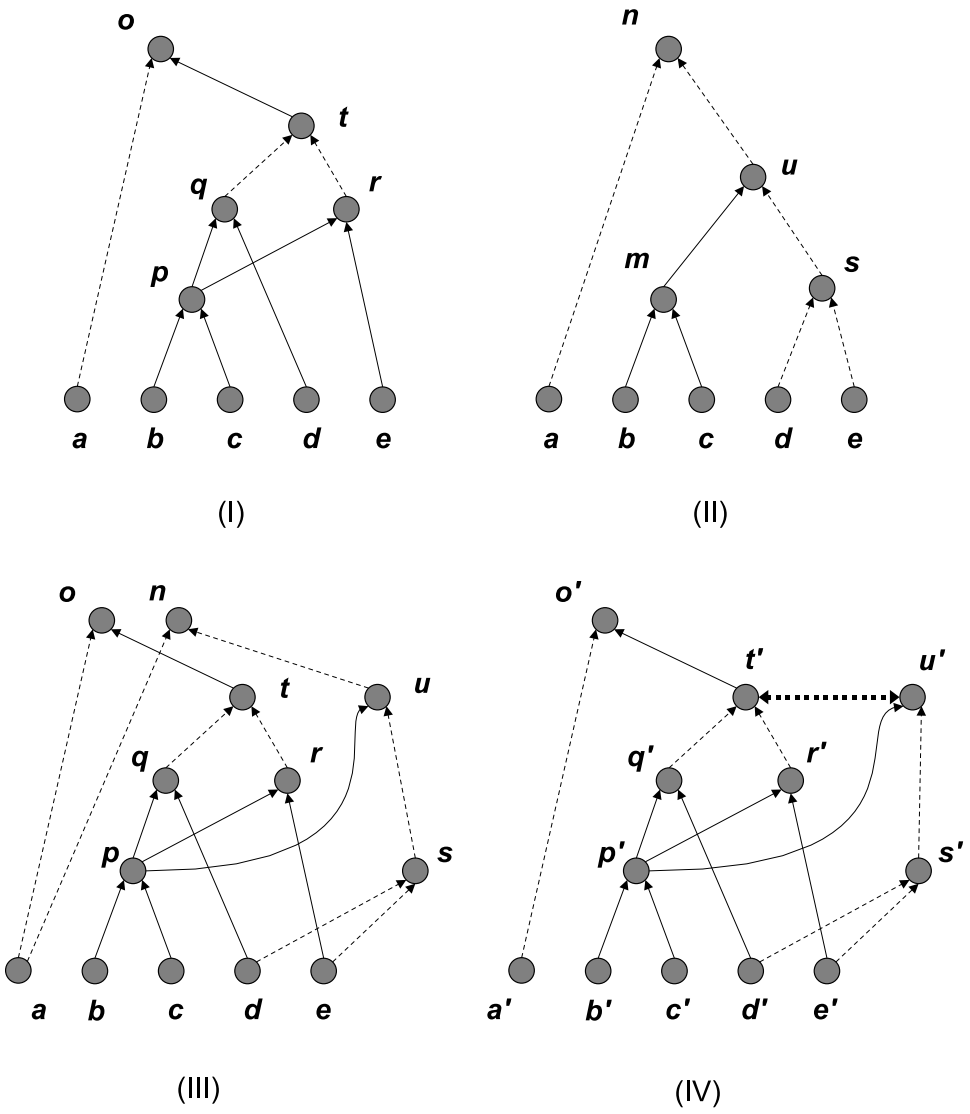


Figure 2.6: An example of using the choice operator to combine the two AIGs (I) and (II) to create an AIG with choice (IV). The AIG (III) is created from (I) and (II) as an intermediate structure on which BUILD-CHOICE-AIG is invoked to construct (IV).

and to construct an AIG from it. This is done by decomposing each library gate (or look-up table) into 2-input AND gates and inverters. The resultant AIG may then be further optimized by technology independent optimizations such as rewriting [MCB06a]. Using the choice operator, this optimized network can be combined with the original AIG with choice used for the mapping. Iterating this process improves the quality of mapping. (If match selection were perfect, then iteration would never make the quality worse since the original matches are always present.) An experiment along these lines is described in Section 7.1.2.

Output Edges There is a subtlety regarding output edges when dealing with AIGs with choice. Recall from Section 2.3 that the output edges of an AIG (without choice) are the edges of the AIG that correspond to the output ports of the TI network from which it was constructed.

Suppose T_1 and T_2 are two functionally equivalent TI networks. Note that they must have the same output ports. Let P be a port of T_1 and T_2. Now, let G_1 be an AIG derived from T_1 and (n_1, i_1) be the output edge in G_1 corresponding to P. Similarly, let G_2 be an AIG derived from T_2, and (n_2, i_2) be the output edge in G_2 corresponding to P.

Now consider an AIG with choice $\widehat{G}$ constructed from G_1 and G_2. In general, the output edges (n_1, i_1) and (n_2, i_2) correspond to different edges in $\widehat{G}$. However, we would like to define a single output edge corresponding to the port P. It is natural to require this single output edge corresponding to port P to be a representative edge.

Note that this is always possible if the choice operator is used to construct the AIG. To see this, observe that n_1 and n_2 must be functionally equivalent up to complementation (otherwise T_1 and T_2 are not equivalent). Therefore, if $\widehat{G}$ is constructed using the choice operator, n_1 and n_2 must be in the same equivalence class. Without loss of generality, suppose n_1 is the representative of that class. Then the edge (n_1, i_1)

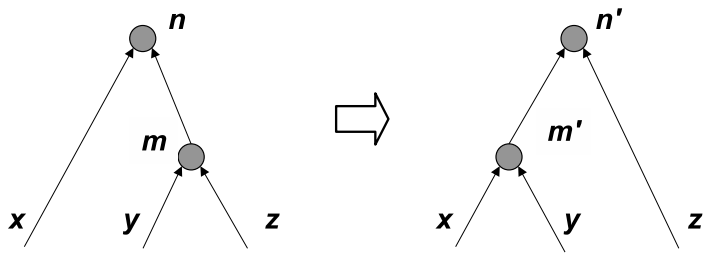


Figure 2.7: An example of a re-write rule (Section 2.7.2).

would be the output edge corresponding to P in $\widehat{G}$.

Once again, although the definition of the output edges depends on the TI networks T_1 and T_2, in what follows we shall simply talk of the output edges of an AIG with choice, trusting that the TI networks are unambiguous. (They are the functionally equivalent networks used to construct the AIG with choice.)

2.7.2 Logic Re-writing

Logic re-writing provides a different way of adding choices. Instead of combining different AIGs to construct an AIG with choice, we directly add choices to an AIG by re-writing the logic.

Lehman *et al.* proposed a way to do this based on algebraic re-writing where the associativity of And is used to add choices [LWGH95]. In their technique, maximal multi-input AND gates[10] are detected in the AIG, by identifying maximal subgraphs that do not contain inverted arcs. Once such a multi-input AND gate is detected, different decompositions of the gate into two-input AND gates are added exhaustively as choices.

Alternatively, the same effect can be obtained by local processing where a local re-writing rule can be used to create choices. Figure 2.7 shows and example of a

[10]In practice the size of the AND gate is limited to 10 inputs.

re-write rule that identifies a form $x \cdot (y \cdot z)$ in the AIG, and introduces a choice by adding nodes corresponding to the form $(x \cdot y) \cdot z$.

More formally, let G be an AIG, and let n and m be nodes in G s.t. the edge $(m, 0)$ is an input of n_1. Let x be the other input of n, and let y and z be the inputs of m. Now, the rewrite rule says that we can add new nodes n' and m' s.t. z and $(m', 0)$ are the inputs of n' and x and y are the inputs of m'. Note that n and n' are functionally equivalent and thus lead to a choice. Similarly, another choice can be created by grouping x and z first.

Iterating this process until no new nodes can be added would lead to the same result as detecting multiple-input AND gates and adding all decompositions. Note that this can be accomplished rather simply (though inefficiently) in our framework by simply adding the new nodes (using CREATE-AND) for the decompositions to an AIG (without choice) and relying on Procedure BUILD-CHOICE-AIG (Section 2.5) to detect the choices.

Lehman *et al.* also use the local re-writing approach to add choices corresponding to the distributivity of And over Or. In this case, the AIG fragments that have the form $x \cdot y + x \cdot z$ are identified (from among the nodes in the critical path) and the form $x \cdot (y + z)$ is added as a choice.

In this connection, note that the recent approaches to technology independent optimization using local re-writing [BB04; MCB06a] can also be used to add different Boolean decompositions to create choices. Another technique of adding Boolean choices is discussed in the context of supergates in Section 4.7.

As before, even when choices are added through re-writing, it will be convenient to require output edges to be representative edges.

2.8 Related Work

2.8.1 On AIGs as Subject Graphs

The AIG with choice data structure is an extension of the AIG data structure introduced by Kuehlmann and Krohm [KK97]. We note that we use only 1-level canonization as was proposed in the original paper and detailed in [KPKG02, Section III.A]. We do not use the 2-level canonization scheme that they used later [KPKG02, Section III.B]. (In the context of mapping, two functionally equivalent but structurally different 2-level subnetworks may be seen as generating a choice that would be detected by BUILD-CHOICE-AIG.)

The use of an AIG as a subject graph where inverters are represented implicitly as attributes on arcs is a break with tradition. The prior approaches for standard cell mapping have generally used NAND2/INV or AND2/INV representations where inverters were explicit [Keu87; SSL+92; LWGH95]. This choice was motivated by the use of structural matching. Note that Rudell's inverter pair heuristic (Section 1.5) is designed to increase the number of matches in such an explicit representation.

In our approach, the use of implicit inverters leads to a couple of advantages:

1. In our approach, matching in both polarities is done, not by using Rudell's heuristic, but by incorporating it into the Boolean matching procedure as will be explained in Chapter 4. The primary advantage of this over an explicit representation is that the cut computation (to discussed in Chapter 3) is faster since there are fewer cuts.[11] The impact of additional cuts due to explicit inverters on the run-time of cut computation is significant since more cut pairs have to be tried to determine the feasible cuts.

[11] A cut with k inputs in the representation with implicit inverters will correspond to 2^k cuts in a representation with explicit inverters.

2. When choices are introduced, keeping inverters implicit allows us to avoid cycles to assert equivalence of nodes that are equivalent upto complementation. This simplifies the implementation and the theory compared to the approach of Lehman *et al.* and Chen and Cong. (For example, see Section 3.5 of [CC01].)

For completeness, we note here that FPGA mappers have generally allowed arbitrary 2-input gates since inverters in a mapped FPGA network are "free" [CD96].

2.8.2 On Detecting Choices

Lehman *et al.* do not describe how choices are detected in their system, but given the small size of their benchmarks, they may have constructed global BDDs. Chen and Cong used BDDs to detect choices, and consequently were unable to process some small circuits (such as C6288, a multiplier) since global BDDs could not be built for those circuits [CC01, Section 5].

In contrast, choice detection in our approach is based on a combination of random simulation and SAT. Methods based on this combination have been shown to be robust and scalable in the combinational equivalence checking literature [Bra93; Kun93; GPB01; LWCH03]. Consequently, our choice detection procedure BUILD-CHOICE-AIG (Section 2.5) is also robust and scalable. (In particular, there is no problem with C6288 as the experiments in Section 7.2.3 show.)

Procedure BUILD-CHOICE-AIG for detecting choices, as presented in this thesis, is based on Kuehlmann's SAT-sweeping algorithm for combinational equivalence checking [Kue04] with the change that a new AIG with choice is created using the extended set of operations (Section 2.4.2) instead of altering the structure of the original AIG. This is done for simplicity of implementation.

Chapter 3

Cut Computation

3.1 Overview

Given an AIG with choice $\widehat{G}$, the first step of matching is to enumerate cuts of size k or less, called *k-feasible cuts*, in $\widehat{G}$. Intuitively, a k-feasible cut corresponds to a single output subnetwork of $\widehat{G}$ (with k inputs) which may be implemented by a gate or a LUT in the mapped network. For LUT-based FPGA mapping, cut enumeration is the only step for matching: each k-feasible cut can be implemented by a k-LUT[1]. For standard cells, more work is needed to determine if a cut can be implemented by a library gate. This is discussed in Chapter 4. In this chapter, we focus on enumerating all cuts with the goal of developing an efficient enumeration[2] procedure for AIGs with choice.

We begin by defining k-feasible cuts (Section 3.2) and reviewing Pan and Lin's procedure for cut computation for an (ordinary) AIG (Section 3.3). Next, we present some improvements to the basic cut computation procedure:

1. We introduce the notion of *signatures* to efficiently detect dominated cuts during

[1] k depends on the FPGA architecture.
[2] We use the terms "cut enumeration" and "cut computation" interchangeably.

enumeration (Section 3.4). In general, removing dominated cuts improves run-time and reduces memory requirements without affecting mapping results.

2. We show how the function of a cut can be computed during cut enumeration (Section 3.5). In addition to reducing run-time (by avoiding additional traversals for cut function computation) the proposed method is easier to generalize to AIGs with choice.

Finally, we generalize the improved cut computation procedure to work on AIGs with choice (Section 3.6). Since the presence of choices leads to many more cuts, the improvements outlined above play an important role in keeping the run-time of the cut computation procedure reasonable.

We postpone the detailed discussion of related work in the literature to the end of the chapter (Section 3.7).

3.2 k-Feasible cut

Let n be a node in an AIG. A *cut* c of n is a set of nodes in its transitive fan-in such that every path from an Input node to n includes a node in c. A cut c of n is said to be *redundant* if some proper subset of c is also a cut of n. A *k-feasible cut* is an irredundant cut of size k or less.

We consider the set $\{n\}$ to be a k-feasible cut ($k > 1$) for a node n and it is called the *trivial cut* of n.

Example 3.1 Figure 3.1 shows all 3-feasible cuts for each node in the AIG. For example, the set $\{a, b\}$ is a 3-feasible cut of node d, and the set $\{a, d, e\}$ is a 3-feasible cut of node f.

The set $\{a, d, b, c\}$ is an example of a redundant cut of node x since $\{a, b, c\}$ is a 3-feasible cut of x. In this example, no node has a k-feasible cut of size greater than

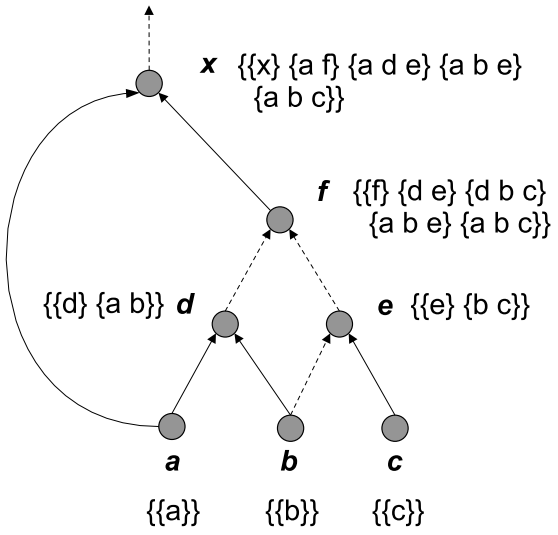


Figure 3.1: An AIG showing the complete set of 3-feasible cuts at each node. All cuts of size 4 or larger are redundant in this example.

3 i.e. any cut with more than 3 nodes is redundant.

3.3 Over-approximate k-Feasible Cut Enumeration

An over-approximation of the set of k-feasible cuts of a node in an AIG can be obtained easily by combining k-feasible cuts of its children.

Let A and B be two sets of cuts. For convenience we define the operation $A \bowtie B$ as follows:

$$A \bowtie B = \{u \cup v \mid u \in A,\ v \in B,\ |u \cup v| \leq k\}$$

Note that $A \bowtie B$ is empty if either A or B is empty.

Let $\Phi(n)$ denote the set of k-feasible cuts of n. If n is an And node, let n_1 and n_2

denote its inputs. We define $\Phi(n)$ as follows:

$$\Phi(n) = \begin{cases} \{\{n\}\} & : \quad n \text{ is an Input node} \\ \{\{n\}\} \cup (\Phi(n_1) \bowtie \Phi(n_2)) & : \quad \text{otherwise} \end{cases} \tag{3.1}$$

Theorem 3.3.1. *$\Phi(n)$ is an over-approximation of the set of k-feasible cuts of node n.*

Proof. By induction. If n is an Input node, the only k-feasible cut is $\{n\}$, and the theorem holds trivially. Suppose n is an And node with inputs n_1 and n_2. By inductive hypothesis assume that $\Phi(n_1)$ and $\Phi(n_2)$ are over approximations of the set of k-feasible cuts of n_1 and n_2 respectively.

Let c be a cut of n. By definition of a k-feasible cut, every path p from an Input node to n passes through a node in c. Since p must pass through n_1 or n_2 to reach n, c induces k-feasible cuts c_1 and c_2 of n_1 and n_2 respectively such that $c_1 \subseteq c$ and $c_2 \subseteq c$. By inductive hypothesis $c_1 \in \Phi(n_1)$ and $c_2 \in \Phi(n_2)$, which implies $c \in \Phi(n)$. □

To see why $\Phi(n)$ is an over-approximation of the set of k-feasible cuts, consider 4-feasible cut computation for the AIG in Figure 3.1. For $n \in \{a, b, c, d, e, f\}$ it is easy to check that $\Phi(n)$ does not contain redundant cuts. Therefore, $\Phi(n)$ is the same as the set of 4-feasible cuts. Now consider

$$\Phi(x) = \{\{x\}\} \cup (\Phi(a) \bowtie \Phi(f))$$

Observe that $\{a\} \in \Phi(a)$. Now since $\{d, b, c\} \in \Phi(f)$, $\{a, d, b, c\} \in \Phi(x)$. On the other hand, since $\{a, b, c\} \in \Phi(f)$, $\{a, b, c\}$ also belongs to $\Phi(x)$. This makes $\{a, d, b, c\}$ a redundant cut of x, and hence not 4-feasible.

More generally, in the presence of reconvergence, Equation 3.1 may produce re-

dundant cuts for some nodes. If these redundant cuts are not removed during the bottom-up cut computation, they will generate redundant cuts for nodes in the transitive fanout.

Equation 3.1 immediately provides a bottom-up algorithm for computing all k-feasible cuts of every node in the network. In topological order, starting from the Input nodes, $\Phi(n)$ is computed for each node. The redundant cuts are removed from $\Phi(n)$ before proceeding to the next node (for computational efficiency).

Let n be an And node with children n_1 and n_2. In the bottom-up algorithm for cut computation, let c be obtained by the union of cuts c_1 of n_1 and c_2 of n_2. We call c_1 and c_2 the *parents* of c.

3.4 Implementation Details

3.4.1 Naïve Cross Product Computation

Cuts are represented as ordered arrays of nodes. The set of cuts of a node is stored as a list of cuts. To compute the cuts of an And node with children n_1 and n_2, we look at every pair of cuts (c_1, c_2) such that $c_1 \in \Phi(n_1)$ and $c_2 \in \Phi(n_1)$. If $c = c_1 \cup c_2$ has k or fewer nodes, c is a *candidate* k-feasible cut. Some candidate k-feasible cuts may turn out to be redundant and we would like to discard them. This is discussed in the next section.

Since c_1 and c_2 have their nodes in sorted order already, c can be easily obtained (with its nodes in sorted order) by a variant of merge sort. Note that this simple approach is inefficient since it requires traversing the cuts c_1 and c_2 in order to decide if c is k-feasible or not. Also, since the number of cuts of size k grows as $O(n^k)$, most k-feasible cuts are of size k. Now, two k-feasible cuts can combine to produce another k-feasible cut only if they are identical, and this can be detected cheaply

using hashing.

3.4.2 Signatures

This brings us to the notion of *signature* of a cut. The signature of a cut is a hash function defined as follows:

$$\text{sign(c)} = \bigoplus_{\text{n}\in\text{c}} 2^{\text{id(n) mod M}} \tag{3.2}$$

where we assume that every node in the AIG has an unique integer id denoted by id(n), and M is a constant equal to the word size of the machine (i.e. $M = 32$ on a 32-bit machine). $\oplus$ denotes bit-wise OR. Thus the signature of a cut is a machine word with 1s in bit positions corresponding to the node ids (modulo M).

The signature is computed when the cut is constructed and stored with the cut. Signatures can be computed incrementally since for a cut $c = c_1 \cup c_2$, $\text{sign(c)} = \text{sign}(\text{c}_1) \oplus \text{sign}(\text{c}_2)$.

Example 3.2 Let $M = 8$ for ease of exposition. The cut c_1 with nodes having ids 32, 68, and 69 has $\text{sign}(c_1) = 0011\ 0001_2$. A second cut c_2 with nodes having ids 32, 68, and 70 has $\text{sign}(c_2) = 0101\ 0001_2$. A third cut c_3 with nodes having ids 36, 64, and 69 has $\text{sign}(c_3) = 0011\ 0001_2$. Observe that c_1 and c_3 have the same signature although they are different cuts. This is called *aliasing*.

Equality Check

To determine if two cuts c_1 and c_2 each of size k can produce a k-feasible cut, we first compare $\text{sign}(c_1)$ and $\text{sign}(c_2)$. If they are different (the common case), then the cuts have different nodes, and hence cannot produce a k-feasible cut. If the signatures

are the same (rare case), then we traverse both cuts to check k-feasibility. Thus signatures allow a quick negative test of equality.

Example 3.3 In Example 3.2, since $\text{sign}(c_1) \neq \text{sign}(c_2)$, we can immediately conclude that $c_1 \neq c_2$). On the other hand, the fact $\text{sign}(c_1) = \text{sign}(c_3)$ tells us nothing. We have to do a node-by-node comparison to check if c_1 and c_3 are equal or not.

Size Check

Note that the number of bits set to 1 in the signature is a lower bound on the size of the cut since each node sets exactly 1 bit to true in formula 3.2. This can be used to filter out cut pairs (c_1, c_2) that cannot produce k-feasible cuts by computing the number of bits set to 1 in $\text{sign}(c_1) \oplus \text{sign}(c_2)$.

3.4.3 Removing Redundant Cuts

The cross product operation described above may produce some redundant cuts that we would like to detect and discard. Given two cuts c_1 and c_2, if $c_1 \subsetneq c_2$ we say c_1 dominates c_2. Thus if a cut is dominated by another cut, it is redundant.

Signatures provide a quick negative test for domination. Given two cuts c_1 and c_2, if $\text{sign}(c_1) \oplus \text{sign}(c_2) \neq \text{sign}(c_2)$, then c_1 does not dominate c_2. Otherwise, a detailed check is needed.

During the cross product, each candidate cut is compared with all cuts that have been generated so far. If the candidate cut dominates another cut, the dominated cut is removed. If the candidate cut is dominated by another cut, then it is immediately discarded.

Example 3.4 Continuing with Example 3.2, consider a fourth cut c_4 with nodes having ids 32, and 33. Therefore $\text{sign}(c_4) = 0000\ 0011_2$, and $\text{sign}(c_4) \oplus \text{sign}(c_1) =$

$0011\ 0011_2$. Since $0011\ 0011_2 \neq \text{sign}(c_1)$, we can conclude that c_4 does not dominate c_1 without doing a detailed check.

3.4.4 Limiting the Number of Cuts

In spite of removing redundant cuts, it is sometimes necessary to set a hard limit on the number of cuts considered at a node to keep the cut computation scalable. We found a limit of 1000 cuts per node to be adequate for $k = 6$ in the sense that it was rarely reached across a large set of benchmarks.

In this context, it is convenient to store the list of cuts of a node in order of increasing size. During the cross product computation, pairs of cuts smaller than k are tried first; then pairs of cuts where one cut is size k and the other smaller than k, and finally pairs of cuts of size k are tried. During this process if the limit is reached, the remaining pairs are not tried. The reason for this order is that most cuts of a node are of size k, and if one cut of a pair being considered is of size k, then the other cut of the pair must be contained in it for the resultant cut to be a candidate. This is unlikely.

3.5 Cut Function Computation

3.5.1 Definition

Intuitively, the function of a cut c of a node m is the function of node m in terms of the nodes in c.

Formally, let $\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_k$ be distinct Boolean variables. An *assignment* σ_c is a map that assigns a Boolean variable $\mathbb{X}_i$ $(1 \leq i \leq k)$ to each node in a k-feasible cut c. Recall from Section 2.2.4 that each node in the AIG has an unique integer id which induces an order on the nodes of the AIG. This allows us to view a cut as an ordered

set of nodes. An *assignment* is *standard* if the first node in the cut maps to $\mathbb{X}_1$, the second node to $\mathbb{X}_2$ and so on.

The *function* of a cut c of node m under an assignment σ, denoted by $\pi_\sigma(m, c)$ is the Boolean function of m according to the AIG under the assignment (in terms of the $\mathbb{X}_i$). In what follows, if we do not specify an assignment for a cut function, the standard assignment should be assumed.

Note that the function of a trivial cut under the standard assignment is always $\mathbb{X}_1$.

Example 3.5 In the AIG of Figure 3.1, let the node ids be assigned in alphabetical order, i.e.

$$\mathrm{id(a)} < \mathrm{id(b)} < \mathrm{id(c)} < \cdots < \mathrm{id(x)}$$

Consider the cut $c_1 = \{b, c\}$ of e. Let σ_1 be the standard assignment, i.e. $\sigma_1(b) = \mathbb{X}_1$ and $\sigma_1(c) = \mathbb{X}_2$. Therefore,

$$\pi_{\sigma_1}(e, c_1) = \neg\mathbb{X}_1 \cdot \mathbb{X}_2$$

i.e. the function of cut c_1 of e is $\neg\mathbb{X}_1 \cdot \mathbb{X}_2$.

Let σ_1' be the (non-standard) assignment defined by $\sigma_1'(b) = \mathbb{X}_2$ and $\sigma_1'(c) = \mathbb{X}_1$. Therefore,

$$\pi_{\sigma_1'}(e, c_1) = \neg\mathbb{X}_2 \cdot \mathbb{X}_1$$

Example 3.6 As a more complicated example, consider once again the AIG in Figure 3.1 with the same ordering of the nodes as in Example 3.5. Consider the cut $c_2 = \{a, b, e\}$ of node f. Let σ_2 be the standard assignment i.e. $\sigma_2(a) = \mathbb{X}_1$, $\sigma_2(b) = \mathbb{X}_2$, and $\sigma_2(e) = \mathbb{X}_3$. Therefore,

$$\pi_{\sigma_2}(f, c_2) = \neg(\mathbb{X}_1 \cdot \mathbb{X}_2 + \mathbb{X}_3)$$

The subtlety here is that in the AIG node b is an input of node e. They are not independent since when b is 1, e must be 0. Nevertheless, we assign independent variables $\mathbb{X}_2$ and $\mathbb{X}_3$ to b and e respectively, and ignore the edge connecting the two nodes when evaluating the function of the cut. Thus the cut function is in some sense over-defined since we are not using the satisfiability don't cares inherent in the structure of the AIG.

3.5.2 Function Computation

In function computation, the goal is to compute the functions of each cut under the standard assignment. For small values of k, truth tables are a convenient representation of the cut functions. Usually, $k \leq 6$ for standard cell mapping and FPGAs, and so the cut functions can be represented by two machine words on a 32-bit machine.

The naïve method to compute the cut function follows from the definition above. The formal variables $\mathbb{X}_1, \mathbb{X}_2, \mathbb{X}_3, \ldots, \mathbb{X}_k$ are represented by truth tables (bit-strings of length 2^k) as follows:

$$\begin{array}{lcccl}
\mathbb{X}_1 & \mapsto & 0101\ 0101 & \cdots & 0101\ 0101 \\
\mathbb{X}_2 & \mapsto & 0011\ 0011 & \cdots & 0011\ 0011 \\
\mathbb{X}_3 & \mapsto & 0000\ 1111 & \cdots & 0000\ 1111 \\
 & & \vdots & & \\
\mathbb{X}_k & \mapsto & 0000\ 0000 & \cdots & 1111\ 1111
\end{array}$$

A truth table is computed for each node in the volume of the cut in topological order as follows. First, each node in the cut is associated with the truth table of the corresponding formal variable under the standard assignment. The truth table of an And node in the volume of the cut is computed by bit-wise AND and negation from the truth tables of its input nodes. In this manner the cut function is built from the leaves up towards the node.

In practice, we use a recursive procedure that starts from the node, and computes the function of the node from the function of its inputs. The base cases of the recursion are the nodes in the cut whose functions are $\mathbb{X}_1, \mathbb{X}_2, \ldots \mathbb{X}_k$.

Example 3.7 Recall the Examples 3.5 and 3.6 of Section 3.5.1 where we computed the functions for a few cuts under different assignments. Those functions would be represented as follows:

Function	Representation
$\pi_{\sigma_1}(e, c_1) = \neg\mathbb{X}_1 \cdot \mathbb{X}_2$	0010 0010 $\cdots$ 0010 0010
$\pi_{\sigma'_1}(e, c_1) = \neg\mathbb{X}_2 \cdot \mathbb{X}_1$	0100 0100 $\cdots$ 0100 0100
$\pi_{\sigma_2}(e, c_2) = \neg(\mathbb{X}_1 \cdot \mathbb{X}_2 + \mathbb{X}_3)$	1110 0000 $\cdots$ 1110 0000

Note that the naïve method requires traversing a part of the AIG to determine the function of a cut. In order to compute the functions for all the cuts, this method is inefficient since it may process the same part of the AIG many times. A second problem arises in conjunction with choices and is discussed in Section 3.6.4.

3.5.3 Incremental Computation

The redundant computation of the naïve method can be avoided by computing the function of a non-trivial cut from the functions of its parents. This allows the cut function to be computed as the cut is constructed during the bottom-up procedure. We begin with two examples from Figure 3.1. As before, node ids are assumed to be in alphabetical order.

Example 3.8 Consider the construction of cut $c_d = \{a, b\}$ of node d from its parents $c_a = \{a\}$ of node a and $c_b = \{b\}$ of node b. From the AIG, the function of c_d under the standard assignment σ is,

$$\pi_\sigma(d, c_d) = \pi_\sigma(a, c_a) \cdot \pi_\sigma(b, c_b)$$

The naïve method would compute $\pi_\sigma(a, c_a)$ and $\pi_\sigma(b, c_b)$ from scratch by traversing the AIG. In incremental computation we wish to avoid this traversal by computing $\pi_\sigma(a, c_a)$ and $\pi_\sigma(b, c_b)$ from the previously computed functions of c_a and c_b respectively.

Let σ_1 be the standard assignment for c_a. In the incremental setting we have already computed the function of c_a, i.e. $\pi_{\sigma_1}(a, c_a)$. We want to use $\pi_{\sigma_1}(a, c_a)$ to compute $\pi_\sigma(a, c_a)$. This is easy, since under both σ and σ_1, the node a is mapped to the same formal variable ($\mathbb{X}_1$). Therefore,

$$\pi_\sigma(a, c_a) = \mathbb{X}_1 = \pi_{\sigma_1}(a, c_a)$$

In the case of c_b, the situation is more complicated. Let σ_2 be the standard assignment for c_b. Under σ_2, node b maps to $\mathbb{X}_1$, whereas, under σ, node b maps to $\mathbb{X}_2$. Thus $\pi_\sigma(b, c_b) = \mathbb{X}_2$ but $\pi_{\sigma_1}(b, c_b) = \mathbb{X}_1$. In this simple example, it is clear that the desired function $\pi_\sigma(b, c_b)$ is the same as the cut function $\pi_{\sigma_1}(b, c_b)$ except for a substitution of variables, *viz.* $\mathbb{X}_1$ is replaced by $\mathbb{X}_2$.

Example 3.9 As a less trivial example, consider the construction of cut $c_f = \{a, b, c\}$ of node f from its parents $c_d = \{a, b\}$ and $c_e = \{b, c\}$. Let σ be the standard assignment for the cut c_f. From the AIG it is clear that

$$\pi_\sigma(f, c_f) = \neg\pi_\sigma(d, c_d) \cdot \neg\pi_\sigma(e, c_e)$$

The functions of cuts c_d and c_e, however, are computed under *their* standard assignments: σ_1 for c_d and σ_2 for c_e. Note that σ_1 agrees with σ since $\sigma_1(a) = \sigma(a) = \mathbb{X}_1$ and $\sigma_1(b) = \sigma(b) = \mathbb{X}_2$. Therefore one can check from the AIG that,

$$\pi_\sigma(d, c_d) = \mathbb{X}_1 \cdot \mathbb{X}_2 = \pi_{\sigma_1}(d, c_d)$$

However, σ and σ_2 disagree since, in particular, $\sigma(b) = \mathbb{X}_2$ whereas $\sigma_2(b) = \mathbb{X}_1$. Therefore, $\pi_\sigma(e, c_e) \neq \pi_{\sigma_2}(e, c_e)$. Indeed, from the AIG it is easy to check that $\pi_\sigma(e, c_e) = \neg\mathbb{X}_2 \cdot \mathbb{X}_3$ and $\pi_{\sigma_2}(e, c_e) = \neg\mathbb{X}_1 \cdot \mathbb{X}_2$.

Once again, observe that the desired function $\pi_\sigma(e, c_e)$ is the same as the cut function $\pi_{\sigma_2}(e, c_e)$ except for a change of variables, *viz.* $\mathbb{X}_1$ is replaced by $\mathbb{X}_2$, and, $\mathbb{X}_2$ by $\mathbb{X}_3$.

More generally, let n be an And node with children n_1 and n_2. Let c be a cut of n, c_1 a cut of n_1, and c_2 a cut of n_2 such that c_1 and c_2 are parents of c i.e. $c = c_1 \cup c_2$. Let σ_1 and σ_2 be the standard assignments on c_1 and c_2 respectively. We are given the functions $\pi_1 = \pi_{\sigma_1}(n_1, c_1)$ and $\pi_2 = \pi_{\sigma_2}(n_2, c_2)$, and asked to compute $\pi = \pi_\sigma(n, c)$.

From the above examples, it is clear that the main complication is due to the fact that π_1 and π_2 are computed under assignments σ_1 and σ_2 which may differ from the desired assignment σ for π. To rectify this, we need to re-express π_1 and π_2 in terms of the assignment σ corresponding to π. Call these π_1' and π_2', i.e. $\pi_1' = \pi_\sigma(n_1, c_1)$ and $\pi_2' = \pi_\sigma(n_2, c_2)$.

The algorithm to compute π given the cuts c, c_1 and c_2 is shown in Listing 3.1. It begins by computing π_1' by iterating over the nodes of c_1. For a node m, the Boolean variable $\sigma_1(m)$ is replaced by the Boolean variable $\sigma(m)$ using Shannon decomposition. The only subtlety is that the last variable (i.e. the variable corresponding to the node with highest id) under the assignment σ_1 is replaced first, the second last variable next, and so on. (The π_1 obtained at the end of this iteration is π_1', the function of node n_1 in terms of the cut c_1 under the assignment σ.) Similarly, the next loop computes π_2'.

Once π_1' and π_2' are computed, it is straightforward to compute the function of node n under the assignment σ.

Listing 3.1 INCREMENTAL-CUT-FUNCTION(n, c, c_1, c_2)

Input: Cut c of node n with input edges (n_1, i_1) and (n_2, i_2); cut c_1 of n_1 and cut c_2 of n_2 where $c = c_1 \cup c_2$

Output: The cut function π of c

Let π_1 be the function of c_1 and σ_1 the standard assignment of c_1

for each node m in c_1 in descending order by id **do**

assert $\sigma_1(m) = \sigma(m)$ or π_1 is independent of $\sigma(m)$

$\pi^+ \leftarrow$ pos cofactor of π_1 w.r.t. $\sigma_1(m)$
$\pi^- \leftarrow$ neg cofactor of π_1 w.r.t. $\sigma_1(m)$
$\pi_1 \leftarrow \sigma(m) \cdot \pi^+ + \neg\sigma(m) \cdot \pi^-$

end for

Let π_2 be the function of c_2 and σ_2 the standard assignment of c_2

for each node m in c_2 in descending order by id **do**

assert $\sigma_2(m) = \sigma(m)$ or π_2 is independent of $\sigma(m)$

$\pi^+ \leftarrow$ pos cofactor of π_2 w.r.t. $\sigma_2(m)$
$\pi^- \leftarrow$ neg cofactor of π_2 w.r.t. $\sigma_2(m)$
$\pi_2 \leftarrow \sigma(m) \cdot \pi^+ + \neg\sigma(m) \cdot \pi^-$

end for

if $(i_1 = 1)$ **then** $\pi_1 \leftarrow \neg\pi_1$ **endif**
if $(i_2 = 1)$ **then** $\pi_2 \leftarrow \neg\pi_2$ **endif**

$\pi \leftarrow \pi_1 \cdot \pi_2$

3.6 Cut Computation with Choices

3.6.1 Overview

In this section we extend cut computation to an AIG with choice $(G, \simeq, \mathcal{X})$. An AIG with choice may be seen as a compact encoding of a set $\mathcal{G}$ of AIGs (without choice). Now pick a particular AIG $H \in \mathcal{G}$ and compute k-feasible cuts for each node in H. Imagine collecting all cuts for a node $n \in G$ from all the individual cut computations over the elements in $\mathcal{G}$. Intuitively, this corresponds to the set of cuts of n in the AIG with choice. In this section, we show how this set of cuts can be computed directly from the AIG with choice without explicitly generating $\mathcal{G}$.

3.6.2 Algorithm

Let $\widehat{G} = (G, \simeq, \mathcal{X})$ be an AIG with choice such that the $\rightsquigarrow$ relation (Section 2.5.4) does not have a cycle. Let X be the set of nodes of AIG G. We compute cuts for both equivalence classes in $X_\simeq$ and nodes in X by modifying (3.1). The set of cuts for an equivalence class $N \in X_\simeq$ is

$$\Phi_\simeq(N) = \bigcup_{n \in N} \Phi(n) \tag{3.3}$$

If n is an And node, let N_1 and N_2 be the equivalence classes of its inputs. Then we define

$$\Phi(n) = \begin{cases} \{\{n\}\} & : \ n \text{ is an Input node} \\ \{\{n\}\} \cup (\Phi_\simeq(N_1) \bowtie \Phi_\simeq(N_2)) & : \ n \in \mathcal{X} \text{ and } n \text{ is an And node} \\ \Phi_\simeq(N_1) \bowtie \Phi_\simeq(N_2) & : \ \text{otherwise} \end{cases} \tag{3.4}$$

Note that the trivial cut is added only for nodes that are representatives.[3] This is because only one trivial cut is needed for an equivalence class. A consequence of this is that only representative nodes may belong to a cut.

The acyclicity of the $\rightsquigarrow$ relation is necessary for the mutually recursive definitions of $\Phi(n)$ and $\Phi_{\simeq}(n)$ to be valid. To compute $\Phi(n)$ for every node n in G, we first fix a topological order on the nodes according to the $\rightsquigarrow$ relation using a depth-first traversal. Next, we process every node n in this topological order, and use (3.4) to compute its set of cuts. If n is a representative of an equivalence class N, then we compute $\Phi_{\simeq}(N)$ according to (3.3), and store it at n. Note that at this point $\Phi(m)$ has been computed for all nodes $m \in N$, since we are processing the nodes in topological order by $\rightsquigarrow$.

Example 3.10 As we saw in Example 2.4, every AIG may be viewed as an AIG with choice where each equivalence class N has exactly one member n. In this case, the reader can check that $\Phi(n)$ computed according to equations (3.3) and (3.4) is equal to $\Phi(n)$ computed according to (3.1).

Example 3.11 Figure 3.2 shows the AIG with choices from Example 2.5 annotated with $\Phi(n)$ for every node for $k = 4$. We note that there are cuts that "cut across" equivalence classes. For example, consider the cuts $\{a, p, s\}$ and $\{a, q, r\}$ of node o. The first cut is due to a cut of u ($\{p, s\}$), whereas the second cut is due to a cut of t ($\{q, r\}$).

3.6.3 Implementation Details

When combining the cut sets of the nodes in an equivalence class, there may be duplicates. Therefore, signatures can be used to detect and remove duplicates and dominated cuts as described in Section 3.4.2.

[3] Recall from Section 2.4.1 that Input nodes are assumed to be representatives.

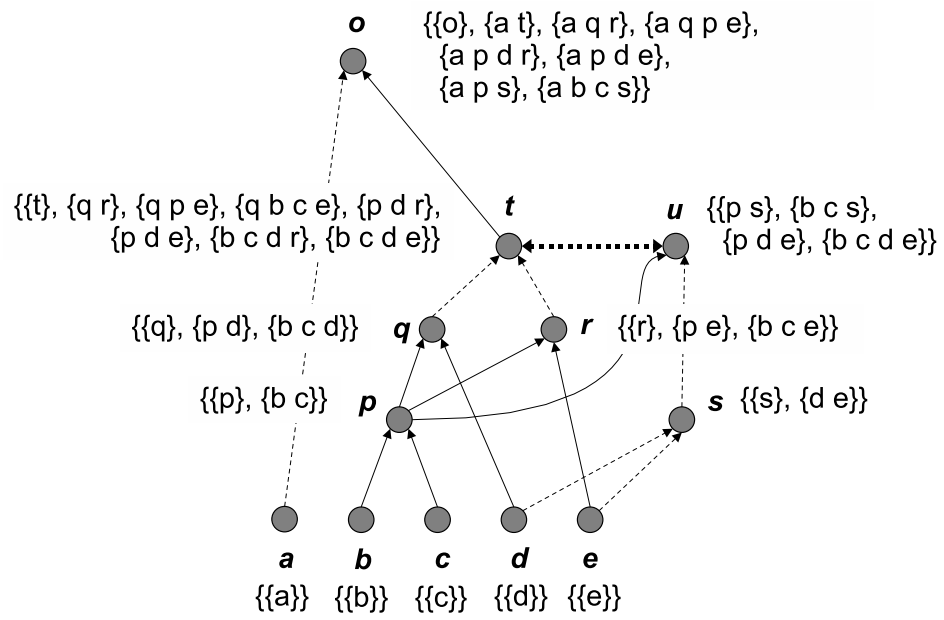


Figure 3.2: An AIG with choice annotated with $\Phi(n)$ for every node n for $k = 4$. In this AIG, all equivalence classes other than $\{t, u\}$ are trivial, and are of the form $\{n\}$ and $\Phi_{\simeq}(\{n\}) = \Phi(n)$. Since $\Phi_{\simeq}(\{t, u\}) = \Phi(t) \cup \Phi(u)$, we have $\Phi(o) = \{o\} \cup (\{a\} \bowtie (\Phi(t) \cup \Phi(u))$.

Example 3.12 In Example 3.11, the set $\{b, c, d, e\}$ is a cut of both nodes t and u. Therefore, when the cuts sets of t and u are merged to construct $\Phi_{\simeq}(\{t, u\})$, this duplication can be detected and eliminated.

We also note that once $\Phi_{\simeq}(N)$ is computed for an equivalence class N, there is no further need to store $\Phi(n)$ for $n \in N$ and this memory can be re-used. (In the subsequent steps of the mapping procedure, we deal only with cuts of equivalence classes.)

Furthermore, as discussed in Section 3.4.4, it is necessary to limit the number of cuts at an equivalence class to keep the computation scalable.

3.6.4 Incremental Cut Function Computation

As seen in Example 3.11 a cut may "cross" an equivalence class. This adds to the inefficiency of the naïve approach to cut function computation since additional information has to be stored with a cut to pick the correct member of an equivalence class during the recursive traversal described in Section 3.5.2.

On the other hand, the incremental function computation of Section 3.5.3 requires little modification to handle choices. Recall that in the incremental function computation, the function of a cut c of an And node n is computed from functions of its parent cuts c_1 and c_2. c_1 and c_2 are cuts of the input nodes n_1 and n_2 of n respectively.

In an AIG with choice, the function of a cut c of node n is computed from its parents c_1 and c_2 as before. It is possible, however, that c_1 and c_2 are not cuts of the actual inputs n_1 and n_2 of n, but are cuts of nodes n_1' and n_2' that are only functionally equivalent up to complementation with n_1 and n_2. In particular, n_1 and n_1' could be complements of each other; similarly n_2 and n_2'.

Example 3.13 In Example 3.11, the parents of cut $\{a, p, s\}$ of node o are the cut $\{a\}$ of node a and the cut $\{p, s\}$ of node u. Although a is an input of node o, u is not an input of o. Indeed, node u is the complement of t which is an input of o.

Therefore, when computing the function of a cut from its parents, we need to possibly complement the cut function of the parent. In practice, this is done when combining the cut sets of the members of an equivalence class N. Let r be the representative of N. If $n \in N$ and $f(n) = \neg f(r)$ (f is the valuation function from Section 2.2.2), then we invert the functions of the cuts of n when adding the cuts to $\Phi_{\simeq}(N)$.

Example 3.14 In Example 3.11, suppose the node ids are ordered alphabetically with the exception of o. o has the highest id. The function of cut $\{p, s\}$ of u is

$\mathbb{X}_1 \cdot \neg\mathbb{X}_2$. Since t is the representative of $\{t, u\}$, and $f(u) = \neg f(t)$, when adding $\{p, s\}$ to $\Phi_{\simeq}(\{t, u\})$, we invert its function to get the new function $\neg(\mathbb{X}_1 \cdot \neg\mathbb{X}_2)$.

To compute the function of cut $\{a, p, s\}$ of o, from the function of cut $\{a\}$ (of the equivalence class $\{a\}$) and cut $\{p, s\}$ (of the equivalence class $\{t, u\}$), we first express the new function of $\{p, s\}$ in the standard assignment of $\{a, p, s\}$ to get $\neg(\mathbb{X}_2 \cdot \neg\mathbb{X}_3)$. The function of cut $\{a\}$ in the standard assignment of $\{a, p, s\}$ is $\mathbb{X}_1$. Now, as before, we combine these functions to obtain the function $\neg\mathbb{X}_1 \cdot \neg(\mathbb{X}_2 \cdot \neg\mathbb{X}_3)$ as the function of $\{a, p, s\}$ of o.

Compare this with the construction of cut $\{a, q, r\}$ of o from cuts $\{a\}$ (of the equivalence class $\{a\}$) and $\{q, r\}$ (of the equivalence class $\{t, u\}$). Since $\{q, r\}$ is a cut of t which is also the representative, we do *not* invert the function of $\{q, r\}$ when adding it to $\Phi_{\simeq}(\{t, u\})$. Thus the function of $\{q, r\}$ remains $\neg\mathbb{X}_1 \cdot \neg\mathbb{X}_2$. Expressing this in the standard assignment of $\{a, q, r\}$, the function obtained is $\neg\mathbb{X}_2 \cdot \neg\mathbb{X}_3$. Thus we obtain the function $\neg\mathbb{X}_1 \cdot \neg\mathbb{X}_2 \cdot \neg\mathbb{X}_3$ as the function of $\{a, q, r\}$.

3.7 Related Work

The basic cut enumeration procedure we use (Section 3.3) is based on the simple algorithm proposed by Pan and Lin [PL98] which was a significant improvement on the previous algorithm based on spanning trees [CD93]. Our contribution here is the efficient detection and removal of redundant cuts during the cut computation using signatures (Section 3.4.2). Conceptually, the use of signatures is similar to the use of Bloom filters for encoding sets [Blo70] and to the use of signatures for comparing CNF clauses for subsumption in the context of SAT solving [EB05].

The previous work on standard cell mapping using Boolean matching did not use cuts, but instead relied on computing *clusters* [MM90; SSBSV92]. Clusters are like cuts in that they correspond to single-output sub-graphs of the subject graph, and are

represented by their input nodes. Clusters of a node r are enumerated in a top-down fashion starting with the trivial cluster containing only r. New clusters are obtained by expanding existing clusters. Expansion is done by removing an input node of the cluster and adding its fanins to the cluster. If in this process an existing cluster is obtained, or the number of inputs exceeds a limit k, then the expanded cluster is simply discarded.

Thus clusters of a node r computed with a limit of k input nodes form a subset of the set of k-feasible cuts of r (i.e. every cluster is a cut). However, the top-down approach means that certain deep k-feasible cuts of r may be missed if the circuit is re-convergent. Indeed, such cuts may be the best kind for mapping since they are deep and absorb reconvergences.

The use of the cut-based approach for standard cell mapping motivated the need for efficient cut function computation described in Section 3.5. In FPGA mapping, cut function computation need not be efficient since only the functions of the cuts that are finally selected in the mapped network need to be computed. However, for Boolean matching in standard cells, the functions of all the cuts are needed.

Finally, we point out some recent attempts at improving cut computation by reducing the resources required for enumeration. The work on factor cuts provides a technique to avoid full cut enumeration by combining bottom-up cut computation with top-down expansion [CMB06a]. The work on priority cuts takes this idea further where only a few cuts are computed during mapping [MCCB07]. A different approach is taken with BDDcut where BDDs are used to store cuts [LZB07]. This reduces the memory required for storing cuts by sharing common subcuts. The main benefits of these three approaches are obtained for large values of k (say 8 or larger). So far, the use of these improved cut enumeration ideas has been mainly for LUT-based FPGA mapping for large LUTs. Adapting these ideas to standard cell mapping or mapping of macrocells in FPGAs which more constrained than FPGA mapping appears to be

an interesting problem.

Chapter 4

Boolean Matching and Supergates

4.1 Overview

In this chapter we present our simplified Boolean matching algorithm to match k-feasible cuts with library gates for standard cell mapping.[1] To the reader familiar with Boolean matching algorithms, the simplified algorithm presented here may appear unorthodox: Rather than use $\mathcal{NPN}$-canonization, we enumerate *configurations* which are essentially different Boolean functions that a library gate can realize under permutations and negations. To address the reader's atavistic fear of enumeration, we summarize certain advantages of the proposed method:

1. Since most gates used in standard cells mapping usually have a few inputs (say 6 or less),[2] matching based on enumeration is *faster* than canonization. The reason for this is described in Section 4.8.

2. Enumeration of configurations is easier to implement than canonization.

[1] No such technique is needed for FPGAs since a k-LUT can implement any k-feasible cut.

[2] There are very few gates used in practice that have more than 6 inputs; these are typically multiplexers (e.g. 8 to 1 MUX) and are usually handled by specialized structural techniques which suffer from local structural bias.

3. This way of looking at the matching problem makes the concept of supergates a natural extension for increasing the set of matches.

We begin with a few preliminary remarks and definitions relating to standard cell libraries and gates (Sections 4.2 and 4.3). Then, we first look at a simplified matching problem using *p-configurations* where we ignore inversions (Section 4.4). Next, we introduce *np-configurations* which generalize p-configurations, and show how np-configurations lead to more matches (Section 4.5). Finally, we introduce supergates as a generalization of np-configurations and show how they can further increase the number of matches (Section 4.6). We also briefly describe the connection between supergates and choices (Section 4.7).

We postpone the detailed discussion of related work in the literature to the end of the chapter (Section 4.8).

4.2 Preliminary Remarks

In standard cell designs, the combinational logic is implemented by gates that have fixed functions. Thus we have a library of different gate *types* such as INV (inverter), NAND2 (two input NAND gate), AND3 (three input AND gate), MUX41 (4-1 multiplexer), etc. Typically each type of gate is actually a family of gates with different electrical properties and timing characteristics. For instance there may be several different AND3 type gates with different drive strengths, and different skews (i.e. input pin to output pin delay characteristics).

For the purposes of matching, only the (logical) functionality of the gate matters and not its electrical and timing characteristics. Therefore, as a first step we group gates into functional families. This is described in more detail in Section 4.3.3.

The gates commonly used in practice to implement combinational logic have 6 or fewer inputs. The Boolean matching technique described in this chapter is particu-

larly well-suited for matching such gates. For gates with substantially more inputs, the method considered here is inadequate. Furthermore, the gates used to implement combinational logic usually have only one output. Multi-output gates for combinational logic are rare except for an important case: half-adder cells. These are typically used on data-path circuits such as adders and multipliers. The method considered here may be extended to handle such cells, but in what follows we restrict our attention to single-output gates.

4.3 The Function of a Gate

4.3.1 Functional Expression

The functionality of a (single-output) gate g is typically provided by an expression[3] in terms of its input pins, called the *functional expression* of g and denoted by $\text{expr}(g)$. For instance, for an NAND2 with input pins a and b, could have $\text{expr}(g)$ equal to `not`$(`$`and`$(a, b))$. An AOI21 gate (a three input And-Or-Invert gate) with inputs $a$, $b$ and $c$ may have $\text{expr}(g)$ equal to `not`$(`$`or`$(`$`and`$(a, b), c))$.

Formally, we assume that the functional expression of a gate g is generated by a grammar such as:

$$\texttt{<expr>} ::= p \mid \texttt{not(<expr>)} \mid \texttt{and(<expr>, <expr>)} \mid \texttt{or(<expr>, <expr>)}$$

where p ranges over the names of the input pins of g. Now, if μ is a function that maps the name of each input pin p of g to a Boolean function $\mu(p)$, then we define

[3] We distinguish expressions from functions here for reasons that will shortly become clear.

the *function* of an expression e *under* μ, denoted by $\pi(e, \mu)$ as follows:

$$\begin{aligned} \pi(p, \mu) &= \mu(p) \\ \pi(\texttt{not}(e), \mu) &= \neg\pi(e, \mu) \\ \pi(\texttt{and}(e_1, e_2), \mu) &= \pi(e_1, \mu) \cdot \pi(e_2, \mu) \\ \pi(\texttt{or}(e_1, e_2), \mu) &= \pi(e_1, \mu) + \pi(e_2, \mu) \end{aligned}$$

We shall use this general notion of the function of an expression under a map μ to define the functions of different structures associated with gates.

4.3.2 Function under Assignment

The function of a gate g is defined in a manner analogous to that of the function of a cut (Section 3.5.1). Suppose g has k inputs. Let $\mathbb{X}_1, \mathbb{X}_2, \ldots, \mathbb{X}_k$ be distinct Boolean variables. An *assignment* σ is a bijective map that assigns a Boolean variable $\mathbb{X}_i$ $(1 \leq i \leq k)$ to each input pin of a gate. The *function* of a gate g under the assignment σ, denoted by $\pi_\sigma(g)$, is simply $\pi(\text{expr}(g), \sigma)$.

Example 4.1 Let g be the NAND2 gate with inputs a and b, and $\text{expr}(g) = \texttt{not}(\texttt{and}(a, b))$. Let σ_1 be the assignment that maps a to $\mathbb{X}_1$ and b to $\mathbb{X}_2$. Therefore,

$$\pi_{\sigma_1}(g) = \neg(\mathbb{X}_1 \cdot \mathbb{X}_2)$$

Under a different assignment σ_2 that maps a to $\mathbb{X}_2$ and b to $\mathbb{X}_1$, we have,

$$\pi_{\sigma_2}(g) = \neg(\mathbb{X}_2 \cdot \mathbb{X}_1)$$

In the above example, we obtained the same *function* for both assignments σ_1 and σ_2 due to the symmetry of the NAND2 gate. This is not true in general as the following example shows.

Example 4.2 Let h be the AOI21 gate with inputs a, b and c, and $\text{expr}(h) = \texttt{not}(\texttt{or}(\texttt{and}(a,\ b),\ c))$. Let σ_1 be the assignment that maps a to $\mathbb{X}_1$, b to $\mathbb{X}_2$ and c to $\mathbb{X}_3$. Therefore,

$$\pi_{\sigma_1}(h) = \neg((\mathbb{X}_1 \cdot \mathbb{X}_2) + \mathbb{X}_3)$$

Under a different assignment σ_2 that maps a to $\mathbb{X}_2$, b to $\mathbb{X}_3$ and c to $\mathbb{X}_1$, we have,

$$\pi_{\sigma_2}(h) = \neg((\mathbb{X}_2 \cdot \mathbb{X}_3) + \mathbb{X}_1)$$

Now, $\pi_{\sigma_2}(h) \neq \pi_{\sigma_1}(h)$.

As the above example shows, the function of a gate is *not* unique: it depends on the assignment. (This is why we distinguished the function of a gate from the expression.) This is similar to the function of a cut which is also not unique but depends on the assignment used.

4.3.3 Implementation Details

In the implementation, the gates are read in from a library file. The functional expression of a gate is parsed to get the abstract syntax tree (AST) of the expression. The function of a gate under a specific assignment is obtained directly from the AST. Functions are represented by truth tables as in the case of cuts (Section 3.5.2).

There may be multiple gates with the same functionality, but different timing properties. The matching process is more efficient if such gates are collected together into *functional families* and only one member per functional family is used for matching. Later, during match selection when the timing properties are important, the different members of a functional family are considered.

An easy way to automatically detect such functional families is to compute the function of the gates under a *standard assignment* that maps the first input of the

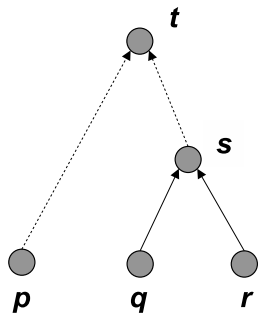


Figure 4.1: The AIG for Example 4.3.

gate to $\mathbb{X}_1$, the second to $\mathbb{X}_2$ and so on. Gates that have the same function under the standard assignment can be grouped together into a functional family.

4.4 Matching Under Permutations

4.4.1 Motivation

The Boolean Matching problem for standard cells asks if a cut c can be implemented with a gate g. Let f be the function of c under the standard assignment. We could use the standard assignment σ_g for g (as defined above) to check for a match. Now, if $\pi_{\sigma_g}(g) = f$, then clearly c can be implemented with g. However, by restricting ourselves to a particular assignment for g, we may not detect a possible match. The following example illustrates this.

Example 4.3 Consider the cut $c = \{p, q, r\}$ of node t in the AIG of Figure 4.1. The node ids are in alphabetical order. Let σ_c be the standard assignment on c. Thus we have,

$$\pi_{\sigma_c}(t, c) = \neg\mathbb{X}_1 \cdot \neg(\mathbb{X}_2 \cdot \mathbb{X}_3)$$

Let h be the AOI21 gate from Example 4.2. Under the assignment that maps the i-th

input to $\mathbb{X}_i$, i.e. the assignment σ_1, we have

$$\pi_{\sigma_1}(h) = \neg((\mathbb{X}_1 \cdot \mathbb{X}_2) + \mathbb{X}_3)$$

Thus g does *not* match c under this assignment. However, under the assignment σ_2, g matches c, since we have,

$$\pi_{\sigma_2}(h) = \neg((\mathbb{X}_2 \cdot \mathbb{X}_3) + \mathbb{X}_1) = \pi_{\sigma_c}(t, c)$$

Therefore, we cannot restrict our attention to the standard assignment for a gate, but need to consider multiple assignments. Formally, we say a gate g matches a cut with function f, if there exists an assignment σ such that $\pi_\sigma(g) = f$. We call this the problem of matching under permutations.

4.4.2 p-configuration, p-match and Validity

If g is a gate, and σ an assignment, the pair (g, σ) is called a *p-configuration*. The *function* of (g, σ) is defined as the function of g under σ. Note that two different p-configurations p_1 and p_2 can have the same function, and they are called functionally equivalent. Now, the problem of matching under permutations reduces to finding a matching p-configuration.

Let f_c be the function of cut c of node n in the AIG. Let (g, σ) be a p-configuration with function f. If $f = f_c$, then the tuple (c, g, σ) is called a *p-match* for node n. For convenience, we define a formal p-match called PI-MATCH that only matches the Input nodes of an AIG.

We define the *support* of a p-match $m = (c, g, \sigma)$ to be c (viewed as a set of nodes). We define support(PI-MATCH) to be the empty set. Now if a match $m \neq$ PI-MATCH is to be used in a mapping, there must be p-matches for the nodes in support(m).

Listing 4.1 BUILD-HASH-TABLE-P($\mathcal{L}$)

Input: Library of gates $\mathcal{L}$
Output: Hash table $\mathcal{H}$

```
for each gate g in L do
  for each assignment σ of g do
    f ← function of g under assignment σ
    P ← LOOKUP(H, f)
    INSERT(H, f, P ∪ {(g, σ)})
  end for
end for
```

This leads us to the notion of validity: A p-match m is *valid* if the set of p-matches for each node $n \in \text{support}(m)$ is non-empty.

4.4.3 Matching Algorithm

We solve the matching problem in a simple brute-force manner. We enumerate all p-configurations corresponding to a gate, by enumerating all possible assignments of the gate. Let g be a gate with k inputs. Since each assignment is a bijection from the set of inputs of the gate to the set $\{\mathbb{X}_1, \mathbb{X}_2, ..\mathbb{X}_k\}$, there are $k!$ different assignments (and therefore $k!$ different p-configurations). Note that enumerating p-configurations is feasible since we restrict ourselves to $k \leq 6$.

The matching procedure is split into two sub-procedures BUILD-HASH-TABLE-P and FIND-P-MATCHES. Procedure BUILD-HASH-TABLE-P runs first and processes the gates in the library $\mathcal{L}$ to construct a hash table $\mathcal{H}$. $\mathcal{H}$ maps a function f to the set of p-configurations whose function is f. The pseudocode is shown in Listing 4.1. Note that this procedure is a pre-computation in the sense that it does not depend on the AIG being mapped. Also, note that the number of keys in $\mathcal{H}$ is quite small in practice since many gates are symmetric and for a symmetric gate, multiple p-configurations

Listing 4.2 FIND-P-MATCHES($\widehat{G}, \mathcal{H}$)

Input: AIG with choice $\widehat{G}$, hash table $\mathcal{H}$ from BUILD-HASH-TABLE-P
Output: match[r] for every representative r in $\widehat{G}$

```
Compute k-feasible cuts Φ_≃(N) for each equivalence class N in Ĝ
for each representative node r in Ĝ in topological order do
  if r is a Input node then
    match[r] ← {PI-MATCH}
  else assert r is an And node
    match[r] ← ∅
    Let N be the equivalence class of r
    for each non-trivial cut c ∈ Φ_≃(N) do
      Let f be the function of c
      for each p-configuration (g, σ) ∈ LOOKUP(H, f) do
        if VALID(c, σ) then match[r] ← match[r] ∪ {(c, g, σ)}
      end for
    end for
  end if
end for
```

have the same function (c.f. Example 4.1).

The second procedure FIND-P-MATCHES uses $\mathcal{H}$ to find matches. Starting with an AIG with choice $\widehat{G}$, it computes the set of k-feasible cuts as described in Section 3.6.2. The value of k is set to the maximum number of inputs of a gate in the library. As noted above, this is typically 5 or 6.

Next it iterates over representative nodes in $\widehat{G}$ to find matches. If a representative r is an Input node, its only match is the PI-MATCH. Otherwise, the procedure iterates over the *non-trivial* cuts of the equivalence class of r. For a cut c, the function of c is used to index into the hash table $\mathcal{H}$ to find the set of p-configurations $\mathcal{P}$ that can

be used to implement the cut. For each p-configuration $(g, \sigma) \in \mathcal{P}$, it checks if the p-match (c, g, σ) is valid. (Recall that a p-match is valid if every node in its support has at least one p-match.) If the match is valid, it is added to the set of matches for r. In this manner, the procedure constructs a list of matches for each representative (and thereby each equivalence class) of $\widehat{G}$.

Remark As an important implementation detail, we note that memory used for storing matches can be significantly reduced by not storing p-matches (as described above) directly; instead, the same information can be represented by storing the pairs $(c, \mathcal{P})$.

4.4.4 Two Remarks on Matching Under Permutations

Remark 1 It would appear that a cut with k nodes can only match a gate with k inputs. However, in practice it is possible that a cut c with k nodes matches a p-configuration (g, σ) where g has *less* than k inputs. This happens with a badly structured AIG where the function of a cut does not depend on all k formal variables. In particular, if the function is independent of the last variable $\mathbb{X}_k$, a match may be found with a gate with fewer than k variables.

The converse situation where a gate with k inputs matches a cut with less than k nodes is also possible if there is a gate whose output expression does not involve all of its input pins (such as tri-state elements). Such gates are usually excluded since they cannot be meaningfully used by the matcher.

Remark 2 The single input gates (inverters and buffers) are never matched (except in the degenerate case described in Remark 1) by procedure FIND-P-MATCHES since it does not consider trivial cuts. Therefore, the matching is not complete in the sense that given a functionally complete library, there is no guarantee that a match will be found for every node in the AIG.

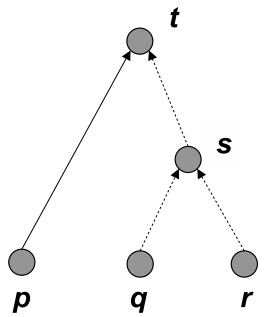


Figure 4.2: The AIG for Example 4.4.

4.5 Matching Under Permutations and Negations

4.5.1 Motivation

Example 4.3 showed that by considering different assignments to the inputs of a gate, we could detect more matches between cuts and gates. Different assignments correspond to different input permutations of a gate. As the following example shows, by considering input negations in addition to permutations, we can detect even more matches.

Example 4.4 Figure 4.2 shows an AIG similar to that of Example 4.3, except that the arcs (p, t), (q, s) and (r, s) are complemented. Suppose our library contains an inverter INV, and the AOI21 gate h as before. It is easy to see that h would *not* match at cut $c = \{p, q, r\}$ of node t under the assignment σ_2 considered before. The function of c under the standard assignment σ_c in this AIG is

$$\pi_{\sigma_c}(t, c) = \mathbb{X}_1 \cdot \neg(\neg\mathbb{X}_2 \cdot \neg\mathbb{X}_3)$$

The function of AOI21 under assignment σ_2 remains, as before,

$$\pi_{\sigma_2}(h) = \neg((\mathbb{X}_2 \cdot \mathbb{X}_3) + \mathbb{X}_1)$$

Therefore, $\pi_{\sigma_c}(t, c) \neq \pi_{\sigma_c}(t, c)$ in this example.

However, since we have an inverter, we could *create* a new gate i comprising the AOI21 gate and an inverter at each input. The function of this *new* gate under the assignment σ_2, is

$$\pi_{\sigma_2}(i) = \neg((\neg\mathbb{X}_2 \cdot \neg\mathbb{X}_3) + \neg\mathbb{X}_1)$$

Now, $\pi_{\sigma_2}(i) = \pi_{\sigma_c}(t, c)$, and this new gate i matches the cut c.

Similarly, by adding an inverter to the output of an existing gate, we may find more matches than otherwise.

The idea of creating new gates by combining gates in the library to improve matching is a powerful one, and we shall explore it more fully in Section 4.6. However, for now, we present a different method to achieve the same effect w.r.t. input and output negations. Instead of creating gates by adding inverters to inputs or outputs, we match each representative node of the AIG in both positive and negative phases. This is equivalent to matching *edges* of the AIG instead of nodes.

Why do we prefer matching both phases over creating new gates with inverters? We discuss this in Section 4.5.5 after the presentation of the matching algorithm.

4.5.2 np-configuration and np-match

Let g be a gate with k inputs, and σ an assignment for g. Let τ be a map that maps an input pin of g to an element of $\{0, 1\}$. Informally, $\tau(i) = 1$ means that input pin i is inverted. The pair (σ, τ) is called a *generalized assignment*. Given a generalized

assignment (σ, τ), define the map μ on the input pins of g as follows:

$$\mu(i) = \begin{cases} \sigma(i) & \text{if } \tau(i) = 0 \\ \neg\sigma(i) & \text{if } \tau(i) = 1 \end{cases}$$

The *function* of a gate g under the generalized assignment (σ, τ) is defined to be $\pi(\text{expr}(g), \mu)$.

Example 4.5 Let h be the AOI21 gate with inputs a, b and c, and $\text{expr}(h) = \texttt{not}(\texttt{or}(\texttt{and}(a,\ b),\ c))$. Let $\sigma = \{a \mapsto \mathbb{X}_2, b \mapsto \mathbb{X}_3, c \mapsto \mathbb{X}_1\}$ and $\tau = \{a \mapsto 1, b \mapsto 0, c \mapsto 1\}$. The function of h under the generalized assignment (σ, τ) is

$$\pi_{(\sigma,\tau)}(h) = \neg((\neg\mathbb{X}_2 \cdot \mathbb{X}_3) + \neg\mathbb{X}_1)$$

Example 4.6 Let i be an inverter with input a. $\text{expr}(i) = \texttt{not}(a)$. There is only one possible assignment for i which is $\sigma = \{a \mapsto \mathbb{X}_1\}$. There are two possible τ maps: $\tau_1 = \{a \mapsto 0\}$ or $\tau_2 = \{a \mapsto 1\}$ Thus we have two possible generalized assignments with the following functions:

$$\pi_{(\sigma,\tau_1)}(i) = \neg\mathbb{X}_1$$

and

$$\pi_{(\sigma,\tau_2)}(i) = \neg\neg\mathbb{X}_1 = \mathbb{X}_1$$

Similarly, a buffer b with input a, $\text{expr}(b) = a$, leads to two possible generalized assignments:

$$\pi_{(\sigma,\tau_1)}(b) = \mathbb{X}_1$$

and

$$\pi_{(\sigma,\tau_2)}(b) = \neg\mathbb{X}_1$$

Now, if g is a gate and (σ, τ) a generalized assignment, the tuple (g, σ, τ) is called a np-configuration. The function of the np-configuration is the function of the g under (σ, τ).

When matching using np-configurations, we associate matches with edges (instead of nodes as in the case of p-configurations). Let f_c be the function of cut c of node n in an AIG. Let (g, σ, τ) be a np-configuration with function f. If $f = f_c$, then the tuple (c, g, σ, τ) is an *np-match* for the edge $(n, 0)$. If $f = \neg f_c$, then (c, g, σ, τ) is an *np-match* for the edge $(n, 1)$. As in the case of p-matches we add a formal match PI-MATCH that only matches Input nodes.

Intuitively, the support of an np-match is the appropriate set of edges obtained from the nodes in the cut and the inversions specified by τ. Let $m = (c, g, \sigma, \tau)$ be an np-match with $|c| = k$. Let i_1, i_2, ..., i_l, $l \leq k$, be inputs pins of g in order. Let $n_1, n_2, .., n_k$ be the sequence of nodes in c such that $\mathrm{id}(n_i) < \mathrm{id}(n_{i+1})$, $1 \leq i < k$. Now, we define support(m) in the following manner:

$$\mathrm{support}(m) = \{(n_1, \tau(i_1)), (n_2, \tau(i_2)), ..., (n_k, \tau(i_k))\}$$

The support of a PI-MATCH is defined to be the empty set.

4.5.3 Validity of an np-match

The notion of validity for an np-match is more complicated than that for p-matches (Section 4.4.2). It is not enough to check that the edges in the support of an np-match have np-matches: Since we now allow single input gates such as inverters and buffers, there could be cycles.

Example 4.7 As an extreme example, suppose we have a degenerate library consisting of only an inverter, and we are asked to map to an AIG containing only a

single And node n (and its two input nodes). For the edge $(n, 0)$, an inverter is the only np-match. Call it m. It is easy to check that support$(m) = \{(n, 1)\}$. Now for edge $(n, 1)$, again, the only np-match is the inverter. Call this match m'. Now, support$(m') = \{(n, 1)\}$. In this case, although the edges in the support of both np-matches also have matches, it is clear that a valid mapped network cannot be constructed, since the matches m and m' are mutually dependent, i.e. form a cycle.

To avoid such cycles stemming from mutually dependent matches due to single input gates, it is convenient to divide np-matches into two groups: *singular* matches and *regular* matches. Informally, singular matches are matches to single input gates such as buffers and inverters. Regular matches are matches to multi-input gates and PI-MATCHes. To avoid cycles, we insist that for a singular match to be valid, the edge in its support must have at least one regular match.

For a regular match to be valid, the edges in its support could have either regular or singular matches. Either way, cycles would not be introduced. However, the practical implementation can be greatly simplified by the following restriction. For a regular match to be valid, the edges in its support must have singular matches. Separately, we ensure that whenever, an edge has a regular match, it automatically gets a singular match. This is done by adding a fake gate to the library called WIRE.

The gate WIRE is *functionally* a buffer, i.e. a single input, single output gate with the identity function. As one might expect, the WIRE gate will be treated specially during match selection (with regard to cost evaluation) since it is not a regular gate.[4]

Remark We want to emphasize that WIRE is added to simplify the implementation of later operations of match selection, especially of match selection using exact area (Section 6.7.2), rather than the matching procedure itself.[5]

[4]For e.g. WIRE transmits the load at the output pin to the input pin as opposed to, say, a buffer gate. This distinction matters during match selection for load-based delay-oriented mapping.

[5]For this reason, WIRE is added to the library even when a buffer (which is functionally equivalent)

Listing 4.3 BUILD-HASH-TABLE-NP($\mathcal{L}$)

Input: Library of gates $\mathcal{L}$
Output: Hash table $\mathcal{H}$

```
Add special gate WIRE to L
for each gate g in L do
  for each assignment σ of g do
    for each phase assignment τ of g do
      f ← function of g under generalized assignment (σ, τ)
      P ← LOOKUP(H, f)
      INSERT(H, f, P ∪ {(g, σ, τ)})
    end for
  end for
end for
```

Formally, PI-MATCH is defined to be regular. An np-match (c, g, σ, τ) is *regular* if $|c| > 1$ and is *singular* if $|c| = 1$. We base these definitions on the size of the cut rather than number of inputs of the gate, due to the rare situation of a single-input gate matching a non-trivial cut as described in Remark 1 of Section 4.4.4. A singular match is *valid* if the edge in its support has at least one regular match. A regular match is *valid* if the every edge in its support has at least one singular match.

4.5.4 Matching Algorithm

The matching algorithm is similar to that for matching under permutations: we generate all np-configurations of a gate. The procedure is split into two sub-procedures. The first sub-procedure BUILD-HASH-TABLE-NP generates all np-configurations from each gate in the library and constructs the hash table $\mathcal{H}$ of their functions. The pseudocode is shown in Listing 4.3. Suppose g is a gate with k inputs. the number of np-configurations is $k! \cdot 2^k$. (Since, there are $k!$ p-configurations, and each

already exists in the library.

Listing 4.4 FIND-MATCHES-NP($\widehat{G}, \mathcal{H}$)

Input: AIG with choice $\widehat{G}$, hash table $\mathcal{H}$ from BUILD-HASH-TABLE-NP
Output: singular and regular matches for every representative r in $\widehat{G}$

```
Compute k-feasible cuts Φ_≃(N) for each equivalence class N in Ĝ

for each representative node r in Ĝ in topological order do

  — compute regular matches —

  if r is a Input node then

    regular[(r, 0)] ← {PI-MATCH}
    regular[(r, 1)] ← ∅        —  no matches

  else assert r is an And node

    regular[(r, 0)], regular[(r, 1)] ← ∅

    Let N be the equivalence class of r

    for each non-trivial cut c ∈ Φ_≃(N) do

      Let f be the function of c

      for each np-configuration (g, σ, τ) ∈ LOOKUP(H, f) do
        if VALID(c, g, σ, τ) then regular[(r, 0)] ← regular[(r, 0)] ∪ {(c, g, σ, τ)}
      end for

      for each np-configuration (g, σ, τ) ∈ LOOKUP(H, ¬f) do
        if VALID(c, g, σ, τ) then regular[(r, 1)] ← regular[(r, 1)] ∪ {(c, g, σ, τ)}
      end for

    end for

  end if

  — compute singular matches —

  singular[(r, 0)], singular[(r, 1)] ← ∅

  for each np-configuration (g, σ, τ) ∈ LOOKUP(H, 𝕏_1) do
    if VALID(c, g, σ, τ) then singular[(r, 0)] ← singular[(r, 0)] ∪ {({r}, g, σ, τ)}
  end for

  for each np-configuration (g, σ, τ) ∈ LOOKUP(H, ¬𝕏_1) do
    if VALID(c, g, σ, τ) then singular[(r, 1)] ← singular[(r, 1)] ∪ {({r}, g, σ, τ)}
  end for

end for
```

p-configuration leads to 2^k np-configurations, since the co-domain of the τ functions is $\{0, 1\}$.) Since $k \leq 6$, the maximum number of np-configurations is $720 \cdot 64 = 46080$ per gate. As noted before, the set of keys of $\mathcal{H}$ is smaller due to symmetries. We also note that procedure BUILD-HASH-TABLE-NP adds the WIRE gate to the library.

The second procedure FIND-MATCHES-NP processes the cuts of the AIG with choices $\widehat{G}$ to find matches using the hash table $\mathcal{H}$ computed by BUILD-HASH-TABLE-NP. The pseudocode is shown in Listing 4.4. The first step is to compute all k-feasible cuts of the equivalence classes as described in Section 3.6.2.

Next, for each representative r in $\widehat{G}$, the procedure first finds matching regular np-configurations for each edge $(r, 0)$ and $(r, 1)$. If r is an Input node, then PI-MATCH is the only regular match for $(r, 0)$, and there are no regular matches for $(r, 1)$. If r is an And node, then the matching regular np-configurations for each edge are found by processing the non-trivial cuts. As in the case of matching under permutations, only valid matches are stored.

After the regular np-matches have been computed, the procedure finds the singular np-matches. Since an np-configuration of a single input gate can either be $\mathbb{X}_1$ or $\neg\mathbb{X}_1$ (Example 4.6), it uses those two functions to index into the hash table to obtain singular np-matches. Again, only valid matches are stored.

To test his understanding of the matching algorithm, the reader may convince himself that no singular np-match with a buffer or WIRE can be found for the edge $(r, 1)$ if r is an Input node.

4.5.5 Why Matching in Two Polarities is Better

Matching a node in both polarities is better than creating new gates from the library gates by adding inverters (Section 4.5.1). The latter leads to duplicated logic in the mapped circuit, as the following example shows:

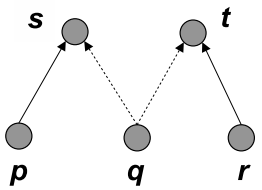


Figure 4.3: The AIG for Example 4.8.

Example 4.8 Consider the AIG shown in Figure 4.3. Let the node ids be in alphabetical order. The function of cut $c_s = \{p, q\}$ of node s is $\mathbb{X}_1 \cdot \neg\mathbb{X}_2$, and that of cut $c_t = \{q, r\}$ of node t is $\neg\mathbb{X}_1 \cdot \mathbb{X}_2$.

Suppose our library consists of an AND2 gate g with inputs a and b, and an inverter i with input a. It is easy to see that when matching under permutations we would not find any matching p-configurations for nodes s and t. (The two p-configurations of the AND2 gate have the same function $\mathbb{X}_1 \cdot \mathbb{X}_2$.)

However, we could construct a new gate h by adding the inverter to input a of the AND2 gate. The expression for h would be `and`(`not`(a), b). Let σ_1 be the assignment $\{a \mapsto \mathbb{X}_1, b \mapsto \mathbb{X}_2\}$. The function of the p-configuration (h, σ_1) is $\neg\mathbb{X}_1 \cdot \mathbb{X}_2$ and therefore it is a match for cut c_t. Indeed it is the *only* match for c_t. The other assignment $\sigma_2 = \{a \mapsto \mathbb{X}_2, b \mapsto \mathbb{X}_1\}$ of h has the function $\neg\mathbb{X}_2 \cdot \mathbb{X}_1$. Therefore, it is the (only) match for c_s. Therefore, the AIG can only be mapped with two instances of h. Since each instance of h has an inverter, there are two inverters in the mapped circuit. As may be verified, both inverters in the mapped circuit compute the same function, and hence there is a redundancy.

In contrast when mapping under permutations and negations, gate i matches edge $(q, 1)$ under the generalized assignment $(\{a \mapsto \mathbb{X}_1\}, \{a \mapsto 0\})$. Let τ_1 be the map $\{a \mapsto 1, b \mapsto 0\}$. The generalized assignment (σ_1, τ_1) matches cut c_t. Similarly, if $\tau_2 = \{a \mapsto 0, b \mapsto 1\}$ then the generalized assignment (σ_2, τ_2) matches cut c_s. Now

in the mapped circuit, there is only one inverter.

Thus by mapping nodes in both polarities, we can avoid unnecessary duplication of logic. Note that it is possible to avoid duplication by analyzing the mapped circuit for redundancies. However, since cost estimates are incorrect during match selection, the overall result is inferior (even after redundancies have been detected and eliminated). We shall return to this subject again in Section 4.7.

4.6 Supergates

4.6.1 Motivation

We now return to the idea of combining library gates to form new gates in order to find more matches. We call such gates constructed from other library gates *supergates*. Informally, a supergate is a single-output network of gates from the library. In Example 4.4 we saw how simple supergates constructed by adding inverters to the inputs and outputs of library gates can result in more matches being detected. However, in that case, the same result can be obtained by mapping in both polarities as discussed above. Therefore, we now present a more interesting example to demonstrate the usefulness of supergates.

Example 4.9 Consider the AIG shown in Figure 4.4 (I). Now, suppose we are given a library consisting of an INV gate (inverter), AND2 gate, a MUX21 (2-1 multiplexer), and an XNOR2 gate and our task is to find all matches for every edge in the AIG. Note that the existence of the AND2 gate and the inverter ensures functional completeness: Procedure 4.4 will find matches for every edge.

Now consider the edge $(v, 1)$ of the AIG. Node v has a single cut of size 3, *viz* $\{t, r, s\}$. It is easy to verify that *no* np-configuration of MUX21 is a match for $\{t, r, s\}$.

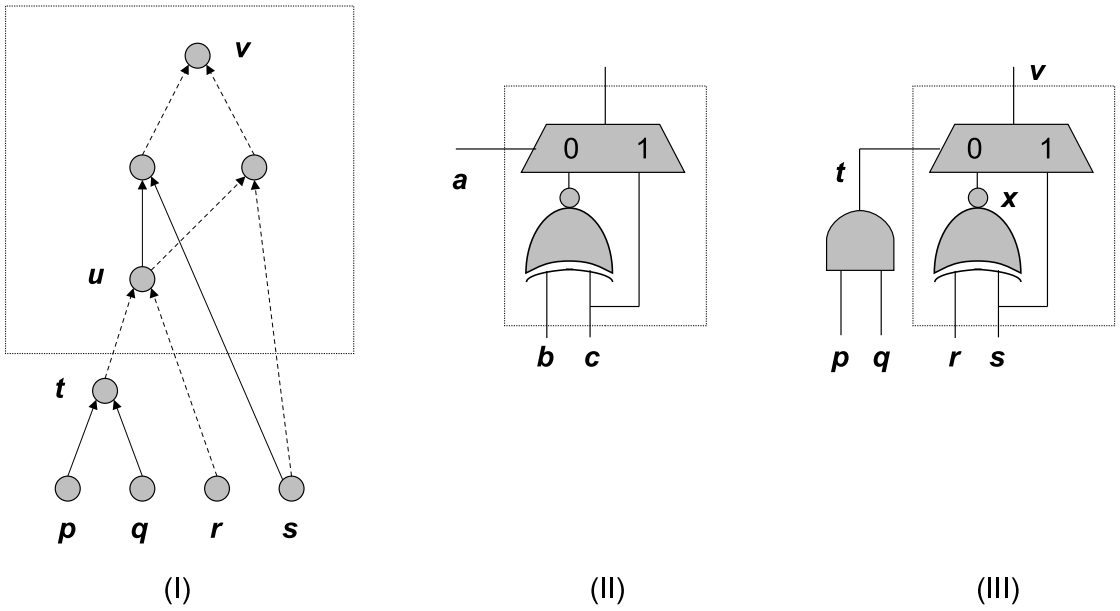


Figure 4.4: Example illustrating how supergates can be used to find more matches. (I) An AIG where no np-configuration of a MUX21 is a match for edge $(v, 0)$. (II) A supergate constructed from the MUX21 and the XNOR2 gate. (III) The result of mapping using the supergate.

However, the MUX21 can be matched by constructing a supergate. Consider the supergate shown in Figure 4.4 (II) consisting of the MUX21 fed by the XNOR2 gate. If we use a, b, and c to denote its input pins, we may describe the functional expression of the supergate by the following expression:

$$\texttt{or}(\texttt{and}(\texttt{not}(\text{a}), \texttt{not}(\texttt{xor}(\text{b}, \text{c}))), \texttt{and}(\text{a}, \text{c}))$$

It is easy to verify that if the supergate were to be added to the library, then Procedure 4.4 would match the supergate to the cut $\{t, r, s\}$.

Figure 4.4 (III) shows a mapping of the AIG using the supergate and the AND2 gate which matches the edge $(t, 0)$. It provides an intuitive explanation for why the MUX21 cannot be directly matched to $(v, 0)$. Consider the node x in the mapped

network. If a node with the same functionality were present in the AIG, then the MUX21 match could have been found directly. Since that corresponding node does not exist in the AIG, a supergate is need to find the match.

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 AIG. Furthermore, as this example illustrates, supergates are useful even with standard cell libraries that are functionally rich.

4.6.2 Formal Definition

We are given a library of gates $\mathcal{L}$. Let k be the largest number of inputs allowed for a supergate (typically 6 or fewer).

4.6.2.1 Level-0 Supergates

Let $\mathbb{X}_1, \mathbb{X}_2, .., \mathbb{X}_k$, be distinct Boolean variables. We define the set of all level-0 supergates $\mathcal{S}_0$ as follows:

$$\mathcal{S}_0 = \{\mathbb{X}_1, \mathbb{X}_2, .., \mathbb{X}_k, \neg\mathbb{X}_1, \neg\mathbb{X}_2, .., \neg\mathbb{X}_k\}$$

Note that we think of the Boolean variable $\mathbb{X}_i$ as a supergate. For convenience, we define the function $\pi(s)$ of a level-0 supergate s to be s itself, i.e. the function of $\mathbb{X}_1$ viewed as a level-0 supergate is $\mathbb{X}_1$.

4.6.2.2 Level-1 Supergates

Informally, a level-1 supergate is constructed by connecting level-0 supergates to the inputs of a library gate. Formally, if $g \in \mathcal{L}$ is a gate with j inputs, $j \leq k$, let ρ be a bijective map from the input pins of g to $\mathcal{S}_0$. The pair (g, ρ) is called a level-1

supergate and g is called the *root*. The function of a level-1 supergate (g, ρ), is defined to be $\pi(\text{expr}(g), \rho)$ (Section 4.3.1). We abuse notation and use $\pi(s)$ to denote the function of a level-1 supergate s.

We use $\mathcal{S}_1$ to denote the set of all level-1 supergates that can be generated from $\mathcal{L}$ and $\mathcal{S}_0$, i.e.

$$\mathcal{S}_1 = \{(g, \rho) \mid g \in \mathcal{L} \text{ and } (g, \rho) \text{ is a level-1 supergate}\}$$

Note that the set $\mathcal{S}_1$ implicitly depends on $\mathcal{L}$ and k (through $\mathcal{S}_0$) and to be precise we should incorporate this dependence into the notation. However, to keep the notation simple, we omit $\mathcal{L}$ and k, trusting that they are clear from the context.

Example 4.10 Set $k = 4$. Let $\mathcal{L}$ consist of a single AND2 gate g with input pins a and b and $\text{expr}(g) = \texttt{and}(a, b)$. Consider $\rho = \{a \mapsto \neg\mathbb{X}_3, b \mapsto \mathbb{X}_1\}$. Now, (g, ρ) is a supergate and its function $\pi((g, \rho))$ is $\neg\mathbb{X}_3 \cdot \mathbb{X}_1$.

In this case, $\mathcal{S}_1$ contains $\binom{8}{2} = 28$ level-1 supergates, a few of which are degenerate. For example, 4 supergates have the constant 0 function. They are of the type $(g, \{a \mapsto \mathbb{X}_i, b \mapsto \neg\mathbb{X}_i\})$.

Level-1 supergates are a generalization of np-configurations in the following sense: If (g, σ, τ) is an np-configuration with function f, then there is a ρ s.t. the level-1 supergate (g, ρ) has function f. Proof: Pick ρ as follows:

$$\rho(i) = \begin{cases} \neg\sigma(i) & \text{if } \tau(i) = 0 \\ \sigma(i) & \text{if } \tau(i) = 1 \end{cases}$$

However, not every level-1 supergate has a corresponding np-configuration with the same function. If a gate g has $j \leq k$ inputs, then an assignment σ (in an np-configuration) can only map an input pin p of g to a variable $\mathbb{X}_i$ where $1 \leq i \leq j$. It

cannot map p to a $\mathbb{X}_i$, $j < i \leq k$. Thus there is no np-configuration with the same function as the supergate of Example 4.10.

It is instructive to consider the reason for this difference between np-configurations and level-1 supergates: The np-configurations are directly used to match with cuts. Now since the variables assigned to a cut c of size j are $\mathbb{X}_i$, $1 \leq i \leq j$, np-configurations are restricted likewise. In other words, generalizing np-configurations to use the remaining variables would not result in more matches (except for the rare situation described in Section 4.4.4).

On the other hand, level-1 supergates serve a second purpose beyond matching cuts. They are used to build larger supergates. In this context, the more general definition is useful since more logic structures can be explored, and a greater variety of supergates can be constructed.

Example 4.11 In Example 4.9, suppose we are interested in all supergates having 3 or fewer inputs, i.e. $k = 3$. One of the level-1 supergates is $s_1 = (\text{XNOR2}, \rho_1)$ where $\rho_1 = \{a \mapsto \mathbb{X}_2, b \mapsto fa_3\}$. This level-1 supergate does not match any cut in the AIG, but is used to build a level-2 supergate that does.

4.6.2.3 Level-2 Supergates

Let $g \in \mathcal{L}$ be a gate with j inputs. Let ρ be a bijective map from the input pins of g to $\mathcal{S}_0 \cup \mathcal{S}_1$. The pair (g, ρ) is called a *level-2 supergate* if (g, ρ) is not a level-1 supergate.[6] The gate g is called the *root* of the supergate (g, ρ). Define the map μ on the input pins of g as follows:

$$\mu(i) = \pi(\rho(i))$$

The function of a level-2 supergate (g, ρ) is defined to be $\pi(\text{expr}(g), \mu) = \pi(\text{expr}(g), \pi \circ \rho)$. If s is a level-2 supergate, once again, we use $\pi(s)$ to denote the function of s.

[6] i.e. for at least one input pin i of g, $\rho(i)$ is a level-1 supergate

We use $\mathcal{S}_2$ to denote the set of all level-2 supergates that can be generated from $\mathcal{L}$.

Example 4.12 The supergate comprising the MUX21 and the XNOR gate used in Example 4.9 is a level-2 supergate s_2 obtained by connecting the MUX21 to the level-1 supergate s_1 obtained in Example 4.11. Formally, $s_2 = (\text{MUX21}, \rho_2)$ where $\rho_2 = \{s \mapsto \mathbb{X}_1, d_0 \mapsto s_1, d_1 \mapsto \mathbb{X}_3\}$. (The input pins of the MUX21 are s, d_0 and d_1, and the functional expression of the MUX21 is $\texttt{or}(\texttt{and}(\texttt{not}(s),\ d_0),\ \texttt{and}(s,\ d_1))$.)

4.6.2.4 Level-n Supergates

Continuing in this way, if ρ is a bijective map from the input pins of g to $\bigcup_{i=0}^{n-1} \mathcal{S}_i$, then (g, ρ) is called a level-n supergate if (g, ρ) is not a level-k supergate for any $k < n$. The function of the supergate, denoted by, $\pi((g, \rho))$ is given by

$$\pi((g, \rho)) = \pi(\text{expr}(g), \pi \circ \rho)$$

Once again, g is called the *root* of the supergate (g, ρ).

4.6.3 Generating Supergates

4.6.3.1 Overview

The formal definition of supergates immediately provides a representation for supergates in programs. The collection of all supergates is represented as a directed acyclic graph (DAG), where each node represents a supergate. The leaf nodes in this DAG are the level-0 supergates. In the first round, one iterates over each gate g in the library, and constructs all level-1 supergates that can be constructed from g. Each level-1 supergate thus obtained is represented by a new node in the DAG. In the second round, one again iterates over each gate g in the library, and constructs all level-2 supergates that can be constructed from g. Once again, each level-2 supergate

is represented by a new node in the DAG. In this manner, the set of supergates is computed in a series of round until a user-specified amount of time elapses.

The adjacency information of the DAG of supergates is stored as follows: If node s represents a supergate (g, ρ), and if g has j inputs $x_1, x_2, .. x_j$, then there are exactly j edges $(\rho(x_i), s)$, $1 \leq i \leq j$ in the DAG. These edges are represented by storing the $\rho(x_i)$ at s.

4.6.3.2 Pruning

The number of supergates generated by the procedure described above is huge especially in modern libraries which contain hundreds of gates. Exhaustive generation of supergates therefore is generally infeasible in mapping that uses a load-based model of delay. However, if a constant-delay model is used for mapping[7] then supergates can be exhaustively generated for a few levels, provided some pruning is used. The following example illustrates the need for pruning even if there are a few gates.

Example 4.13 Suppose during supergate generation, after level-1 supergates have been computed there are 1000 supergates in all. Now, if the library contains a NAND4 gate (i.e. a NAND gate with 4 inputs), then there are roughly $\binom{1000}{4}$ possible level-2 supergates. In this case it would be impractical to enumerate all level-2 supergates generated by the NAND4 gate.

Therefore, during supergate generation, an attempt is made to reduce the number of supergates considered through pruning. Pruning is of two types:

1. **Pruning by Dominance.** As we saw in Example 4.10, some supergates may be degenerate and have the same function as a constant. During enumeration,

[7]This means that there are fewer gates in the library since gates with different drive-strengths are represented with one gate.

we can detect these degenerate gates, and remove them from the set. This prevents them from being used to build other supergates.

The idea of detecting and removing degenerate gates can be generalized by the notion of dominance. Suppose during mapping we are interested in minimizing two cost functions: delay and area. If we have two supergates s_1 and s_2 that have the same function, such that

- the area of s_1 is not (strictly) greater than the area of s_2, and,
- the input to output delay of each input pin of s_1 is not (strictly) greater than that of the corresponding delay in s_2[8]

then we say that s_1 *dominates* s_2. Any mapping solution that uses s_2 can be improved in terms of area and delay by using s_1 instead of s_2.

Example 4.14 Suppose the library has an AND2 gate with input pins a and b and output z. Suppose the AND2 gate has unit area, and that the delay from a to z is 1 unit, and that from b to z is 2 units[9]. Now consider the supergates $s_i = (\text{AND2}, \rho_i)$ defined as follows:

s_i	ρ_i	$\pi(s_i)$
s_1	$\{a \mapsto \mathbb{X}_1, b \mapsto \mathbb{X}_2\}$	$\mathbb{X}_1 \cdot \mathbb{X}_2$
s_2	$\{a \mapsto \mathbb{X}_2, b \mapsto \mathbb{X}_3\}$	$\mathbb{X}_2 \cdot \mathbb{X}_3$
s_3	$\{a \mapsto s_1, b \mapsto \mathbb{X}_3\}$	$(\mathbb{X}_1 \cdot \mathbb{X}_2) \cdot \mathbb{X}_3 = \mathbb{X}_1 \cdot \mathbb{X}_2 \cdot \mathbb{X}_3$
s_4	$\{a \mapsto s_1, b \mapsto s_2\}$	$(\mathbb{X}_1 \cdot \mathbb{X}_2) \cdot (\mathbb{X}_2 \cdot \mathbb{X}_3) = \mathbb{X}_1 \cdot \mathbb{X}_2 \cdot \mathbb{X}_3$

Now, the area of s_3 is 2 units whereas that for s_4 is 3 units. Furthermore, the delays from input to output are as follows:

[8] We assume the constant-delay model here for simplicity.
[9] We do not consider separate rise and fall delays here for simplicity.

Input	Delay in s_3	Delay in s_4
$\mathbb{X}_1$	2 units	2 units
$\mathbb{X}_2$	3 units	3 units
$\mathbb{X}_3$	4 units	2 units

Thus s_3 dominates s_4.

To be complete, we point out here that two supergates obtained by permutation need not dominate each other.

During supergate generation, we can detect all dominated gates by constructing a hash table $\mathcal{H}$ indexed by the function of the supergate. When a new supergate is constructed, we lookup in the hash table to find all supergates that have the same function. If any existing supergate dominates the newly generated supergate, then it is not added to the set of supergates. Otherwise, it is added to the set of supergates, and also inserted into the hash table.

2. **Pruning by Resource Limitations.** A second method of pruning arises by imposing artificial resource limitations on delay or area. For instance, only supergates of a certain area or less are considered. Any supergate that exceeds this limit is not added to the set of supergates (and hence does not participate in the subsequent rounds).

   A very useful resource limitation in practice is to set a limit n on the number of inputs a root may have. Thus in each round *after* the first, only library gates with n or fewer inputs are used to construct supergates. In practice $n = 3$ is a reasonable limit, and even $n = 2$ leads to a good set of supergates. (Note that this method of pruning directly prevents the combinatorial blow-up seen in Example 4.13.)

4.6.4 Matching Using Supergates

A level-k supergate is called *proper* if $k > 0$. Only proper supergates are used for matching i.e. level-0 supergates are not used for matching.

Proper supergates can be used in the matching process by a simple modification of the process used to match regular library gates using np-configurations. First, a hash table $\mathcal{H}$ is created which maps a Boolean function f to the set of proper supergates whose function is f. This is similar to the hash table created by Procedure BUILD-HASH-TABLE-NP (Listing 4.3) for np-configurations. Second, by a procedure similar to Procedure FIND-MATCHES-NP (Listing 4.4) the cuts are matched with corresponding supergates using $\mathcal{H}$. In this context, the singular matches will be matches to the level-1 supergates generated by inverters and buffers in the library.

Remark Note that we do not match to level-0 supergates since we exclude them from the hash table $\mathcal{H}$. This is because they do not correspond to real library gates, but play a role similar to that of np-configurations to permit matching in both polarities (Section 4.6.2.2). In particular, the level-0 supergate $\neg\mathbb{X}_1$ should be used to match since it is possible the library might not have an inverter. (If the library has an inverter then there will be a level-1 supergate with the function $\neg\mathbb{X}_1$. This supergate *will* be used in matching.)

4.7 Supergates and Choices

Supergates can be seen as a technique for creating choices. Unlike the choices due to algebraic rewriting or lossless synthesis, the choices introduced by supergates come from the library. Thus, if a supergate matches at an edge r in an AIG with choice $\widehat{G}$, it is possible to create a new AIG with choice $\widehat{G}'$ from $\widehat{G}$ such that mapping *without* supergates (i.e. using just np-configurations of library gates) on $\widehat{G}'$ leads to the match

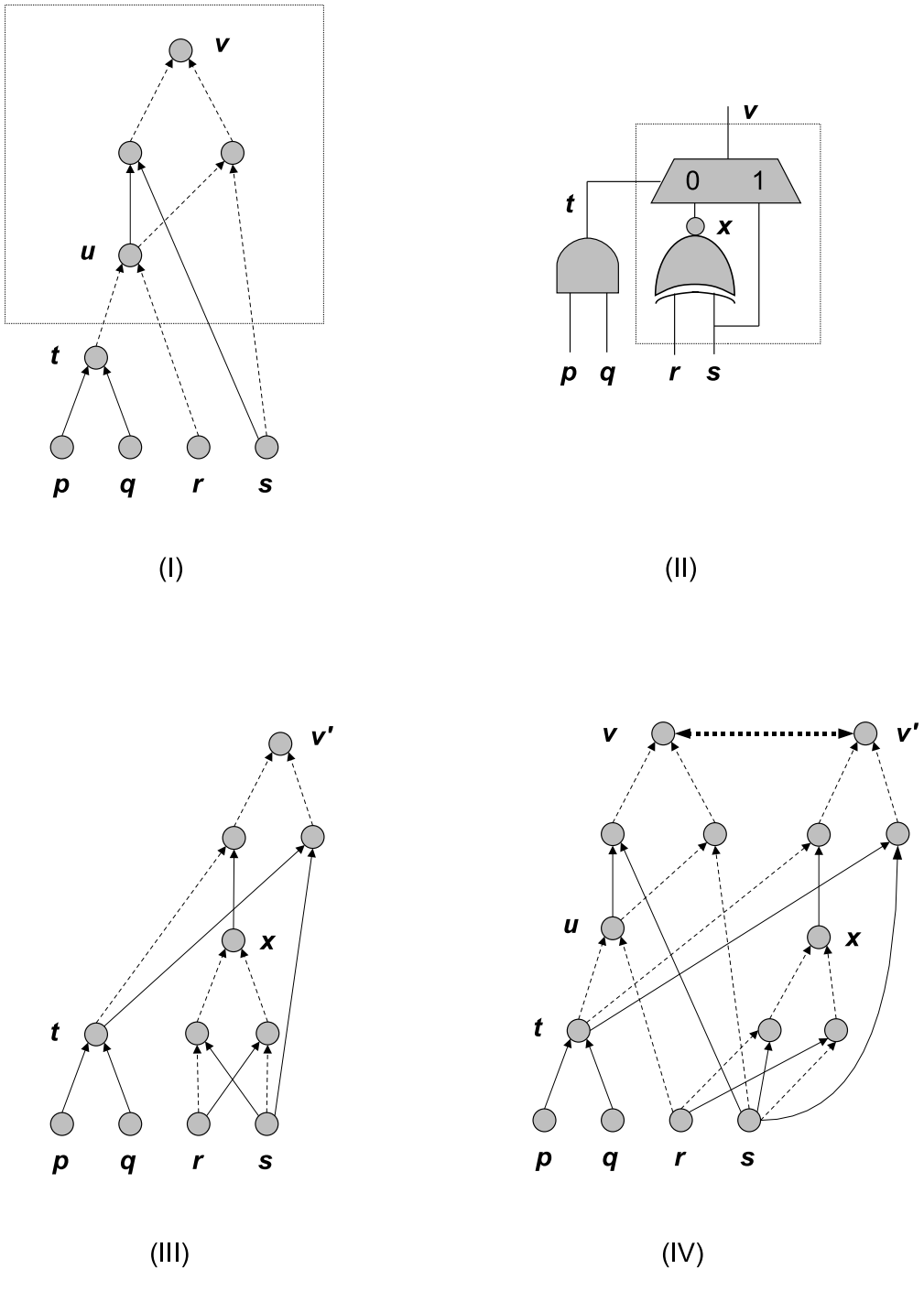


Figure 4.5: Example illustrating how supergates lead to choices.

that was found using supergates.

Example 4.15 Figure 4.5 (I) reproduces the AIG of Example 4.9, and Figure 4.5 (II) shows a mapped network M corresponding to the AIG that uses the supergate constructed in that example. Now consider an AIG obtained from M by decomposing its constituent library gates into two input AND gates and inverters. This AIG is shown in Figure 4.5 (III). Note that the function of node v' in (III) is the complement of the function of node v in (I). Now, if the choice operator (Section 2.6) is used to combine these two AIGs to form a single AIG with choice $\widehat{G}'$, we obtain the structure shown in Figure 4.5 (IV) where nodes v and v' belong to the same equivalence class, say V.

Now, if mapping is performed on $\widehat{G}'$ using only np-configurations (i.e. without using supergates) then we find that the MUX21 is a match for class V since the cut $\{s, t, x\} \in \Phi_{\simeq}(V)$.

Remark 1 In the light of the above discussion, it is possible to see supergate matching as a semi-technology dependent synthesis step – call it *superchoicing* – for adding choices using library information. Although experiments indicate that superchoicing leads to good results, the main problem is that too many choices are added due to the large number of supergate matches.

Remark 2 Our experience indicates that supergates are mainly useful for delay-oriented mapping. They tend to increase area since common logic between two supergates is not correctly accounted for during the selection process. We saw the reason for this in Section 4.5.5 for the special case of supergates constructed using only inverters and library gates. An interesting line of research would be to explore the use of superchoicing in reducing the area overhead of using supergates.

4.8 Related Work

4.8.1 On $\mathcal{NPN}$-Canonization

Mailhot and De Micheli first proposed the use of Boolean matching (instead of structural matching) for technology mapping [MM90]. However, their method to detect when a library gate matches a cut function[10] is different from ours. It is instructive to look at their method since it motivated much of the subsequent work on Boolean matching.

Two Boolean functions are $\mathcal{NPN}$-equivalent if one can be obtained from the other by negation of output, permutations of inputs, or negations of inputs (or any combination thereof). To decide if a cut matches a library gate, Mailhot and De Micheli enumerate all functions that are $\mathcal{NPN}$-equivalent to the cut function to see if any one of them is equal to *the*[11] function of the library gate. They also present some heuristics to speed up the process by terminating this enumeration early.

In the terminology of this chapter, they essentially enumerate the different "np-configurations" of a *cut* function[12] to see which matches the function of a given library gate. For each cut, this procedure is repeated for every library gate.

It is easy to see why this procedure is inefficient: It repeats the enumeration of $\mathcal{NPN}$-equivalent functions of each cut functions once for each library gate. Thus its run-time is $O(c \cdot l)$ where c is the number of cuts and l is the number of gates in the library. Furthermore, the constant hidden by the big-O notation is large. In contrast, in our method, we pre-compute the different np-configurations of each library gate *once* (an $O(l)$ procedure with a large constant), and then each cut function just requires two lookups in the hash table (an $O(c)$ procedure with a small constant).

[10] In fact they used clusters instead of cuts (Section 3.7) but that difference is not important here.
[11] In our terminology this would be the function of the gate under an arbitrary assignment.
[12] Except they also consider the output negation which is not captured in the np-configuration. We handle that by matching both the cut function and its complement.

Since $c \gg l$, our method is significantly faster.

Burch and Long improved upon Mailhot and De Micheli by proposing the use of $\mathcal{NPN}$-canonization [BL92]. Since $\mathcal{NPN}$-equivalence is an equivalence relation, it splits the set of all Boolean functions on k variables into equivalence classes. One member of each class is designated as the representative of that class. Now, if we have a procedure that given a function f returns the representative of the $\mathcal{NPN}$-equivalence class of f, then the following procedure can be used. First, we compute the $\mathcal{NPN}$-representative of the function of each library gate, and construct a hash table of these representatives. Next, for each cut function, we compute the canonical representative and look it up in the hash table to find a library gate that matches it.

However, the problem of finding the $\mathcal{NPN}$-representative of a given function is non-trivial, and much of the recent research in this area has addressed the problem of making this efficient [CS03; DS04; AP05; CK06]. (In fact the problem of determining canonical representatives is an old one [Gol59].[13]) However, we note that in the worst case (for some classes of functions), all these methods lead to a search where different $\mathcal{NPN}$-equivalent functions are enumerated in order to find the representative. Thus, although these methods can work with gates with more than 6 inputs, these methods also have a limit. With current techniques this limit is around 12 inputs.

There are two main problems with the use of canonization in the context of technology mapping:

1. Every cut function needs to be canonized before the hash-table lookup, and this is non-trivial. In contrast, in our method, there is no need to canonize. On the other hand, in our method, the hash table contains more elements; but that does not significantly increase the time for lookups.

2. After a match has been found, different symmetries of the function need to be

[13] We thank Donald Chai for this reference.

explored to find the different matches. (For instance, a multi-input And gate will in general have different input pin to output delays; thus different matches need to be explored.)

In summary, in the context of standard cell matching, our method is more efficient and also simpler to implement than methods based on canonization.

On the other hand for matching problems where the number of inputs is larger (say between 8–12), canonization-based methods are useful. However, note that in such cases, it may be too expensive to actually evaluate the different matches during selection. Also, exhaustive cut computation in such cases is not possible. For an example of a mapping problem with these characteristics, see the work on factor cuts [CMB06a].

We note here that the work on *functional matching* in the BDDmap system is similar to ours in that different permutations are enumerated [KDNG92]. However, they use BDDs instead of truth tables to represent functions. Thus, only those gates which are highly symmetric or have a small number of inputs (say 3) are matched using functional matching in order "to avoid possible exponential blowup of [BDDs]." All other gates are matched structurally. Somewhat paradoxically, by using truth tables instead of BDDs, we can use Boolean matching for a much larger class of gates.

The work on BDDmap also addresses the question of mapping to both polarities in the context of Boolean matching. However, they pick the solution of creating new gates with inverters as described in Section 4.5.1. That method is inferior to our method for reasons described in Section 4.5.5.

4.8.2 On Supergates

To our knowledge, the use of supergate-like constructs to increase the number of matches has not been reported in the literature.

Hinsberger and Kolla consider increasing the set of possible matches by constructing *configurations* from library gates [HK98]. These configurations are constructed by bridging (i.e. connecting certain input pins together) or connecting constants to certain inputs. Although the use of these configurations will increase the number of matches, it is not clear what the benefit is. For instance, bridging inputs of a 4-input AND gate will lead to smaller AND gates. However, in a realistic library, these smaller AND gates are likely to be present anyway, and the smaller gates will have better area and delay compared to the larger gate with bridging. (In the absence of further data, we must conclude that their improvements compared to SIS stem from the use of Boolean matching as opposed to the use of configurations.)

Note that in our framework, we can accommodate bridging by *not* requiring assignments to be bijective in Section 4.3.2. Setting some inputs to constants can be accommodated by allowing level-0 supergates to also include the constant 0 and constant 1 functions in Section 4.6.2.1.

The bottom-up supergate generation may be seen as an attempt at exact multi-level synthesis for small functions. In this connection the reader may be interested in some early work on exact multi-level synthesis of small functions using exhaustive enumeration [Hel63; Dav69].

4.8.3 Supergates and Algebraic Choices

Recall from Section 2.7.2 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.* [LWGH95].

The search space explored by supergates is different from the search space explored by algebraic 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.

Chapter 5

Match Selection for FPGA Mapping

5.1 Overview

In this chapter we look at the match selection problem in the context of FPGA mapping with a special focus on area-oriented selection. The goal of selection is to construct the best mapped network by selecting a suitable set of matches. Although this chapter deals only with FPGA mapping, it may be of interest to the standard cell reader since the main ideas behind the area-oriented selection algorithms are easier to grasp in the simpler FPGA setting.

We begin with a review of the cost models for FPGA mapping (Section 5.2). Next, we define the notion of a *cover* (Section 5.3). A cover identifies a subset of matches that correspond to a valid mapped network, and in the rest of the chapter we deal with mapped networks through covers. We begin our discussion of selection algorithms with a generic selection procedure where the cost function is abstracted away (Section 5.4). Then, we review depth-oriented selection in this context (Section 5.5). Then, we present the main contribution of this chapter which is the area-oriented selection algorithm for FPGA mapping (Section 5.6). After that, we discuss briefly how

area and depth costs are used to break ties during selection (Section 5.7). Finally, we look at how area-oriented selection may be combined with depth-oriented selection to improve area under delay constraints (Section 5.8).

We postpone the detailed discussion of related work in the literature to the end of the chapter (Section 5.9).

Notation Let $\widehat{G}$ be an AIG with choice, and let $\mathcal{X}$ be its set of representatives. Given a node $r \in \mathcal{X}$, we abuse notation and use $\Phi_{\simeq}(r)$ to denote $\Phi_{\simeq}(N)$ where N is the equivalence class of r.

5.2 The Cost Model

Delay. We assume the commonly used unit delay model where each LUT has constant (unit) delay from any pin to output pin [CD96, Section 2.3]. The proposed algorithms can also work with a non-unit delay model where the input-to-output delays are different for different input pins of a LUT. We point out the necessary modifications at the appropriate place.

Area. We assume that the area of a LUT depends on the actual function implemented by the LUT. We abstract this dependence away by assuming a function GET-LUT-AREA that given a cut c returns the area required by a LUT if it implements the function of c.

For many FPGA architectures such as Actel ProASIC III [Act07] (3-LUT), Altera Stratix [Alt06] (4-LUT) and Xilinx Virtex [Xil07a] (4-LUT) the area of a LUT is independent of the function implemented by it. In this cases, for any k-feasible cut c (k depending on the architecture),

$$\text{GET-LUT-AREA}(c) = 1$$

For some newer architectures such as the Altera Stratix II [LAB+05] and Xilinx Virtex-5 [Xil07b], a 6-input LUT may be used to implement any Boolean function of 6 inputs or may be used to implement two Boolean functions of 5 inputs or fewer. For these architectures, for a 6-feasible cut c we have,

$$\text{GET-LUT-AREA}(c) = \begin{cases} 1 & : \text{ if } |c| \leq 5 \\ 2 & : \text{ if } |c| = 6 \end{cases}$$

5.3 Cover

Let $\widehat{G}$ be an AIG with choice. Intuitively, the set of k-feasible cuts for all equivalence classes of $\widehat{G}$ is an implicit description of a set of mapped networks. Let $\mathcal{M}$ denote this set of mapped networks. The task of selection is to explicitly identify one mapped network in $\mathcal{M}$ that is optimal w.r.t. the given cost function.

The notion of a cover is used to specify an arbitrary element $M \in \mathcal{M}$. Let $\mathcal{X}$ be the set of representatives of $\widehat{G}$. Let O be the set of output edges of $\widehat{G}$. Let ν be a map that associates with every $r \in \mathcal{X}$ either a cut $c \in \Phi_{\simeq}(r)$ or the symbol NIL. If $\nu(r) \neq \text{NIL}$, then r is said to be *selected* in ν, and $\nu(r)$ is said to be the selected cut of r, or simply *the* cut of r. The map ν is a *cover* if the following three conditions are satisfied:

1. If $(r, i) \in O$ then r is selected in ν.
2. If r is selected in ν then every $r' \in \nu(r)$ is also selected in ν.
3. If r is selected in ν and r is an And node, then $\nu(r)$ is a non-trivial cut.

A cover ν is a complete description of a mapped network in the following manner: For each representative r selected in ν, there is a net n_r in the mapped network. If r is an Input node, n_r is driven by the input port corresponding to r in the mapped

network. Otherwise, n_r is driven by a LUT l_r corresponding to r in the mapped network. The configuration bits of l_r are set according to the function of the cut $\nu(r)$. The input pins of l_r are connected to the nets corresponding to the representatives in $\nu(r)$ (which by Condition 2 are also selected in ν).

The three conditions above ensure that the mapped network corresponding to a cover ν is a valid one. Condition 1 ensures that there is a net corresponding to each output in the mapped network. Condition 2 ensures that the input nets of a LUT are present in the mapped network. Condition 3 ensures that the trivial cuts are only allowed for Input nodes (note that there are no LUTs corresponding to the input nodes); all other representatives are implemented in the mapped network with LUTs.

To summarize, a cover is a complete description of a mapped network. In the subsequent discussion, we shall deal with mapped networks though covers. The goal of selection now becomes to select the optimal cover from among the possible covers.

5.4 Overview of the Selection Procedure

In this section we present an overview of the selection algorithm. For simplicity of exposition, in this section, we do not deal with specific cost functions. The input to the selection procedure is an AIG with choices $\widehat{G}$. We assume that the set of cuts $\Phi_{\simeq}(N)$ of each equivalence class N in $\widehat{G}$ has already been constructed as described in Section 3.6. The output of the selection procedure is a cover that optimizes the given cost function.

The selection procedure is divided into two sub-procedures EVALUATE-MATCHES and COVER. The sub-procedure EVALUATE-MATCHES (Listing 5.1) is called first. It processes the representative nodes in topological order starting from inputs. For each representative r, it assigns the best[1] non-trivial cut to best[r] and the corresponding

[1]according to the cost function abstracted here by the function EVAL-MATCH-COST

Listing 5.1 EVALUATE-MATCHES($\widehat{G}$)

Input: AIG with choice $\widehat{G}$
Output: best[r] for every And representative r in $\widehat{G}$

for each representative node r in $\widehat{G}$ in topological order **do**
  if r is a Input node **then**
    cost[r] $\leftarrow$ SET-INPUT-COST(r)
    best[r] $\leftarrow \{r\}$
  else assert r is an And node
    cost[r] $\leftarrow \min_{c \in \Phi_{\simeq}(r) \setminus \{\{r\}\}}$ EVAL-MATCH-COST(c)
    best[r] $\leftarrow \arg\min_{c \in \Phi_{\simeq}(r) \setminus \{\{r\}\}}$ EVAL-MATCH-COST(c)
  end if
end for

Listing 5.2 COVER($\widehat{G}, O,$ best[$\cdot$])

Input: AIG with choice $\widehat{G}$, set of output edges O, best[r] for each representative r
Output: nref[r] for every representative r in $\widehat{G}$

for each representative node r in $\widehat{G}$ **do**
  nref[r] $\leftarrow 0$
end for
for each edge $(n, i) \in O$ **do**
  nref[n] $\leftarrow$ nref[n] + 1
end for
for each representative node r in $\widehat{G}$ in reverse topological order **do**
  if nref[r] > 0 and r is an And node **then**
    for each $n \in$ best[r] **do**
      nref[n] $\leftarrow$ nref[n] + 1
    end for
  end if
end for

cost to cost[r]. Although we defer the specification of the specific cost function to later, one should think of the cost of a cut as a "cumulative" measure that includes the cost of the subnetwork below it. Thus cost[r] is used by EVAL-MATCH-COST to determine the cost of cuts (of representatives in the transitive fanout of r) that contain r. For now, also note that input nodes are treated specially.

Intuitively, EVALUATE-MATCHES constructs part of the cover. For each representative r, best[r] stores the selected cut of r if r were to be selected in the cover. To complete the construction of the cover, the sub-procedure COVER is called next. The pseudo-code of COVER is shown in Listing 5.2. It constructs a cover in a straightforward manner by enforcing Conditions 1 and 2 of Section 5.3. (Condition 3 is enforced by EVALUATE-MATCHES.) Note that the procedure COVER does not depend on the cost function.

For each representative r, COVER also computes nref[r] which is the number of fanouts of the LUT corresponding to r in the mapped network. (This fanout count includes fanouts to output ports of the mapped network in addition to the fanouts to other LUTs in the network.) The maps best[$\cdot$] and nref[$\cdot$] define a cover ν in the following manner:

$$\nu(r) = \begin{cases} \text{best}[r] & : \text{ if nref}[r] > 0 \\ \text{NIL} & : \text{ otherwise} \end{cases}$$

Finally, we remind the reader that only representative nodes in an AIG with choice may belong to a cut (Section 3.6.2). Therefore, the selection procedures EVALUATE-MATCHES and COVER only deal with representative nodes.

5.5 Depth-Oriented Selection

In depth-oriented selection the goal is to choose a mapped network (or, equivalently, a cover) with minimum depth i.e. minimum arrival time at the outputs. This is done

in the standard manner by choosing for each representative r a cut that minimizes the arrival time.

Listing 5.3 EVALUATE-MATCHES-DEPTH($\widehat{G}$)

Input: AIG with choice $\widehat{G}$
Output: best[r] for every And representative r in $\widehat{G}$

```
for each representative node r in Ĝ in topological order do
  if r is a Input node then
    arrival[r] ← arrival time at input r
    best[r] ← {r}
  else assert r is an And node
    arrival[r] ← min_{c ∈ Φ≃(r)\{{r}}} ARRIVAL-TIME(c)
    best[r] ← arg min_{c ∈ Φ≃(r)\{{r}}} ARRIVAL-TIME(c)
  end if
end for
```

As discussed in Section 5.4, depth-oriented selection is split into two sub-procedures. Procedure EVALUATE-MATCHES-DEPTH is run first to select the best cut for each representative r. Next, the COVER procedure (Listing 5.2) is invoked to complete the construction of the cover. As is well known in the literature, this procedure is optimal i.e. produces a cover with minimum depth.

The pseudo-code for EVALUATE-MATCHES-DEPTH is shown in Listing 5.3. It is a specialization of the generic EVALUATE-MATCHES procedure shown in Listing 5.1 for depth-oriented selection. For a representative r, arrival[r][2] is the minimum arrival time possible for r. If r is an Input node, arrival[r] is set to the given arrival time for that input. If r is an And node, arrival[r] is set to the minimum of the arrival times from among the cuts of r. The arrival time of a cut c is computed in the unit delay

[2]arrival[r] corresponds to the generic cost[r] in Listing 5.1 in the context of depth-oriented mapping

model (Section 5.2) as follows:

$$\text{ARRIVAL-TIME}(c) = 1 + \max_{r \in c} \text{arrival}[r]$$

If the delays from input pins to output pin are different, then the arrival time of a cut can be computed by assigning the input pin with smallest delay to output to the latest arriving cut node, the input pin with second smallest delay to the second latest arriving cut node, and so on.

5.6 Area-Oriented Selection

5.6.1 Overview

Area-oriented selection is done using two separate selection procedures which use different heuristics to minimize area. The first area-oriented selection heuristic is called *area-flow*. Area-flow is used to construct an initial cover with "good" area. This cover is then greedily optimized by a second heuristic called *exact area*.

5.6.2 Area-Flow

Similar to depth-oriented selection, area-oriented mapping using the area-flow heuristic is split into two sub-procedures. Procedure EVALUATE-MATCHES-AREA-FLOW runs first and assigns a best cut to each representative r. Next, procedure COVER is invoked which completes the construction of the cover.

The pseudocode for EVALUATE-MATCHES-AREA-FLOW is shown in Listing 5.4. For a representative r, area$[r]$ measures the area (i.e. the cost) to implement r in the mapped network. For an Input node r, area$[r]$ is zero. For an And node r, area$[r]$ is the minimum area-flow cost among all cuts c of r. The area-flow cost of a cut c is

Listing 5.4 EVALUATE-MATCHES-AREA-FLOW($\widehat{G}$)

Input: AIG with choice $\widehat{G}$
Output: best[r] for every And representative r in $\widehat{G}$

for each representative node r in $\widehat{G}$ in topological order **do**
  if r is a Input node **then**
    area[r] $\leftarrow 0$
    best[r] $\leftarrow \{r\}$
  else assert r is an And node
    area[r] $\leftarrow \min_{c \in \Phi_{\simeq}(r) \setminus \{\{r\}\}}$ AREA-FLOW(c)
    best[r] $\leftarrow \arg\min_{c \in \Phi_{\simeq}(r) \setminus \{\{r\}\}}$ AREA-FLOW(c)
  end if
end for

given by the following formula:

$$\text{AREA-FLOW}(c) = \text{GET-LUT-AREA}(c) + \sum_{r \in c} \frac{\text{area}[r]}{\text{fanouts}(r)}$$

where fanouts(r) is the number of fanouts of the representative r in the AIG with choice $\widehat{G}$. (Note that since $r \in c$, r fans out to at least one node. Therefore, fanouts(r) $\neq 0$.)

Intuitively, this definition of the area cost takes into account possible sharing at a multi-fanout node. The cost of LUTs is distributed among the different fanouts. This would be completely accurate if fanouts(r) would reflect the actual fanout of the LUT chosen for r in the mapped network. However, since the fanouts of r are mapped after the best cut for r is selected, we simply *estimate* the fanout by assuming it to be the same as that in the AIG. This estimation is particularly inaccurate in the presence of choices, since choices increase fanout.

Note that in the special case where $\widehat{G}$ is a tree, and every equivalence class has only one representative (i.e. $\widehat{G}$ is an AIG without choices), the area flow metric is the

same as area since fanouts(r) = 1 for all representatives r.

5.6.3 Exact Area

Unlike area-flow, exact area does not create a cover from scratch. Instead, it greedily and incrementally optimizes an existing cover to increase sharing. The pseudocode for the main exact area procedure IMPROVE-COVER-EXACT-AREA is shown in Listing 5.5. It invokes two auxiliary procedures RECURSIVE-SELECT and RECURSIVE-DESELECT whose pseudocode is presented in Listings 5.6 and 5.7.

Procedure RECURSIVE-SELECT when invoked on a representative r is used to compute the *incremental* area cost of selecting best[r] assuming that the rest of the cover is fixed. The incremental area cost is the cost of selecting the LUTs that are only used to implement best[r] (and not used anywhere else in the mapped network). In other words this is the cost of the *maximum fanout-free cone* of r in the mapped network. This cost is estimated by recursively selecting the best cuts for the nodes in best[r] that are not currently selected. As a side effect of this routine, the nref[$\cdot$] array is updated to reflect the new fanout counts of the nodes after best[r] has been selected.

Procedure RECURSIVE-DESELECT is the opposite of RECURSIVE-SELECT. When invoked on a representative r, RECURSIVE-DESELECT frees the LUTs that have been selected exclusively to implement best[r]. Like RECURSIVE-SELECT, it returns the incremental area needed to select best[r]. As a side effect it updates nref[$\cdot$] array to reflect the new fanout counts of the nodes after best[r] has been de-selected.

The main IMPROVE-COVER-EXACT-AREA procedure processes each representative in topological order. For each representative And node r, it iterates through the non-trivial cuts. If c is a non-trivial cut of r, it sets c to be the best cut of r, and then invokes RECURSIVE-SELECT to estimate the incremental cost of implementing

Listing 5.5 IMPROVE-COVER-EXACT-AREA($\widehat{G}$, best[·], nref[·])

Input: AIG with choice $\widehat{G}$, a cover of $\widehat{G}$ specified through best[·] and nref[·]
Output: new cover for $\widehat{G}$ in best[·] and nref[·]

```
for each representative node r in Ĝ in topological order do
  if r is an And node then
    if nref[r] ≠ 0 then RECURSIVE-DESELECT(Ĝ, r, best, nref) fi
    area ← ∞
    best ← ∅
    for each non-trivial cut c ∈ Φ_≃(r) do
      best[r] ← c
      area1 ← RECURSIVE-SELECT(Ĝ, r, best, nref)
      area2 ← RECURSIVE-DESELECT(Ĝ, r, best, nref)
      assert area1 = area2
      if area1 < area then
        best ← c
        area ← area1
      end if
    end for
    area[r] ← area
    best[r] ← best
    if nref[r] ≠ 0 then RECURSIVE-SELECT(Ĝ, r, best, nref) fi
  end if
end for
```

Listing 5.6 SELECT($\widehat{G}, r, \text{best}[\cdot], \text{nref}[\cdot]$)

Input: AIG with choice $\widehat{G}$, representative r, best[·] and nref[·]
Output: Returns area and updates best[·] and nref[·]

```
if r is Input node then
  area ← 0
else assert r is And node
  area ← GET-LUT-AREA(best[r])
  for each node n ∈ best[r] do
    nref[n] ← nref[n] + 1
    if nref[n] = 1 then area += RECURSIVE-SELECT(Ĝ, n, best, nref) fi
  end for
end if
return area
```

Listing 5.7 RECURSIVE-DESELECT($\widehat{G}, r, \text{best}[\cdot], \text{nref}[\cdot]$)

Input: AIG with choice $\widehat{G}$, representative r, best[·] and nref[·]
Output: Returns area and updates best[·] and nref[·]

```
if r is Input node then
  area ← 0
else assert r is And node
  area ← GET-LUT-AREA(best[r])
  for each node n ∈ best[r] do
    nref[n] ← nref[n] − 1
    if nref[n] = 0 then area += RECURSIVE-DESELECT(Ĝ, n, best, nref) fi
  end for
end if
return area
```

c. It then invokes RECURSIVE-DESELECT to undo the changes made by RECURSIVE-SELECT. In this manner, the incremental cost of each cut of r is computed, and the best cut is selected and assigned to best[r] at the end.

Remark 1 Procedure IMPROVE-COVER-EXACT-AREA maintains the invariant that the cover (represented by best[$\cdot$] and nref[$\cdot$]) is valid before and after a representative has been processed. Therefore, the area cost estimated for a cut during exact area is accurate (even in the presence of choices).

Remark 2 If the representative r is already in the cover, it temporarily deselects best[r] before evaluating the incremental cost of the different cuts. Finally, after all the cuts of r have been processed, it restores the cover by selecting the (new) best cut for r.

Convergence. From Remarks 1 and 2, it follows that after processing a representative r that is initially selected in the cover, the cost of the new cover is never greater than the cost of the initial cover. Therefore, after IMPROVE-COVER-EXACT-AREA has been invoked the cost of the new cover is less than or equal to that before. Therefore, IMPROVE-COVER-EXACT-AREA may be iterated until there is no further reduction in cost. In practice, the most dramatic reductions in cost is seen in the first couple of iterations. The subsequent iterations lead to diminishing returns.

5.7 Breaking Ties during Selection

During depth-oriented selection, several cuts of a representative node may have minimum depth. In such cases, the area-flow cost can be used to break ties. Similarly, in area-oriented selection using area-flow or exact area, when several cuts have the same cost (or costs within an ϵ for area-flow), the depth can be used to break ties.

5.8 Minimizing Area Under Depth Constraints

5.8.1 Overview

Depth-oriented selection and the area-oriented selection may be combined together to optimize the area under depth constraints in the following manner:

1. Compute the depth-oriented cover using EVALUATE-MATCHES-DEPTH and COVER.
2. For each selected node r in the cover, compute required times required$[r]$. For remaining nodes, set required$[r] = \infty$. (The procedure to compute required times is presented in Section 5.8.2.)
3. Compute best$[r]$ for each representative r using a modified version of EVALUATE-MATCHES-AREA-FLOW that ensures that arrival$[r] \leq$ required$[r]$. (The modification is described in Section 5.8.3.)
4. Compute the cover using COVER.
5. Compute required times again (as in Step 2).
6. Improve area-oriented cover using a modified version of IMPROVE-COVER-EXACT-AREA that ensures that arrival$[r] \leq$ required$[r]$ for every representative r. (The modification is described in Section 5.8.3.)

In the above process, ties are broken at each step as described in Section 5.7. The last step (Step 6) may be iterated several times to reduce area as discussed in Section 5.6.3.

Note that in Step 2 if the required time of an output is less than the corresponding arrival time computed in Step 1, then there is no mapped network that satisfies the user-specified delay constraints. However, if the constraints are met in Step 2, then the final network at the end of Step 6 also satisfies the delay constraints. This is discussed in Section 5.8.3.

5.8.2 Computing Required Times

Listing 5.8 COMPUTE-REQUIRED-TIMES($\widehat{G}, O$, best[·], nref[·])

Input: AIG with choice $\widehat{G}$, set of output edges O, a cover of $\widehat{G}$ specified through best[·] and nref[·]
Output: required[r] for every representative r in $\widehat{G}$

```
for each representative node r in Ĝ do
  required[r] ← ∞
end for
for each edge (r, i) ∈ O do
  required[r] ← required time for corresponding output
end for
for each representative node r in Ĝ in reverse topological order do
  if nref[r] > 0 then
    if r is an And node then
      for each n ∈ best[r] do
        required[n] ← min(required[n], required[r] − 1)
      end for
    end if
  end if
end for
```

For completeness, in this section we review the procedure to compute required times. The pseudocode is shown in Listing 5.8. Initially, the required time for every representative is set to ∞. Then the nodes corresponding to the outputs are assigned the corresponding user-specified required times. Then, in the main loop of the procedure, the representatives are visited in reverse topological order. For a representative r that is selected in the cover, the required time is propagated to the nodes in best[r]. (The required time of a node n is the minimum of the required time propagated from each of its fanouts.)

Note that this procedure only assigns meaningful required times to the repre-

sentatives selected in the cover. For the other representatives, the required time is ∞.

5.8.3 Modifications to Area-Flow and Exact Area

For area-oriented selection under delay constraints, the area-flow and exact area procedures (ie. EVALUATE-MATCHES-AREA-FLOW and IMPROVE-COVER-EXACT-AREA) need to be modified slightly. Instead of picking the minimum area cost cut from among *all* cuts at a representative r, we pick the minimum area cost cut from only those cuts whose arrival times do not exceed the required time at r.

By meeting the required times for every representative r, we ensure that there is *a* cover that meets the required times at the outputs. This is true since, at least the cuts chosen for the selected nodes in the depth-oriented cover ν are guaranteed to meet the required times. (This is by construction — the required times are computed based on these cuts.)

5.9 Related Work

5.9.1 On Depth-Oriented Selection

Cong and Ding presented the first optimal depth-oriented selection algorithm on DAGs for FPGAs in their work on Flowmap [CD92]. They used a network flow algorithm to compute the minimum depth k-feasible cut for every node in the subject graph. The main limitation of this approach was the fact that the network flow algorithm produced only a single cut; therefore, it was difficult to optimize area under depth constraints. This motivated the early work on cut enumeration in an attempt to use different cuts to obtain better area without degrading depth [CD93]. Chen and Cong later extended the cut enumeration-based approach to optimally map for depth on a

subject graph with choices due to different algebraic decompositions [CC01].

5.9.2 On Area-Oriented Selection

Farrahi and Sarrafzadeh showed that the area-oriented selection problem is NP-hard [FS94], and most[3] of the work has been on finding efficient heuristics.

The early work on area-oriented selection in the Chortle system followed Keutzer's scheme of partitioning the DAG into a forest of trees and mapping each tree optimally for area using dynamic programming [FRC90]. (Although the notion of cuts is not explicitly present in this work, they essentially enumerate the cuts of a tree.) An interesting feature of Chortle is the ability to explore different decompositions of multi-input gates in a tree to find the best mapping. This is done without having explicit choices which makes the algorithm more difficult to understand; but note that the essential idea behind choices is already present in Chortle! A later improvement to Chortle, Chortle-crf, suggested using a bin-packing technique to avoid exhaustive enumeration of different decompositions to improve the speed of Chortle [FRV91]. (This is the "c" in Chortle-crf.)

Partitioning a DAG into a forest of trees prevents optimizations that can take place across tree boundaries, and there are two benefits in particular of *not* partitioning. First, short reconvergent paths may be covered by a single LUT, thus reducing the total number of LUTs required. This was first explored in Chortle-crf (the "r"), and later by Cong and Ding in their work on duplication-free mapping [CD93]. Cong and Ding show that duplication-free mapping can be done optimally by a dynamic programming approach. Second, by allowing some multi-fanout nodes to be part of two or more trees (i.e. allowing them to be duplicated), the total number of LUTs required for the mapping may be reduced. Again, this was explored in Chortle-crf

[3] Chowdhury and Hayes formulated the selection problem as a mixed integer linear program. However, large circuits had to be partitioned in order to keep the run-time reasonable [CH95a].

(the "f") and by Cong and Ding [CD93]. We note that Cong and Ding report better results than Chortle-crf, in part due to the use of optimal duplication-free mapping followed by post-processing to introduce duplication wherever beneficial.

The subsequent work in area-oriented selection, therefore, explores different heuristics for mapping directly on DAGs. The work on CutMap explores an network flow-based algorithm, similar to FlowMap, but with the goal of reducing the number of LUTs by choosing cuts whose inputs have already been selected in the cover [CH95b].

The work of Cong *et al.* on Praetor introduces the notion of area-flow[4] for optimal duplication-free mapping [CWD99]. For each node n in the subject graph, they enumerate all k-feasible cuts of n that are contained within the maximum fanout free cone (MFFC) of n. They show that if the cost of such a cut is its area-flow, then the mapping obtained by dynamic programming is the optimal duplication-free mapping. They then try this heuristic directly with a larger set of k-feasible cuts (by relaxing the requirement that the cuts be limited to the MFFC) and report good results.[5]

The work on IMAP pushes the area-flow concept further with a heuristic to estimate the fanout of a LUT in the mapped network [MBV04; MBV06]. In each mapping iteration, the fanout estimate of the previous round and the actual fanout of the mapped network obtained in the previous round are used to predict the fanout of the next round. Manohararajah *et al.* show that fanout estimation helps improve the area by about 4% over several rounds.

Finally, the work on DAOmap explores a large number of heuristics for minimizing area under delay constraints [CC04]. These include adjusting the area-flow cost with some additional metrics to capture the effect of reconvergent paths, etc. and an iterative cut selection procedure that, during covering, tries to choose cuts whose inputs have already been chosen.

[4] They call it *effective area cost*.

[5] The gains of this generalization are reported in the work on IMAP [MBV06]. The gains range from about 10% for $k = 4$ to about 30% for $k = 6$ which also agrees with our experience.

In our approach, following the work on Praetor and IMAP, we use area-flow as a starting point, and then iteratively improve it using exact area. Compared to DAOmap, an obvious advantage of our approach is simplicity. We obtain better results (on average, but not always — see Section 7.1.1) even without lossless synthesis using only two simple heuristics which do not involve empirically determined magic constants. The simplicity of the heuristics also leads to faster run-time. A less obvious advantage is ability of our approach to handle choices. The heuristics used by DAOmap would be difficult to extend to an AIG with choice where there are many "fake" fanouts due to choices.

We have surveyed only the most relevant literature on the selection problem. We refer the reader to the survey by Chen *et al.* for a more comprehensive overview of the literature, and in particular for selection criteria other than area and depth [CCP06, Chapter 3].

Chapter 6

Match Selection for Standard Cells

6.1 Overview

In this chapter we look at the match selection problem in the context of standard cell mapping with a special focus on area-oriented selection. The goal of selection is to construct the best mapped network by selecting a suitable set of matches. Although this chapter does not depend on Chapter 5, the reader is encouraged to read Chapter 5 if he has not done so since the area-oriented selection is easier to understand in the FPGA setting.

We begin with reviews of cost models for standard cell mapping, and of the timing model and associated static timing analysis operations (Section 6.2). Next, we introduce the notion of *electrical edges* (Section 6.3). Although the selection algorithm can be formulated without the concept of electrical edges, using it makes the formulation particularly simple, especially w.r.t. matches corresponding to single-input gates. Correct handling of such matches is tricky in implementations, and most bugs in our early implementations were due to incorrect handling of single-input matches. Next, we use the notion of electrical edges to define a *cover* for standard cell mapping

(Section 6.4).

The rest of the chapter follows the plan of Chapter 5. We begin our discussion of selection algorithms with a generic selection procedure where the cost function is abstracted away (Section 6.5). Then, we review delay-oriented selection in this context (Section 6.6). Then, we present the main contribution of this chapter which is the area-oriented selection algorithm for standard cell mapping (Section 6.7). After that, we discuss briefly how area and delay costs are used to break ties during selection (Section 6.8). Finally, we look at how area-oriented selection may be combined with delay-oriented selection to improve area under delay constraints (Section 6.9).

We postpone the detailed discussion of related work in the literature to the end of the chapter (Section 6.10).

Remark For simplicity of presentation, in this chapter we deal with np-matches instead of supergates, and we shall often say "match" to mean np-match. Extending the selection algorithms to handle supergates is straightforward.

6.2 Cost Model

We assume the load-independent model of delay where the delay of a gate is independent of the capacitive load driven by the gate in the network [GLH+95]. This model is useful since during technology mapping it is difficult to estimate the load driven by a gate. Hence it is common to create a load-independent abstraction of the actual technology library where each functional family of gates (where the members of the family differ in their drive-strengths) is collapsed into a single gate with a fixed delay that does not depend on the load. After mapping to this abstract library, the actual drive-strengths for gates in the mapped network are chosen later on in a separate sizing operation (which may be done in conjunction with buffering and placement).

In what follows we assume that the actual library has been abstracted to create a load-independent library.

Delay. Let g be a single-output gate with output pin o. In the constant delay model, the *pin-to-pin delay* from an input pin p of g to o is specified by a tuple of four extended real numbers[1] $d_p = (d_p^{\mathrm{RR}}, d_p^{\mathrm{FR}}, d_p^{\mathrm{RF}}, d_p^{\mathrm{FF}})$. In this tuple, d_p^{RR} is the delay from the **r**ising edge at p to the **r**ising edge at the output of g, d_p^{FR} is the delay from the **f**alling edge at p to the **r**ising edge at the output of g, etc. The pin-to-pin delays are specified in the library file.

Example 6.1 Let g be a NAND2 gate with input pins a and b and output z. Now, $\mathrm{expr}(g) = \texttt{not}(\texttt{and}(a, b))$. The load independent delay from pin a to pin z for a 90 nm technology might be specified by the tuple $(-\infty, 44, 33, -\infty)$ where the units of time are picoseconds. Note that since input a is negative unate, some components of the delay tuple are $-\infty$.

The delay of the (single-input) WIRE gate introduced for matching purposes in Section 4.5.3 is $(0, -\infty, -\infty, 0)$.

In delay-oriented selection the goal is to obtain a mapped network that has the "smallest" arrival time at the outputs. In the constant delay model, an arrival time a is specified by a pair of extended real numbers (r, f) where r is the arrival time of the rising edge and f is the arrival time of the falling edge. Since an arrival time is a pair of extended reals, the set of arrival times has a natural partial order. However, for delay-oriented selection, we need to impose a total order on arrival times. This is done by defining an absolute value of an arrival time $a = (r, f)$, denoted by $|a|$ as follows:

$$|a| = \max(r, f)$$

[1] This is the set of real numbers extended with ∞ and $-\infty$.

Now, we can define $\leq$ operator as follows:

$$a_1 \leq a_2 \quad \text{iff} \quad |a_1| \leq |a_2|$$

Finally, we define the minimum of two arrival times a_1 and a_2 as follows:

$$\min(a_1, a_2) = \begin{cases} a_1 & : \text{ if } a_1 \leq a_2 \\ a_2 & : \text{ otherwise} \end{cases}$$

Next, we define the sum of an arrival time $a = (r, f)$ and a pin-to-pin delay $d = (d^{\text{RR}}, d^{\text{FR}}, d^{\text{RF}}, d^{\text{FF}})$ to be an arrival time $a' = (r', f')$ where

$$\begin{aligned} r' &= \max(r + d^{\text{RR}}, f + d^{\text{FR}}) \\ f' &= \max(r + d^{\text{RF}}, f + d^{\text{FF}}) \end{aligned}$$

and we write $a' = a + d$ for convenience. Let P be the set of input pins of g. For $p \in P$, let a_p be the arrival time at p. The arrival time a at the output of g is given by

$$a = \max_{p \in P} (a_p + d_p)$$

where the maximum of two arrival times $a_1 = (r_1, f_1)$ and $a_2 = (r_2, f_2)$ is $\max(a_1, a_2) = (\max(r_1, r_2), \max(f_1, f_2))$.

Area. The area of a gate g is a real number denoted by $\text{AREA}(g)$. The area of the gate WIRE introduced in Section 4.5.3 is 0.

6.3 Electrical Edge

We now introduce the concept of an electrical edge and some related notions in preparation for the formal definition of a cover. The motivation for this machinery is

to avoid cycles in the mapped network (and also in the cover). The reader may wish to review the discussion on cycles in Section 4.5.3 at this point.

Let E be the set of edges in an AIG with choice $\widehat{G}$. We call members of the set $L = E \times \{\text{singular}, \text{regular}\}$ *electrical edges*. Intuitively, from now on we shall associate singular matches of the edge r in $\widehat{G}$ with the electrical edge $(r, \text{singular})$ and regular matches of r with the electrical edge $(r, \text{regular})$. An electrical edge (r, d) is called a *representative electrical edge* if r is a representative edge.

We extend the notion of support of an np-match m to include indicate electrical edges. If m is an np-match, then we define the function esupport(m) that returns the set of *electrical* edges in the support of m as follows:

$$\text{esupport}(m) = \begin{cases} \{(r, \text{regular}) \mid r \in \text{support}(m)\} & : \quad m \text{ is singular} \\ \{(r, \text{singular}) \mid r \in \text{support}(m)\} & : \quad m \text{ is regular} \end{cases}$$

The key point to note is that the electrical edge in the esupport of a singular match is the regular electrical edge, and vice versa. The reason for this is as follows: Whenever we select a singular match m (a buffer, inverter or WIRE) for an edge e, we also need to select the regular match of the edge in support(m).[2] With the introduction of electrical edges, this is enforced by requiring the match for the electrical edge in esupport(m) to be chosen. Since we associate regular matches with the regular electrical edges and singular with singular electrical edges, the electrical edge in esupport(m) is defined to be regular. Similarly, the reader can verify that this definition works for the case of regular matches.

[2] Since the edge in support(m) is either e or COMPLEMENT(e), selecting another singular match for it may lead to cycles. This was discussed in detail in Section 4.5.3.

6.4 Cover

Let $\widehat{G}$ be an AIG with choice. Intuitively, the collection of regular and singular matches for every representative is an implicit description of a set of mapped networks. Let $\mathcal{M}$ denote this set of mapped networks. The task of selection is to explicitly identify one mapped network in $\mathcal{M}$ that is optimal w.r.t. the given cost function.

The notion of a cover is used to specify an arbitrary element $M \in \mathcal{M}$. Let $\mathcal{E}$ be the set of representative electrical edges of $\widehat{G}$. Let O be the set of output edges of $\widehat{G}$. Let ν be a map that associates with every $(r, d) \in \mathcal{E}$ either an np-match m of r or the symbol NIL. If $\nu((r, d)) \neq$ NIL, then (r, d) is said to be *selected* in ν and $\nu((r, d))$ is said to be the selected match of (r, l), or simply *the* match of (r, d). The map ν is a *cover* if the following four conditions are satisfied:

1. If $r \in O$ then $(r, \text{singular})$ is selected in ν.
2. If (r, d) is selected in ν then (r', d') is also selected in ν for every $(r', d') \in \text{esupport}(\nu((r, d)))$.
3. If $l = (r, \text{singular})$ is selected in ν then $\nu(l)$ is a singular match.
4. If $l = (r, \text{regular})$ is selected in ν then $\nu(l)$ is a regular match.

A cover ν is a complete description of a mapped network in the following manner: For each representative electrical edge l selected in ν, there is a net n_l in the mapped network. If $\nu(l) =$ PI-MATCH then n_l is driven by the input port corresponding to l in the mapped network. Otherwise, n_l is driven by a gate g_l in the mapped network. Clearly, g_l is an instance of the gate associated with the match $\nu(l)$.[3] The input pins of g_l are connected to the nets corresponding to the electrical edges in the esupport$(\nu(l))$ (which by Condition 2 are also selected in ν).

[3]We continue to think of WIRE as a gate, but it is realized as a wire in the mapped network.

The four conditions above ensure that the mapped network corresponding to a cover ν is a valid one. Condition 1 ensures that there are nets corresponding to each output in the mapped network. Condition 2 ensures that the inputs to each gate are also present in the mapped network. Conditions 3 and 4 capture the idea of separating singular matches from regular matches to prevent cycles as discussed in Section 6.3.

To summarize, a cover is a complete description of a mapped network. In the subsequent discussion, we shall deal with mapped networks though covers. The goal of selection now becomes to select the optimal cover from among the possible covers.

6.5 Overview of the Selection Procedure

In this section we present an overview of the selection algorithm. For simplicity of exposition, in this section, we do not deal with specific cost functions. The input to the selection procedure is an AIG with choices $\widehat{G}$. We assume that for each representative r, the set of singular and regular np-matches i.e. singular$[r]$ and regular$[r]$ has already been computed by Procedure FIND-MATCHES-NP as discussed in Section 4.5.4. The output of the selection procedure is a cover that optimizes the given cost function.

For later use, we define an auxiliary function matches$(\cdot)$ on electrical edges to associate singular matches with singular electrical edges and regular matches with regular edges. If $l = (r, d)$ is an electrical edge, then

$$\text{matches}(l) = \begin{cases} \text{singular}[r] & : \quad \text{if } d = \text{singular} \\ \text{regular}[r] & : \quad \text{otherwise} \end{cases}$$

The selection procedure is divided into two sub-procedures EVALUATE-MATCHES and COVER. The sub-procedure EVALUATE-MATCHES (Listing 6.1) is called first. For

Listing 6.1 EVALUATE-MATCHES($\widehat{G}$)

Input: AIG with choice $\widehat{G}$
Output: best[l] for every representative electrical edge l in $\widehat{G}$

```
for each representative node r in Ĝ in topological order do
  for each d in [regular, singular] in order do
    for each i ∈ {0, 1} do
      l ← ((r, i), d)
      if matches(l) ≠ ∅ then
        cost[l] ← min_{m ∈ matches(l)} EVAL-MATCH-COST(m)
        best[l] ← arg min_{m ∈ matches(l)} EVAL-MATCH-COST(m)
      else
        cost[l] ← ∞
        best[l] ← NIL
      end if
    end for
  end for
end for
```

each representative electrical edge l, it evaluates the best[4] np-match and stores it in best[l]. This is done by processing the representative nodes of $\widehat{G}$ in topological order starting from inputs. For each representative r, it first computes the best regular np-match for each edge. Next, it computes the best singular np-match for each edge. Thus the best matches for all 4 electrical edges associated with a node r are computed. It is important to note that the evaluation of regular matches at a node precedes the evaluation of singular matches. (The two edges associated with a node can be processed in either order.)

Although we defer the specification of the specific cost function to later, one should think of the cost of a match as a "cumulative" measure that includes the cost of the

[4] according to the cost function abstracted here by the function EVAL-MATCH-COST

Listing 6.2 COVER($\widehat{G}, O, \text{best}[\cdot]$)

Input: AIG with choice $\widehat{G}$, set of output edges O, best[l] for each representative electrical edge l
Output: nref[l] for every electrical representative l in $\widehat{G}$

```
for each representative electrical edge l in Ĝ do
  nref[l] ← 0
end for
for each edge r ∈ O do
  nref[(r, singular)] ← nref[(r, singular)] + 1
end for
for each representative node r in Ĝ in reverse topological order do
  for each d in [singular, regular] in order do
    for each i ∈ {0, 1} do
      l ← ((r, i), d)
      if nref[l] > 0 then
        for each l′ ∈ esupport(best[l]) do
          nref[l′] ← nref[l′] + 1
        end for
      end if
    end for
  end for
end for
```

subnetwork below it. Thus cost[l] is used by EVAL-MATCH-COST to determine the cost of matches (in the fanout) whose esupport contains l. For now, also note that input nodes are treated specially.

If no matches exist for an electrical edge l (such as $((r, 1), \text{regular})$ where r is an Input node), then it is assigned NIL as the best match with infinite cost. Note that l will not be in the esupport of any other match m (in the transitive fanout of l) since m would not be valid (Section 4.5.4).

Intuitively, EVALUATE-MATCHES constructs part of the cover. For each electrical

representative l, best[l] stores the selected match of l if l were to be selected in the cover. To complete the construction of the cover, the sub-procedure COVER is called next. The pseudo-code of COVER is shown in Listing 6.2. It constructs a cover in a straightforward manner by enforcing Conditions 1 and 2 of Section 6.4. (Conditions 3 and 4 are enforced by EVALUATE-MATCHES.) Note that the procedure COVER does not depend on the cost function.

For each electrical representative l, COVER also computes nref[l] which is the number of fanouts of the gate corresponding to l in the mapped network. (This fanout count includes fanouts to output ports of the mapped network in addition to the fanouts to other gates in the network.) The maps best[$\cdot$] and nref[$\cdot$] define a cover ν in the following manner:

$$\nu(l) = \begin{cases} \text{best}[l] & : \text{ if nref}[l] > 0 \\ \text{NIL} & : \text{ otherwise} \end{cases}$$

6.6 Delay-Oriented Selection

In delay-oriented selection the goal is to choose a mapped network (or, equivalently, a cover) with minimum delay i.e. minimum arrival time at the outputs. This is done in the standard manner by choosing for each electrical representative l a match that minimizes the arrival time.

As discussed in Section 6.5, delay-oriented selection is split into two sub-procedures. Procedure EVALUATE-MATCHES-DELAY is run first to select the best match for each electrical representative l. Next, the COVER procedure (Listing 6.2) is invoked to complete the construction of the cover.

The pseudo-code for EVALUATE-MATCHES-DELAY is shown in Listing 6.3. It is a specialization of the generic EVALUATE-MATCHES procedure shown in Listing 6.1 for delay-oriented selection. For an electrical representative l, arrival[l] records the best

Listing 6.3 EVALUATE-MATCHES-DELAY($\widehat{G}$)

Input: AIG with choice $\widehat{G}$
Output: best[l] for every representative electrical edge l in $\widehat{G}$

```
for each representative node r in Ĝ in topological order do
  for each d in [regular, singular] in order do
    for each i ∈ {0, 1} do
      l ← ((r, i), d)
      if matches(l) ≠ ∅ then
        arrival[l] ← min_{m ∈ matches(l)} ARRIVAL-TIME(m)
        best[l] ← arg min_{m ∈ matches(l)} ARRIVAL-TIME(m)
      else
        arrival[l] ← ∞
        best[l] ← NIL
      end if
    end for
  end for
end for
```

possible arrival time at l. It is decided by examining all the matches at l, and picking the match with the least[5] arrival time l, and this match is stored in best[l].

The arrival time of a match m is computed as follows. If m is a PI-MATCH then ARRIVAL(m) is set to the user supplied arrival time for the input. For an np-match $m = (c, g, \sigma, \tau)$, arrival(m) is given by

$$\text{ARRIVAL-TIME}(m) = \max_{p \in P} (a_p + d_p)$$

where P is the set of pins of gate g, a_p is the arrival time at the electrical edge $l \in$ esupport(m) corresponding to the pin p of match m, and d_p is the delay from pin p to output of g as specified in the library.

[5] using the definitions of $\leq$ and min set forth in Section 6.2

6.7 Area-Oriented Selection

Area-oriented selection is done using two separate selection procedures which use different heuristics to minimize area. The first area-oriented selection heuristic is called *area-flow*. Area-flow is used to construct an initial cover with "good" area. This cover is then greedily optimized by a second heuristic called *exact area*.

6.7.1 Area-Flow

Similar to delay-oriented selection, area-oriented mapping using the area-flow heuristic is split into two sub-procedures. Procedure EVALUATE-MATCHES-AREA-FLOW runs first and assigns a best match to each electrical representative l. Next, procedure COVER is invoked which completes the construction of the cover.

The pseudocode for EVALUATE-MATCHES-AREA-FLOW is shown in Listing 6.4. For an electrical representative l, area$[l]$ measures the area (i.e. the cost) to implement l in the mapped network and is the minimum area-flow cost among all the matches of l. For PI-MATCH matches the area-flow is zero. Otherwise, for an np-match $m = (c, g, \sigma, \tau)$, the area-flow cost is given by the following formula:

$$\text{AREA-FLOW}(m) = \text{AREA}(g) + \sum_{l' \in \text{esupport}(m)} \frac{\text{area}[l']}{\text{fanouts}(l')}$$

where area(g) is the area of gate g and fanouts(l) for an electrical edge $l = ((r, i), d)$ is defined as follows:

$$\text{fanouts}(l) = \begin{cases} \text{fanouts}(r) & : \text{ if } d = \text{singular} \\ 1 & : \text{ otherwise} \end{cases}$$

where fanouts(r)[6] is the number of fanouts of the representative node r in the AIG

[6] Observe that we do not distinguish between the fanouts of the positive edge of r and the negative

Listing 6.4 EVALUATE-MATCHES-AREA-FLOW($\widehat{G}$)

Input: AIG with choice $\widehat{G}$
Output: best[l] for every representative electrical edge l in $\widehat{G}$

```
for each representative node r in Ĝ in topological order do
  for each d in [regular, singular] in order do
    for each i ∈ {0, 1} do
      l ← ((r, i), d)
      if matches(l) ≠ ∅ then
        area[l] ← min_{m ∈ matches(l)} AREA-FLOW(m)
        best[l] ← arg min_{m ∈ matches(l)} AREA-FLOW(m)
      else
        area[l] ← ∞
        best[l] ← NIL
      end if
    end for
  end for
end for
```

with choice $\widehat{G}$. (Note that since r is in the support of m, r fans out to at least one node. Therefore, fanouts(r) $\neq 0$.)

Intuitively, this definition of the area cost takes into account possible sharing at a multi-fanout node. The cost of implementing the node is distributed among the different fanouts. This would be completely accurate if fanouts(l) would reflect the actual fanout of the gate chosen for l in the mapped network. However, since the fanouts of l are mapped after the best match for l is selected, we simply *estimate* the fanout by assuming it to be the same as that in the AIG. This estimation is particularly inaccurate in the presence of choices, since choices increase fanout.

Note that in the special case where $\widehat{G}$ is a tree, and every equivalence class has

edge of r.

only one representative (i.e. $\widehat{G}$ is an AIG without choices), the area flow metric is the same as area since fanouts(l) = 1 for all electrical representatives l.

6.7.2 Exact Area

Unlike area-flow, exact area does not create a cover from scratch. Instead, it greedily and incrementally optimizes an existing cover to increase sharing. The pseudocode for the main exact area procedure IMPROVE-COVER-EXACT-AREA is shown in Listing 6.5. It invokes two auxiliary procedures RECURSIVE-SELECT and RECURSIVE-DESELECT whose pseudocode is presented in Listings 6.6 and 6.7.

Procedure RECURSIVE-SELECT when invoked on an electrical representative l is used to compute the *incremental* area cost of selecting best[l] assuming that the rest of the cover is fixed. The incremental area cost is the cost of selecting the gates that are only used to implement best[l] (and not used anywhere else in the mapped network). In other words this is the cost of the *maximum fanout-free cone* of l in the mapped network. This cost is estimated by recursively selecting the best matches for the nodes in best[l] that are not currently selected. As a side effect of this routine, the nref[$\cdot$] array is updated to reflect the new fanout counts of the nodes after best[l] has been selected.

Procedure RECURSIVE-DESELECT is the opposite of RECURSIVE-SELECT. When invoked on a representative l, RECURSIVE-DESELECT frees the gates that have been selected exclusively to implement best[l]. Like RECURSIVE-SELECT, it returns the incremental area needed to select best[l]. As a side effect it updates nref[$\cdot$] array to reflect the new fanout counts of the nodes after best[l] has been de-selected.

The main IMPROVE-COVER-EXACT-AREA procedure processes each representative node in topological order. For each representative r, it processes the four electrical edges associated with r. For each match m of an electrical edge l associated with r, it

Listing 6.5 IMPROVE-COVER-EXACT-AREA($\widehat{G}$, best[·], nref[·])

Input: AIG with choice $\widehat{G}$, a cover of $\widehat{G}$ specified through best[·] and nref[·]
Output: new cover for $\widehat{G}$ in best[·] and nref[·]

```
for each representative node r in Ĝ in topological order do
  for each d in [regular, singular] in order do
    for each i ∈ {0, 1} do
      l ← ((r, i), d)
      if matches(l) ≠ ∅ then
        if nref[l] ≠ 0 then RECURSIVE-DESELECT(Ĝ, l, best, nref) fi
        area ← ∞
        best ← ∅
        for each np-match m ∈ matches(l) do
          best[l] ← m
          area1 ← RECURSIVE-SELECT(Ĝ, l, best, nref)
          area2 ← RECURSIVE-DESELECT(Ĝ, l, best, nref)
          assert area1 = area2
          if area1 < area then
            best ← m
            area ← area1
          end if
        end for
        area[l] ← area
        best[l] ← best
        if nref[l] ≠ 0 then RECURSIVE-SELECT(Ĝ, l, best, nref) fi
      end if
    end for
  end for
end for
```

Listing 6.6 SELECT($\widehat{G}, l, \text{best}[\cdot], \text{nref}[\cdot]$)

Input: AIG with choice $\widehat{G}$, electrical representative l, best[$\cdot$] and nref[$\cdot$]
Output: Returns area and updates best[$\cdot$] and nref[$\cdot$]

```
assert best[l] ≠ NIL
if best[l] = PI-MATCH then
  area ← 0
else assert best[l] is an np-match (c, g, σ, τ)
  area ← AREA(g)
  for each electrical representative l' ∈ esupport(best[l]) do
    nref[l'] ← nref[l'] + 1
    if nref[l'] = 1 then area += RECURSIVE-SELECT(Ĝ, l', best, nref) fi
  end for
end if
return area
```

assigns m temporarily as the best match for l, and then invokes RECURSIVE-SELECT to estimate the incremental cost of implementing m. It then invokes RECURSIVE-DESELECT to undo the changes made by RECURSIVE-SELECT. In this manner, the incremental cost of each match of l is computed, and the best match is selected and assigned to best[l] at the end.

Remark 1 Procedure IMPROVE-COVER-EXACT-AREA maintains the invariant that the cover (represented by best[$\cdot$] and nref[$\cdot$]) is valid before and after an electrical representative has been processed. Therefore, the area cost estimated for a cut during exact area is accurate (even in the presence of choices).

Remark 2 If the representative l is already in the cover, it temporarily deselects best[l] before evaluating the incremental cost of the different matches. Finally, after all the matches of l have been processed, it restores the cover by selecting the (new)

Listing 6.7 RECURSIVE-DESELECT($\widehat{G}, l, \text{best}[\cdot], \text{nref}[\cdot]$)

Input: AIG with choice $\widehat{G}$, electrical representative l, best[$\cdot$] and nref[$\cdot$]
Output: Returns area and updates best[$\cdot$] and nref[$\cdot$]

```
assert best[l] ≠ NIL
if best[l] = PI-MATCH then
  area ← 0
else assert best[l] is an np-match (c, g, σ, τ)
  area ← AREA(g)
  for each electrical representative l' ∈ esupport(best[l]) do
    nref[l'] ← nref[l'] − 1
    if nref[l'] = 0 then area += RECURSIVE-DESELECT(Ĝ, l', best, nref) fi
  end for
end if
return area
```

best match for l.

Convergence. From Remarks 1 and 2, it follows that after processing an electrical representative l that is initially selected in the cover, the cost of the new cover is never greater than the cost of the initial cover. Therefore, after IMPROVE-COVER-EXACT-AREA has been invoked the cost of the new cover is less than or equal to that before. Therefore, IMPROVE-COVER-EXACT-AREA may be iterated until there is no further reduction in cost. In practice, the most dramatic reductions in cost is seen in the first couple of iterations. The subsequent iterations lead to diminishing returns.

6.8 Breaking Ties during Selection

During delay-oriented selection, several matches of an electrical representative may have minimum delay (or near-minimum delay). In such cases, the area-flow cost can be used to break ties. Similarly, in area-oriented selection using area-flow or exact area, when several matches have the same cost (or costs within an ϵ), the delays of the matches can be used to break ties.

6.9 Minimizing Area Under Delay Constraints

6.9.1 Overview

Delay-oriented selection and the area-oriented selection may be combined together to optimize the area under delay constraints in the following manner:

1. Compute the delay-oriented cover using EVALUATE-MATCHES-DELAY and COVER.
2. For each selected electrical edge l in the cover, compute required times required[l]. For remaining electrical edges, set required[l] = (∞, ∞). (The procedure to compute required times is presented in Section 6.9.2.)
3. For each electrical representative l, compute best[l] using a modified version of EVALUATE-MATCHES-AREA-FLOW that ensures that arrival[l] $\leq$ required[l]. (The modification is described in Section 6.9.3.)
4. Compute the cover using COVER.
5. Compute required times again (as in Step 2).
6. Improve area-oriented cover using a modified version of IMPROVE-COVER-EXACT-AREA that ensures that arrival[l] $\leq$ required[l] for every representative l. (The modification is described in Section 6.9.3.)

In the above process, ties are broken at each step as described in Section 6.8. The last step (Step 6) may be iterated several times to reduce area as discussed in Section 6.7.2.

Note that in Step 2 if the required time of an output is less than the corresponding arrival time computed in Step 1, then there is no mapped network that satisfies the user-specified delay constraints. However, if the constraints are met in Step 2, then the final network at the end of Step 6 also satisfies the delay constraints. This is discussed in Section 6.9.3.

6.9.2 Computing Required Times

For completeness, in this section we review the procedure to compute required times. In the constant delay model, a required time q is specified by a pair of extended real numbers (r, f) where r is the required time for the rising edge and f is the required time for the falling edge. Before describing the procedure to compute required times, we define two operations on required times.

First, we define the difference of a required time $q = (r, f)$ and a pin-to-pin delay $d = (d^{\mathrm{RR}}, d^{\mathrm{FR}}, d^{\mathrm{RF}}, d^{\mathrm{FF}})$ to be a required time $q' = (r', f')$ where

$$\begin{aligned} r' &= \min(r - d^{\mathrm{RR}}, f - d^{\mathrm{FR}}) \\ f' &= \min(r - d^{\mathrm{RF}}, f - d^{\mathrm{FF}}) \end{aligned}$$

and we write $q' = q - d$ for convenience.

Second, we define the minimum of two required times $q_1 = (r_1, f_1)$ and $q_2 = (r_2, f_2)$ as follows:

$$\min(q_1, q_2) = (\min(r_1, r_2), \min(f_1, f_2))$$

We note that the min and $-$ operations on required times are analogous to the max and $+$ operations on arrival times. (The min operation on required times is not the same as the min operation on arrival times.)

Listing 6.8 COMPUTE-REQUIRED-TIMES($\widehat{G}, O, \text{best}[\cdot], \text{nref}[\cdot]$)

Input: AIG with choice $\widehat{G}$, set of output edges O, a cover of $\widehat{G}$ specified through best[$\cdot$] and nref[$\cdot$]
Output: required[l] for every electrical representative l in $\widehat{G}$

```
for each electrical representative node l in Ĝ do
  required[l] ← (∞, ∞)
end for

for each edge r ∈ O do
  required[(r, singular)] ← required time for corresponding output
end for

for each representative node r in Ĝ in reverse topological order do
  for each d in [singular, regular] in order do
    for each i ∈ {0, 1} do

      l ← ((r, i), d)

      if nref[l] > 0 then

        if best[l] ≠ PI-MATCH then

          Let g be the gate associated with best[l]

          for each l' ∈ esupport(best[l]) do

            Let p be the input pin corresponding to l in best[l]

            Let d be the delay of the pin p to output of g

            required[l'] ← min(required[l'], required[l] − d)

          end for

        end if

      end if

    end for
  end for
end for
```

The procedure to compute required times is shown in Listing 6.8. Initially, the required time for every electrical representative is set to (∞, ∞). Then the singular electrical edges corresponding to the output edges are assigned the corresponding user-specified required times. Then, in the main loop of the procedure, the representatives nodes are visited in reverse topological order. For a representative r, its singular edges are processed before its regular edges. For an electrical edge l that is selected in the cover, the required time for l is propagated down to the electrical edges in the esupport of best[l]. (The required time of an electrical edge l is the minimum of the required time propagated from each of its fanouts.)

Note that this procedure only assigns meaningful required times to the electrical edges of representatives selected in the cover. For the other representatives, the required time is (∞, ∞).

6.9.3 Modifications to Area-Flow and Exact Area

For area-oriented selection under delay constraints, the area-flow and exact area procedures (ie. EVALUATE-MATCHES-AREA-FLOW and IMPROVE-COVER-EXACT-AREA) need to be modified slightly. Instead of picking the minimum area cost match from among *all* matches at an electrical representative l, we pick the minimum area cost match from only those matches whose arrival times do not exceed the required time at l.

By meeting the required times for every electrical representative l, we ensure that there is *a* cover that meets the required times at the outputs. This is true since, at least the matches chosen for selected nodes in the delay-oriented cover ν are guaranteed to meet the required times. (This is by construction — the required times are computed based on these matches.)

6.10 Related Work

6.10.1 On Delay-Oriented Selection

As discussed in Section 1.7, Keutzer showed that if the subject graph is partitioned into trees, then each tree can be mapped optimally for area and delay [Keu87]. Lehman *et al.* showed that even if choices are added, each tree can be mapped optimally for area and delay [LWGH95]. (All delay optimality results are on a model where the rise and fall delays are the same; Murgai showed that with different rise and fall delays the problem is NP-hard [Mur99].)

However, as noted in Section 5.9 for FPGA mapping, partitioning the subject graph into trees prevents optimizations that can be performed across tree boundaries. For delay-oriented mapping, Kukimoto *et al.* presented an optimal algorithm for DAGs by adapting the FlowMap algorithm from FPGA mapping [KBS98].

6.10.2 On Area-Oriented Selection

For area-oriented selection for standard cells, the problem is NP-hard on DAGs.[7] However, there appears to be little literature on heuristics to solve this problem. Furthermore, even though the corresponding FPGA problem is very similar (although some what easier due to the absence of inverters), there seems to be little cross-fertilization between the two areas. At this point we invite the reader to review Section 5.9 (on related work for area-oriented FPGA mapping), and in particular, the notion of duplication-free mapping.

Returning to the standard cell mapping literature, we find two papers that address this problem. The first paper anticipates the area-flow (or effective area) heuristic

[7] An arbitrary area-oriented k-LUT FPGA mapping problem can be trivially converted into a standard cell mapping problem (by constructing a library with all functions of k variables), and area-oriented FPGA mapping is NP-hard [FS94].

of FPGA mapping and even mentions a result similar to the optimal duplication-free result in passing [KDNG92]. The authors also suggest a modification to area-flow to account for duplications. However, somewhat surprisingly, they obtain only a 4% or so improvement over the tree mapper in MIS [Rud89]. The second paper also uses the area-flow heuristic and introduces the notion of *crossing numbers* to capture the effects of duplication [JWBO00]. The notion of crossing numbers appears very similar to the modification proposed by Kung *et al.*. The main idea is to detect (multi-fanout) nodes inside the match that would be duplicated if the match were selected, and to add the extent of these duplications to fanouts of the input nodes of the match (in addition to the fanouts of the input nodes as is already done in area-flow).

In addition, Jongeneel *et al.* consider the use of choices in the manner of Lehman *et al.*, though they limit the choices to be those obtained from different decompositions of multi-input AND gates. By limiting choices to this particular form, they are able to better estimate the crossing numbers by accounting for fanouts that arise only from choices. The main limitation of their approach is the restriction of the types of choices. In particular, their restriction would preclude lossless synthesis. Our work differs from theirs in that we do not restrict the type of choices that can be considered. Furthermore, we use area-flow to obtain an *initial* cover which is optimized further using exact area. Also note that it is possible to use exact area in conjunction with their technique to further reduce area. (Since exact area is greedy and incremental, it never makes the area worse.)

Finally, we note that there has been work on delay- and area-oriented selection using different delay models such as load-based [Rud89; Tou90] and gain-based (i.e. based on the theory of logical effort) [HWKMS03; KS04]. (The constant delay model used in this work may be seen as a special case of gain-based synthesis where every instance of a library gate is assigned a fixed gain.)

For a review of the selection problem for other cost functions such as wire-length, we refer the reader to the survey by Stok and Tiwari [HS02, Chapter 5].

Chapter 7

Experimental Results

What you cannot enforce, do not command.

— Sophocles

7.1 FPGA Mapping

7.1.1 Comparison of Area-Oriented Selection with DAOmap

In this section we compare the area-oriented selection presented in Section 5.6 against the state-of-the-art reported in the literature. The area-oriented selection procedure has been implemented in the FPGA mapping package in the ABC logic synthesis and verification system. The comparison was performed against the DAOmap system which has been demonstrated to be superior to previously reported systems [CC04].[1]

The benchmarks used for the comparison were pre-optimized in SIS using `script.algebraic` [SSL+92] followed by decomposition into two-input gates using command `dmig` in RASP [CPD96]. To ensure identical starting network structures, we used the same pre-optimized benchmarks that Chen and Cong used for their paper. Both

[1] We thank Deming Chen and Jason Cong, the authors of DAOmap, for their help with this experiment.

Name	DAOmap			Baseline		
	Depth	Area	Run-time	Depth	Area	Run-time
alu4	6	1065	0.5	6	984	0.16
apex2	7	1352	0.6	7	1216	0.19
apex4	6	931	0.7	6	899	0.18
bigkey	3	1245	0.6	3	805	0.18
clma	13	5425	5.9	13	4483	0.82
des	5	965	0.8	5.0	957	0.24
diffeq	10	817	0.6	10	830	0.17
dsip	3	686	0.5	3.0	694	0.17
elliptic	12	1965	2.0	12	2026	0.31
ex1010	7	3564	4.0	7	3151	0.59
ex5p	6	778	1.0	6	752	0.26
frisc	16	1999	1.9	15	2016	0.39
misex3	6	980	0.8	6	952	0.19
pdc	7	3222	4.6	7	2935	0.65
s298	13	1258	2.4	13	828	0.21
s38417	9	3815	3.8	9	4198	0.73
s38584	7	2987	27	7	3084	0.58
seq	6	1188	0.8	6	1099	0.22
spla	7	2734	4.0	7	2518	0.60
tseng	10	706	0.6	10	759	0.17
Ratio	1.00	1.00	1.00	1.00	0.94	0.22

Table 7.1: Comparison with DAOmap (Section 7.1.1).

mappers were given the task of mapping to 5-input LUTs to obtain the best area under depth constraints. The depth constraints were set as follows: Each benchmark was first mapped by each mapper for minimum depth. Then the arrival time of the latest output was used as the required time for each output of the benchmark. After mapping, the mapped networks were compared to the original Boolean networks using the equivalence checker in ABC [MCBE06].

The results of the comparison are presented in Table 7.1. The section of the table

labeled "DAOmap" collects the statistics for DAOmap, and that labeled "baseline"[2] for our mapper. The columns labeled "depth" in Table 7.1 give the number of logic levels of the resulting mapped networks. The values in these columns agree in all but one case (benchmark `frisc`). The agreement is expected since both mappers obtain the optimum depth for a given network structure. The one difference may be explained by minor variations in the manipulation of the subject graph (such as AIG balancing performed by ABC).

The columns labeled "area" show the number of LUTs in the resulting mapped networks. The data show that the proposed area-oriented selection procedure is competitive with DAOmap leading to about 6% fewer LUTs on average. For 13 benchmarks the proposed technique performs better than DAOmap (sometimes significantly, e.g. `clma`, `pdc`); for 6 benchmarks it does slightly worse ($< 1\%$); and only for 1 benchmark is it significantly worse (`s38417`).

The columns labeled "run-time" report the run-times of both programs in seconds. DAOmap was run on a 4 CPU 3.00 GHz computer with 510 MB of memory under Linux. ABC was run on a 1.6 GHz laptop with 1 GB of memory under Windows. We note that the run-time includes the time for reading the input network, constructing the subject graph and performing technology mapping. We find that the implementation in ABC is significantly faster (given the difference in CPU speed) than that in DAOmap and attribute the speed-up to differences in the basic data structures, improved cut computation, and simplicity of the proposed area-oriented selection method.

7.1.2 Validation of Lossless Synthesis

In this section we present experimental results that demonstrate the usefulness of lossless synthesis in the context of FPGA mapping. As discussed in Section 1.6.3, we

[2]We label our mapper "baseline" since we do not use lossless synthesis in this experiment.

do not study different ways of doing lossless synthesis, but instead demonstrate a particular method that leads to better mapped networks.

For these experiments, we use the same set of benchmarks as in Section 7.1.1 as inputs. We compare three different modes of mapping:

1. **Baseline.** This corresponds to the simple mapping without lossless synthesis as described in Section 7.1.1.

2. **Choice.** In this mode we combine three different AIGs to form an AIG with choice which is used for mapping. The first AIG corresponds to the original network. The second and third AIGs were obtained as snapshots during logic synthesis on the original network: The second network is the snapshot taken after applying the `resyn` script in ABC whereas the third network is the snapshot taken after applying the `resyn2` script to the second network. (Both `resyn` and `resyn2` are based on iterative application of AIG rewriting [MCB06a].)

3. **Choice 5x.** In this mode we study the effect of repeated application of mapping with choices. After the network has been mapped as described under mode Choice, we decompose the resulting mapped network into an AIG (by factoring the logic functions of the LUTs). The resulting network is again subjected to logic synthesis and mapping with choices as described under mode Choice. This process is repeated five times. Note that at each point in this process, the previous best mapped network is "available" since it (or, more precisely, an AIG corresponding to it) is included as one of the three AIGs that are choiced.

The results of this experiment are shown in Table 7.2. As can be seen by comparing the columns labeled "depth" and "area," the quality of the mapped network improves substantially after several iterations of choicing and mapping with choices. Each iteration generates structural variations on the currently selected best mapping

Name	Baseline			Choice			Choice 5x		
	Depth	Area	Time	Depth	Area	Time	Depth	Area	Time
alu4	6	984	0.16	6	971	1.36	6	889	6.58
apex2	7	1216	0.19	7	1170	1.52	6	1046	7.27
apex4	6	899	0.18	6	890	1.11	6	852	5.76
bigkey	3	805	0.18	3	805	1.05	3	695	7.00
clma	13	4483	0.82	11	3695	10.54	11	2788	32.83
des	5	957	0.24	5	997	1.92	5	914	10.74
diffeq	10	830	0.17	9	785	1.37	9	761	5.48
dsip	3	694	0.17	3	694	0.93	3	693	5.64
elliptic	12	2026	0.31	12	2085	1.81	12	2048	13.20
ex1010	7	3151	0.59	7	2973	3.80	7	2749	21.42
ex5p	6	752	0.26	5	671	1.60	5	515	6.92
frisc	15	2016	0.39	14	1971	2.21	13	1937	15.07
misex3	6	952	0.19	6	923	1.30	5	814	6.26
pdc	7	2935	0.65	7	2592	5.28	7	2310	29.49
s298	13	828	0.21	10	771	1.90	9	716	7.05
s38417	9	4198	0.73	8	3144	7.58	7	3035	24.87
s38584	7	3084	0.58	7	2754	6.66	6	2641	24.02
seq	6	1099	0.22	5	1035	1.67	5	819	8.04
spla	7	2518	0.60	7	2244	4.78	7	1922	23.17
tseng	10	759	0.17	9	725	0.86	8	725	4.23
Ratio	1.00	1.00	1.00	0.94	0.94	7.27	0.90	0.86	36.1

Table 7.2: Comparison with Lossless Synthesis (Section 7.1.2).

Mode	Depth	Area	Run-time
DAOmap	1.00	1.00	1.00
Baseline	1.00	0.94	0.22
Associative	0.95	0.96	-
Choice	0.94	0.88	1.60
Choice 5x	0.90	0.81	7.94

Table 7.3: Summary comparison of different mappers and mapping modes (Section 7.1.2).

and allows the mapper to combine the resulting choices even better by mixing and matching different logic structures. Iterating the process tends to gradually evolve structures that are good for the selected LUT size.

The run-times for Choice and Choice 5x includes the time for logic synthesis, detecting choices, and mapping with choices. Although the run-time increases significantly compared to Baseline, it is not a problem in practice since the run-time for Baseline is very small. For the largest benchmarks, the run-time of Choice 5x is about half a minute. (As before, these run-times are on a 1.6 GHz laptop with 1 GB of memory under Windows.)

We also compared our lossless synthesis with the technique in proposed by Chen and Cong where different decompositions for multi-input AND-gates are added as choices [CC01]. Table 7.3 shows a comparison of associative choices with the other modes and mappers considered so far. We observe that the lossless synthesis modes lead to better depth and area compared to adding associate choices.

Although we do not show the run-time for mapping with associative choices (our implementation for constructing the AIG with choice was inefficient), we note here that exhaustively adding associative decompositions greatly increases the total number of choices, which lead to many cuts. This slows down the mapper more than the relatively few choices added by the proposed lossless synthesis.

In summary, the above experiments demonstrate that lossless synthesis can substantially reduce depth and area of the mapped networks, both as a stand-alone mapping procedure and as a post-processing step applied to an already computed FPGA mapping.

7.2 Standard Cell Mapping

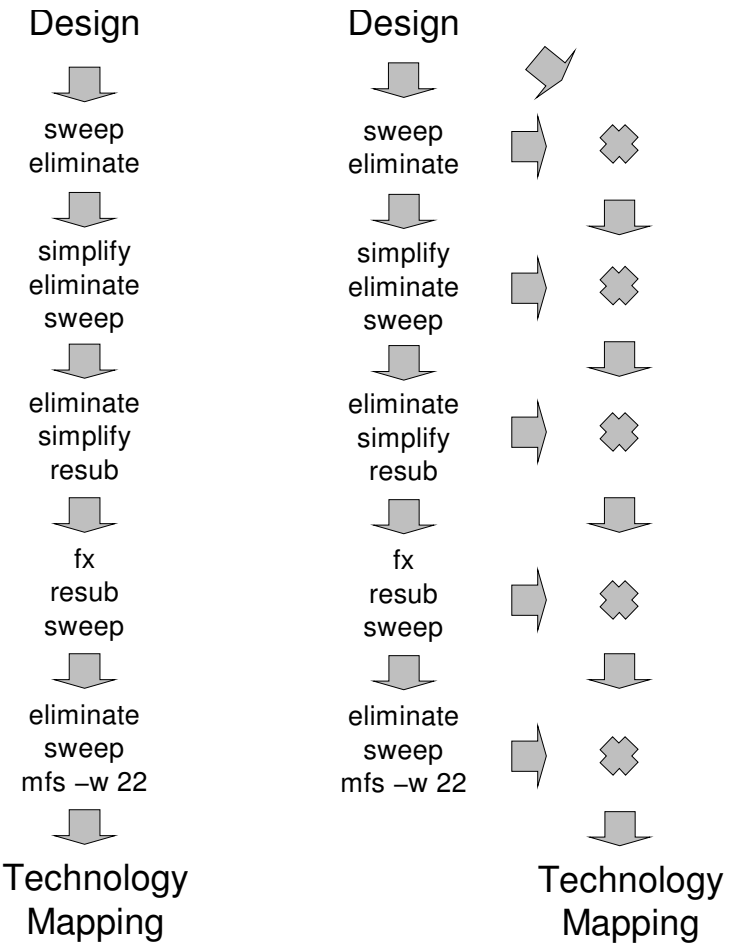
In the first four experiments (Sections 7.2.1—7.2.4), delay is the primary objective, and area is secondary. Area recovery is done using the scheme described in Section 6.9. In the last experiment (Section 7.2.5), area is used as the primary objective and delay is secondary. In all experiments, the secondary objective is used to break ties (Section 6.8).

7.2.1 Comparison of Structural Bias Reduction Techniques

The first two experiments were done to evaluate the efficacy of the different methods to reduce structural bias. The benchmarks chosen were 15 large publicly available benchmarks and they were mapped onto the *mcnc.genlib* library.

The benchmarks were initially processed using the technology independent script shown in Figure 7.1(a) and the resulting networks were converted into AIGs. During this decomposition, multi-input AND gates were decomposed in a balanced fashion to minimize delay.

To evaluate the usefulness of lossless synthesis, various snapshots were obtained during the technology independent synthesis. Each snapshot was converted into an AIG (once again, multi-input AND gates were decomposed in a balanced fashion to minimize delay). The different AIGs were combined into a single AIG with choice before mapping. The schedule according to which snapshots were collected is shown



(a) Conventional Synthesis (b) Lossless Synthesis

Figure 7.1: In lossless synthesis intermediate networks are combined to create a choice network which is then used for mapping. The intermediate networks are first converted into AIGs as described in Section 2.3 and then combined using the choice operator defined in Section 2.6.

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 7.4: Comparison of various mapper modes. B is Baseline, L is Lossless synthesis, S is Supergates (Section 7.2.1). 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.

in Figure 7.1(b). The schedule was selected so that snapshots are collected after "downhill" moves during technology independent optimization.

To evaluate the usefulness of supergates, a set of 5-input supergates was generated from the library subject to resource constraints. About 35,000 supergates were generated from the original 22 gates in the library. (About 1.5 million supergates were considered during the generation process out of which only 35,000 were non-dominated.) The supergate generation was allowed to run for 10 seconds, and in that time all level 3 supergates (with root gates having had 3 or fewer inputs) were considered. Some level 4 supergates were generated before the time out of 10 seconds was reached.

Table 7.4 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 7.4 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.

7.2.2 Experimental Comparison with Algebraic Re-writing

Since a direct comparison with the implementation of Lehman *et al.* [LWGH95] was not possible, we performed a simple experiment to estimate the effect of algebraic re-writing. As per Section 2.7.2, 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 4.8.3 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.

7.2.3 Comparison with the Mapper in SIS

Table 7.5 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 [SSL+92] 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. We note that the *run-time* comparison with the SIS mapper is not entirely fair since it is

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	0.97	3.12	1.89	16.68
b15	1.57	74.50	0.51	1.38	14579	1.09	3.75	2.08	19.22
bigkey	1.39	11.40	0.61	1.30	5284	1.45	7.50	0.40	39.18
C5315	1.44	27.20	0.71	1.28	2733	1.05	4.00	0.35	22.69
C6288	1.96	82.60	0.66	0.98	6455	1.24	2.28	1.23	8.42
C7552	1.52	23.50	0.72	1.35	3393	1.14	3.11	0.58	17.09
clma	1.47	34.30	0.73	1.25	19968	1.17	5.29	2.10	55.70
clmb	1.28	36.20	0.70	1.26	19686	1.17	5.43	2.10	55.84
dsip	1.47	8.70	0.75	1.21	5205	1.26	9.03	0.31	9.68
pj1	1.68	41.00	0.58	1.31	24345	1.11	4.54	3.15	33.15
pj2	1.72	14.80	0.77	1.25	4957	1.16	7.36	0.38	36.63
pj3	1.70	28.50	0.67	1.28	15461	1.26	4.25	2.05	36.63
s15850	1.47	33.10	0.76	1.24	5963	0.97	7.33	0.45	32.76
s35932	1.52	9.00	0.84	1.30	15242	1.02	8.77	1.38	19.93
s38417	1.59	22.00	0.71	1.26	18057	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 7.5: Comparison with SIS on public benchmarks (Section 7.2.3). The numbers for Baseline are absolute; those for SIS and B+L+S are relative to Baseline. Run-time is in seconds on a 1.6 GHz Intel laptop.

designed for load-dependent mapping.

7.2.4 Comparison with (Proprietary) 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.

Table 7.6 shows a comparison of the mapper with two other state-of-the art mappers: DAG mapper [KBS98] and GraphMap which is an independent implementation of the algorithm described by Lehman *et al.* [LWGH95] 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 7.7 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 [KBS98] that DAG mapper can produce significantly faster circuits compared to the traditional tree mapping approach [Keu87]. 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.

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
Ratio	1.00	1.00	1.20	0.85	0.94	0.81

Table 7.6: Comparison with the other mappers on industrial benchmarks (Section 7.2.4).

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
Ratio	1.00	1.00	1.00	0.76	0.88	0.04

Table 7.7: Comparison on large industrial circuits (Section 7.2.4).

7.2.5 Comparison with a Commercially Available Tool

In this experiment we compared our area-oriented selection procedure with a state-of-the-art commercial tool DC using 63 benchmarks from Altera's QUIP toolkit.[3] The benchmarks were obtained in gate-level format (from Quartus) and technology independent optimization was performed using DC by compiling it to a subset of our library consisting of only a NOR2 gate and an inverter.

The resulting circuits were then mapped using three different methods:

- **DC.** In this method, the DC mapper was invoked (using the compile command with both effort and area-recovery effort set to high) without any timing constraints, but with an area constraint of zero to ensure that the tool produces a solution with minimum area.

- **Baseline.** In this method, our mapper was invoked in area mode. Area-oriented selection was done with one pass of area-flow followed by 4 passes of exact area optimization. Ties were broken based on delay.

- **Choice 1x.** In this method, we obtained an initial mapping as in Baseline. The resulting mapped circuit was unmapped and resynthesized using two iterations of the `resyn2` script in ABC. Three snapshots were collected: the initial mapped network, the result of the first `resyn2` and the final network, and these snapshots were combined into one AIG with choice. This was used for the final mapping (again initial area-flow followed by 4 passes of exact area).

Each of the mapped circuits was then processed by DC in incremental mode to fix design rule violations (due high fanout nets), and then area and delay were measured in DC.

[3] `http://www.altera.com/education/univ/research/unv-quip.html`

Table 7.8 shows the final area in the three cases. On average Baseline is better by about 4% and Choices 1x is better by about 9% than DC. Note that Baseline is better than DC in all but 12 cases. Among those, only three cases are significantly bad: mux8_128bit, mux8_64bit, and ts_mike_fsm. Although the library does not have any 8 to 1 MUXes, it may be possible that DC performs some MUX specific optimization. Furthermore, Choices 1x is better than DC in all but two cases (mux8_128bit and ts_mike_fsm).

Table 7.9 shows the final delay in the three cases. Although delay is only a secondary objective, our method generally outperforms DC on delay. The delay with both Baseline and Choices 1x is about 16% better.

Although comparing run-times would not be fair, we note that on the largest benchmark `uoft_raytracer` (about 150K AIG And nodes) our implementation takes about 5 minutes for Baseline and 50 minutes for Choice 1x (which includes some unnecessary overhead due to the use of different programs for mapping and choicing) on a 3 GHz Pentium 4 processor. In contrast, DC takes about 130 minutes on a 1 GHz 4 processor Opteron.

Table 7.8: Comparison of area-oriented mapping with a commercial tool using Baseline and Choice 1x (Section 7.2.5). The column labelled "Impr." indicates improvements in area over DC. For example, for barrel16, Baseline leads to a 14% improvement over DC.

Name	Area				
	DC	Baseline	Impr.	Choice 1x	Impr.
barrel16	1226.02	1058.09	0.14	991.13	0.19
barrel16a	1250.17	1228.21	0.02	1087.72	0.13
barrel32	3145.72	3015.10	0.04	2891.08	0.08
barrel64	8302.27	7704.02	0.07	6854.51	0.17
fip_cordic_cla	5087.38	4833.83	0.05	4580.29	0.10
fip_cordic_rca	5336.54	4658.21	0.13	4452.96	0.17
fip_risc8	14388.73	14151.70	0.02	13551.30	0.06
mux32_16bit	4066.60	3855.86	0.05	3784.52	0.07
mux64_16bit	8376.90	7762.21	0.07	7556.97	0.10
mux8_128bit	7443.88	9245.08	-0.24	8329.69	-0.12
mux8_64bit	3761.47	4009.54	-0.07	3551.83	0.06
nut_000	2594.72	2628.75	-0.01	2412.52	0.07
nut_001	10541.46	9834.54	0.07	9370.28	0.11
nut_002	2152.39	2179.83	-0.01	2071.17	0.04
nut_003	5662.54	5467.15	0.03	5157.62	0.09
nut_004	1375.29	1350.05	0.02	1202.97	0.13
oc_aes_core_inv	28799.90	28376.16	0.01	26842.77	0.07
oc_aes_core	26545.31	25446.52	0.04	23047.05	0.13
oc_aquarius	59252.11	54515.74	0.08	52769.40	0.11
oc_ata_ocidec1	4470.52	4347.58	0.03	4179.66	0.07
oc_ata_ocidec2	4848.10	4852.47	0.00	4698.82	0.03
oc_ata_ocidec3	9643.58	9568.95	0.01	9228.66	0.04
oc_ata_v	2279.72	2165.56	0.05	2163.37	0.05
oc_ata_vhd_3	9722.63	9592.00	0.01	9254.99	0.05
oc_cfft_1024x12	31034.53	26954.52	0.13	26336.59	0.15
oc_cordic_p2r	29152.18	27218.10	0.07	28047.94	0.04
oc_cordic_r2p	37079.33	34875.41	0.06	34131.17	0.08
oc_correlator	5122.50	4708.70	0.08	4696.63	0.08
oc_dct_slow	3312.55	2956.93	0.11	2827.41	0.15
oc_des_area_opt	9111.25	9169.41	-0.01	8011.40	0.12

Continued on next page

Table 7.8 continued from previous page

Name	Area				
	DC	Baseline	Impr.	Choice 1x	Impr.
oc_des_des3area	13151.70	13004.63	0.01	12014.55	0.09
oc_des_des3perf	265356.31	246740.59	0.07	216520.44	0.18
oc_des_perf_opt	74142.63	75459.08	-0.02	65843.59	0.11
oc_ethernet	27094.12	26379.53	0.03	24946.01	0.08
oc_fcmp	1305.05	1335.78	-0.02	977.96	0.25
oc_fpu	53503.70	52455.48	0.02	48611.45	0.09
oc_gpio	1996.53	1977.87	0.01	1954.82	0.02
oc_hdlc	5535.21	5518.73	0.00	5345.31	0.03
oc_i2c	2555.21	2458.62	0.04	2401.55	0.06
oc_mem_ctrl	40058.54	38701.84	0.03	38334.07	0.04
oc_minirisc	5553.86	5297.03	0.05	5120.31	0.08
oc_miniuart	1594.81	1623.35	-0.02	1508.10	0.05
oc_mips	45187.86	44924.46	0.01	42607.31	0.06
oc_oc8051	28338.87	27514.53	0.03	26179.77	0.08
oc_pavr	41313.11	39909.22	0.03	38493.28	0.07
oc_pci	23850.54	23247.93	0.03	22660.69	0.05
oc_rtc	3117.18	3043.65	0.02	2891.08	0.07
oc_sdram	2634.24	2616.68	0.01	2528.87	0.04
oc_simple_fm_receiver	9401.01	7485.65	0.20	7393.46	0.21
oc_ssram	849.54	849.54	0.00	814.42	0.04
oc_vga_lcd	20761.71	20044.94	0.03	19295.26	0.07
oc_video_dct	123085.73	93931.36	0.24	94914.60	0.23
oc_video_huffman_dec	5120.31	5242.14	-0.02	4766.88	0.07
oc_video_huffman_enc	6007.18	6181.67	-0.03	5317.87	0.11
oc_video_jpeg	169283.53	138585.77	0.18	134959.64	0.20
oc_wb_dma	40129.92	39779.72	0.01	37689.79	0.06
os_blowfish	17845.18	17745.35	0.01	17774.99	0.00
os_sdram16	3448.65	3567.20	-0.03	3339.99	0.03
radar12	77860.90	71953.72	0.08	76520.13	0.02
radar20	203818.91	167486.92	0.18	176606.81	0.13
ts_mike_fsm	138.30	144.88	-0.05	143.79	-0.04
uoft_raytracer	542491.12	401099.09	0.26	402091.44	0.26
xbar_16x16	2577.16	2247.88	0.13	2177.64	0.16
Average			0.04		0.09

Table 7.9: Comparison of area-oriented mapping with a commercial tool using Baseline and Choice 1x (Section 7.2.5). The column labelled "Impr." indicates improvements in delay over DC. For example, for barrel16, Baseline leads to a 13% improvement over DC.

Name	Delay				
	DC	Baseline	Impr.	Choice 1x	Impr.
barrel16	0.70	0.61	0.13	0.60	0.14
barrel16a	0.73	0.59	0.19	0.47	0.36
barrel32	0.86	0.86	0.00	0.82	0.05
barrel64	1.25	0.97	0.22	0.91	0.27
fip_cordic_cla	3.57	2.90	0.19	3.12	0.13
fip_cordic_rca	3.93	3.49	0.11	3.30	0.16
fip_risc8	4.28	3.67	0.14	3.09	0.28
mux32_16bit	0.74	0.66	0.11	0.77	-0.04
mux64_16bit	1.16	0.98	0.16	1.01	0.13
mux8_128bit	1.11	0.55	0.50	0.81	0.27
mux8_64bit	0.91	0.44	0.52	0.47	0.48
nut_000	2.95	2.37	0.20	2.17	0.26
nut_001	4.17	3.15	0.24	3.37	0.19
nut_002	1.02	0.92	0.10	0.93	0.09
nut_003	1.57	1.32	0.16	1.28	0.18
nut_004	0.46	0.43	0.07	0.52	-0.13
oc_aes_core_inv	1.56	1.21	0.22	1.13	0.28
oc_aes_core	1.50	1.28	0.15	1.19	0.21
oc_aquarius	8.89	6.77	0.24	7.15	0.20
oc_ata_ocidec1	1.18	1.15	0.03	0.99	0.16
oc_ata_ocidec2	1.26	1.15	0.09	0.97	0.23
oc_ata_ocidec3	1.35	1.24	0.08	1.07	0.21
oc_ata_v	0.94	1.16	-0.23	0.97	-0.03
oc_ata_vhd_3	1.40	1.24	0.11	1.02	0.27
oc_cfft_1024x12	2.07	1.79	0.14	1.89	0.09
oc_cordic_p2r	2.40	1.76	0.27	1.72	0.28
oc_cordic_r2p	2.74	2.03	0.26	1.96	0.28
oc_correlator	2.16	1.74	0.19	1.92	0.11
oc_dct_slow	1.49	1.28	0.14	1.19	0.20
oc_des_area_opt	1.96	1.63	0.17	1.42	0.28

Continued on next page

Table 7.9 continued from previous page

Name	Delay				
	DC	Baseline	Impr.	Choice 1x	Impr.
oc_des_des3area	3.19	2.74	0.14	2.35	0.26
oc_des_des3perf	1.75	0.99	0.43	0.77	0.56
oc_des_perf_opt	1.08	1.01	0.06	0.69	0.36
oc_ethernet	2.11	2.47	-0.17	1.94	0.08
oc_fcmp	2.02	1.36	0.33	1.29	0.36
oc_fpu	103.67	111.98	-0.08	98.83	0.05
oc_gpio	0.85	0.77	0.09	0.75	0.12
oc_hdlc	0.90	0.88	0.02	0.70	0.22
oc_i2c	1.20	0.92	0.23	1.07	0.11
oc_mem_ctrl	2.56	1.86	0.27	2.40	0.06
oc_minirisc	2.10	1.52	0.28	1.43	0.32
oc_miniuart	0.81	0.51	0.37	0.50	0.38
oc_mips	7.08	6.19	0.13	5.86	0.17
oc_oc8051	4.52	3.68	0.19	3.44	0.24
oc_pavr	3.83	3.80	0.01	3.75	0.02
oc_pci	2.99	2.95	0.01	2.54	0.15
oc_rtc	2.22	1.83	0.18	1.75	0.21
oc_sdram	0.95	0.60	0.37	0.82	0.14
oc_simple_fm_receiver	2.68	2.26	0.16	2.55	0.05
oc_ssram	0.07	0.07	0.00	0.13	-0.86
oc_vga_lcd	1.98	1.70	0.14	2.27	-0.15
oc_video_dct	3.15	2.82	0.10	2.98	0.05
oc_video_huffman_dec	1.43	0.93	0.35	0.96	0.33
oc_video_huffman_enc	1.05	1.04	0.01	0.93	0.11
oc_video_jpeg	3.10	2.74	0.12	2.96	0.05
oc_wb_dma	3.54	2.64	0.25	1.90	0.46
os_blowfish	2.96	3.00	-0.01	3.25	-0.10
os_sdram16	1.31	0.87	0.34	1.11	0.15
radar12	4.59	4.02	0.12	3.94	0.14
radar20	4.73	4.02	0.15	3.94	0.17
ts_mike_fsm	0.34	0.25	0.26	0.23	0.32
uoft_raytracer	12.52	8.81	0.30	9.67	0.23
xbar_16x16	0.39	0.36	0.08	0.38	0.03
Average			0.16		0.16

Chapter 8

Conclusion

We believe that the techniques developed in this thesis lead to significant improvements over existing approaches to technology mapping. In this chapter we review the main benefits of these techniques, and also look at their limitations in order to identify directions for future research.

8.1 Local Structural Bias

Simplified Boolean Matching. For standard cell mapping, the Boolean matching technique based on enumerating cuts and matching cuts to library cells using configurations is attractive for both for its simplicity of implementation and for its fast run-time. This makes Boolean matching very practical and – in our opinion – a viable replacement for structural matching in most cases.

Future Directions. In fact, structural matching is useful only for gates with many inputs, and for those gates it is useful only because we do *not* expect it to be "complete." Instead, we just look for a particular decomposition of the library gate in the subject graph. An interesting problem would be to allow this sort of relaxation of

completeness in the context of Boolean matching. An obvious relaxation presents itself: do not enumerate all configurations of such a library cell. However, enumerating all cuts with as many inputs as the library cell is likely to be impractical. Here, we may relax completeness by enumerating only some cuts (instead of all). The interested reader is referred to the work on factor cuts for an exploration of this idea in the context of macrocell matching [CMB06a].

Handling Inverters. The explicit handling of single-input gates, especially inverters, using the notions of singular and regular matches and electrical edges makes the subsequent match selection problem easier to solve. This way of handling single input gates is especially useful for area-oriented selection using exact area.

Supergates. Local structural bias can be significantly reduced by using supergates which allow many more matches to be detected. Even the simple, bottom-up procedure that we use to generate supergates leads to improved mapping quality.

The combination of supergates and cut-based Boolean matching may also be seen as a technique for technology independent logic optimization. For example, by constructing supergates using only AND gates and inverters it is possible to resynthesize AIGs for better area and obtain results comparable to those obtained by traditional synthesis scripts in a fraction of the run-time [MCB06a].

Future Directions. Our supergate generation 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 good supergates for frequently occurring cut functions. To this end, some techniques proposed in the literature for synthesizing multi-level circuits for Boolean functions of few variables could be explored [RK62; Hel63; Dav69; MKLC89]. (A more comprehensive survey of such techniques appears in [HS02, Chapter 2].)

8.2 Global Structural Bias

Detecting Choices. AIGs with choice are an useful extension of the AIG data structure for encoding multiple networks into one subject graph for technology mapping. AIGs with choice can be created very efficiently from different networks obtained during logic synthesis and mapping using SAT- and simulation-based techniques from combinational equivalence checking.

Future Directions. An interesting engineering problem in this connection is to altogether *avoid* using equivalence-checking to detect choices: Instead of the current system of detecting choice after synthesis, the synthesis algorithms themselves could be modified to detect and preserve choices.

Creating Choices for Lossless Synthesis. The main advantage of lossless synthesis over exhaustively adding local choices through algebraic re-writing is that the choices are more global. Therefore, the results are better in spite of having fewer choices. Furthermore, the run-time is reasonable since fewer inferior choices are examined during selection. That said, the global choices obtained by lossless synthesis appear to be qualitatively different from the local choices obtained by algebraic re-writing, and it is possible to use both together to obtain the best results although at the expense of longer run-time.

Future Directions. In this work we have only used a couple of techniques for obtaining different networks to combine for lossless synthesis. More work is needed to identify the best set of technology independent scripts and algorithms for lossless synthesis.

8.3 Match Selection

Area flow and exact area are effective heuristics for area-oriented mapping for both standard cells and FPGAs. Although in this thesis our focus was on area, we expect that these heuristics will also be effective in reducing power when used with suitable cost functions.

A key advantage of using exact area is its effectiveness in the presence of choices. Since choices add additional fanouts to the subject graph, other heuristics that are based on estimating fanouts in the mapped network using fanout information in the subject graph may fail. However, since exact area is greedy and incremental, and does not try to estimate fanouts, it is effective even in the presence of choices.

A good area-oriented mapper can also be seen as a more general purpose tool. For instance, by suitably defining the notion of area, area-oriented mapping using our heuristics can be used to generate a good set of CNF clauses from a circuit for faster equivalence checking [EMS07].

Future Directions. The main limitation of the current selection algorithms is that they are not layout-aware. Layout-awareness is particularly important in the context of delay-oriented mapping.

In the case of FPGAs, since most of the delay is in the routing network, it is important to take layout and routing into account in order to accurately characterize the performance of the mapped network. However, usefully integrating placement information into mapping appears to be very challenging.

In the case of standard cell mapping, we have used the constant delay model in this work. This model is useful for mapping in a flow where sizing, fanout-tree construction and buffering are done after mapping in conjunction with placement. However, the proposed methods can be used with a load-based delay model. (In fact, our implementation supports this.) However, there are a number of challenges

with incorporating a load-based mapping in the context of mapping directly to DAGs without partitioning them into trees:

- During the selection, it is difficult to estimate accurate arrival times at the inputs of the match since the total load at the input (due to other fanouts) is not known. It seems difficult to address this without an iterative procedure, and with an iterative procedure there may be a problem with convergence.

- Better results may be obtained by integrating fanout-tree construction with mapping as is done in the tree case [Tou90, Chapter 4], but this appears challenging without *a priori* splitting the DAG into trees.

- Since many more matches are explored (due to reduced structural bias) in our approach, match selection with a load-based library is slow. In this connection, exploring techniques for library abstraction (where a number of gates are collected into a group based on delay similarity) followed by subsequent refinement (to pick the best gate) might be interesting.

Finally, there remains the problem of incorporating wire delays arising from layout into the mapping procedure. (Since these delays arise from the capacitive loading due to long wires, it may be important to address the problems listed above first.) Furthermore, adapting previously proposed methods for layout-aware standard cell mapping [PB91; KS01; GKSV01; SSS06] to use AIGs with choice also appears to be an interesting problem. These methods rely on obtaining a companion placement using the subject graph, and naïvely placing an AIG with choice leads to a distorted placement.

Bibliography

[Act07] Actel. *ProASIC3 Flash Family FPGAs Datasheet v2.1.* Actel Corporation, Mountain View, CA, USA, 2007.

[Alt06] Altera. *Stratix Device Handbook Vol. I.* Altera Corporation, San Jose, CA, USA, 2006.

[AP05] Afshin Abdollahi and Massoud Pedram. A new canonical form for fast Boolean matching in logic synthesis and verification. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 379–384, New York, NY, USA, 2005. ACM Press.

[BB04] P. Bjesse and A. Boralv. DAG-aware circuit compression for formal verification. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 42–49, Washington, DC, USA, 2004. IEEE Computer Society.

[BL92] Jerry R. Burch and David E. Long. Efficient boolean function matching. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 408–411, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[Bra87] R. K. Brayton. Factoring logic functions. *IBM J. Res. Dev.*, 31(2):187–198, 1987.

[Bra93] Daniel Brand. Verification of large synthesized designs. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 534–537, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[CC01] Gang Chen and Jason Cong. Simultaneous logic decomposition with technology mapping in FPGA designs. In *FPGA '01: Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 48–55, New York, NY, USA, 2001. ACM Press.

[CC04] D. Chen and J. Cong. Daomap: a depth-optimal area optimization mapping algorithm for fpga designs. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 752–759, Washington, DC, USA, 2004. IEEE Computer Society.

[CCP06] Deming Chen, Jason Cong, and Peichan Pan. FPGA design automation: A survey. *Foundations and Trends in Electronic Design Automation*, 1(2):195–330, 2006.

[CD92] Jason Cong and Yuzheng Ding. An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. In *ICCAD '92: Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design*, pages 48–53, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[CD93] Jason Cong and Yuzheng Ding. On area/depth trade-off in LUT-based FPGA technology mapping. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 213–218, New York, NY, USA, 1993. ACM Press.

[CD96] Jason Cong and Yuzheng Ding. Combinational logic synthesis for lut based field programmable gate arrays. *ACM Trans. Des. Autom. Electron. Syst.*, 1(2):145–204, 1996.

[CH95a] Amit Chowdhary and John P. Hayes. Technology mapping for field-programmable gate arrays using integer programming. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 346–352, Washington, DC, USA, 1995. IEEE Computer Society.

[CH95b] Jason Cong and Yean-Yow Hwang. Simultaneous depth and area minimization in LUT-based FPGA mapping. In *FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*, pages 68–74, New York, NY, USA, 1995. ACM Press.

[CK06] Donald Chai and Andreas Kuehlmann. Building a better boolean matcher and symmetry detector. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 1079–1084, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[CMB+05] Satrajit Chatterjee, Alan Mishchenko, Robert Brayton, Xinning Wang, and Timothy Kam. Reducing structural bias in technology mapping. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 519–526, Washington, DC, USA, 2005. IEEE Computer Society.

[CMB06a] Satrajit Chatterjee, Alan Mishchenko, and Robert Brayton. Factor cuts. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 143–150, New York, NY, USA, 2006. ACM Press.

[CMB+06b] Satrajit Chatterjee, Alan Mishchenko, Robert Brayton, Xinning Wang, and Timothy Kam. Reducing structural bias in technology mapping. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(12):2894–2903, 2006.

[CP92] K. Chaudhary and M. Pedram. A near optimal algorithm for technology mapping minimizing area under delay constraints. In *DAC '92: Proceedings of the 29th ACM/IEEE conference on Design automation*, pages 492–498, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[CPD96] Jason Cong, John Peck, and Yuzheng Ding. RASP: A general logic synthesis system for SRAM-based FPGAs. In *FPGA*, pages 137–143, 1996.

[CS03] Jovanka Ciric and Carl Sechen. Efficient canonical form for Boolean matching of complex functions in large libraries. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 22(5):535–544, May 2003.

[CWD99] Jason Cong, Chang Wu, and Yuzheng Ding. Cut ranking and pruning: enabling a general and efficient FPGA mapping solution. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 29–35, New York, NY, USA, 1999. ACM Press.

[Dav69] Edward Davidson. An algorithm for NAND decomposition under network constraints. *IEEE Trans. on Computers*, EC-18(12):1098–1109, 1969.

[DS04] Debatosh Debnath and Tsutomu Sasao. Efficient computation of canonical form for Boolean matching in large libraries. In *ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation*, pages 591–596, Piscataway, NJ, USA, 2004. IEEE Press.

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[EMS07] Niklas Eén, Alan Mishchenko, and Niklas Sörensson. Applying logic synthesis for speeding up SAT. In João Marques-Silva and Karem A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2007.

[ES03] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[FRC90] Robert J. Francis, Jonathan Rose, and Kevin Chung. Chortle: A technology mapping program for lookup table-based field programmable gate arrays. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 613–619, New York, NY, USA, 1990. ACM Press.

[FRV91] Robert Francis, Jonathan Rose, and Zvonko Vranesic. Chortle-crf: Fast technology mapping for lookup table-based FPGAs. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 227–233, New York, NY, USA, 1991. ACM Press.

[FS94] Amir H. Farrahi and Majid Sarrafzadeh. Complexity of the lookup-table minimization problem for FPGA technology mapping. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(11):1319–1332, 1994.

[GKSV01] Wilsin Gosti, Sunil P. Khatri, and Alberto L. Sangiovanni-Vincentelli. Addressing the timing closure problem by integrating logic optimization and placement. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 224–231, Piscataway, NJ, USA, 2001. IEEE Press.

[GLH+95] Joel Grodstein, Eric Lehman, Heather Harkness, Bill Grundmann, and Yosinatori Watanabe. A delay model for logic synthesis of continuously-sized networks. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 458–462, Washington, DC, USA, 1995. IEEE Computer Society.

[Gol59] Solomon Golomb. On the classification of Boolean functions. *IRE Transactions on Information Theory*, IT-5:176–186, 1959.

[GPB01] E. Goldberg, M. Prasad, and R. Brayton. Using SAT for combinational equivalence checking. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 114–121, Piscataway, NJ, USA, 2001. IEEE Press.

[Hel63] Leo Hellerman. A catalog of three-variable Or-Invert and And-Invert logical circuits. *IEEE Trans. on Electronic Computers*, EC-12(3):198–223, 1963.

[HK98] Uwe Hinsberger and Reiner Kolla. Boolean matching for large libraries. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 206–211, New York, NY, USA, 1998. ACM Press.

[HS02] Soha Hassoun and Tsutomu Sasao (editors), editors. *Logic Synthesis and Verification*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[HWKMS03] Bo Hu, Yosinori Watanabe, Alex Kondratyev, and Malgorzata Marek-Sadowska. Gain-based technology mapping for discrete-size cell libraries. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 574–579, New York, NY, USA, 2003. ACM Press.

[JWBO00] Dirk-Jan Jongeneel, Yosinori Watanbe, Robert K. Brayton, and Ralph Otten. Area and search space control for technology mapping. In *DAC '00: Proceedings of the 37th conference on Design automation*, pages 86–91, New York, NY, USA, 2000. ACM Press.

[KBS98] Yuji Kukimoto, Robert K. Brayton, and Prashant Sawkar. Delay-optimal technology mapping by DAG covering. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 348–351, New York, NY, USA, 1998. ACM Press.

[KDNG92] D. S. Kung, R. F. Damiano, T. A. Nix, and D. J. Geiger. BDDMAP: a technology mapper based on a new covering algorithm. In *DAC '92:*

Proceedings of the 29th ACM/IEEE conference on Design automation, pages 484–487, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Keu87] Kurt Keutzer. DAGON: technology binding and local optimization by DAG matching. In *DAC '87: Proceedings of the 24th ACM/IEEE conference on Design automation*, pages 341–347, New York, NY, USA, 1987. ACM Press.

[KK97] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 263–268, New York, NY, USA, 1997. ACM Press.

[KPKG02] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 21(12):1377–1394, 2002.

[KS00] Victor N. Kravets and Karem A. Sakallah. Constructive library-aware synthesis using symmetries. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 208–215, New York, NY, USA, 2000. ACM Press.

[KS01] Thomas Kutzschebauch and Leon Stok. Congestion aware layout driven logic synthesis. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 216–223, Piscataway, NJ, USA, 2001. IEEE Press.

[KS04] S. K. Karandikar and S. S. Sapatnekar. Logical effort based technology mapping. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 419–422, Washington, DC, USA, 2004. IEEE Computer Society.

[Kue04] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 50–57, Washington, DC, USA, 2004. IEEE Computer Society.

[Kun93] Wolfgang Kunz. HANNIBAL: an efficient tool for logic verification based on recursive learning. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 538–543, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[LAB+05] David Lewis, Elias Ahmed, Gregg Baeckler, Vaughn Betz, Mark Bourgeault, David Cashman, David Galloway, Mike Hutton, Chris Lane, Andy Lee, Paul Leventis, Sandy Marquardt, Cameron McClintock, Ketan Padalia, Bruce Pedersen, Giles Powell, Boris Ratchev, Srinivas Reddy, Jay Schleicher, Kevin Stevens, Richard Yuan, Richard Cliff, and Jonathan Rose. The Stratix II logic and routing architecture. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 14–20, New York, NY, USA, 2005. ACM Press.

[LWCH03] Feng Lu, Li-C. Wang, Kwang-Ting Cheng, and Ric C-Y Huang. A circuit SAT solver with signal correlation guided learning. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10892, Washington, DC, USA, 2003. IEEE Computer Society.

[LWGH95] Eric Lehman, Yosinori Watanabe, Joel Grodstein, and Heather Harkness. Logic decomposition during technology mapping. In *ICCAD '95: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 264–271, Washington, DC, USA, 1995. IEEE Computer Society.

[LZB07] Andrew Ling, Jianwen Zhu, and Stephen Brown. BddCut: Towards scalable cut enumeration. In *Proceedings of the 2007 Conference on Asia South Pacific Design Automation: ASP-DAC 2007, Yokohama, Japan, January 23-26, 2007*, 2007.

[MBV04] Valavan Manohararajah, Stephen Dean Brown, and Zvonko G. Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. In *Proceedings of the International Workshop in Logic Synthesis*, pages 14–21, 2004.

[MBV06] Valavan Manohararajah, Stephen Dean Brown, and Zvonko G. Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(11):2331–2340, 2006.

[MCB06a] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-aware AIG rewriting a fresh look at combinational logic synthesis. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 532–535, New York, NY, USA, 2006. ACM Press.

[MCB06b] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Improvements to technology mapping for LUT-based FPGAs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 41–49, New York, NY, USA, 2006. ACM Press.

[MCB07] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Improvements to technology mapping for LUT-based FPGAs. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(2):240–253, 2007.

[MCBE06] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to combinational equivalence checking. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 836–843, New York, NY, USA, 2006. ACM Press.

[MCCB07] Alan Mishchenko, Sungmin Cho, Satrajit Chatterjee, and Robert Brayton. Priority cuts. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design (to appear)*, 2007.

[MKLC89] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method-design of logic networks based on permissible functions. *IEEE Trans. Comput.*, 38(10):1404–1424, 1989.

[MM90] Frédéric Mailhot and Giovanni De Micheli. Technology mapping using boolean matching and don't care sets. In *EURO-DAC '90: Proceedings of the conference on European design automation*, pages 212–216, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[Mur99] Rajeev Murgai. Performance optimization under rise and fall parameters. In *ICCAD '99: Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design*, pages 185–190, Piscataway, NJ, USA, 1999. IEEE Press.

[MWK03] Alan Mishchenko, Xinning Wang, and Timothy Kam. A new enhanced constructive decomposition and mapping algorithm. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 143–148, New York, NY, USA, 2003. ACM Press.

[PB91] Massoud Pedram and Narasimha Bhat. Layout driven technology mapping. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 99–105, New York, NY, USA, 1991. ACM Press.

[PL98] Peichen Pan and Chih-Chang Lin. A new retiming-based technology mapping algorithm for LUT-based FPGAs. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 35–42, New York, NY, USA, 1998. ACM Press.

[RK62] J. Paul Roth and Richard Karp. Minimization over boolean graphs. *IBM J. Res. Dev.*, 6(2):227–238, 1962.

[Rud89] Richard Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, EECS Department, University of California, Berkeley, Berkeley CA 94720, May 1989.

[SSBSV92] Hamid Savoj, Màrio J. Silva, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Boolean matching in logic synthesis. In *EURO-DAC '92: Proceedings of the conference on European design automation*, pages 168–174, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[SSL+92] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Sis: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.

[SSS06] Rupesh S. Shelar, Prashant Saxena, and Sachin S. Sapatnekar. Technology mapping algorithm targeting routing congestion under delay constraints. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(4):625–636, 2006.

[Tou90] Hervé Touati. *Performance-Oriented Technology Mapping*. PhD thesis, EECS Department, University of California, Berkeley, Berkeley CA 94720, Dec 1990.

[Xil07a] Xilinx. *Virtex-4 Family Overview v2.0*. Xilinx Corporation, San Jose, CA, USA, 2007.

[Xil07b] Xilinx. *Virtex-5 User Guide*. Xilinx Corporation, San Jose, CA, USA, 2007.