University of California
Santa Cruz

Rectangle Replacement and Variable Ordering: Two Techniques for Logic Minimization Using If-Then-Else DAGs

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

Doctor of Philosophy

in

Computer Engineering

by

Søren Søe

June 1994

The dissertation of Søren Søe is approved:

Kevin Karplus

Martine Schlag

Glen Langdon

Dean of Graduate Studies and Research

Copyright © by

Søren Søe

1994

Rectangle Replacement and Variable Ordering: Two Techniques for Logic Minimization Using If-Then-Else DAGs

Søren Søe

ABSTRACT

This thesis explores logic minimization techniques for Boolean functions represented as if-then-else DAGs. In particular the thesis presents algorithms for two areas of multi-level logic minimization: Rectangle Covering and Variable Ordering.

Rectangle covering is the process of factoring and extracting common sub-expressions in Boolean functions. Boolean functions are represented as Boolean matrices, and rectangles of these matrices represent either a factor of a function or a sub-expression that can be shared among several functions. An efficient heuristic algorithm *two-column rectangle replacement* for finding rectangles of a Boolean matrix is presented. The heuristic is particular well suited for optimizing circuits for area, while controlling the delay. A slight variation of the heuristic optimizes with respect to delay.

Variable ordering is a problem specific to canonical if-then-else DAGs and ordered binary decision diagrams. This thesis presents an improved depth-first ordering heuristic based on reconvergent fanout. This heuristic is fast and produces variable orders resulting in smaller canonical forms than previously published traversal-based ordering heuristics, and is suitable when using canonical form for verification purposes. The thesis also presents a new ordering heuristic called *SplitOrder*, which is especially well suited for finding good variable orders for ordered binary decision diagrams (OBDDs). SplitOrder constructs the variable order by building an OBDD top-down one level at a time, choosing the next variable such that the corresponding level in the OBDD has few nodes and represents expressions that are as small as possible. SplitOrder is compared to several depth-first ordering heuristics and show that when converting to OBDDs SplitOrder averages 25% fewer nodes than taking the best of 8 depth-first techniques (37% better than the best single depth-first technique). Finally it

is demonstrated that applying SplitOrder to a set of expressions already represented as an OBDD often results in significantly better variable orders, thus making it beneficial to iterate SplitOrder.

All of the algorithms presented in this thesis have been implemented, tested, and installed as part of the logic minimizer ITEM.

Acknowledgements

I first of all want to thank my research advisor Kevin Karplus for his support and encouragement through the past 5 years. Kevin has been a source of inspiration ever since I joined the Ph.D. program at UC Santa Cruz—without his insight, ideas, and valuable suggestions this thesis would have been impossible. I have been very fortunate to have Kevin as a research advisor and I truly will miss his sharp comments on the work I do in the future. Thank you Kevin.

I also wish to thank Martine Schlag for serving on my thesis committee, reading drafts of this thesis, providing useful comments, and catching errors. My thanks also goes to professor Glen Langdon for taking time to serve on my thesis committee and reading my thesis.

Several fellow graduate students have been involved in the ITEM project, and here I especially want to thank Merhdad Parsa for many inspiring discussions during the development of ITEM.

I have several funds to thanks. Most of all I must thank Rotary International for selecting me as a recipient of a one year scholarship, without this scholarship I most likely would never have come to study in the US. Through Rotary I also met friends who introduced me to the American way of living. After Rotary I was funded in part by the Danish Research Academy for three full years. The Danish Research Academy is a government institution, which graciously supports Danish Ph.D. students in foreign countries by covering parts of their tuition and living expenses. They even make sure you don't loose touch with your home country by providing a round-trip ticket for each of the three years you are supported. Yes, we do pay a lot of taxes in Denmark, but it's worth it, and one day I will pay my share. My thanks also goes to the following private Danish funds who have supported me and my wife during my studies at UC Santa Cruz: *Augustinus Fondon*, *Thomas B. Thriges Fond*, *Otto Mønsteds Fond*, *Fisker & Nielsens Fond*, *Laurits Andersens Fond*, and *Henry og Mary Skovs Fond*.

My friend Søren Christensen, who works at Sun Microsystems, also has to be thanked. Søren supported me with valuable computer resources, with which I have spend so many hours. His hardware support made it possible for me to work from home and spend more time with my wife.

Finally, but not least, I would like to thank my wife Jeanette for her patience and support during the time it has taken to get this thesis done. She showed me that there was a life outside school, but at the same time she encouraged me to stay long hours in front of my computer making sure I got some work done.

Contents

Abstract iii

Acknowledgements v

1. Introduction 1

1.1 Silicon compilation . . . 2

1.2 Logic synthesis . . . 3

1.3 Logic minimization . . . 5

1.4 Organization of the thesis . . . 8

2. Representing Boolean expressions 10

2.1 Sum-of-products form . . . 12

2.2 Factored form . . . 13

2.3 Binary decision diagrams . . . 14

2.4 If-Then-Else DAGs . . . 15

2.5 Two input NORs . . . 17

2.6 Representing Boolean networks . . . 17

3. Logic minimization techniques 21

3.1 Factoring . . . 21

3.2 Sub-expression extraction . . . 23

3.3 Canonical form, verification and testability . . . 25

4. ITEM 30

4.1 Reading and writing different formats . . . 31

4.2 Optimizations in ITEM . . . 31

4.2.1 LocalFactor . . . 32

4.2.2 Block covering 33
4.2.3 Conversion to canonical form 33
4.3 Mapping 34

5. Rectangle Replacement **35**
5.1 Introduction 35
5.2 Blocks and rectangles 37
5.2.1 Boolean matrices and blocks of Boolean matrices 38
5.2.2 Replacing rectangles in a Boolean matrix 40
5.2.3 The rectangle replacement problem 42
5.3 Two-column rectangle replacement and if-then-else DAGs 42
5.3.1 Creating Boolean matrices from if-then-else DAGs 43
5.3.2 Rectangle replacement algorithm 44
5.3.3 Selecting rectangles 45
5.3.4 Tree balancing 47
5.3.5 Expanding IF-expressions 48
5.4 Results 49
5.5 Conclusions and future work 53

6. Variable ordering **55**
6.1 Introduction 55
6.2 Background 56
6.3 Depth-first ordering heuristics 61
6.3.1 Children of a node 62
6.3.2 Incremental ordering heuristics 62
6.3.3 Reconvergent ordering heuristics 66
6.3.4 Results 69
6.4 SplitOrder heuristic 74

6.4.1 Choosing the cost function 76

6.4.2 Computing $e|_v$ and $e|_{v'}$ 78

6.4.3 Complexity 79

6.4.4 Inefficiencies in computing given that 82

6.4.5 Results 83

6.5 The effect of variable order on other transformations 98

6.6 Conclusion 100

7. Conclusions and future research **102**

7.1 Two-column rectangle replacement 102

7.2 Variable ordering 103

7.3 Future work 104

References **106**

List of Figures

1.1 Different abstraction levels for silicon compilation. 3

2.1 Binary decision diagram for $abc + \neg ad + \neg bd$ 15

2.2 If-then-else DAG for $abc + \neg ad + \neg bd$. 16

2.3 Boolean network with three primary outputs and 4 primary inputs. The expressions are represented are $f_1 = a + b$, $f_2 = a + b + c$, and $f_3 = a + b + c + d$. 18

2.4 Multiply-rooted if-then-else DAG. The three expressions represented are $f_1 = a + b$, $f_2 = a + b + c$ and $f_3 = a + b + c + d$. 19

3.1 Binary decision diagram for $abc + \neg ad + \neg bd$, with respect to the ordering (c, d, a, b). 27

3.2 Canonical if-then-else DAG for $abc + \neg ad + \neg bd$, with respect to the ordering (c, d, a, b). 29

4.1 General outline of ITEM. 30

4.2 One view of the optimization phase in ITEM. 32

5.1 Boolean matrix for the functions $f_1 = a + b + de$, $f_2 = b + de + c(f + g)$, and $f_3 = cf + cg$. 39

5.2 Boolean matrix for the functions f_1, f_2, and f_3, after the new column $b + de$ has replaced the columns b and de. 42

5.3 The FindInputs routine. 44

5.4 Multiply-rooted if-then-else DAG representing the three expressions $f_1 = a+b$, $f_2 = a + b + c$, and $f_3 = a + b + c + d$. 46

5.5 Boolean matrix for $f_1 = a + b$, $f_2 = a + b + c$, and $f_3 = a + b + c + d$, showing overlapping rectangles. 46

5.6 Replacing the highest valued two-column rectangle in the matrix of Figure 5.5. 47

5.7 Replacing the highest valued two-column rectangle in the matrix of Figure 5.6. 47

5.8 Replacing the only remaining rectangle with positive value in the matrix of Figure 5.7. 47

5.9 Comparison of the count and height of circuits minimized by LocalFactoring and by LocalFactoring plus two-column rectangle replacement optimizing for area. 51

5.10 Comparison of the count and height of circuits minimized by LocalFactoring and by LocalFactoring plus two-column rectangle replacement optimizing for delay. 51

5.11 Comparison of the count and height of circuits minimized by DMIG and by LocalFactoring plus two-column rectangle replacement for delay. 52

5.12 Comparison of the count and height of circuits minimized by two-column rectangle replacement with and without expanding if-expressions. 53

6.1 The ratio of the sizes of canonical if-then-else DAG when using two random orders. The x-axis is the size of when using the first random order. 57

6.2 General outline of the optimization loop in ITEM. 60

6.3 General depth-first ordering algorithm. 61

6.4 Merging of the orders for the three subDAGs of an if-then-else node in reconvergent ordering heuristics. 67

6.5 Reconvergent merging of the orders $b < c < a$ and $d < a < b$. 68

6.6 Split order algorithm. 75

6.7 Computing $e|_v$ and $e|_{v'}$ in one traversal of e. 81

6.8 The result of computing $f|_a$ in an if-then-else DAG representing the expression $f = ab' + (c + a')(b + c')$. 85

6.9 The size of OBDD when using SplitOrder divided by the size when using the best of the depth-first ordering heuristics. 86

6.10 The size of canonical if-then-else DAG when using SplitOrder divided by the size when using the best of the depth-first ordering heuristics. 88

6.11 The size of OBDDs when using SplitOrder on unoptimized examples divided by the size when using the SIS optimized examples. 91

6.12 The size of canonical if-then-else DAGs when using SplitOrder on unoptimized examples divided by the size when using the SIS optimized examples. 92

6.13 The number of nodes (`size`) in canonical if-then-else DAGs divided by the the number of nodes (`siscount`) in OBDDs. 92

6.14 Optimization script used with ITEM to show the effect of variable order on other optimizations by LocalFactor. 100

List of Tables

5.1 Operators for determining the meanings of blocks in Boolean matrices of different types. 40

5.2 The result of two-column rectangle replacement applied to a network optimized by LocalFactor. 50

6.1 Results when converting to canonical if-then-else DAGs using a depth-first variable order. 71

6.2 Results when converting to OBDDs using a depth-first variable order. 72

6.3 Results when converting large examples to canonical if-then-else DAGs using a depth-first variable order. 73

6.4 Results when converting large examples to OBDDs using a depth-first variable order. 73

6.5 Comparing depth-first ordering heuristics. 74

6.6 Same as Table 6.5 except that we here compare OBDDs. 74

6.7 Comparing the cardinality of *exprSet* computed by GivenThat with the minumum achievable. 84

6.8 The size and height after applying the various ordering heuristics to an initial if-then-else DAG. 87

6.9 Comparing the result of applying SplitOrder to unoptimized examples and examples that have been optimized with SIS using the standard script. . . . 90

6.10 The result of normalizing each heuristic to the best of all heuristics. 94

6.11 Result of SplitOrder on examples from the ISCAS benchmark set. 95

6.12 Result of iterating SplitOrder. 97

6.13 Results from iterating the script in Figure 6.14. 99

1. Introduction

Logic synthesis and in particular logic minimization becomes increasingly more important as Very Large Scale Integration (VLSI) continues to offer more and more complexity in a single integrated circuit. As the complexity grows so does the number of different design styles. Application specific intergrated circuits (ASICs) designed from scratch are still very popular, but recently many varieties of programmable logic devices have gained increasing popularity. The field-programmable gate array (FPGA) market is today the single fastest growing area of the logic market, and everyday new and more complex FPGAs are invented.

The challenge is to develop a synthesis system that can handle all aspects of logic devices efficiently. Conventional logic synthesis tools are based on a two-level logic representation, which has proven to be the right representation style when the targeted logic device also represents logic in a two-level form. When logic is represented in multiple levels and by devices that in no way resembles two-level logic components, it is not clear that sum-of-products form is the best representation.

In this thesis we develop logic minimization algorithms using the if-then-else DAG data structure to represent Boolean logic. We demonstrate that Boolean expressions can be represented efficiently using if-then-else DAGs and that the if-then-else DAG data structure is a flexible representation style when applying higher-level logic minimizations techniques to minimize Boolean expressions. The thesis addresses two major areas in logic minimization: *extraction of common sub-expressions* and *conversion to canonical form under different variable orders*. Techniques are demonstrated using the logic synthesis tool ITEM, where ITEM stands for *If-Then-Else Minimizer*.

Although this thesis concentrates on higher-level technology-independent optimizations, published papers [Kar91d, Kar91a, Kar93] show that the if-then-else DAG representation style indeed is efficient when targeting various FPGA style logic components.

In order to put the work presented in this thesis into perspective we describe the role of logic minimization in a computer-aided design environment for production of very large

scale integrated circuits (VLSI). Such an environment, in which logic minimization is just one small part, is also known as silicon compilation.

1.1 Silicon compilation

Silicon compilation is the translation from behavior to silicon. Depending on the level of abstraction a behavioral description can range from a high-level algorithmic description to a low-level description in terms of Boolean equations. Dave Johannsen [Joh79] first used the term silicon compilation in 1979 for an automatic synthesis system that assembled parameterized pieces of layout. Since then the term has been used in a much broader sense to define the translation process from a higher-level description into layout [GDP86]. The main purpose of silicon compilation can be summarized in three points:

- To broaden the scope of designers who can construct ASICs. Using a silicon compiler, a designer is working at a higher level of abstraction and need not have any detailed knowledge about IC design, thus more designers can construct ICs.
- To improve design quality. Ideally, any component instantiated by a silicon compiler is free of errors and satisfies design rules. Errors can only propagate from a higher level, whereas manual design leaves many possibilities for introducing errors.
- To increase design productivity. Design productivity is increased as a result of shorter design time due to the higher level of abstraction.

Of course there are disadvantages involved in silicon compilation. A silicon compiler designed to cover all the aspects of a design and for a wide variety of applications, will have to make compromises that affect the quality of the final design.

In silicon compilation the term *synthesis* denotes the process of converting a functional representation of a circuit into a structural representation. Silicon compilation is separated into several abstraction levels as illustrated in Figure 1.1. The figure illustrates that logic minimization is a part of logic synthesis.

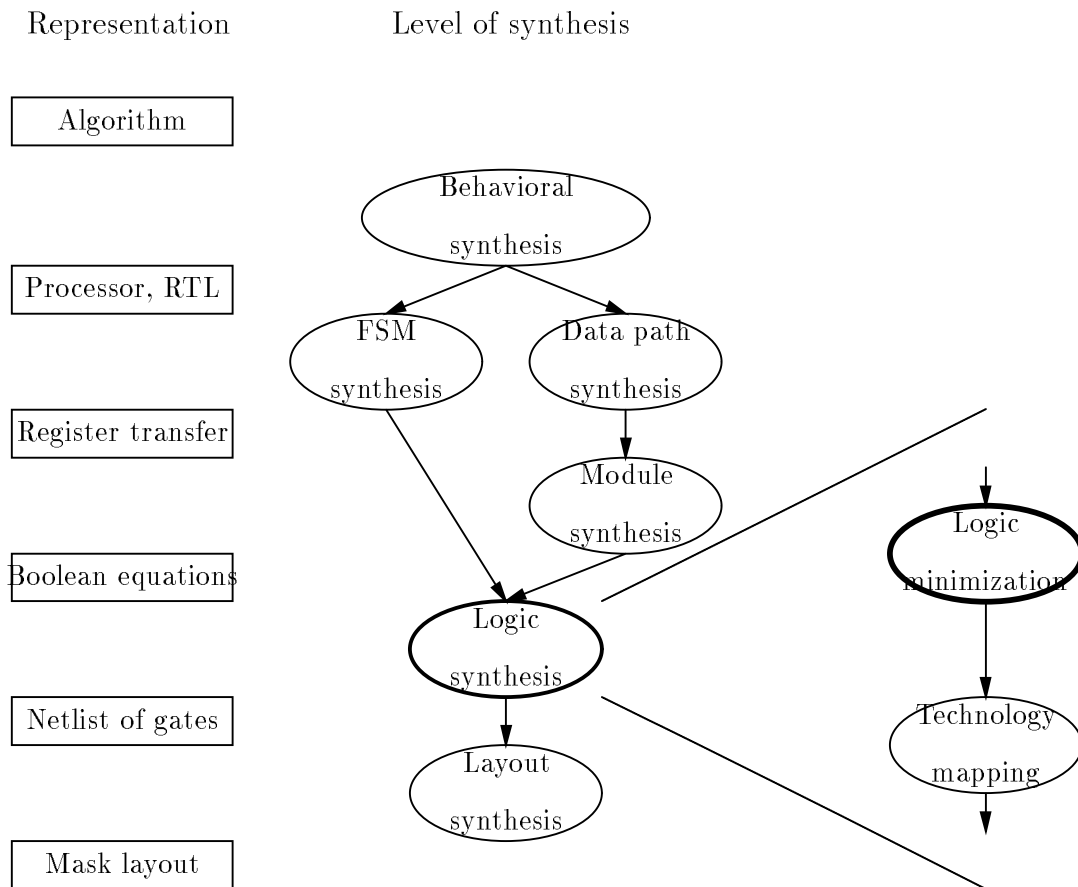

Figure 1.1: Different abstraction levels for silicon compilation. The behavioral description of a design is synthesized into control and data path structures during behavioral synthesis. In finite-state-machine synthesis the control structures are state encoded. Data path synthesis partitions the data structures into states and assigns registers to variables that are used in multiple states. The operations performed in each state are synthesized into functional units, which during module synthesis are converted into building blocks available in the target technology. In logic synthesis the combinational blocks of logic are synthesized into a netlist of gates that implements the desired functions, meeting area and timing constraints. Finally, the netlist of gates is synthesized into a mask layout ready for production.

1.2 Logic synthesis

In *logic synthesis* combinational blocks of logic are synthesized. If the register transfer description contains any storage constructs, they will be set aside during and reconnected after the logic synthesis.

The task of logic synthesis is to convert a description of a set of Boolean functions into

a netlist of gates that implements the functions, meeting area and timing constraints or testing requirements. Logic synthesis is divided into two important areas:

Logic minimization and optimization, which involves changing the Boolean functions such that circuits built from them are better. The key parts of logic minimization are *factoring* to reduce complexity and *sharing common subexpressions* to reduce redundant circuitry.

Technology mapping, which takes the result of logic minimization, and finds a realization of the minimized equations in a particular circuit technology producing a netlist of gates.

Logic synthesis can be compared to language compilers; the first part, logic minimization, consists of technology-independent optimizations, which corresponds to language-independent optimizations and technology mapping corresponds to code generation and peephole optimizations. K. Keutzer [Keu87] related the problem of technology mapping to that of code generation in language compilers. From a set of decomposed functions (the result of logic minimization) a circuit graph (known as the subject graph) in a simple base technology is constructed. The base technology should be as simple as possible to keep the number of different nodes low. In [Keu87] the base technology is two-input NAND-gates and inverters, while in work done by Karplus the if-then-else operator has been used as the base technology [Kar89]. The logic function for each library gate is also represented as a graph (known as the pattern graph) in the base technology. For each logic function there are many different representations using the base technology, and hence many different pattern graphs. Technology mapping now amounts to finding the minimum cost covering of the subject graph by choosing from the library of pattern graphs. This problem is well known from code generation in language compilers, where each machine instruction is decomposed into a DAG of atomic operations (the pattern graph), and where the optimized program itself is a DAG of atomic operations (the subject graph). As in language compilers, special-purpose techniques for specific targets often work better than general-purpose technology mappers.

1.3 Logic minimization

The work in this thesis will concentrate on the logic synthesis level, with primary focus on technology-independent logic minimization. We do not distinguish between logic *minimization* and *optimization*—both denote the changing of the representation of a Boolean function such that the representation is as "good" as possible. Even though minimization leads one to think of methods that make the function representation small, we will use the terms to also include the cases where we in fact enlarge the functions to meet other goals, such as testing requirements.

Logic minimization is divided into two-level and multi-level logic minimization. In *two-level logic minimization* the combinational logic is represented in a two-level form that corresponds directly to a physical representation in terms of Or-And (sum-of-products) logic or And-Or (products-of-sum) logic. In the sum-of-products form the goal of two-level logic minimization is to reduce the number of products and the number of inputs to each product. In VLSI a common method for implementing a two-level form uses a Programmable Logic Array (PLA). A PLA consists of an And-plane and an Or-plane. The And-plane produces the products by performing an And operation on the inputs. The Or-plane produces the output signals by performing an Or-operation on the products fed by the And-plane. The height of a PLA is determined by the number of products, and the width is determined by the number of inputs and outputs.

The area of two-level logic minimization is mature and near-minimum PLA realizations can almost always be found [BHMS84, Rud89]. Unfortunately, there are many designs for which a two-level representation is inappropriate. Not only can the number of products be exponential in the number of inputs, but a two-level representation of a design may also be considerably slower than a representation using multiple levels of logic.

Multi-level logic minimization minimizes with the object of implementing the final circuit in random logic. Most previous work in multi-level logic minimization is based on extensions to two-level logic minimization techniques [Bra87a, BRSW87a, BHJ+87, BCGH86]; a notable example is the misII multi-level logic minimization system [BRSW87a]. Many

multi-level minimizers use two-level minimization as a subroutine, usually based on the espresso two-level minimizer [BHMS84]. The objectives in multi-level logic minimization are

- To minimize area of the fabricated circuit.
- To minimize critical path delay.
- To make routability in layout synthesis easier.
- To make testability of the final circuit easier, and in some cases, to provide a test set.

Logic minimization is the technology-independent part of logic synthesis, but some knowledge of the target technology is useful for guiding the minimization in the right direction.

Testing optimizations, or synthesis for testability, refers to optimizations that ensure the design is testable with a small set of test patterns. Logic synthesis has made it possible to produce 100% testable circuits automatically. The result of logic minimization is a circuit that, ideally, is irredundant, and therefore testable. Some techniques of logic minimization can be proven to preserve testability [Kar91b], and thus if the starting point is a testable circuit the final circuit will also be testable.

The objectives of logic minimization have been solved using two different approaches:

- The local-transformation/rule-based approach.
- The algorithmic approach.

The *local-transformation/rule-based* approach is a compiler-like approach, where the circuit is represented as a graph. Transformations are applied to small parts of the graph to transform it into a functionally equivalent, but simpler graph. Local transformations are a rather ad hoc set of rules, and it can be difficult to assure that the number of included rules is in fact sufficient. There is no limit to the number of rules that can be added to a system in order to achieve logic minimization. An example of a rule-based system is LSS [DBG+84], where the level of specification ranges from low-level close to hardware, through register-transfer-level to very high-level descriptions with no assumptions of structural decisions.

The *algorithmic* approach is more global in the sense that small changes may affect the entire circuit. The main global techniques are factorization, extraction of common subexpression extraction, and various algorithms for finding common subexpressions and factors. The most notable example of an algorithmic system is misII [BRSW87a], which evolved from algorithms developed for two-level logic synthesis [BHMS84].

More recently the local-transformation/rule-based and algorithmic approach has been combined, where the algorithmic approach is used in the initial phase of logic minimization, and the rule-based approach is used towards the end and in particular for technology mapping. SOCRATES [BCGH86] and more resent versions of LSS [BT88] are examples of systems combining the two approaches. A slightly different approach is to use local transformations and make them have global effect. This approach is used in ITEM [Kar91c, Kar89] and is accomplished by using a global symbol table to store unique expressions. If a transformation transforms a part of the circuit into a form that already exists somewhere in the circuit, the common form will be explicitly shared. However, there is no guarantee that identical expressions will be recognized and transformed to a common form, and thus ITEM also incorporates some global techniques [SK91, SK93].

Extracting common sub-expressions in a set of Boolean functions is important for minimizing the area occupied by the logic equations. In thesis a new algorithm *two-column rectangle replacement* for factoring and extracting common sub-expressions in Boolean functions is presented. The algorithm is an improved variant of Brayton's Brayton's rectangle covering problem [Bra87a, BRSW87b], and it is particularly well suited for optimizing circuits for area, while controlling delay. We also present a slight variation of the heuristic, which optimizes with respect to delay.

Canonical form representation of Boolean expressions is an important part of logic synthesis and verification. In canonical form, two expressions representing the same logic function have identical structure. The identical structure of logically equivalent expressions make canonical forms useful for tautology checking and Boolean verification. Boolean verification is used frequently in logic synthesis to verify that minimization has not changed

the function of a Boolean expression. If-then-else DAGs and binary decision diagrams have very convenient canonical forms that are easy to compute. Unfortunately the sizes of a canonical if-then-else DAG and ordered binary decision diagrams are very sensitive to the variable order, and hence finding a good order is essential for efficient converting to canonical form.

A major part of this thesis investigates techniques for computing variable orders that result in small canonical forms. We have developed a new improved depth-first ordering heuristics based on reconvergent fanout. This heuristic is fast and produces variable orderings that are superior compared to variable orders found by other depth-first ordering heuristics. We also introduce a new ordering heuristic called *SplitOrder*, which is especially well suited for finding good variable orders for ordered binary decision diagrams (OBDDs). A nice property of the SplitOrder heuristic (not found with the depth-first ordering heuristics) is that it can be iterated to improve the resulting variable order.

1.4 Organization of the thesis

The research in this thesis is concerned with technology-independent optimization. Chapters 2 and 3 are overviews of what has been done in logic minimization. Chapter 2 summarizes different methods for representing Boolean expressions. Chapter 3 shows some different techniques for logic minimization.

In Chapter 4, the UC Santa Cruz If-Then-Else Minimizer ITEM is described briefly. ITEM is an interactive multi-level logic synthesis tool, which contains several algorithms for technology-independent minimization and technology mapping. A major part of the work involved in this thesis has been designing and implementing parts of ITEM, and all the ideas presented in this thesis have been implemented in ITEM.

Chapter 5 and 6 are the main contributions to the area. Chapter 5 presents a new algorithm for extracting common sub-expressions from a logic network. The technique, which we call *two-column rectangle replacement*, builds on the rectangle covering problem introduced by Brayton *et al.* [BRSW87b].

Chapter 6 is concerned with variable ordering for ordered binary decision diagrams (OBDDs) and canonical if-then-else DAGs. The chapter presents new heuristics for finding variable orders that result in small canonical DAGs. The first part of the chapter investigates depth-first, traversal-based ordering heuristics and a new improved depth-first ordering heuristic based on reconvergent fanout is introduced. The second part of the chapter introduces a new ordering heuristic called *SplitOrder*, which is especially well suited for finding variable orders that result in small ordered binary decision diagrams. Experiments are presented that show that even though SplitOrder is targeted for ordered binary decision diagrams it also results in good orders for canonical if-then-else DAGs.

2. Representing Boolean expressions

When designing a logic synthesis system, the first thing to decide is how to represent Boolean expressions internally. There is no specific representation scheme that is best for all tasks of logic synthesis. One representation scheme may be the best for finding common subexpressions and another may be better for verifying that two Boolean expressions are equivalent. In this chapter we will summarize four of the most common representation schemes used and we will demonstrate how these schemes are used when representing a network of functions.

Boolean expressions

Given a Boolean algebra $\mathbf{B}$, the set of Boolean expression on n symbols $x_1, x_2, \cdots, x_n$ is defined using the following recursive rules:

- The elements of $\mathbf{B}$ are Boolean expressions.
- The symbols $x_1, x_2, \cdots, x_n$ are Boolean expressions.
- If f and g are Boolean expressions, then so are
  - $(f) + (g)$
  - $(f)(g)$
  - $(f)'$

Expressions defined by the above rules are referred to as n-variable Boolean expressions.

We relax the definition somewhat by allowing the removal of pairs of parenthesis $(\cdots)$ where such a removal doesn't introduce ambiguity.

In a finite Boolean algebra (the $\mathbf{B}$ in $(\mathbf{B}, +, \cdot, 0, 1)$ is a finite set), the set of Boolean expressions is infinite.

Boolean functions

A function $f : \mathbf{B}^n \mapsto \mathbf{B}$ is an n-variable Boolean function if and only if it can be expressed by a Boolean expression. In the definition below we need to associate a function

with each n-variable Boolean expression on $\mathbf{B}$. Given a Boolean algebra $\mathbf{B}$, the set of n-variable Boolean functions on $\mathbf{B}$ is defined by the following recursive rules:

- For all elements $b \in \mathbf{B}$, the *constant function* defined by

$$f(x_1, x_2, \cdots, x_n) = b \qquad\qquad \forall (x_1, x_2, \cdots, x_n) \in \mathbf{B}^n$$

  is a n-variable Boolean function.

- For any symbols x_i in the set $(x_1, x_2, \cdots, x_n)$ the function

$$f(x_1, x_2, \cdots, x_n) = x_i \qquad\qquad \forall (x_1, x_2, \cdots, x_n) \in \mathbf{B}^n$$

  is a n-variable Boolean function.

- If f and g are n-variable Boolean functions, then for all $(x_1, x_2, \cdots, x_n) \in \mathbf{B}^n$ the functions

$$\begin{aligned}
(f+g)(x_1, x_2, \cdots, x_n) &= f(x_1, x_2, \cdots, x_n) + g(x_1, x_2, \cdots, x_n) \\
(fg)(x_1, x_2, \cdots, x_n) &= f(x_1, x_2, \cdots, x_n) \cdot g(x_1, x_2, \cdots, x_n) \\
(g')(x_1, x_2, \cdots, x_n) &= (g(x_1, x_2, \cdots, x_n))'
\end{aligned}$$

  are also n-variable Boolean functions.

In a finite Boolean algebra (the $\mathbf{B}$ in $(\mathbf{B}, +, \cdot, 0, 1)$ is a finite set), the set of n-variable Boolean expressions is infinite. However, the set of n-variable Boolean functions is finite. In a function table specifying an n-variable Boolean function there are $|\,\mathbf{B}\,|^n$ rows specifying all possible input combinations. Since each input combination can map to $|\,\mathbf{B}\,|$ values in the range, there must be $|\,\mathbf{B}\,|^{|\mathbf{B}|^n}$ n-variable Boolean functions in a Boolean algebra with $|\,\mathbf{B}\,|$ elements. Since we are dealing with a 2-element algebra, there are 2^{2^n} n-input functions.

The mapping from Boolean expressions to Boolean functions is a many-to-one mapping, and this gives rise to the main purpose of logic minimization which is to find the "best" expression for representing a Boolean function.

2.1 Sum-of-products form

Because of two-level (PLA) minimization, sum-of-products form has become a very popular, and probably the most common, way of representing Boolean expressions in logic synthesis tools. Many multi-level minimization techniques rely on methods developed for two-level minimization and the sum-of-products form seemed the obvious way to go [BRSW87a, BHJ+87]. Throughout this thesis we often use the terminology related to sum-of-products form.

A *variable* is a symbol representing a single coordinate of the Boolean space $\mathbf{B}^n$ (e.g., x).

A *literal* is a variable or its negation, or a constant (element of $\mathbf{B}$) or its negation. (e.g., x or x').

A *cube*, a *product*, or a *term* is either 1, a single literal, or a conjunction of literals in which no variable or constant appears more than once. It is common to view a cube as a *set* C of literals, such that $x \in C$ implies $x' \notin C$. For example $\{x, y, z'\}$ is a cube, but $\{x, y, y'\}$ is not.

A *Boolean expression* is a disjunction of cubes. It is common to view an expression as a *set* F of cubes. For example, $\{\{x\}, \{y, z'\}\}$ is an expression.

A *Boolean function* is a mapping of vertices in the Boolean space to members of $\mathbf{B}$, that is, $f : \mathbf{B}^n \mapsto \mathbf{B}$. The mapping is defined by a Boolean expression.

The *support* of a Boolean expression f (support(f)) is the set of variables, such that for each $v \in$ support(f), $v \in C$ for some cube $C \in f$, or $v' \in C$ for cube $C \in f$.

Two Boolean expressions f and g are said to *orthogonal* or *disjoint* if support(f) $\cap$ support(g)= $\emptyset$.

For a Boolean function g the set of vertices in the Boolean space $\mathbf{B}^n$ that satisfies $g(v) = 1$, is said to be the *on-set* of g. The set of vertices that satisfies $g(v) = 0$, is said to be the *off-set* of g. The set of vertices for which we don't care about the value of g is said to be the *don't-care-set* of g. A Boolean function for which the don't-care-set is empty is a

completely specified function. If the don't-care-set is non-empty then g is an *incompletely* specified function. An incompletely specified function is denoted by the triplet (f, d, r), where f, d, and r are completely specified Boolean functions representing respectively the on-set, don't-care-set, and off-set of the incompletely specified function.

An *implicant* of a Boolean function (f, d, r) is a cube c that is included in the union of the on-set and don't-care-set, $f \cup d$, and such that $c \cap r = \emptyset$.

A Boolean function g is said to *contain* another Boolean function h if each implicant of h is also an implicant of g.

A *prime implicant* of (f, d, r) is an implicant that ceases to be so if any of its literals is removed. This means that a prime implicant cannot be contained in any other implicant of (f, d, r).

An *irredundant expression* for a Boolean function (f, d, r) is a disjunction of prime implicants of (f, d, r) that represents (f, d, r) and ceases to do so if any of its cubes is deleted.

2.2 Factored form

One of the tasks of multi-level logic minimization is factoring of expressions to reduce complexity. Unfortunately the sum-of-products form can not represent factored forms in a simple way. As an example consider the Boolean function $f_1 = bdg + b'd'g + dfg$, which in a multi-level representation could look like

$$\begin{aligned} F &= gs_1 \\ s_1 &= ds_2 + b'd' \\ s_2 &= f + b, \end{aligned}$$

where s_1 and s_2 are factors which are represented in sum-of-products form.

Factored forms are introduced to make multi-level representations easier. The factored form for representing Boolean expressions is defined recursively as follows:

- a literal is a factored form,

- a sum of factored forms is a factored form,
- a product of factored forms is a factored form.

The expression $bdg + b'd'g + dfg$ can be written in factored form as $g(d(f+b)+b'd')$. Note, that factored forms are not unique, for example,

$$(d+c)(a'+b')+(a+b)(c'+d')+c'd+a'b+cd'+ab'$$

$$(a+b+c+d)(a'+b'+c'+d')$$

are distinct Boolean expressions both representing the same Boolean function. By using De Morgan's law the negation of a factored form is easily obtained and is itself a factored form.

The literature has reported several attempts to minimize factored forms, see [BHS90] for a list of references, but unlike sum-of-products form it is hard to determine if a given factored form is optimal. Lawler [Law64] presented an algorithm for obtaining optimal factored forms, but the approach is only feasible for low-complexity functions of few inputs [Wan89].

2.3 Binary decision diagrams

Binary decision diagrams offer an alternative way of representing and manipulating Boolean expressions [Bry86]. They have recently become very popular for verification purposes [Bry86, Bry85, MWBS88], and attempts to use them for logic minimization has also been reported [FFK88].

A binary decision diagram is a directed acyclic graph that use a single universal operator: the if-then-else operator.

Definition 1: *The* if-then-else *operator is a ternary Boolean function, with* (**if** a **then** b **else** c) *defined as* $ab + \neg ac$ *or, equivalently,* $(a+c)(\neg a + b)$.

The if-then-else operator is very flexible and can directly represent 2-input AND, OR, XOR, and IF expressions.

Binary decision diagrams restrict the **if**-part to always being a single variable, and are defined as follows:

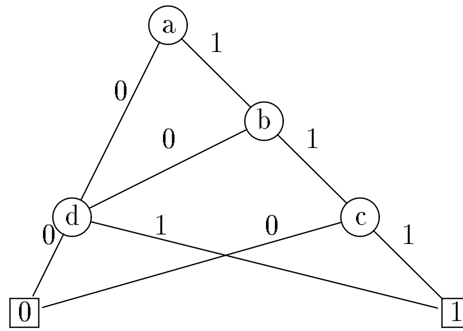

Figure 2.1: Binary decision diagram for $abc + \neg ad + \neg bd$

Definition 2: *A* binary decision diagram *is a binary directed acyclic graph with two leaves* TRUE *and* FALSE*, in which each non-leaf node is labelled with a variable and has two out-edges pointing to the* **then***-part and the* **else***-part. The meaning of a binary decision diagram is defined recursively as* (**if** *label*(*node*) **then** *meaning*(**then**-*part*) **else** *meaning*(**else**-*part*))*.*

Figure 2.1 shows a binary decision diagram for the Boolean expression $abc + \neg ad + \neg bd$. It should be noted that each non-leaf node itself represents a Boolean expression.

Binary decision diagrams are easy to construct [Bry86], but without other restrictions they can be difficult to simplify or compare for equality—$a + b$ can be represented with two different binary decision diagrams: one can have a as root and the other b as root. In Section 3.3 we will show how binary decision diagrams can be canonically represented using *Bryant's canonical form.*

2.4 If-Then-Else DAGs

A major focus of this thesis will be on if-then-else DAGs, which basically are extended binary decision diagrams that allows for sharing of the **if**-part [Kar89, Kar88].

Like binary decision diagrams, if-then-else DAGs use the universal *if-then-else* operator, but unlike binary decision diagrams, there is no restriction that the **if**-part must be a single variable:

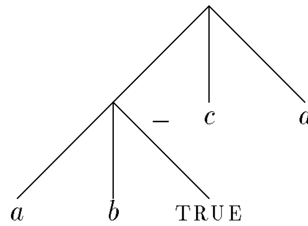

Figure 2.2: If-then-else DAG for $abc + \neg ad + \neg bd$. The left branch from a node points to the **if**-part, the center to the **then**-part, and the right to the **else**-part.

Definition 3: *An* if-then-else DAG *is a directed acyclic graph in which each leaf is labelled with* TRUE *or a variable, and each internal node has three out-edges pointing to the* **if**-*,* **then**-*, and* **else**-*parts. Each edge is labelled with either* **plus** *or* **minus***. The* meaning *of a node in the* DAG *is defined recursively:*

- *The meaning of a leaf node is the label on the node.*
- *The meaning of a pointer is the meaning of the node pointed to (if the label on the edge is* **plus***) or its negation (if the label on the edge is* **minus***).*
- *The meaning of an internal node is*

  (**if** meaning(**if**-part) **then** meaning(**then**-part) **else** meaning(**else**-part)).

Two nodes are equivalent *if their meanings are logically equivalent.*

An edge with a minus label pointing to TRUE will sometimes be referred to as an edge pointing to FALSE. In all figures only minus labels are shown.

Figure 2.2 shows the if-then-else DAG for the Boolean expression $abc + \neg ad + \neg bd$, here it is represented as (**if** (**if** a **then** b **else** FALSE) **then** c **else** d).

Like BDDs, if-then-else DAGs are impractical to manipulate if they can appear in any form. Several restrictions are placed on if-then-else DAGs to make them canonical, refer to Section 3.3. In practice however, we only require the following subset of the canonical form restrictions to be satisfied:

Systematic-negation conditions: A systematic choice must be made between the equivalent DAGs (**if** a **then** b **else** c) and (**if** a' **then** c **else** b) and between (**if** a **then** b **else** c) and (**if** a **then** b' **else** c')$'$. We require that **if**- and **then**-parts of a node be pointers

labeled **plus**, with negation allowed only for the **else**-part or the pointer to a root of the DAG.

Weak distinct-cases condition: The **then**- and **else**-parts of a node must be different pointers or the **else**-part must be a pointer labeled with **minus**. In canonical form the restriction implies that the **then**- and **else**-parts are different Boolean functions.

No-constant-if condition: Triples whose **if**-part points to TRUE are prohibited, and should be replaced by the **then**-part.

No-two-constant condition: Triples in which both the **then**- and **else**-parts point to TRUE (with either plus or minus labels) are prohibited. The triple should be replaced by an appropriately labeled pointer to the **if**-part or to TRUE.

2.5 Two input NORs

If-then-else DAGs and binary decision diagrams use the single universal if-then-else operator. An alternative single universal operator, which has been used in LSS [DBG+84] is the *two-input NOR gate*. Obviously it is less flexible than the if-then-else operator as it can only represent only 5 Boolean functions: 0, 1, x_1, x'_1, and $x'_1x'_2$. However, it has proven to be useful and successful in a rule-based logic synthesis system and it is also quite popular as the fine-grain network for technology mapping [Rud89].

2.6 Representing Boolean networks

One of the tasks of logic minimization is to find common subexpressions in a network of Boolean functions constituting the entire block of logic we are optimizing. We therefore also need a way to represent a network of Boolean functions as an entity, this entity is called a *Boolean network*.

One way of representing a set of Boolean functions is as a directed acyclic graph, in which each node represents a Boolean expression and all leafs are simple variables or constants

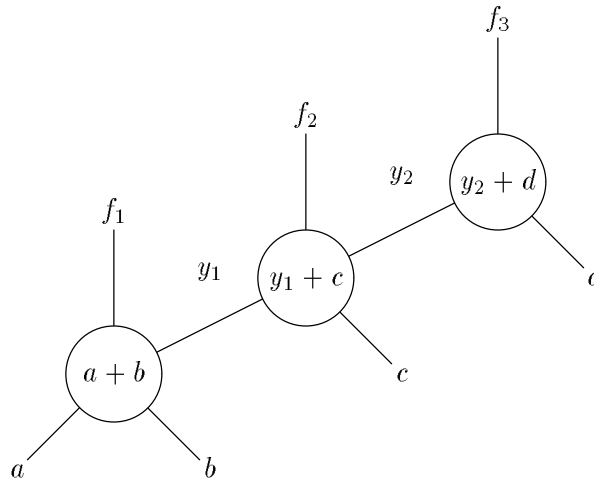

Figure 2.3: Boolean network with three primary outputs and 4 primary inputs. The expressions are represented are $f_1 = a+b$, $f_2 = a+b+c$, and $f_3 = a+b+c+d$.

(constants are elements of the Boolean algebra, which are TRUE and FALSE for the two-element Boolean algebra). Some nodes are designated as output nodes and are referred to as *primary outputs*; these nodes are associated with the Boolean functions we are representing. Similarly all leaf node variables, except the constants, are denoted *primary inputs*.

In sum-of-products (or factored) form a Boolean network is DAG in which each node is associated with a variable y_i and a sum-of-products (or factored) representation of a function f_i. An arc from node i to node j indicates that y_i is used explicitly in the representation of f_j. Figure 2.3 shows a Boolean network for the set of functions $f_1 = a + b$, $f_2 = a + b + c$, and $f_3 = a + b + c + d$. The functions are decomposed into $f_1 = a + b$, $f_2 = f_1 + c$, and $f_3 = f_2 + d$.

A Boolean network is a gate-level representation of a set of Boolean functions—each node is a gate, which is allowed to fanout to several other gates. Minimization of a Boolean network consist of two steps:

- Rearranging the overall structure of the Boolean network. This involves finding common subexpressions, which are then extracted and added to Boolean network as separate nodes.
- Minimizing each node of the Boolean network separately.

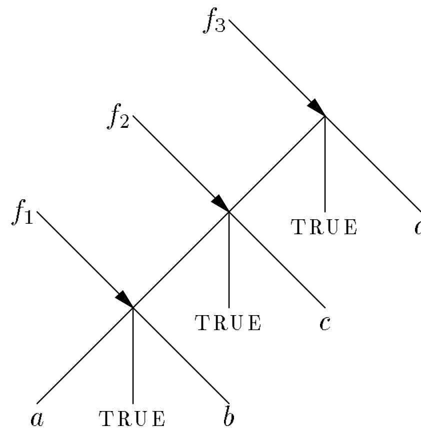

Figure 2.4: Multiply-rooted if-then-else DAG. The three expressions represented are $f_1 = a + b$, $f_2 = a + b + c$ and $f_3 = a + b + c + d$.

The disadvantages of using a Boolean network like the one shown in Figure 2.3 is that different representations are used for the DAG itself and for the nodes of the DAG. The Boolean network is a graph structure, whereas the nodes are some representation of Boolean expressions. Due to this different representation, each node must be labeled with an intermediate variable y_i, which is then used explicitly by other nodes. By using y_i in the representation of f_j (node i fans out to node j) rather than the expression that represents y_i, information is lost. The lost information is summarized in the *intermediate don't-care set*, IM_{DC}:

$$IM_{DC} = \sum_{n_i} y_i \oplus f_i$$

An if-then-else DAG can represent a network of functions directly. Instead of having one root we allow an if-then-else DAG to be multiply-rooted, where each root represents a primary output. Figure 2.4 shows an example of a multiply-rooted if-then-else DAG. Notice, that the representation is the same as in Figure 2.2, and the only extra information that needs to be kept is a table of pointers to the multiple roots.

Minimization of a multiply-rooted if-then-else DAG usually consists of

- Minimization using local transformations at any level in the DAG.
- Finding common subexpressions.

The advantage of if-then-else DAGs is that the universal if-then-else operator is the level of representation for both a single Boolean expression and a network of Boolean functions. This means that there is no artificial partitioning into “gates”, and hence the intermediate don't-care set is non existent.

Binary decision diagrams can represent network of functions in a manner similar to if-then-else DAGs, however, since a BDD cannot directly represent all 2-input gates and selectors, it is less flexible than an if-then-else DAG, and some rearrangements are necessary when constructing a multiply-rooted BDD.

3. Logic minimization techniques

We now turn towards solving the objectives of logic minimization. This chapter gives an overview of previous and current work by other researchers and new work done in connection with this thesis.

Some of the methods described in this section have been briefly touched upon in preceding sections, but here we will be slightly more detailed. For each method we first describe its purpose in the context of logic minimization, and we then outline some techniques that have been used for implementing the method and solving the particular problem.

Before starting out it may be helpful to review the objectives of logic minimization as they were mentioned in Section 1.3:

Area, minimize the area and cost of the fabricated circuit.

Delay, minimize the critical path delay.

Routability, make routing of the final layout easier.

Testability, make testing of the fabricated circuit easier, and in some cases provide a test set.

All of the above objectives interact in that it is almost always impossible to find an implementation of a function that is optimal in all objectives. Optimizing for area is usually at the cost of increased delay and more difficult routing. Testability can be hard to preserve when certain powerful transformations, such as the *generalized bypass transform* by McGeer [MBSS91], are used.

3.1 Factoring

Factoring is the process of transforming a Boolean expression to a factored form, see Section 2.2. For example,

$$G = bdg + b'd'g + dfg$$

can be factored to

$$G = g(d(f + b) + b'd')$$

In a Boolean network, see Section 2.6, where the underlying representation is sum-of-products form, factorization is one of the main techniques for minimizing the node functions. Another important technique used prior to factorization is 2-level logic minization as in *espresso* [BHMS84].

In sum-of-products form, factorization of a function usually consists of finding a "factor candidate" and then "dividing" the function by this candidate. Division is not defined in a Boolean algebra, since the only operators are the binary $+$ and $*$ (also known as *disjunction* and *conjunction*). However, we can define an operation, which behaves like the division operation. Given two Boolean functions f and p, *division* of f by p generates a *quotient* q and a *remainder* r, such that the following equation is satisfied:

$$f = pq + r \ .$$

Obviously such a division operation can not be unique.

The function p is called a *Boolean divisor* of f, or if $f = pq$ then p is called a *Boolean factor* of f. The number of Boolean divisors and factors is clarified by the following propositions borrowed from [BHS90]:

Proposition 1: *A logic function p is a Boolean factor of a logic function f if and only if $f \neg p = 0$, i.e., if f is in sum-of-products form then every term of f contains p.*

Proposition 2: *A logic function p is a Boolean divisor of a logic function f if and only if $fp \neq 0$, i.e., some term in f must not contain $\neg p$.*

Any function containing f is a Boolean factor of f. Any function not orthogonal to f is a Boolean divisor of f. Factorization of f is usually a recursive procedure, where we first find a factor candidate p, then perform the division to generate q and r, which are then factored recursively. The factor candidates are sought among the divisors of f, but only a few of the many divisors are suitable for factorizing f, and the real problem is to determine these suitable divisors—performing the division is a simple task.

The first way to restrict the number of divisors is to consider only *algebraic divisors.* The function p is an *algebraic* divisor of f if $f = qp + r$, where qp is non-null and q and p are orthogonal (support(p)$\cup$ support(q) = $\emptyset$), so that the multiplication pq can be carried out without considering Boolean identities and complements (pq is an algebraic product). Brayton [BHS90] presents an algorithm that performs algebraic division, that is, given f and p it uniquely determines q and r such that,

- pq is an algebraic product,
- r has a few cubes as possible, and
- $pq + r$ and f are the same expression (having the same number of cubes).

This algorithm is know as *WEAK_DIV* and denoted by the symbol /.

The number of candidate divisors can be further limited by considering only *kernels.* A *kernel* k of an expression f is an expression such that

- k is the quotient of f and a cube c, k=f/c (c is called the *co-kernel*),
- k is *cubefree*, meaning that k does not contain any factors that are simple cubes.

Kernels provide a useful set of divisors to choose from when factoring a function, and various algorithms for finding kernels are presented in [Bra87b].

3.2 Sub-expression extraction

Consider a Boolean function f represented by the Boolean expression F. Define a *sub-expression* of F as an expression G, such that f can be written as

$$f = QG + R \tag{3.1}$$

where Q and R are Boolean expressions and Q is non-zero. Sometimes G is referred to as a *factor* or *divisor* of F and Q is referred to as the *quotient* of F with respect to G.

Given two Boolean expressions F and G and their associated Boolean functions, a *common sub-expression* of F and G is an expression C such that the functions f and g can be written as

$$f = Q_1 C + R_1 \quad (3.2)$$

$$g = Q_2 C + R_2 \quad (3.3)$$

Sub-expression extraction is the problem of finding and extracting common sub-expression in a network of functions. A new node is created for each common sub-expression and the node is the made to fan out explicitly to all the expressions that use the common sub-expression.

The purpose of sub-expression extraction is to maximize sharing in the network of functions, and thus extracting sub-expressions will tend to reduce the area needed to implement the functions. Adding nodes to the network will increase the number of levels in the DAG and maybe increase the number of levels on the critical path, hence sub-expression extraction may increase the delay. Extracting sub-expression tends to make placement and routability more difficult since common nodes have multiple fanout and perhaps can't be placed close to all fanout nodes at once. Testability can be preserved if all products in the rewritten expressions (3.1)–(3.3) are limited to be algebraic products [HJKM89].

Types of sub-expressions

In sum-of-products forms, common sub-expressions are sought among the set of cubes and kernels. Recall that a cube is the either 1, a single literal, or a conjunction of literals. A kernel of a function f is a cubefree (it contains more than one cube) divisor of f.

In misII [BRSW87a] the common subexpressions are sought in the set of cubes and kernels, known as *common-cube extraction* and *kernel-intersection extraction.* In common-cube extraction divisors are cube intersections common to two or more expressions. For example, the functions

$$f_1 = abm$$

$$f_2 = abk$$

have the cube ab in common, and hence both f_1 and f_2 are divided by ab to obtain

$$f_1 = xm$$
$$f_2 = xk$$
$$x = ab$$

In kernel-intersection extraction divisors are kernel intersections common to two or more expressions. For example, the functions

$$f_1 = abk + abl + abm$$
$$f_2 = cdk + cdl + cdn$$

have the the cubefree expression $k+l$ in common. This expression, which is the intersection of the kernels $k+l+m$ and $k+l+n$, is divided into both f_1 and f_2 to obtain

$$f_1 = abx + abm$$
$$f_2 = cdx + cdn$$
$$x = k + l$$

3.3 Canonical form, verification and testability

In canonical form the representations of two logically equivalent expressions are identical. We can distinguish between *weak canonical forms*, in which logically equivalent expressions have identical structure, but may occur in different locations in memory, and *strong canonical forms*, in which expressions in different locations represent different Boolean functions. Strong canonical forms are particularly useful, because they guarantee that any explicitly represented subexpression is shared by all expressions that need it. Thus one of the most common tasks of logic minimization, that of finding common subexpressions, can be achieved in part by converting an expression to strong canonical form. Unfortunately, the major limitation of canonical forms is that subexpressions are not necessarily explicit, and one challenge is to come up with a canonical form that is capable of expressing as much sharing as possible.

Some representation schemes have more convenient canonical forms than others. The most *inconvenient* canonical form is for the sum-of-products and factored form, where all implicants are reduced[1] to minterms, that is, the canonical form for a function (f, d, r) consist of all the minterms covered by the $f \cup d$. For example, the canonical form for the completely specified function $f(x, y, z) = x'y + z' + xyz$ is

$$\begin{aligned} f_c(x, y, z) &= x'y(z + z') + (x + x')(y + y')z' + xyz \\ &= x'yz + x'yz' + xyz' + xy'z' + x'y'z' + xyz \end{aligned}$$

Due to the large number of minterms (up to $2^n - 1$) the sum-of-products canonical form quickly becomes impractical, and more recently researchers have adapted the binary decision diagram representation for verification purposes [MWBS88, MF89].

Binary decision diagrams and if-then-else DAGs have very convenient canonical forms. Bryant formulated a canonical form for binary decision diagrams [Bry86]. As originally described, it is a weak canonical form, but adding a permanent symbol table to give unique ids to each node makes it a strong canonical form. Bryant's canonical form is obtained by ordering the set of variables, and constructing the binary decision diagram such that the variable at each node in the diagram is earlier in the order than the variables of its children. A second restriction requires that the BDD is *reduced*, meaning that it contains no nodes of the form (**if** i **then** x **else** x), nor does it contain distinct nodes representing the same Boolean function. The *reduced* condition ensures that a single binary decision diagram will be in strong canonical form, but since each expression is handled separately, two independently built expressions may occupy different memory locations, but be logically equivalent. The size of a canonical binary decision diagram is very dependent on the variable ordering and finding the best ordering is a co-NP-complete problem [Bry86]. The binary decision diagram in Figure 2.1 is in canonical form with respect to the ordering (a, b, c, d),

[1] An implicant is reduced by adding literals to the implicant—this awkward terminology was introduced in espresso [BHMS84]

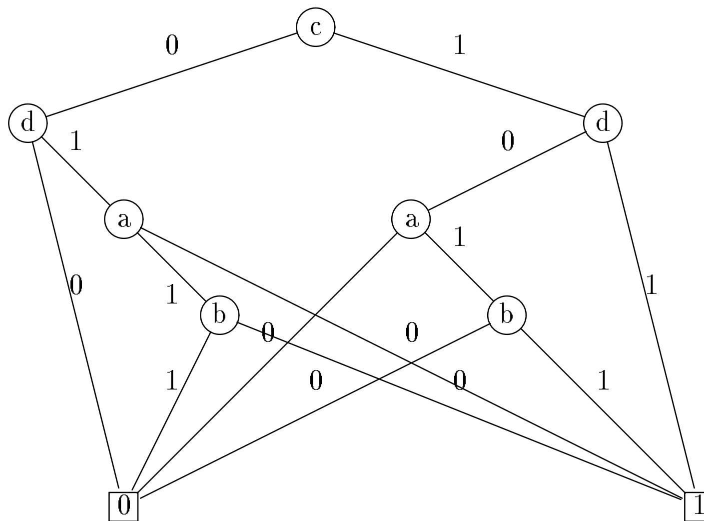

Figure 3.1: Binary decision diagram for $abc + \neg ad + \neg bd$, with respect to the ordering (c, d, a, b).

whereas the DAG in Figure 3.1 shows the same function in canonical form with respect to the ordering (c, d, a, b).

Karplus [Kar89] formulated a different strong canonical form for if-then-else DAGs. Conversion to canonical form consist of 7 rules, three of which (the *systematic-negation condition*, the *no-constant-if condition*, and the *no-two-constant condition*) are usually used in all representations of if-then-else DAGs, even non-canonical ones, refer to Section 2.4. The remaining 4 rules are

Variable ordering condition: A total ordering is imposed on the variables, and all the variables in the **if**-part must be earlier in the order than all variables in the **then**- and **else**-parts.

A weaker restriction, that the variables of the **if**-part be disjoint from those of the **then**- and **else**-parts is not enough to make the if-then-else DAG canonical, but is all that is needed for path-delay-fault testability. This weaker restriction is referred to as the *separate-support condition.*

Distinct-cases condition: The **then**- and **else**-parts of a node must be distinct Boolean functions—exactly as in Bryant's canonical form.

No-common-cut condition: In the triple (**if** a **then** b **else** c), b and c must not share both **then**- and **else**-parts. If b = (**if** b_a **then** b_b **else** c_c) and c = (**if** c_a **then** b_b **else** c_c), then the correct representation is (**if** (**if** a **then** b_a **else** c_a) **then** b_b **else** c_c). If b = (**if** b_a **then** b_b **else** b_c) and c = (**if** c_a **then** b_c **else** b_b), then use (**if** (**if** a **then** b_a **else** c'_a) **then** b_b **else** b_c).

No-collapsed-cut condition: In the triple (**if** a **then** b **else** c), b must not contain c as a **then**- or **else**-part. If b = (**if** b_1 **then** b_b **else** c) or b = (**if** b_2 **then** c **else** b_c), then the DAG should be changed to (**if** (**if** a **then** b_1 **else** FALSE) **then** b_b **else** c) or (**if** (**if** a **then** b_2 **else** TRUE) **then** c **else** b_c). If c is a constant (TRUE or FALSE), then this restriction amounts to choosing left-associativity for commutative AND or OR operations. The symmetric test for c = (**if** c_1 **then** c_b **else** b) or c = (**if** c_2 **then** b **else** c_c) is also needed.

The *variable ordering* and *distinct-cases* conditions correspond directly to the restrictions Bryant imposed on BDDs to make them canonical.

As in OBDDs the rule with the most influence on the size of the DAG is the *variable ordering condition.* The if-then-else DAG in Figure 2.2 is in canonical form with respect to the ordering (a, b, c, d), whereas the DAG in Figure 3.2 shows the same function in canonical form with respect to the ordering (c, d, a, b). Note, that Figure 3.2 is in fact an OBDD, since each **if**-branch is a single variable. The main difference between Figure 3.1 and Figure 3.2 is that the if-then-else DAG uses systematic negation, which allows an expression and its negation to be represented by the same subDAG.

Karplus [Kar91b] recently showed that if-then-else DAGs in canonical form are 100% path-delay fault testable, and hence testable for single and multiple stuck-at faults. By converting to canonical form and using testability-preserving transformations we can optimize for testability.

In strong canonical form, identical expressions are stored in the same memory location and hence common subexpressions will be explicitly shared. Unfortunately conversion to canonical form isn't guaranteed to give the best decomposition of a set of functions—this is clearly illustrated by Figure 2.2 and 3.2. Even if the best variable order is found, the size

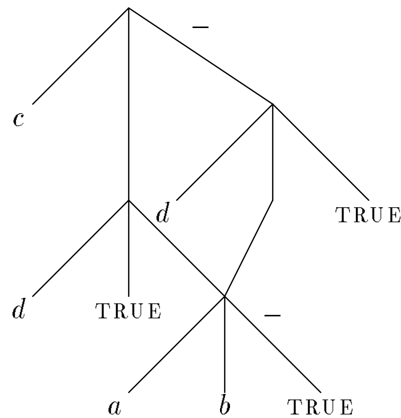

Figure 3.2: Canonical if-then-else DAG for $abc + \neg ad + \neg bd$, with respect to the ordering (c, d, a, b).

of the canonical DAG may be larger than a corresponding non-canonical DAG.

In both OBDDs and canonical if-then-else DAGs equivalence checking can be done by a simple traversal of the DAG (taking $O(n)$ time), or, if in strong canonical form by comparing, two pointers ($O(1)$). Because equivalence checking is fast in canonical form, but equivalence checking in non-canonical form is equivalent to the complement of the NP-complete problem SATISFIABILITY, refer to Gary and Johnson [GJ79, page 261], we are almost guaranteed that conversion to canonical form is exponential in the worst case.

The NP completeness result guarantees that some functions will have exponentially large OBDDs or canonical if-then-else DAGs. Bryant proved that some useful functions (the middle output of an integer multiplier) have exponentially large OBDDs for any variable order [Bry91]. His results apply equally well to if-then-else DAGs, though there are canonical if-then-else DAGs with exponentially larger OBDDs.

4. ITEM

ITEM is our if-then-else minimizer used as an environment for testing various logic synthesis techniques. ITEM is implemented in C++ and currently consists of approximately 50,000 lines of code. ITEM is an interactive system like the MISII minimizer [BRSW87a].

Figure 4.1 shows a general overview of ITEM. ITEM constructs its initial multiply-rooted if-then-else DAG from some network description. Various optimizations are then applied to the if-then-else DAG to improve area, delay, or testability of the circuit. Once optimized the circuit is mapped to a target technology. ITEM currently supports mapping to field-programmable gate-arrays and complex gates. The optimizations and the mapping can be iterated to improve results. After the final mapping, ITEM can output a netlist in different formats, which then can be passed to other synthesis tools.

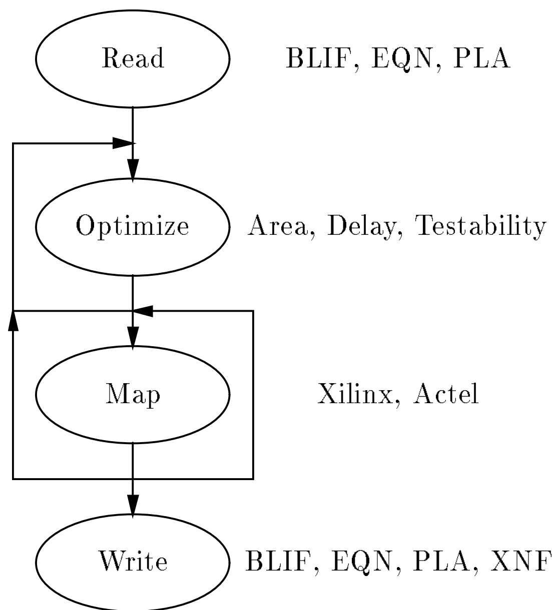

Figure 4.1: General outline of ITEM. After reading a description of a network, various optimizations are applied. The mapping phase supports mapping to field-programmable gate-arrays and complex gates. Both the mapping phase and the optimization phase can be iterated to improve results. Finally, ITEM can output a netlist in different formats, which then can be passed to other synthesis tools.

4.1 Reading and writing different formats

Most of the standard benchmarks are available in either the Berkeley Logic Interchange Format (BLIF, for short) or in the Berkeley equation format (EQN, for short). BLIF is capable of expressing both combinational and sequential logic, whereas EQN can only express combinational logic.

To allow comparisons with other tools, particularly MISII, we have chosen to use the BLIF and EQN file formats as our main interchange formats for both input and output. We also use other formats as needed (for instance, we can output XNF format after mapping to Xilinx cells).

The IO module of ITEM reads a textual file containing a description of a circuit, and converts to the internal if-then-else DAG format. Because the different formats have different underlying models of the circuits, conversions are more than just a simple textual substitution. For example, conversion of the sum-of-products format in BLIF to if-then-else DAGs requires something roughly equivalent to single-cube factoring [Kar89].

4.2 Optimizations in ITEM

After constructing the initial multiply-rooted if-then-else DAG we apply optimizations to produce a DAG that meets the requirements we have set forth. Generally the optimizations are technology independent, but some knowledge of the target technology can be used in guiding the optimizations in the right direction. For instance, in a technology, where routing is a problem, it might be beneficial to minimize the number of edges in the DAG. Some other technologies would perhaps benefit from minimizing the number of nodes in the DAG. ITEM currently offers the following optimizations:

- *Minimizing* using local transformations.
- *Conversion to canonical form* for verification or to create a fully testable circuit.
- *Finding shared sub-expressions* using local and global techniques.

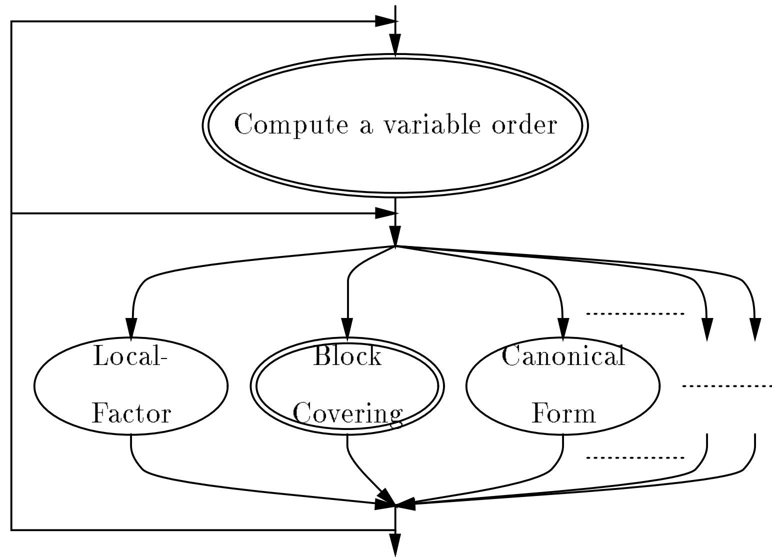

Figure 4.2: One view of the optimization phase in ITEM. First some variable order is computed, then different optimizations are applied. The variable ordering step can be skipped if the optimization step itself does not rely on variable order. If no variable is computed the system will use a default order if needed. The highlighted areas are the subjects of Chapter 5 and Chapter 6 of this thesis.

Figure 4.2 shows one way to organize the optimizations in ITEM. Many of our optimizations rely on an ordering of the input variables before they can be applied, hence the first step is to compute a variable order. The various optimizations can be iterated to improve results. The double circled areas of Figure 4.2 are the original contributions and are the topics of the remaining chapters in this thesis.

4.2.1 LocalFactor

Our initial work in logic minimization used local transformations applied to all portions of the DAG, generally in a depth-first traversal from the outputs. We came up with two sets of transformations: *Printform* and *LocalFactor*.

The Printform transformations [Kar89] preserves testability, but are not good at minimizing circuit area. The reason they are not good circuit minimizers is that they were originally designed to minimize the size of printed Boolean expressions, not multi-level circuits. They attempt to minimize the *pcount* measure [Kar89], which predicts printing size

quite well, but which is not a good predictor of circuit area or delay.

The LocalFactor transformations are a rather *ad hoc* collection of transformations that do an adequate job of minimizing circuit area [Kar89]. Unfortunately, they do not preserve path-delay-fault testability. LocalFactor relies on variable order, since one of its most powerful transformations is conversion to canonical form applied small parts of the DAG. Unlike conventional sum-of-products minimizers, which minimize the number of literals in the network, LocalFactor can be used to minimize whatever estimator we have decided best predicts the property (area or delay) that we are trying to minimize.

4.2.2 Block covering

The local transformation techniques often find interesting ways to rearrange functions, but the resulting expressions often have common subexpressions that have not been merged.

The block covering[1] algorithms developed for MISII [BRSW87b, BRSW87a] are very effective at finding shared expressions, but are too expensive to apply to entire circuits (MISII applies them only at the gate level). We have developed a cheaper variant, *two-column rectangle replacement*, which can be applied to Boolean matrices derived from entire large circuits, and which is quite effective at finding sharing. The heuristics of the replacement strategy can be tuned to maximize sharing (minimizing circuit area) or to balance operator trees (minimizing delay) [SK91]. Two-column rectangle replacement preserves path-delay-fault testability [Kar91b]. Two-column rectangle replacement does not rely on variable order. Chapter 5 presents the two-column rectangle replacement we have developed.

4.2.3 Conversion to canonical form

Conversion to canonical form has several advantages. As pointed out in Section 3.3 an if-then-else DAG in canonical form is 100% path-delay fault testable, and hence testable for single- and multiple stuck at faults. This means that converting to canonical form can

[1] Block covering is also referred to as *rectangle covering*

be the first step in synthesis for testability, which can then be followed by applying only testability-preserving transformations. The most effective testability-preserving transformation available in ITEM appears to be conversion to canonical form with different variable orderings, optionally followed by two-column rectangle replacement.

Strong canonical form also offers the detection of shared subexpressions, since logically equivalent expressions are stored in the same memory location. This is exactly what LocalFactor makes use of when it converts small parts of the DAG to canonical form.

Unfortunately the size of an if-then-else DAG in canonical form is very sensitive to the variable ordering, and hence finding a good ordering is essential for efficient conversion to canonical form, and for achieving good minimization. In Chapter 6 we present several variable ordering heuristics for finding variable orders that result in small canonical forms.

4.3 Mapping

ITEM currently supports mapping to complex gates and field-programmable gate arrays [Kar91d, Kar91a]. Field-programmable gate array mappers have been developed for Xilinx-style arrays (*Xmap* [Kar91d], *Xcmap* and *Xtmap* [Kar93]) and for Actel-style arrays (*Amap* [Kar91a]). Both *Amap* and *Xmap* preserve testability.

5. Rectangle Replacement

This chapter describes the use of rectangle replacement for multi-level logic minimization on functions represented as if-then-else DAGs. We define the concept of Boolean matrices, and give formal definitions of blocks and rectangles and their meanings. We introduce a new heuristic, *two-column rectangle replacement* for finding rectangle coverings of Boolean matrices. This heuristic is particularly well suited for optimizing circuits for area, while controlling the delay. A slight variation of the heuristic optimizes with respect to delay. The results of using two-column rectangle replacement on if-then-else DAGs are reported for several benchmark examples.

5.1 Introduction

This chapter is concerned with factoring and recognizing shared subexpressions in Boolean functions. We use a technique we call *two-column rectangle replacement of Boolean matrices*. Boolean functions are represented as Boolean matrices, and rectangles of these matrices represent either a factor of a function or a subexpression that can be shared among several functions. The rectangle replacement problem is a variant of Brayton's rectangle-covering problem [Bra87a, BRSW87b]. In both we find sets of rectangles that cover all the 1's of the Boolean matrix—rectangle replacement differs from rectangle covering in the way rectangles are replaced. In Brayton's rectangle covering, rectangles were replaced in parallel—the aim was to find a set of rectangles that best covered a Boolean matrix, and then replace all of these rectangles in one step. In rectangle replacement and two-column rectangle replacement, we replace rectangles in sequence, that is, we find and replace one rectangle at a time, until the Boolean matrix is covered. Because the Boolean matrix changes after each replacement, the solution to the two problems may differ significantly.

We show how to create Boolean matrices from functions represented as if-then-else DAGs, and we give formal definitions of blocks and their meanings. We introduce a new heuristic, *two-column rectangle replacement*, a simple yet efficient method for optimizing

multi-level logic. Even though our starting point is if-then-else DAGs, two-column rectangle replacement does not rely on the if-then-else DAG representation—it could be applied any time that rectangle covering is useful.

In Section 5.2 we focus on Boolean matrices and blocks and rectangles of Boolean matrices. We formally define the semantics of *blocks* and *rectangles*, and we show how to replace rectangles of the matrix with simpler rectangles, while maintaining the meaning of any blocks in the matrix. We finally define the rectangle replacement problem, which consists of finding and replacing rectangles in the right order.

In Section 5.3, we use two-column rectangle replacement in ITEM. We first show how to create Boolean matrices from if-then-else DAGs, and then we give an algorithm for solving the rectangle-replacement problem. This algorithm is based on a heuristic method for selecting the order of rectangle replacement. Two-column rectangle replacement consists of finding rectangles with exactly two columns in the matrix. These two columns have associated Boolean expressions, which will be combined into a new expression using an associative and commutative logic operator.

Two-column rectangle replacement is well suited for optimizing multi-level logic with respect to both delay and area. For all the cases we are considering in this chapter, an expression is created from several other expressions by pairwise combining expressions with an associative and commutative operator. It has been shown [Kar89] that the height of an if-then-else DAG is a usable, though not very good, delay estimate for the final circuit. The easiest way to keep the height under control is to balance the DAG when we are creating it. Clearly, if we combine the if-then-else DAGs such that low height DAGs gets combined first, we achieve a form of tree balancing that will keep the delay under control. Thus it is possible to optimize for delay by replacing two-column rectangles in the order of increasing height. A similar tree balancing approach named DMIG, has been used in DAGMAP [CCD+92] to transform an arbitrary Boolean network into a n-input network of minimum height, where all gates have at most n inputs. ITEM also includes a direct implementation of DMIG and in the results section we compare two-column rectangle replacement with DMIG. The original

idea behind DMIG is due to Wang [Wan89], who proposed a timing-driven decomposition algorithm for timing optimization.

Section 5.4 presents some results of minimizing multi-level logic benchmarks using two-column rectangle replacement.

Motivation

The original motivation behind two-column rectangle replacement was the need for a global optimization technique that would merge logically equivalent if-then-else DAGs.

Local factoring techniques can be used to factor if-then-else DAGs [Kar89]. If factoring results in two identical if-then-else sub-DAGs they will be merged into a single copy. Unfortunately local factoring techniques fail to give us the global view needed to identify that that two expression can be identical even though they are represented differently. For instance, $a + b + c$ could be represented as (**if** (**if** a **then** TRUE **else** b) **then** TRUE **else** c) or as (**if** a **then** TRUE **else** (**if** c **then** TRUE **else** b)). If both the representations are used, they will not be recognized as being the same expressions, unless they happen to be transformed to a common form. Although we have used canonical forms to merge such common subexpressions, the computation of canonical forms is often too expensive, as they will sometimes be exponentially large, even for some common circuits such as multipliers [Bry91].

In this chapter we explore rectangle replacement, a variant of rectangle covering [Bra87a, BRSW87b], which has been useful for finding common subexpressions in sum-of-products minimizers, and see how it can be applied to if-then-else DAGs. We use rectangle replacement primarily to recognize commonality in commutative and associative operations—roughly the equivalent of the common-cube extraction operation of MISII.

5.2 Blocks and rectangles

The *rectangle-covering problem* applied to logic synthesis was first presented by Robert Brayton [Bra87a]. He showed how a set of Boolean functions could be represented as a Boolean matrix, and that finding "rectangles" of this matrix was the same as finding

factors of expressions and common subexpressions of a network of functions. In this section we introduce rectangle replacement, a serialized version of rectangle covering.

We define only the concepts needed to introduce and prove the correctness of our two-column rectangle replacement algorithm.

5.2.1 Boolean matrices and blocks of Boolean matrices

A *Boolean matrix* is a two-dimensional matrix representing a logic expression or a set of logic expressions. Each row and each column in the matrix is associated with its own expression. The row expressions are built out of some combination of the column expressions. For example, if the rows are sum-of-products expressions, then each column would correspond to a product term (cube) in some row expression. We are not limited to sum-of-products or product-of-sums representations, but allow arbitrary expressions on the columns and any associative, commutative operation to combine the column expressions into rows.

An entry B_{rc} in the Boolean matrix takes the values 1, 0, and d depending on whether the expression for row r contains column c (1), doesn't contain it (0), or we don't care whether it contains it or not (d).

Consider the functions

$$\begin{aligned} f_1 &= a + b + de \\ f_2 &= b + de + c(f + g) \\ f_3 &= cf + cg, \end{aligned}$$

which are all sum-of-products. A Boolean matrix representing these functions is shown in Figure 5.1—the matrix is referred to as an OR-matrix, since each row is the OR of the corresponding columns.

We need to be able to talk about parts of the matrix as single entities, and so we define a block:

OR-matrix	a	b	de	c(f+g)	cf	cg
$f_1 = a + b + de$	1	1	1	0	0	0
$f_2 = b + de + c(f+g)$	0	1	1	1	0	0
$f_3 = cf + cg$	0	0	0	0	1	1

Figure 5.1: Boolean matrix for the functions f_1, f_2, and f_3. This is an OR-matrix, in that each row is the OR of the corresponding columns. Note that the columns are not limited to simple AND-terms.

Definition 4: *A* block *of a Boolean matrix B is any subset R of rows and any subset C of columns in B.*

The meaning of a row or a block depends on the operation that relates the row and column expressions in the Boolean matrix.

Definition 5: *After setting each don't-care independently to either 0 or 1, the* meaning of a row *in an OR-matrix is the expression obtained by or-ing together all the column expressions for columns that have a 1 in the row. That is,*

$$\text{meaning_row}(r, C, B) = \bigvee_{c \in C, B_{rc}=1} c \ .$$

Definition 6: *The* meaning of a block *in an OR-matrix is the expression obtained by and-ing together the meanings of each row in the block. The operator should be a true Boolean operator, so that $abab = ab$. That is,*

$$\text{meaning_block}((R, C), B) = \bigwedge_{r \in R} \text{meaning_row}(r, C, B) \ .$$

The definitions for the meaning of rows and blocks in AND and XOR matrices are similar—we just change the operators according to Table 5.1. We require that both operators be commutative and associative, but do not require distributivity.

In order to handle the don't-cares correctly when we define replacement of blocks, we need to define acceptable row expressions:

Definition 7: *Row r_2 is an* acceptable replacement *for row r_1 if for every setting s_2 of the d's in r_2 there exists a setting s_1 of the d's in r_1, such that the meaning of r_1 based on the setting s_1 is the same as the meaning of r_2 based on the setting s_2. Note that the rows do not have to use the same columns.*

	XOR-matrix	AND-matrix	OR-matrix
row from columns	$\oplus$	$\bigwedge$	$\bigvee$
block from rows	$\bigwedge$	$\bigvee$	$\bigwedge$

Table 5.1: Operators for determining the meanings of blocks in Boolean matrices of different types. Each row expression is the first operator applied to the selected column expressions. The meaning of entire block is the second operator applied to the row expressions.

We can similarly define a block to be an acceptable replacement *if for every setting of don't-cares in the new block, there is a setting of the don't-cares in the original block that gives the two blocks the same meaning.*

Finally, we need the definition of a *rectangle*[1]. In Figure 5.1 the first two rows of the matrix have two columns that have 1's in both rows. We call such combinations of rows and columns a *rectangle*:

Definition 8: *We say that a block* (R, C) *of a Boolean matrix* B *is a* rectangle *if for every* $r \in R$ *and* $c \in C$*, we have* $B_{rc} \neq 0$*.*

5.2.2 Replacing rectangles in a Boolean matrix

The main operation in rectangle replacement is to add a new column to a matrix corresponding to some rectangle of the matrix, and to replace 1's in the original rectangle with the 1's in the new column.

The replacement should preserve the meaning of important blocks of the Boolean matrix. There are two particularly interesting cases: each row represents some function we wish to compute, or the entire matrix interpreted as a block is the function which we wish to compute. In this chapter we consider only the first case, which corresponds to common cube extraction in MISII.

Lemma 1 (Replacement): *A rectangle of a matrix can be replaced by adding a new column* C_{new}*, whose associated expression is the meaning of the rectangle, to the matrix. Putting a 1 in* C_{new} *in each row contained in the rectangle and changing the 1's in the*

[1] We prefer the name *full block*, but for compatibility with Brayton's work we have used *rectangle* throughout this chapter.

rectangle to d's, makes any row in the new matrix an acceptable replacement for the same row in the old matrix.

In fact, the meaning of any block in the old matrix containing all the columns of the rectangle can be acceptably replaced by the corresponding block with column C_{new} added in the new matrix.

The proof follows immediately from the definition of the meaning of a block and the definition of acceptable replacement, by setting d's in the rectangle before replacement equal to 1, and setting other d's to match the setting in the new block. Note that the lemma is essentially the correctness proof for MISII's cube extraction algorithm.

Note, that Lemma 1 is for the OR-matrix and AND-matrix only. For the XOR-matrix, we must make sure that each 1 in the matrix is covered an odd number of times. Alternatively, we can modify the definition of a rectangle, so that it is not allowed to contain d's.

Replacing a rectangle in the matrix reduces the number of 1's in the matrix by an amount we call the value of a rectangle:

Definition 9: *The* value *of a rectangle is equal to the difference in the number of 1's in the matrix before and after replacement of the rectangle. Since the replacement of a rectangle results in one new column with a 1 for each row of the rectangle, the value of a rectangle is the same as the number of 1's in the rectangle minus the number of rows in the rectangle.*

As an example, consider the Boolean matrix shown in Figure 5.1. If we replace the rectangle consisting of the first and second row and the second and third column (value=2), we end up with the matrix shown in Figure 5.2. Rectangles that span more than one row and more than one column are particularly useful to replace, as the new column, in this case $b + de$, will then be explicitly used as a shared subexpression for the rows of the rectangle. Note that this technique unfortunately does not recognize the commonality of $c(f + g)$ and $cf + cg$. MISII finds such commonality using kernel extraction, however, this is a fairly expensive technique and is only applied to small expressions. We are still looking for methods to find such common subexpressions at an acceptable cost.

OR-matrix	a	b	de	$c(f+g)$	cf	cg	$b+de$
$f_1 = a + b + de$	1	d	d	0	0	0	1
$f_2 = b + de + c(f+g)$	0	d	d	1	0	0	1
$f_3 = cf + cg$	0	0	0	0	1	1	0

Figure 5.2: Boolean matrix for the functions f_1, f_2, and f_3, after the new column $b + de$ has replaced the columns b and de.

5.2.3 The rectangle replacement problem

In Section 5.2.2 we saw that a rectangle could be taken out of the Boolean matrix and replaced by a block consisting of one column representing the meaning of the original rectangle, and containing the same rows. We define the *rectangle-replacement problem* to be the problem of sequentially replacing rectangles of a Boolean matrix until all the rectangles have value zero or less. This termination condition guarantees that each row has at most one 1 in it, and rows that start out with 1's end up with exactly one 1. The order in which we replace rectangles affects the quality of the resulting representation for the functions. The rectangle-replacement problem is to choose an ordering for the replacement of rectangles that minimizes the predicted area or delay for the circuit. Solving the rectangle-replacement problem also solves the rectangle-covering problem as we have covered all the 1's in the Boolean matrix.

For the rest of this chapter we will focus on replacement problems in which each row of the Boolean matrix represents a function we need to compute (the equivalent of misII's cube extraction). After solving the rectangle-replacement problem, we replace each row expression by the column expression for which the row has a 1. A column expression may contain a row expression as a subexpression, in which further substitution is done. We do the replacement by a depth-first traversal of the DAG from the roots (outputs). Each edge of the final DAG is traversed only once, and so the replacement is quickly done.

5.3 Two-column rectangle replacement and if-then-else DAGs

In this section we will show how we apply the rectangle-replacement problem to minimization using if-then-else DAGs. The technique that will be described corresponds roughly

to cube extraction in MISII.

Section 5.3.1 will describe how we create Boolean matrices from multiply-rooted if-then-else DAGs. Section 5.3.2 will outline our algorithm for solving the rectangle-replacement problem, and Section 5.3.3 gives our new heuristic for choosing the order of rectangle replacement.

5.3.1 Creating Boolean matrices from if-then-else DAGs

An if-then-else triple can directly represent five different Boolean operators: NOT, AND, OR, XOR, and IF. Since we consistently require *systematic negation* to be satisfied, refer to Section 2.4, we have eliminated all NOT-triples, and so all triples can be classified into one of the other four types.

Three of these triples, the AND, OR, and XOR, are associative and commutative, and consequently can represent the same expression in more than one way. By using De Morgan's laws we merge the sets of AND- and OR-expressions, and create two Boolean matrices: the OR-matrix for the combined AND- and OR-expressions, and the XOR-matrix for XOR-expressions.

The OR-matrix and the XOR-matrix are built by traversing the if-then-else DAG from each root and looking for AND-, OR-, and XOR-triples. Whenever we find one, we check to see if any of the children of the triple (its inputs) are triples of the same type, in which case we recursively include them in the row expression we are about to create. The inputs to the expression become columns of the matrix.

For example, in traversing f_3 of Figure 2.4, we would discover that the triple pointed to by f_2 is also an OR-expression and similarly with f_1, thus giving us $a + b + c + d$ as one row in matrix.

The routine *FindInputs* shown in Figure 5.3 is used when finding the inputs of a commutative triple e. The routine determines the immediate inputs of e (the non-constant children of e), and recursively finds the inputs of these. The recursion stops when a child-triple is of different type than its parent. *FindInputs* has two optional arguments, which can

```
FindInputs(e, op, prop, limit)
    if (Operator(e) ≠ op)
        return {e}
    if (Operator(e) = IfOp)
        return {e.i,e.t,e.e}
    if (Property(e,prop) ≥ limit)
        return {e}
    (i1,i2) ← the two inputs to the operator represented by e
    return
        {FindInputs(i1,op,prop,limit) ∪ FindInputs(i2,op,prop,limit)}
```

Figure 5.3: FindInputs takes as argument an if-then-else triple e and the operator *op* we are finding inputs for. The argument *prop* represents an integer property associated with e, which if greater than *limit*, stops the recursion. Both *prop* and *limit* are optional arguments.

be used to stop the recursive traversal at triples that satisfy some constraint. For example, if a triple has a high fanout, it may be beneficial to avoid restructuring the triple by changing its associativity or commutativity, thus we can tell FindInputs to stop recursion at triples with more than n fanouts. The type of constraint is specified by the argument *prop*, which identifies a certain property stored with a triple (like the fanout of a triple). The recursion stops at triples whos property value exceeds or equals the limit *limit*.

5.3.2 Rectangle replacement algorithm

After creating a matrix we apply the rectangle replacement algorithm to it. The algorithm replaces rectangles of a Boolean matrix B sequentially by using two sub-procedures:

select_rectangle, which selects a rectangle based on a heuristic method described in Section 5.3.3,

replace_rectangle, which replaces a rectangle according to the replacement strategy presented in Section 5.2.2.

The algorithm itself is fairly simple:

```
replace_rectangles(B) =
    while (∃ rectangles with value>0 in B) do
    {
        rect = select_rectangle(B)
        replace_rectangle(rect, B)
    }
```

When the algorithm terminates, each row contains exactly one 1 (a row with n 1's contains a rectangle with value $n-1$). We create a new multiply-rooted if-then-else DAG by traversing the old one from the roots, replacing each sub-DAG that corresponds to a row with the column expression for which that row has a 1. We continue the traversal with the children of the column expression, so that all necessary replacements are done in one traversal.

5.3.3 Selecting rectangles

Selecting rectangles for replacement is the most difficult part of rectangle covering [Bra87a]. The main step in rectangle replacement is to add a new column to the Boolean matrix, where the new column is an acceptable replacement for some rectangle of the matrix. Because the if-then-else DAG representation forces n-ary associative operators to be represented as binary trees, we create new columns from exactly two existing columns. This means that we need only look at two-column rectangles, rather than multi-column rectangles where several columns are combined arbitrarily. The two-column rectangle replacement heuristic is formulated as follows:

Replacement method 1 (Two-column rectangle replacement): *As long as there are rectangles containing exactly 2 columns with value>0, replace the rectangle of greatest value. If two or more rectangles have the same value, choose the one in which the new column would have the earliest estimated arrival time.*

By considering only two-column rectangles and only one rectangle at a time, we have reduced the problem of finding rectangles. Also the work associated with finding the right two-column rectangle is considerably less than that of finding the right multi-column rectangle (a *prime* rectangle [Bra87a, BRSW87b]). In a matrix with n columns there are $\binom{n}{2}$ possible two-column rectangles and 2^n possible prime rectangles. Hence, we can afford

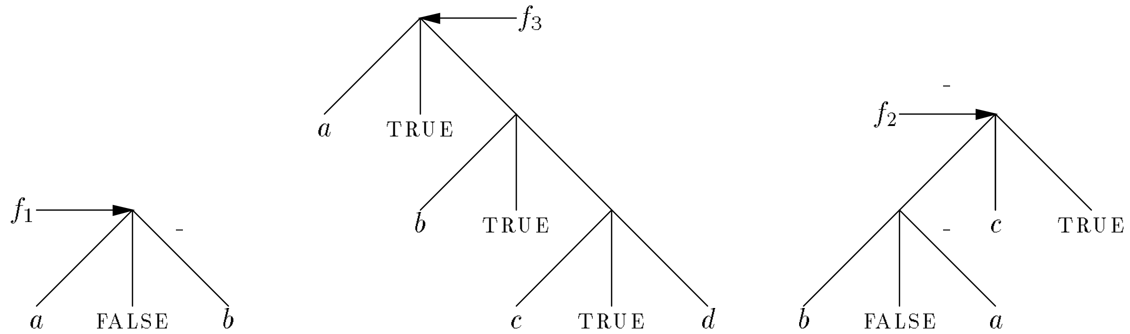

Figure 5.4: Multiply-rooted if-then-else DAG representing the three expressions $f_1 = a + b$, $f_2 = a + b + c$, and $f_3 = a + b + c + d$.

OR-matrix	a	b	c	d
$f_1 = a + b$	1	1	0	0
$f_2 = a + b + c$	1	1	1	0
$f_3 = a + b + c + d$	1	1	1	1

Figure 5.5: Boolean matrix for $f_1 = a + b$, $f_2 = a + b + c$, and $f_3 = a + b + c + d$, showing overlapping rectangles.

to enumerate all two-column rectangles and choose the best, whereas choosing a prime rectangle must resort to a limited enumeration.

As was noted in Section 5.2.2, rectangles spanning more than one row are particularly useful to replace. This is also reflected in the two-column rectangle replacement method, as we specifically choose the rectangle with the highest value (the value of a two-column rectangle of all 1's is equal to the number of rows the rectangle covers). Choosing the two-column rectangle of highest value for replacement also handles overlapping rectangles. To see this, consider the matrix show in Figure 5.5 for the multiply-rooted DAG in Figure 5.4. Replacing the highest valued two-column rectangle results in the matrix shown in Figure 5.6, where $a + b$ is explicitly used by all rows. Continuing to replace the highest valued two-column rectangle of Figure 5.6, results in Figure 5.7, and finally, replacing the last two-column rectangle gives us the matrix shown in Figure 5.8. An if-then-else DAG representation corresponding to the expressions in the last matrix is shown in Figure 2.4.

The primary goal of two-column rectangle replacement is to optimize with respect to area, but if two or more rectangles have the same value, we choose the one that will result

	a	b	c	d	$a+b$
$f_1 = a+b$	d	d	0	0	1
$f_2 = a+b+c$	d	d	1	0	1
$f_3 = a+b+c+d$	d	d	1	1	1

Figure 5.6: Replacing the highest valued two-column rectangle in the matrix of Figure 5.5.

	a	b	c	d	$a+b$	$(a+b)+c$
$f_1 = a+b$	d	d	0	0	1	0
$f_2 = a+b+c$	d	d	d	0	d	1
$f_3 = a+b+c+d$	d	d	d	1	d	1

Figure 5.7: Replacing the highest valued two-column rectangle in the matrix of Figure 5.6.

	a	b	c	d	$a+b$	$(a+b)+c$	$d+((a+b)+c)$
$f_1 = a+b$	d	d	0	0	1	0	0
$f_2 = a+b+c$	d	d	d	0	d	1	0
$f_3 = a+b+c+d$	d	d	d	d	d	d	1

Figure 5.8: Replacing the only remaining rectangle with positive value in the matrix of Figure 5.7.

in the earliest arrival time of the new column expression. We have previously seen that the height of a DAG is a usable delay estimate for the final circuit [Kar89], and so we use height as our arrival time estimator. Using the height of DAGs to break ties results in a primitive form of tree balancing.

5.3.4 Tree balancing

Tree balancing can be carried a little further than in Section 5.3.3. By changing the two-column rectangle replacement method only slightly, we can optimize for delay instead of area:

Replacement method 2 (Two-column rectangle replacement, optimizing for delay): *As long as there are rectangles containing exactly 2 columns with value>0, replace the rectangle in which the new column would have the earliest arrival time. If two or more rectangles result in the same arrival time, choose the one with the greatest value.*

If we use the height of the DAGs as our delay estimate, this replacement method will balance the DAG. It is possible to use a weighted sum of height and value to sacrifice area for delay, or delay for area.

5.3.5 Expanding IF-expressions

In our main algorithm for minimizing a multiply-rooted if-then-else DAG, we first apply rectangle replacement to the XOR-matrix, then we create the OR-matrix and apply rectangle replacement to this matrix. In creating the OR-matrix we have found that it exposes more sharing if we expand IF- and XOR-triples that are inputs to OR- or AND-expressions. An IF-expression is expanded to either $ab + \neg ac$ or $(a+c)(\neg a+b)$ depending on whether it is input to an OR- or AND-expression. However, if the expressions resulting from the expansion are used only for the original IF- or XOR-triple, we have lost the compact IF-expression without gaining more sharing, and we should recover the original IF-expression.

In the matrix, a column expression that is not shared will occur as a column with exactly one 1 (the column is used in exactly one row—it is an unshared column). If a row r contains two or more such columns they will form a rectangle *rect* spanning only the row r. If we replace *rect* before we replace other rectangles of perhaps greater value, we can check to see if any pair of columns in *rect* form an IF-expression when combined.

If we precede OR-matrix covering by replacing rectangles of unshared columns, we have an easy way of recovering exactly those IF-expressions and XOR-expressions that were expanded uselessly. Since no other row expression uses the part of the expanded expression we are free to change it back to an IF- or XOR-triple.

After replacing rectangles containing unshared columns, we can remove all unshared columns, since they will never be part of a rectangle again. This removal will result in a speedup of the rectangle replacement algorithm, since there are fewer columns to consider. By using the described technique we achieved anywhere from 0% to 80% reduction in the number of columns over the benchmark examples we ran. The mean reduction in column count was 36%, which shows that the matrices are generally sparse.

5.4 Results

In this section we show the results of using two-column rectangle replacement on a set of examples from the 1989 International Workshop on Logic Synthesis [Lis88].

In Table 5.2 we present the results of applying two-column rectangle replacement to examples optimized by *LocalFactor* in ITEM (refer to Section 4.2.1). The columns headed with *count* reports the *count* of the optimized if-then-else DAG. The *count* metric is our technology independent area predictor, corresponding roughly to (number of outputs) + (number of literals in factored form) - (number of gates) [Kar89]. The columns headed with *height* reports the height of the optimized if-then-else DAG. Columns 2 and 3 are the results of using only LocalFactor. Columns 4 and 5 are the results of using LocalFactor followed by two-column rectangle replacement optimizing for area. The next two columns are when using two-column rectangle replacement optimizing for delay as described in Section 5.3.4. Finally the last two columns reports the result of applying DMIG [CCD+92] to the networks optimized by LocalFactor. DMIG is a tree-balancing technique that rebalances associative operators to minimize the height of the network. DMIG is guaranteed to give the smallest possible height, but while rebalancing the network it completely ignores other properties of the network, such as keeping area under control.

The results are summarized in Figure 5.9, Figure 5.10, and Figure 5.11. The first figure compares the circuits produced by two-column rectangle replacement with those produced by LocalFactor alone. We plot the ratios of the count measure on the x-axis, and the ratio of the heights on the y-axis. On the average we achieved a 10.2% reduction in count. The height of the network on the average remained unchanged.

Figure 5.10 compares LocalFactor to two-column rectangle replacement optimizing for delay. We see that the tree-balancing scheme used by two-column rectangle replacement actually does well, reducing the height in all but 10 examples where there was no change. On average the height was reduced by 30%, whereas the *count* measure varied by ±20%—but the average *count* didn't change significantly.

	LocalFactor		LocalFactor TwoColumn area		LocalFactor TwoColumn delay		LocalFactor DMIG	
Example	count	height	count	height	count	height	count	height
alu2	404	11	402	11	404	11	412	11
alu4	764	30	737	34	792	24	935	23
apex6	792	17	779	20	862	13	968	13
apex7	245	13	243	14	265	11	313	11
b9	117	10	108	10	112	8	121	8
c8	175	9	164	10	166	7	192	7
cc	67	5	64	5	72	5	76	4
cht	184	3	184	3	184	3	184	3
cm138a	24	5	24	5	25	3	26	3
cm151a	35	9	35	10	36	8	36	8
cm152a	22	3	22	3	22	3	22	3
cm162a	41	7	40	8	42	6	54	6
cm163a	40	6	39	7	40	6	47	6
cm42a	28	3	27	3	28	2	30	2
cm85a	43	6	43	10	47	6	52	6
cmb	40	9	40	11	41	5	43	5
count	143	18	143	18	160	7	242	7
cu	71	7	52	7	57	6	63	6
decod	46	4	42	3	47	3	46	3
example2	313	13	305	13	355	8	431	7
f51m	90	6	88	7	90	6	91	6
frg1	235	21	216	15	216	14	236	14
frg2	1279	15	980	15	1244	10	1426	9
lal	105	12	94	10	102	6	130	6
ldd	121	7	110	8	113	6	128	6
pcle	64	9	64	9	70	6	87	6
pcler8	88	11	88	11	111	7	146	6
pm1	59	6	49	6	51	5	65	5
sct	95	10	70	9	74	6	93	6
term1	231	17	211	21	235	12	268	12
ttt2	247	7	243	8	245	7	259	7
unreg	128	3	128	3	128	3	128	3
vda	1321	39	1083	24	1176	12	1490	12
x1	343	14	300	14	311	10	361	10
x2	52	7	51	7	53	6	61	5
x3	941	18	864	18	908	12	976	11
x4	511	11	391	10	463	7	516	7
z4ml	67	7	65	10	65	7	68	7
TOTAL	9571	408	8588	410	9412	287	10822	280

Table 5.2: The result of two-column rectangle replacement applied to a network optimized by LocalFactor. The table shows the results of optimizing for area (minimizing the *count* metric), and optimizing for delay (minimizing the *height* of the DAG). The table also shows the result of applying DMIG to the same starting point as two-column rectangle replacement optimizing. DMIG ensures minimum height, and hence makes it easy to verify how well two-column rectangle replacement for delay performed.

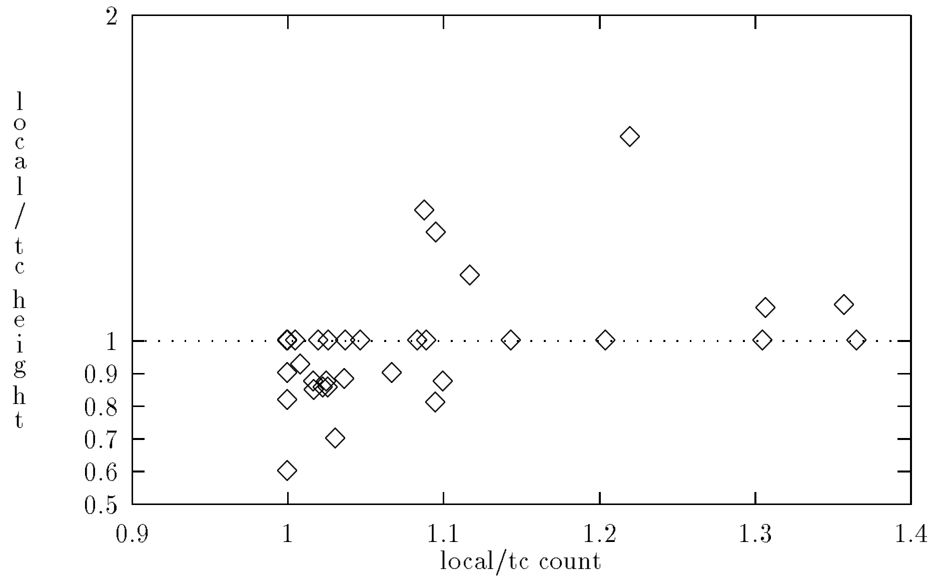

Figure 5.9: Comparison of the count and height of circuits minimized by LocalFactoring (local) and by LocalFactoring plus two-column rectangle replacement optimizing for area (tc).

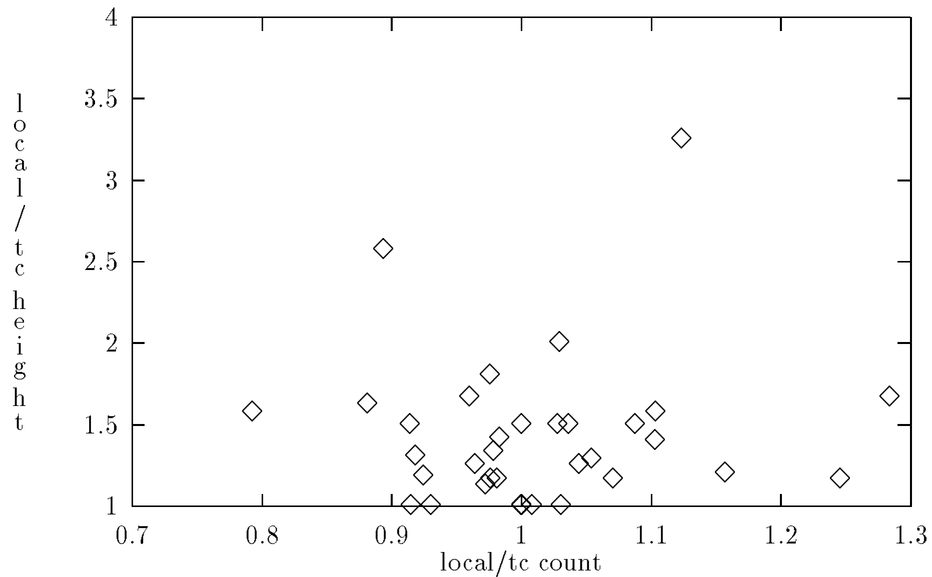

Figure 5.10: Comparison of the count and height of circuits minimized by LocalFactoring (local) and by LocalFactoring plus two-column rectangle replacement optimizing for delay (tc).

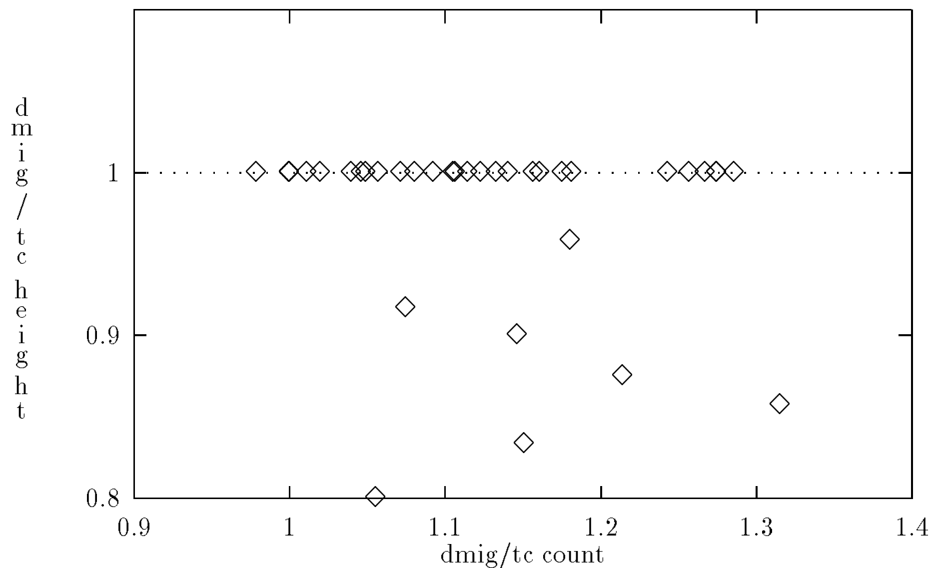

Figure 5.11: Comparison of the count and height of circuits minimized by DMIG (dmig) and by LocalFactoring plus two-column rectangle replacement for delay (tc).

Figure 5.11 compares the circuits produced by two-column rectangle replacement with those produced by DMIG. We see that two-column rectangle replacement performs remarkable well. Totally DMIG only decreases the height by 2.5% when compared to two-column rectangle replacement for delay. For almost all examples DMIG resulted in an increase in the area estimate—overall DMIG increased *count* by 13.5%. When compared to LocalFactor alone DMIG increased the *count* measure in all but 5 examples.

Expanding if-expressions

An experiment was performed to see the effects of expanding if-triples as described in Section 5.3.5. We use the same examples as before and show the results in the scatter diagram in Figure 5.12. The diagram compares two-column rectangle replacement optimizing for area with and without expanding if-expressions. Again we plot the ratios of the count measure on the x-axis, and the ratio of the heights on the y-axis.

We see that expanding if-expressions generally results in a smaller *count*s at the expense of a significant increase in height. Totally the improvement in *count* was no more than

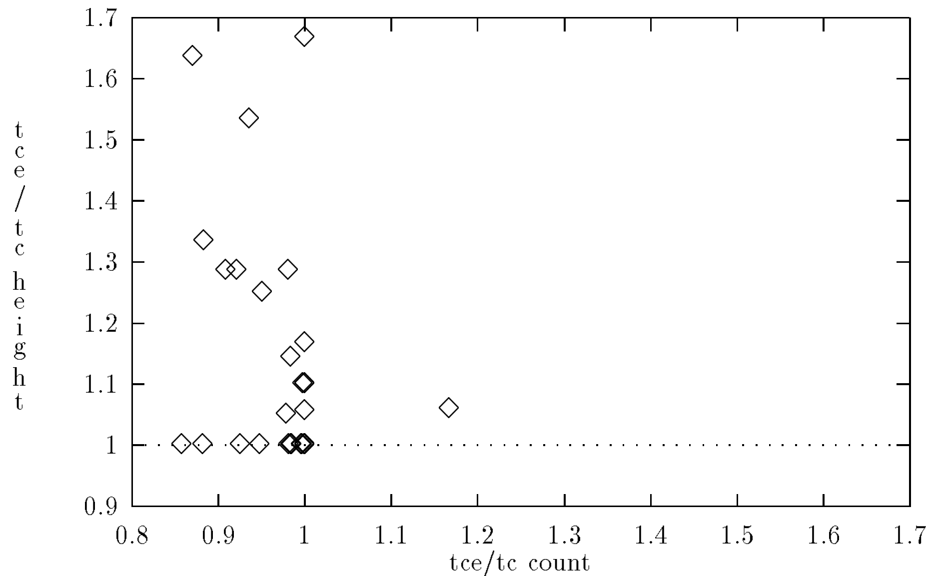

Figure 5.12: Comparison of the count and height of circuits minimized by two-column rectangle replacement with (tce) and without (tc) expanding if-expressions.

2.6%, and one example even resulted in an increase in *count*.

5.5 Conclusions and future work

We have shown that rectangle replacement can be effectively applied to other representations than sum-of-products form, particularly if-then-else DAGs. Our new heuristic for solving the rectangle replacement problem appears to be effective in finding common subexpressions. Two-column rectangle replacement, which was inspired by the structure of if-then-else DAGs, is particular interesting as it combines finding common subexpressions with balancing operator trees.

We are still looking for better ways to select rectangles for replacement. When optimizing for area the optimal replacement may be the one that covers all one's in the Boolean matrix with the fewest replacements.

When optimizing for delay we have the problem that the height of the DAG is not a very good delay estimate, and the heuristic we are using does not always balance the DAG. This however, is solved in the current implementation, since we can ask the program to use any

available cost function as delay estimator. Thus if we have more knowledge of the target technology, we may decide to use a different delay estimator (like `lutheight` for FPGA table-lookup architectures).

6. Variable ordering

6.1 Introduction

The area of variable ordering is a problem specific to binary decision diagrams and if-then-else DAGs. As we saw in Section 3.3 the size of an ordered binary decision diagram (OBDD) or canonical if-then-else DAG is very dependent on the order selected for the input variables—a bad order may result in exponentially many nodes, whereas a good order may result in canonical DAG taht is a very small representation of the logic—indeed some of our best minimization results come from converting to canonical if-then-else DAGs.

In this chapter we investigate techniques for finding variable orders that keep the canonical DAG as small as possible, or make transformations that depend on variable order perform better. First some background is given as to what other researchers have done in the area. The primary focus of this chapter is on two types of heuristics: the *traversal-based heuristics* and the *split-order heuristics*.

Traversal-based heuristics, Section 6.3, all traverse an initial network representation of the Boolean expressions we wish converted to a canonical DAG. The traversal methods we have implemented are all depth-first; they differ in the order in which branches are chosen for traversal and in how multiple variable orders are merged into one total variable order. In Section 6.3 we present a depth-first ordering algorithm, which generalizes the depth-first ordering heuristics of Fujita [FFK88], Malik [MWBS88], Karplus [Kar90] and the new *reconvergent* ordering heuristics introduced in Section 6.3.3.

The split order heuristics, Section 6.4, are not traversal-based, instead they construct the total order one variable at a time, where the next variable in the order is chosen among the remaining variables depending on some cost estimate. After a variable v has been chosen, each expression e used in estimating its cost is "split" into the two expressions $e|_v$ and $e|_{v'}$, which will now constitute the expressions we are trying to find a variable order for.

The main emphasis in Section 6.3 and Section 6.4 is more on the heuristics themselves than on the context in which they are used. In Section 6.5 we finally show how the variable

order may affect other transformations besides conversion to canonical form.

6.2 Background

Variable order in binary decision diagrams (BDDs) has been an issue ever since Bryant presented a canonical form for BDDs called ordered binary decision diagrams (OBDDs) [Bry86]. In Chapter 3 both Bryant's canonical form for BDDs and Karplus' canonical form for if-then-else DAGs were presented, and we saw that the one condition that can influences the size of the canonical DAG is the variable order condition:

Variable order condition for OBDDs: *A total order is imposed on the variables, and the variable at each node in the* OBDD *is earlier in the order than the variables of its children.*

Variable order condition for if-then-else DAGs: *A total order is imposed on the variables, and all the variables in the* **if**-*part must be earlier in the order than all variables in the* **then**- *and* **else**-*parts.*

As was mentioned in Section 3.3, the size of both Bryant's canonical form and Karplus' canonical form can be exponential regardless of the variable order. However, in most cases there exist variable orders that result in polynomial canonical DAGs. Even when exponential blow-up is avoided, the size of a canonical DAGs remains extremely sensitive to the variable order. Figure 6.1 is a scatter diagram showing the result of using two random variable orders on some arbitrarily chosen examples. The plot shows the ratio of the two sizes obtained when converting each example to a canonical if-then-else DAG. We see that the dots are scattered around $y = 1$ with up to a factor of 2 variation in both directions.

If we are using conversion to canonical form as a transformation for reducing size, it is important that we find the order that results in the smallest canonical DAG. An optimum order could be found by considering all permutations of the variables, and Nair & Brand [NB86] present an $O(n!2^n)$ algorithm that does this for binary decision diagrams. Their algorithm considers all permutations, but avoids constructing all of the corresponding binary decision digrams, cutting the running time in half.

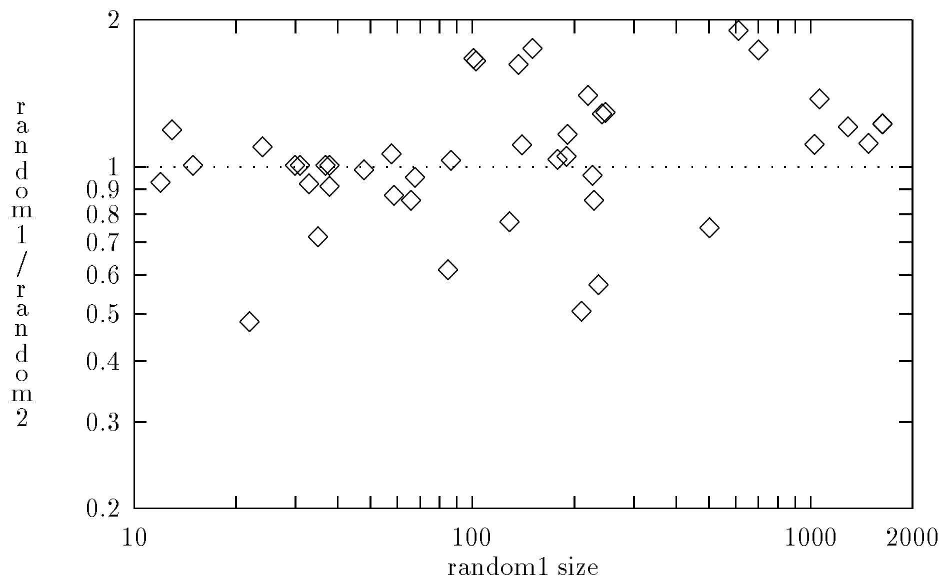

Figure 6.1: The ratio of the sizes of canonical if-then-else DAG when using two random orders. The x-axis is the size of when using the first random order.

Friedman and Supowit [FS87] later came up with an algorithm that finds the optimal order with complexity $O(n^2 3^n)$. They developed a dynamic programming algorithm around the fact that given the optimal order for various sets of k variables ($k \leq n$) the search for the optimal order for sets of $k+1$ variables can be restricted to contain only those sets that were optimal for k variables. That is, if we have 4 sets of k variables such that each set is optimal (the first k levels of the OBDDs constructed from the 4 sets, contain the least possible number of nodes), then the search for optimal sets containing $k+1$ variables are limited to only $4(n-k)$ candidates, namely each of the four optimal k sets with one more variable appended.

Finding an optimal order is only feasible for functions of up to 10 variables, and so we must resort to heuristics for finding a good order. Nair & Brand [NB86] presented some heuristics based on their optimal algorithm, but even then the complexity was exponential in the number of variables. Their main disadvantage is in the representation of binary decision diagrams—all of their heuristics and their optimal algorithm use a complete truth table[1] to

[1] A complete truth table for a function of n variables contains 2^n entries and can be thought of as a binary tree with a unique path from the root to a leaf node for every of the 2^n input combinations.

represent the function they are computing an order for. Hence the space complexity gets out of hand fairly quickly.

Several researchers have presented heuristics for finding variable orders that result in small OBDDs [FFK88, MWBS88, BRKM91, FFM93]. Here the starting point is a Boolean network, which is traversed once in a depth- or breadth-first manner to determine a total order of the variables, which is then used for building the ordered binary decision diagram. The primary advantage of traversal-based methods is that they are fast. A variable order is computed during a single traversal of the initial network, and thus the complexity is $O(n)$ where n is the number of nodes in the network.

Unfortunately, all traversal-based heuristics have one thing in common: no single heuristic is superior. In [BRKM91] various combinations of both breadth- and depth-first heuristics are tried, but one of their conclusions directly contradicts what other researchers have concluded: they conclude that for depth-first traversals, ordering inputs with least (or high) fanout first creates only minor differences in ordering results. Fujita [FFK88, FFM93] presents a depth-first method that orders high fanout inputs before other inputs.

Still, fast ordering heuristics are sometimes more important than spending a lot of time finding the best (or near best) variable order. In Section 6.3 we experiment with two types of depth-first ordering heuristics: one which appends variables to the total order in the order the variables are visited during the traversal, and a new heuristic which uses a merging scheme based on reconvergent fanout for merging intersecting variable orders.

Other approaches for finding a good variable order include

- Sorting the input variables depending on whether they are control or data variables [Bry86]. This method requires knowledge of the circuit function, and so is not feasible in many applications.
- Simulation-based heuristics [BRKM91]. Initially all primary inputs are set to unknown and the state of the circuit is recorded. Next the circuit is simulated by setting each primary input to 0 first and then to 1, counting the number of changes in the simulated

circuit. The variable that is most *controlling* is chosen as the next in the order, and its value is now fixed to either 0 or 1 while the remaining variables are ordered.

- Testability-measure-based heuristics [BRKM91]. Primary inputs are assigned weights based on their observability and sorted in decreasing order of their weights.
- Simulated annealing techniques [MKR92]. Here a heuristically selected order is used initially, and in one annealing step a variable is moved to a randomly chosen position at most ± 5 from its position in the current order. Each time a variable is moved, an *ordered partial decision diagram* OPDD [Ros90, MKR92] is constructed to evaluate the cost of the new order. An OPDD is a sample of the OBDD for the same order. A limit is placed on the largest number of nodes allowed in the OBDD, and if the limit is exceeded, some nodes become undefined. An OPDD tends to represent the top portion of the OBDD containing the shortest paths to TRUE and FALSE.
- Sifting heuristic [Rud93]. This heuristic is a dynamic ordering technique which is applied during the construction of an OBDD. When a certain limit on the number of nodes in the OBDD is reached, the variables are sorted according to their number of occurrences in the OBDD. Then each variable in turn is *sifted* down and up in the OBDD until all positions have been tried, and the best position is chosen for the particular variable.

Our approach to variable ordering

The variable ordering heuristics we present in this chapter have all been implemented in our If-Then-Else Minimizer ITEM. All heuristics fit into a three-step approach, which is also the main optimization loop in ITEM. The three steps (illustrated in Figure 6.2) are

1. Build an if-then-else DAG from the circuit description or a set of equations.
2. Construct a variable order by applying the variable-order heuristic to the constructed DAG or parts of the constructed DAG.
3. Transform the DAG using the obtained variable order. The transformation can be any transformation in ITEM that relies on a good variable order.

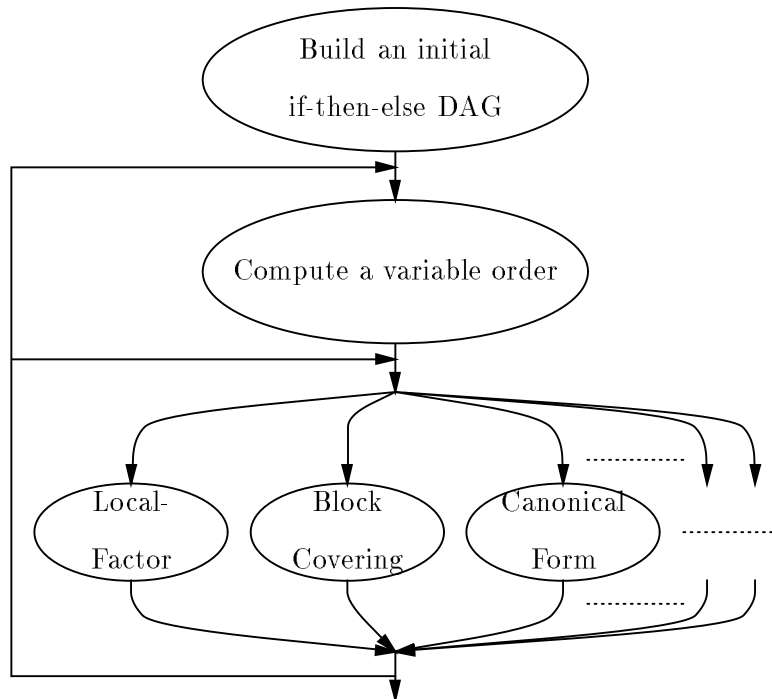

Figure 6.2: General outline of the optimization loop in ITEM. First we construct an if-then-else DAG from a circuit description. We then compute a variable order, and finally apply transformations to the DAG. The variable order computation and the transformations can be iterated to improve results.

The transformation in the third step depends on what objective we are trying to meet. For verification purposes we would transform the DAG to canonical form, but as we will see in Section 6.5, other transformations also rely on good variable orders.

Our three-step approach differs from that of other researchers in that we use an if-then-else DAG as our starting point and that we rely on good variable orders for more than just transformation to canonical form. Most previous work in finding a good order for the variables in ordered binary decision diagrams and if-then-else DAGs has used a Boolean network as starting point. In work done by Malik [MWBS88] the initial DAG is a Boolean network, where nodes are in sum-of-products form.

Our heuristics could be equally well applied to Boolean networks using almost any representation for the logic—we chose if-then-else DAGs because we had significant investment in code for manipulating them. One advantage of the if-then-else DAG approach is that we can iterate the last two steps as illustrated in Figure 6.2 and perhaps get a better order in

```
Order(Node node)
    if (node is a variable)
        return Variable(node)
    <d_1, d_2, ..., d_n> = Sort(Children(node),property)
    o_1 = Order(d_1)
    o_2 = Order(d_2)
    ⋮
    o_n = Order(d_n)
    return MergeOrders(o_1, o_2, ..., o_n)
```

Figure 6.3: General depth-first ordering algorithm.

that way. Iterating in the approach taken by Malik requires a bit more programming effort.

6.3 Depth-first ordering heuristics

Most published work in the area of variable ordering for binary decision diagrams is based on a depth-first traversal of an initial network, followed by constructing the OBDD.

In this section we consider the depth-first ordering heuristics we have implemented in ITEM. The heuristics presented here traverse an initial DAG only once, and hence the complexity of the heuristics are $O(n)$ in the number of nodes.

The second step in our approach to variable ordering, that of constructing a variable order, can be implemented as shown in Figure 6.3. We refer to the algorithm in Figure 6.3 as our general depth-first ordering algorithm, as it generalizes the depth-first ordering heuristics of Fujita [FFK88], Malik [MWBS88], Karplus [Kar90], and the other heuristics presented here.

The algorithm traverses each subDAG (children) of a node (*node*) in an order determined by the routine Sort, which returns a sorted list of the subDAGs of *node*. The list is sorted according to some property associated with each of the subDAGs of *node*. The subDAGs are then traversed in the order returned by Sort, and the variable orders obtained are finally merged into a new order.

By changing the property passed to Sort, we can implement several different depth-first ordering heuristics, differing only in the order in which the branches are traversed.

If the initial DAG is multiply-rooted, as in Figure 2.4, the output nodes (the roots of the DAG) are sorted using the same Sort routine as in Figure 6.3.

The implementation of the algorithm in ITEM is slightly different from the simplified version in Figure 6.3 as it ensures that a node in DAG is visited only once, hence making the algorithm $O(n)$.

6.3.1 Children of a node

In the context of if-then-else DAG we have the option of treating the descendents of a node n in two ways:

Direct children (*DC(n)*) are the subDAGs rooted as the **if**-part, **then**-part, and **else**-part of n.

Commutative children (*CC(n)*) If n is a commutative operator (*Op(n)* is one of XOR, AND, or OR), then the commutative children of n are

$$CC(n) = \{s | s \in DC(n) \wedge Op(s) \neq Op(n)\} \cup CC(s | s \in DC(n) \wedge Op(s) = Op(n))$$

that is, all the inputs to n that are part of the operator n represents.

It should be noted that the formal definition of commutative children corresponds to the algorithm *FindInputs* shown in Figure 5.3.

6.3.2 Incremental ordering heuristics

One class of depth-first ordering heuristics use a very simple MergeOrders procedure, where we first make sure that the subDAG orders do not intersect and then concatenate them to form a total order. This version of MergeOrders would return the order

$$o_1 < (o_2 - o_1) < \cdots < (o_k - o_{k-1} - \cdots - o_1). \tag{6.1}$$

The operator $-$ is left associative in this context, and the result of the subtraction $o_n - o_k$ is the same order as o_n but with the variables of o_k removed.

With this definition of MergeOrders, we can simplify the ordering procedure. We keep a mark with each node once it has been visited, and append a variable to the total order if and only if it has not been marked. The second time we visit a node, we know that all the descendents of that node have also been visited and that the leaf nodes (variables) therefore already appear in the total order.

In our implementation of the incremental ordering heuristic we provide as a parameter the property by which the subDAGs of *node* are sorted, thus giving us as many different heuristics as we have properties. In Section 6.3.4 we provide results using the following four incremental ordering heuristics:

Simple Depth heuristic, where the **if**-part is traversed before the **then**-part, which is traversed before the **else**-part. This heuristic corresponds to the one previously used by Karplus in ITEM [Kar90].

Fanout heuristic, where the subDAGs are sorted in order of decreasing fanout. This heuristic corresponds to the one described by Fujita et al. [FFK88].

Height heuristic, where the subDAGs are sorted in order of decreasing height. This heuristic corresponds to a heuristic described by Malik et al. [MWBS88].

Count heuristic, where the subDAGs are sorted in order of decreasing count. Count is our technology-independent area estimate, corresponding roughly to (number of outputs) + (number of literals in factored form) - (number of gates) [Kar89].

Simple depth heuristic

The simple depth ordering heuristic is the simplest *incremental* heuristic: it traverses the DAG by first traversing the **if**-part followed by the **then**-part and finally the **else**-part. This heuristic is only applicable to if-then-else DAGs and binary decision diagrams. It is easy to see that if applied to a canonical DAG the heuristic will produce the same order as the one used to create the canonical DAG.

The very simple heuristic is particularly effective because of the way ITEM constructs the initial multiply-rooted if-then-else DAG from a circuit description [Kar89]. When constructing an if-then-else DAG from a set of terms (a sum-of-products expression) the terms are factored using a technique similar to simple literal factoring [Bra87a]. This technique will move variables appearing in many terms into **if**-parts of an if-then-else triple. For example, in constructing an if-then-else DAG E for a set of terms T, the first variable v_1 is factored out by grouping together those terms that don't use v_1 (T_d), those that use $\neg v_1$ (T_0), and those that use v_1 (T_1). Then v_1 is stripped off the terms of each group and the routine is applied recursively to get the three expressions E_d, E_0, and E_1. The expression E is constructed as (**if** (**if** v **then** E_1 **else** E_0) **then** TRUE **else** E_d).

When using the simple depth heuristic on a DAG constructed in this way, v_1 will be ordered before the variables in E_1 and E_0, thus we may be able to preserve the factorization in the canonical form.

Fanout heuristic

Fujita et al. [FFK88] used a simple *incremental* depth-first heuristic to determine the variable order. Their starting point was a Boolean network, where the input nets of a node were traversed in order of decreasing fanout.

Fujita et al. justified their heuristic with a theorem, which assures that whenever a network is composed of only AND, OR, and NOT gates and has only one input or gate with a fanout of more than one, then the best order is acquired by a depth-first traversal of the network, but a sub-net with a fanout of more than one should be traversed first.

This theorem is easily related to cube- and kernel factorization of sum-of-products expressions [Bra87a]. If a cube or kernel appears in more than one term the expression can be factored by using the cube as the factor. That is, the expression $abc + (ab)'d$ can be factored into $xc + x'd$ with $x = ab$, and in an if-then-else DAG this factorization can be directly represented as (**if** (**if** a **then** b **else** FALSE) **then** c **else** d), which is in canonical form.

The main problem is of course that a network rarely satisfies the requirement of a fanout of one for all but one of its nodes, and Fujita used a generalized version of the theorem where the requirement is dropped.

We have implemented a *fanout* heuristic similar to the heuristic used by Fujita et al. [FFK88], except that the starting point is an if-then-else DAG. It makes a depth-first traversal of the DAG traversing the subDAGs of a node in order of decreasing fanout.

Height heuristic

Malik et al. [MWBS88] used a strategy similar to Fujita, but they argued that the inputs of a node should be traversed in order of decreasing transitive fanin DAG height. The transitive fanin DAG of a node n consists of n, all the transitive fanins of n (nodes that can be reached through some path from n), and the edges between these nodes. The height of a transitive fanin DAG is the maximum distance from any of its nodes to a primary input.

Malik justified their heuristic intuitively by pointing out a similarity between the intermediate nodes in a Boolean network and the nodes in an OBDD. At a given level in an OBDD the nodes encode information about the variables seen so far. The intermediate nodes are used in subsequent levels to compute the value of the function. In the Boolean network the function of a node n is encoded by the means of the transitive fanin DAGs, suggesting a depth-first traversal of each input DAG of node n in order to compute them before the node n. The order in which to traverse the input DAGs should be in decreasing order of height, thus computing the most compute-intensive nodes first. Berman [Ber91] later related the order of traversal in Malik's heuristic to the problem of register allocation, saying that the height of a fanin DAG can be taken as a rough estimate to the number of registers required to evaluate the function represented by the fanin DAG. Again this is intuitively clear—at any given level in an OBDD the nodes correspond to the amount of information we need in order to compute the rest of the function. Thus if we compute the most *difficult* subDAG s first (requiring k registers) and if none of the variables in s are used in other subDAGs then after computing s we only need one register to hold the value of s thus freeing $k-1$

registers to compute the less difficult subDAGs.

The height heuristic we have implemented is similar to the heuristic used by Malik et al. It traverses the subDAGs of a node in order of decreasing height.

Count heuristic

The count heuristic is a slightly modified version of the height heuristic. Instead of traversing the subDAGs in order of decreasing height, it traverses in order of decreasing count, where count is our technology-independent area estimate. The use of *count* to sort the subDAGs is inspired by Berman's paper of relating ordering heuristics to register allocation [Ber91]. Berman states that Malik's height heuristic is an approximation to the optimal register allocation technique as described in [SU70]. By choosing the highest subDAG first Malik et al. has achieved a rough estimate of the number of registers required to evaluate the logic.

Intuitively we thought that the area estimate of a subDAG would be a better measure for the number of registers, and thus came up with the *count* heuristic.

6.3.3 Reconvergent ordering heuristics

Another class of depth-first ordering heuristics generalized by the general ordering algorithm in Figure 6.3 is the *reconvergent ordering heuristics*. This class of heuristics was introduced in [SK93]. In the reconvergent ordering heuristics the subDAG orders are not made disjoint before merging, and therefore MergeOrders must resolve inconsistencies.

In Figure 6.4 we show how we resolve inconsistencies when merging three intersecting orders (the merging can be generalized to n orders). The variable orders o_1, o_2, o_3 are obtained from traversing the subDAGs returned by Sort in Figure 6.3. Intersecting variables are ordered according to the order of the lowest numbered subDAG order, that is, the variables intersecting all three orders will appear first in the merged order, and they will appear in the same order as in o_1.

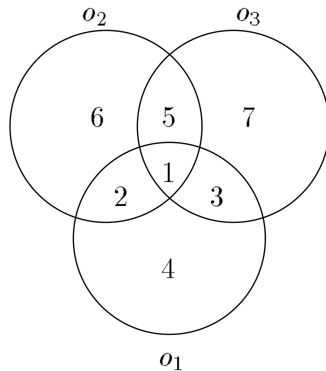

Figure 6.4: Merging of the orders for the three subDAGs of an if-then-else node in reconvergent ordering heuristics. The numbers indicate the order the variables will have in the merged order.

The merging in the reconvergent ordering heuristics makes shared variables come before other non-shared variables in the total order. Having shared variables early in the order increases the likelihood that these variables will appear in an **if**-part in the canonical form, rather than duplicated in the **then**- and **else**-parts. For example, consider the expression $abc + d(a' + b')$ represented as shown in Figure 6.5 (a). After traversing the subDAGs of the root node we may have obtained the order $b < c < a$ for the **if**-part and $d < a < b$ for the **else**-part. When merging these two orders we would obtain the final variable order $b < a < c < d$ and in canonical form this would result in the representation (**if** (**if** b **then** a **else** FALSE) **then** c **else** d) shown in Figure 6.5 (b), thus moving the shared variables to the **if**-part and achieving the desired factorization.

It is easy to verify that none of the incremental ordering heuristics would have obtained the best order with a starting point as shown in Figure 6.5 (a).

One can think of variations to the merging strategy, which would make just as much sense as the one depicted in Figure 6.4:

- Count the number of subDAGs each variable occurs in. Then order the most frequently occurring variables first, while matching the order they have in the lowest numbered subDAG.

We have experimented with variants of the merging strategy, but none were found to

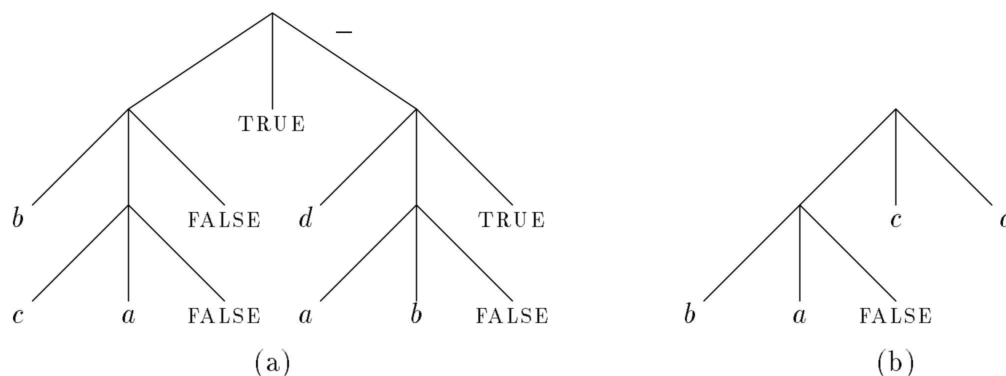

Figure 6.5: (a) An if-then-else DAG representation for the expression $abc+d(a'+b')$. When traversing the **if**-part of the root node we get the order $b < c < a$, for the **else**-part we get the order $d < a < b$. Reconvergent merging results in the order $b < a < c < d$ and (b) shows the same expression in canonical form with respect to this order.

be superior. As will become clear from the result section, all of the depth-first ordering heuristics have the same problem: none of them are universally superior. It is possible to come up of with an unlimited number of variations each of which will perform excellent on a few examples, but very poorly on other examples.

In the result section, the merging strategy shown in Figure 6.4 is the one used.

The reconvergent ordering heuristics are easily related to the fanout heuristic (Section 6.3.2), where the subDAGs are visited in order of decreasing fanout. Since a subDAG with a fanout of more than one is a shared subDAG, the variables of the shared subDAG will come before the variables of the other subDAGs in the total order. The *reconvergent* ordering heuristics do not rely on the fanout to detect shared variables—they always detect shared variables and order them before other variables. The relation to the incremental height, count, and simple depth heuristic is not that obvious.

In the result section we use the reconvergent ordering heuristics corresponding to all the incremental heuristics, that is, we sort the subDAGs using the same cost functions. This gives us:

Reconvergent Simple Depth, where the subDAGs are just the **if**-, **then**-, and **else**-part.

Reconvergent Fanout, where the subDAGs are sorted in order of decreasing fanout.

Reconvergent Height, where the subDAGs are sorted in order of decreasing height.

Reconvergent Count, where the subDAGs are sorted in order of decreasing count.

Again, since the cost function is just passed as an argument to the ordering heuristic, we are not limited to only these four variations of the reconvergent ordering heuristics.

6.3.4 Results

In this section we present results of applying the depth-first ordering heuristics to examples from the benchmark set for the 1989 International Workshop on Logic Synthesis [Lis88]. We provide results both for canonical if-then-else DAGs and ordered binary decision diagrams.

For all examples, our primary goal is to compute a single variable order that works well for all the primary outputs of the network. To do this, we pass all output expressions to the ordering heuristic at once and then convert all the outputs to canonical form using the order returned by the heuristic.

The numbers reported in the tables are nodes in the canonical representation. For canonical if-then-else DAGs we use `size`, which counts the number of if-then-else triples plus distinct variables in the DAG. For ordered binary decision diagrams we use `siscount`, which produces the same counts as `_bdd_size` in SIS, that is, it counts the number of different nodes in the OBDD. It should be noted that `size` can be applied to non-canonical if-then-else DAGs without changing its definition. Similarly, `siscount` can be applied to unordered binary decision diagrams without changing its definition.

We first compare the heuristics on a set of small examples. Table 6.1 shows the number of nodes (using `size`) and the height for canonical if-then-else DAGs. In Table 6.2 we report the same measures (using `siscount`) when converting to ordered binary decision diagrams. In both tables the first column is the name of the example, the next four columns are the 4 incremental depth-first ordering heuristics. Column five through eight are the four reconvergent depth-first ordering heuristics. The column label SIS is the result of running the same examples through SIS using the command `_bdd_create -o dfs`. We modified SIS

to report the variable order it computed, and then used this order in ITEM. Finally, the column labeled *best* is the best (smallest number of nodes) of all the previous columns.

From the tables we see that the most significant difference between canonical if-then-else DAGs and ordered binary decision diagrams is the height, where canonical if-then-else DAGs are on the average 15% lower than OBDDs. In verification the height is not important, but in logic minimization the height is a reasonable estimate for delay, and here if-then-else DAGs have a clear advantages over binary decision diagrams.

From the results we can see that the *reconvergent count* heuristic is the best overall depth-first heuristic. In general the reconvergent ordering heuristics are superior to their incremental counterparts. The *reconvergent count* heuristic is on the average 11% better than the best incremental ordering heuristic.

The bad performance of the *incremental height* heuristic is due to one example, `frg1`, where it is approximately 40 times worse than the best. This illustrates that generally no single traversal-based heuristic is particular efficient, and trying combinations of several will almost always improve the results [BRKM91].

In Table 6.3 and Table 6.4 we compare the depth-first ordering heuristics on examples from the ISCAS benchmarks for testing [Lis88]. The results here are much more mixed and for several of the examples we ran out of memory (entries marked with *oom*) when converting to canonical form. It is interesting to observe that the difference in size and in height between OBDDs and canonical if-then-else DAGs is much less significant for these examples. In fact there are several orderings for which the OBDD is less than the corresponding canonical if-then-else DAG for the same ordering.

In Table 6.5 and Table 6.6 we summarize the results of Table 6.1 and Table 6.2. We compare the totals and extract how many times a given heuristic finds a best depth-first order. A best depth-first order is an order that results in the canonical if-then-else DAG sizes shown as *best* in Table 6.1 and 6.2. The row *Total* is copied directly from the corresponding table, and shows the sum of the nodes over all the examples for the given depth-first heuristic, when converting to canonical from. The rows *Best found* indicates how many

Canonical If-Then-Else DAGs										
	Incremental				Reconvergent					
Example	Simple	Fanout	Height	Count	Simple	Fanout	Height	Count	SIS	best
alu2	153:9	181:9	220:9	175:9	*152:9*	181:9	165:9	165:9	257:9	152:9
alu4	464:12	356:12	472:13	347:13	*302:11*	356:12	687:13	465:13	1197:13	302:11
apex6	838:18	693:19	970:12	1435:17	831:18	*689:19*	1135:16	1120:16	1031:14	689:19
apex7	246:12	345:12	265:12	245:10	241:12	347:12	393:17	*238:17*	473:15	238:17
b9	169:10	168:9	174:9	186:10	165:10	173:9	168:9	*164:9*	178:10	164:9
c8	165:10	153:10	120:9	120:8	199:10	213:10	*102:9*	*102:9*	121:8	102:9
cc	68:5	87:6	64:5	63:5	68:5	87:6	64:5	63:5	*60:4*	60:4
cht	137:4	137:4	145:5	140:4	133:4	*127:4*	145:5	141:4	135:4	127:4
cm138a	42:5	42:5	*22:5*	*22:5*	42:5	42:5	*22:5*	*22:5*	*22:5*	22:5
cm151a	42:7	31:7	31:7	31:7	*28:5*	*28:5*	*28:5*	*28:5*	42:7	28:5
cm152a	*18:3*	*18:3*	36:5	36:5	*18:3*	*18:3*	26:4	26:4	26:4	18:3
cm162a	61:7	51:7	*43:7*	*43:7*	56:7	44:6	*43:7*	44:6	64:8	43:7
cm163a	48:5	*40:7*	45:6	45:6	49:5	*40:7*	45:6	*40:7*	46:5	40:7
cm42a	26:3	24:3	*22:3*	*22:3*	26:3	24:3	*22:3*	*22:3*	*22:3*	22:3
cm85a	*45:9*	*45:9*	47:9	49:9	*45:9*	*45:9*	47:9	47:9	48:9	45:9
cmb	47:11	47:11	*46:11*	49:11	47:11	47:11	*46:11*	49:11	49:11	46:11
count	141:17	*98:18*	131:18	131:18	141:17	*98:18*	131:18	131:18	132:17	98:18
cu	*48:7*	*48:7*	77:10	71:7	*48:7*	*48:7*	59:8	55:8	70:9	48:7
decod	49:4	49:4	*35:4*	*35:4*	49:4	49:4	*35:4*	*35:4*	*35:4*	35:4
example2	364:13	368:12	386:12	*334:12*	364:13	344:12	368:12	390:10	364:12	334:12
f51m	*57:7*	67:7	72:7	72:7	*57:7*	69:7	72:7	72:7	66:7	57:7
frg1	269:22	264:22	5576:21	941:20	178:19	178:19	146:19	*144:19*	227:18	144:19
frg2	2244:19	3126:18	1038:17	941:15	2245:19	3114:18	982:14	937:14	*862:13*	862:13
lal	112:11	*101:10*	103:11	103:11	112:11	105:10	103:11	103:11	116:10	101:10
ldd	*76:6*	*76:6*	96:8	96:8	*76:6*	*76:6*	80:7	80:7	*76:6*	76:6
pcle	82:10	82:10	76:10	76:10	82:10	82:10	*66:10*	*66:10*	73:10	66:10
pcler8	134:10	140:10	143:10	143:10	134:10	140:10	127:11	127:11	*113:10*	113:10
pm1	59:7	59:7	71:8	58:7	59:7	59:7	71:8	61:8	*57:7*	57:7
sct	135:10	119:10	*79:6*	*79:6*	131:10	119:10	*79:6*	*79:6*	119:8	79:6
tcon	41:2	41:2	27:2	27:2	41:2	41:2	27:2	27:2	*25:1*	25:1
term1	248:13	245:13	301:14	300:14	245:13	245:13	*209:14*	*209:14*	376:14	209:14
ttt2	179:12	155:11	171:11	171:9	176:11	*149:10*	168:11	163:9	165:11	149:10
unreg	99:3	*87:4*	135:4	135:4	99:3	*87:4*	135:4	135:4	138:4	87:4
vda	1291:13	563:11	*541:12*	*541:12*	1291:13	563:11	585:12	585:12	570:11	541:12
x1	828:19	734:19	931:20	977:22	816:19	*726:19*	926:20	958:21	1072:19	726:19
x2	49:6	49:6	47:6	45:6	49:6	49:6	46:6	45:6	*44:6*	44:6
x3	938:18	855:18	952:17	1133:18	929:18	*847:18*	1354:19	1174:18	1314:13	847:18
x4	407:9	*396:9*	410:12	413:9	398:9	*396:9*	404:11	429:9	423:13	396:9
z4ml	32:6	32:6	41:6	31:6	41:6	41:6	41:6	*27:6*	42:6	27:6
Nodes	10451	10172	14161	9861	10163	10086	9352	8768	10250	7219
Height	374	373	373	366	367	366	373	366	358	360

Table 6.1: The columns headed with name of heuristics report the total number of nodes and height after converting all the primary outputs to canonical if-then-else DAGs. The two last rows report the total number of nodes and the total height over all examples. We used the cost function `size` to count the number of nodes in an if-then-else DAG. For each example we use italics to indicate which heuristics found the best variable order.

Ordered Binary Decision Diagrams										
	Incremental				Reconvergent					
Example	Simple	Fanout	Height	Count	Simple	Fanout	Height	Count	SIS	best
alu2	*176:9*	180:9	215:9	188:9	177:9	180:9	180:9	180:9	250:9	176:9
alu4	499:13	*446:13*	555:13	505:13	466:13	*446:13*	734:13	547:13	1168:13	446:13
apex6	1000:19	888:19	1066:19	1744:19	991:19	*886:19*	1295:19	1350:19	1346:19	886:19
apex7	483:18	596:18	*453:18*	502:20	484:18	597:18	551:18	537:19	667:18	453:18
b9	185:13	210:12	182:12	192:12	180:12	219:12	*177:12*	184:12	214:12	177:12
c8	207:12	225:12	150:12	140:12	219:11	236:11	*98:9*	113:9	144:12	98:9
cc	82:5	96:6	*78:5*	79:5	82:5	96:6	*78:5*	79:5	*78:5*	78:5
cht	138:4	138:4	145:5	139:4	134:4	*131:4*	145:5	140:4	135:4	131:4
cm138a	*19:5*	*19:5*	39:5	39:5	*19:5*	*19:5*	39:5	39:5	39:5	19:5
cm151a	32:7	23:7	25:7	25:7	*22:5*	*22:5*	*22:5*	*22:5*	32:7	22:5
cm152a	*17:3*	*17:3*	26:5	26:5	*17:3*	*17:3*	18:4	18:4	18:4	17:3
cm162a	65:9	55:9	*45:9*	*45:9*	63:9	51:9	*45:9*	51:9	60:9	45:9
cm163a	52:7	*41:7*	46:8	46:8	49:7	*41:7*	46:8	*41:7*	47:7	41:7
cm42a	*21:3*	*21:3*	23:3	23:3	*21:3*	*21:3*	25:3	25:3	23:3	21:3
cm85a	*40:9*	*40:9*	42:9	42:9	*40:9*	*40:9*	42:9	42:9	42:9	40:9
cmb	*37:11*	*37:11*	*37:11*	*37:11*	*37:11*	*37:11*	*37:11*	*37:11*	*37:11*	37:11
count	233:18	*202:18*	234:19	234:19	233:18	*202:18*	234:19	234:19	235:18	202:18
cu	64:9	64:9	51:10	*41:10*	63:9	64:9	56:9	50:9	68:9	41:10
decod	*33:4*	*33:4*	39:4	39:4	*33:4*	*33:4*	39:4	39:4	39:4	33:4
example2	565:13	*514:13*	746:14	636:13	568:13	531:13	772:13	770:13	754:13	514:13
f51m	*48:7*	69:7	75:7	75:7	53:7	70:7	75:7	75:7	67:7	48:7
frg1	504:23	504:23	5540:24	946:24	168:22	168:22	130:22	*128:22*	186:23	128:22
frg2	3786:21	5055:19	2440:20	2457:20	3787:21	5073:19	2680:20	2448:20	*2103:20*	2103:20
lal	104:12	102:12	105:12	105:12	104:12	105:12	105:12	105:12	*101:12*	101:12
ldd	81:7	81:7	86:8	86:8	81:7	81:7	*74:7*	*74:7*	81:7	74:7
pcle	101:10	*54:10*	96:11	96:11	101:10	*54:10*	71:10	71:10	96:10	54:10
pcler8	189:11	154:11	155:11	155:11	189:11	154:11	*133:11*	*133:11*	141:11	133:11
pm1	53:8	53:8	56:8	51:8	53:8	53:8	56:8	*49:8*	59:8	49:8
sct	142:13	142:12	*75:12*	*75:12*	139:13	142:12	*75:12*	*75:12*	103:12	75:12
tcon	42:2	42:2	27:2	27:2	42:2	42:2	27:2	27:2	*26:1*	26:1
term1	487:19	487:19	386:19	391:19	415:19	415:19	210:19	*208:19*	616:19	208:19
ttt2	199:13	175:13	213:13	189:13	206:13	183:13	215:13	*157:13*	175:13	157:13
unreg	99:3	*86:4*	135:4	135:4	99:3	*86:4*	135:4	135:4	136:5	86:4
vda	1256:13	528:12	*522:12*	*522:12*	1256:13	528:12	555:12	555:12	533:12	522:12
x1	1089:22	941:22	1084:22	1045:22	1106:22	*935:22*	1080:22	1071:22	1165:22	935:22
x2	*41:6*	43:6	44:7	*41:6*	*41:6*	43:6	45:6	*41:6*	43:6	41:6
x3	1112:19	1062:19	*1019:19*	1226:21	1100:19	1052:19	1577:19	1099:19	1492:21	1019:19
x4	908:14	900:14	608:14	755:14	900:14	900:14	603:14	767:14	*596:14*	596:14
z4ml	46:6	46:6	39:6	38:6	39:6	39:6	39:6	*36:6*	38:6	36:6
Nodes	14235	14369	16902	13137	13777	13992	12518	11752	13153	9868
Height	420	417	428	429	415	413	415	414	420	411

Table 6.2: The columns headed with name of heuristics report the total number of nodes and height after converting all the primary outputs to ordered binary decision diagrams. The two last rows report the total number of nodes and the total height over all examples. We used the cost function `siscount` to count the number of nodes in an OBDD. For each example we use italics to indicate which heuristics found the best variable order.

Canonical If-Then-Else DAGs										
	Incremental				Reconvergent					
Example	Simple	Fanout	Height	Count	Simple	Fanout	Height	Count	SIS	best
C432	29069:35	*26686:35*	31870:35	31870:35	29069:35	*26686:35*	31864:35	31864:35	29069:35	26686:35
C499	53889:40	53657:40	*40733:40*	*40733:40*	51585:40	51353:40	62273:40	62273:40	43458:40	40733:40
C880	oom	oom	5124:33	4953:34	oom	oom	7708:37	8462:37	*4477:33*	4477:33
C1355	53889:4	53657:4	*40733:40*	*40733:40*	51585:40	51353:40	62273:40	62273:40	43458:40	40733:40
C1908	18331:30	18353:30	18684:30	15806:30	16693:30	17439:30	18829:30	*12782:30*	12825:30	12782:30
C2670	oom	oom	oom	oom	oom	oom	oom	oom	oom	—
C3540	oom	oom	oom	oom	oom	oom	oom	oom	oom	—
C5315	oom	oom	29810:53	oom	oom	oom	oom	oom	*22692:50*	22692:50

Table 6.3: The columns headed with name of heuristics report the total number of nodes and height after converting all the primary outputs to canonical if-then-else DAGs. The two last rows report the total number of nodes and the total height over all examples. We used the cost function `size` to count the number of nodes in an if-then-else DAG. For each example we use italics to indicate which heuristics found the best variable order.

Ordered Binary Decision Diagrams										
	Incremental				Reconvergent					
Example	Simple	Fanout	Height	Count	Simple	Fanout	Height	Count	SIS	best
C432	31179:35	*29866:35*	32279:35	32279:35	31179:35	*29866:35*	32273:35	32273:35	31179:35	29866:35
C499	53867:40	53635:40	*40659:40*	*40659:40*	51563:40	51331:40	61511:40	61511:40	44299:40	40659:40
C880	oom	oom	*6845:41*	6863:41	oom	oom	9635:40	9643:40	7724:41	6845:41
C1355	53867:40	53635:40	*40659:40*	*40659:40*	51563:40	51331:40	61511:40	61511:40	44299:40	40659:40
C1908	17759:30	19403:30	19499:30	16787:30	16628:30	18760:30	19609:30	*12203:30*	12713:30	12203:30
C2670	oom	oom	oom	oom	oom	oom	oom	oom	oom	—
C3540	oom	oom	oom	oom	oom	oom	oom	oom	oom	—
C5315	oom	oom	36151:53	oom	oom	oom	oom	oom	*26066:53*	26066:53

Table 6.4: The columns headed with name of heuristics report the total number of nodes and height after converting all the primary outputs to ordered binary decision diagrams. The two last rows report the total number of nodes and the total height over all examples. We used the cost function `siscount` to count the number of nodes in an OBDD. For each example we use italics to indicate which heuristics found the best variable order.

times a given heuristic and class of heuristic finds a best depth-first order.

Among the depth-first ordering heuristics the reconvergent merging strategy is superior to the incremental merging strategy finding a best order in 30 versus 18 examples when converting to canonical if-then-else DAGs and 28 versus 22 when converting to OBDDs. *Reconvergent count* appears to give the best overall result, but otherwise no one depth-first heuristic appears to be significantly better than another. The large total size for *incremental height* is due to one bad order.

The results presented in this section show that none of the depth-first heuristics described

Canonical If-Then-Else DAGs									
Class	Incremental				Reconvergent				
Heuristic	Simple	Fanout	Height	Count	Simple	Fanout	Height	Count	SIS
Total Nodes	10541	10172	14161	9861	10163	10086	9352	8768	10250
Best found	5	9	7	7	10 8	14	10	13	10
Best found	18				30				

Table 6.5: This table compares depth-first heuristics only. *Simple* is short for *Simple Depth*, refer to Section 6.3. For each depth-first heuristic the table shows total size when converting to canonical if-then-else DAG over all 39 examples we used. *Best found* indicates the number of examples for which a given heuristic and class of heuristic finds a depth-first order that results in canonical if-then-else DAG sizes as reported under *best* in Table 6.1.

Ordered Binary Decision Diagrams									
Class	Incremental				Reconvergent				
Heuristic	Simple	Fanout	Height	Count	Simple	Fanout	Height	Count	SIS
Total Nodes	14235	14369	16902	13137	13777	13992	12518	11752	13153
Best found	9	11	7	6	6 8	14	9	12	6
Best found	22				28				

Table 6.6: Same as Table 6.5 except that we here compare OBDDs.

here work well as a general ordering heuristic. All the heuristics are far from obtaining the best total, and all fail badly on a few examples. For reasonable sized examples it may be worthwhile to try a number of different heuristics and pick the best result as we have done.

6.4 SplitOrder heuristic

We now turn to the *SplitOrder* heuristic, which is especially well suited for finding good variable orders for ordered binary decision diagrams. Unlike the depth-first ordering heuristics, SplitOrder is not traversal-based—instead it constructs an OBDD top-down one level at a time.

Even though the heuristic is targeted towards finding good orders for OBDDs we found that a good order for an OBDD is rarely a bad order for an if-then-else DAG. We compare SplitOrder to several depth-first ordering heuristics and show that when converting to OBDDs SplitOrder averages 25% fewer nodes than taking the best of 8 depth-first techniques (36%

```
SplitOrder(exprSet,support,order)
    if (support is empty) return
    best_exprSet = ∅
    best_variable = 0
    foreach v ∈ support
        E' = ∅
        foreach e ∈ exprSet
            E' = E' ∪{e|_v, e|_v'}
        if (Cost(E') < Cost(best_exprSet))
            best_exprSet = E'
            best_variable = v
    support = support −{best_variable}
    append best_variable to the end of order
    SplitOrder(best_exprSet,support,order)
```

Figure 6.6: Split order algorithm. The argument *exprSet* is the set of expressions to find a variable order for. Initially this is the set of expressions we want to transform using a variable-order-dependent transformation. The argument *support* is the set of variables still to order. Initially this set contains the primary inputs of the expressions in *exprSet*. The argument *order* is the accumulated total order.

better than the best single depth-first technique). We do not see the same size reduction when converting to canonical if-then-else DAGs.

The split order heuristic constructs the total order one variable at a time, where the next variable in the order is chosen among the remaining variables depending on some cost estimate. The cost of a variable v is based on how much it would cost to "split" each expression e, in the set of expressions for which we are computing an order, into the two expressions $e|_v$ (e given that v is true) and $e|_{v'}$ (e given that v is false). After a variable v has been chosen, the set of $e|_v$ and $e|_{v'}$ expressions will be the new set of expressions we need to compute an order for.

Figure 6.6 shows a pseudo-code version of the split order heuristic. The algorithm is passed a set of expressions *exprSet*, which is used in determining the cost of a variable. Initially *exprSet* is the set of expressions we wish to transform using some transformation that relies on a good variable order (such as conversion to canonical form). The variables that still need to be ordered are passed to SplitOrder as the set *support*. Initially, *support* contains all variables of the expressions in *exprSet*. The list *order* is the accumulated total

order.

In each recursive call to SplitOrder, the algorithm finds the "best" variable in *support*, and appends it to the total order *order*. The best variable v is determined based on how much it would cost to factor each expression e in *exprSet* into $(ve|_v + v'e|_{v'})$. For all expressions in *exprSet*, the expressions $e|_v$ and $e|_{v'}$ are collected in a new expression set E', and if E' is the best seen so far it is stored together with v. On the next recursive call to SplitOrder, *exprSet* is the best of all of the E' we constructed. To avoid recomputing $e|_v$ and $e|_{v'}$ for the same expression e and variable v at each recursive call to SplitOrder, we cache both $e|_v$ and $e|_{v'}$ with e.

6.4.1 Choosing the cost function

Testing the effect of all variables one by one is what makes the SplitOrder targeted towards finding good orders for binary decision diagrams rather than targeted to if-then-else DAGs. If SplitOrder is called with one expression e then at the kth recursive call the kth variable in the order is chosen, and this variable will occur at level k in the OBDD for that expression. This argument is carried a little further in the following lemma, which shows that SplitOrder actually creates the OBDD (*obdd*) for e as it goes along:

Lemma 2: *At level k the number of nodes in an obdd is less than or equal to the cardinality of the expression set* exprSet *passed as argument to SplitOrder in the kth recursive call. In fact, the set of Boolean functions represented by the expressions in* exprSet *is exactly the same as the set of Boolean functions represented by the nodes at level k in the obdd.*

Proof: We start with second part and prove it by induction. Initially *exprSet* contains the one expression we wish to convert to an OBDD. Since the conversion to an OBDD does not change the function represented and since the OBDD at level 0 has one node our base case is ok. Assume that the hypothesis holds for level $k-1$; we will now show that it also holds for level k. At the $k-1^{th}$ recursive call the functions represented by the expressions in *exprSet* are the same as the functions represented by the nodes at level $k-1$ in the obdd. After the $k-1^{th}$ pass through the algorithm, the $k-1^{th}$ variable v_{k-1} in the total order has

been chosen and each expression e in *exprSet* has been factored into $v_{k-1}e|_{v_{k-1}} + v'_{k-1}e|_{v'_{k-1}}$ and the expressions $e|_{v_{k-1}}$ and $e|_{v'_{k-1}}$ have been inserted in *eSet*. Since $e|_{v_{k-1}}$ and $e|_{v'_{k-1}}$ are exactly the functions that appear at level k in *obdd* and since *eSet* is passed to SplitOrder as *exprSet* in the kth calls we have proven the second part of the lemma is true for k provided it is true for $k-1$. This together with the base case proves the second part of the lemma.

The first part of the lemma follows directly from the second part. Since the expressions in *exprSet* are not in canonical form we can not be sure that no two expressions represent the same function, and thus we can not change the *less than or equal* to *equal*.

□

We can use Lemma 2 to select an appropriate cost function for evaluating the cost of a variable. The cost function *Cost* can be chosen so as to optimize for different objectives. Our main goal is to find an order that results in a canonical form that is as small as possible. We currently try to meet two objectives:

Primary objective: Try to minimize the number of nodes in the canonical form by minimizing the number of nodes in the representation of E', the set of expressions remaining after the next variable has been chosen.

Secondary objective: Keep the number of nodes at any given level in the OBDD as small as possible. From Lemma 2 we see that this objective can be met by having "SecCost" return the cardinality of the expression set passed as argument.

To solve these objectives we use a weighted sum of the number of nodes in E' and the cardinality of E' as

$$Cost(E') \quad = \quad a_1\,Nodes(E') + a_2\,|E'|,$$

where *Nodes(E')* counts the number of different nodes in the expressions with roots in E'.

Making a_2 much larger than a_1 would tend to minimize the cardinality of *exprSet* at each recursive call to SplitOrder, while ignoring the size of the expressions rooted in *exprSet*. Even though |*exprSet*| is an estimate on the number of nodes at a given level in the OBDD, it generally is a bad idea to ignore the size of the expressions (see Lemma 3), and we have found that keeping $a_1 = 1$ and $a_2 = 1$ works well overall.

6.4.2 Computing $e|_v$ and $e|_{v'}$

The cost function used by SplitOrder is strongly dependent on the accuracy of the *given that* operator when it computes $e|_v$ and $e|_{v'}$ for each expression e in *exprSet*.

The expression $e|_v$ is computed in one traversal of e. If all the operator does is to traverse the DAG and set v to TRUE or FALSE, but otherwise doesn't change the structure of the DAG, then the cost function will be very inaccurate, and the resulting order will be, as shown in Lemma 3, in increasing order of occurrences in the original set of expressions passed to SplitOrder.

Lemma 3: *Without simplifications by the* given-that *operator, SplitOrder returns a variable order, where the variables are sorted in increasing order of number of occurrences in the expressions in* exprSet*.*

Proof: First observe that the number of nodes in the expressions rooted at *exprSet* remains unchanged at each recursive call to SplitOrder, since *given that* only replaces variables with constants. This simplifies the cost function used by SplitOrder to

$$Cost(E') \quad = \quad constant + a_2\,|E'|.$$

Let s be the subset of expressions in *exprSet* that contains variable v. Each expression $e \in s$ will be split into two expressions $e|_v$ and $e|_{v'}$ and the number of expressions in E' will then be

$$|E'| = 2\,|s| + |exprSet| - |s| = |exprSet| + |s|.$$

Since the cost function will choose the split variable that results in the smallest cost it will then choose the variable that minimizes $|s|$ and this is the variable that occurs in fewest of the expressions in *exprSet*.

□

If, on the other hand, the operator transforms $e|_v$ making use of all the reductions that occur when a variable is set to either TRUE or FALSE, then the cost function can be very accurate. If $e|_v$ and $e|_{v'}$ are converted to canonical form and identical expressions are

merged, then, according to Lemma 2, the cardinality of E' ($|E'|$) will always be exactly the number of nodes at level k in the OBDD.

Obviously, doing no reductions at all and converting to canonical form are two extremes—the first is useless and the second is computationally too expensive. We instead allow *given that* to propagate any constant time simplifications that occur as a result of replacing v with TRUE:

No-two-constant: Triples in which both the **then**- and **else**-parts point to TRUE (with either plus or minus labels) are replaced by an appropriately labeled pointer to the **if**-part or to TRUE.

Weak distinct-cases: Triples whose **then**- and **else**-parts are the same pointer are replaced with just this pointer.

No-constant-if: Triples whose **if**-part points to TRUE (FALSE) are replaced by the **then** (**else**-part).

Distinct-if: Triples whose **if**-part is the same pointer as the **then**- and/or **else**-part are replaced with a triple in which the **then**- and/or **else**-part is an appropriately labeled pointer to TRUE.

Since the *given that* operation does not convert to canonical form, the expression set E' may contain different representations of the same Boolean function. However, the set does not necessarily grow exponentially with the number of variables ordered, since duplicate expressions (same pointers) are eliminated.

6.4.3 Complexity

In this subsection we will analyze the complexity of the split order heuristic. In one pass through the algorithm in Figure 6.6 the running time is bounded by the product of the cardinality of *support* and the time it takes to compute *given that* on all the expressions in *exprSet*. Since each recursive call decreases the size of *support* by one, the running time of the algorithm is:

$$\sum_{i=0}^{i=n-1} i\, G(i),$$

where n is the number of variables and $G(i)$ is the complexity of computing the *given that* operation in the ith recursion through SplitOrder.

Let us examine $G(i)$ a little closer by first looking at how the *given that* operator computes $e|_v$ and $e|_{v'}$. The simple and most straight forward *given that* makes one traversal of the expression e to compute $e|v$, replacing each occurrence of v with TRUE and propagating constant time simplifications. Clearly this has complexity $O(m)$, where m is the number of nodes in the DAG representation of e. We can easily modify the *given that* operator to compute both $e|v$ and $e|_{v'}$ in the same traversal, giving us the algorithm *GivenThat* shown in Figure 6.7.

Each time *GivenThat* visits a node it checks to see if we have already computed the *given that* for the node, in which case we just return the cached versions. Otherwise, we compute *given that* for the node, and cache the results with the node for future reference. The *IfThenElse* operator is called with three expressions (i,t, and e), and returns an expression that is logically equivalent to the expression (**if** i **then** t **else** e). The complexity of *IfThenElse* varies depending on what conditions the returned expressions should satisfy, see Section 2.4 and 3.3. The conditions SplitOrder requires are shown in Section 6.4.2 and can all be carried out in constant time, making *GivenThat* an $O(m)$ algorithm, where m is the number of nodes in the DAG representation of e.

With the version of *GivenThat* in Figure 6.7, we can see that the complexity of $G(i)$ is still $O(m)$, with m being the number of nodes in the multiply-rooted DAG with roots in *exprSet*.

We now need to determine m for the multiply-rooted DAG with roots in *exprSet*. To prove that m is exponential in the worst case is easy, here we just need to consider SplitOrder called with one expression e for which we wish to compute an order, and then use the following two lemmas:

Lemma 4: SplitOrder *constructs all the nodes in the* OBDD *representation for* e *with respect to the order returned by* SplitOrder.

```
GivenThat(Expr e, Variable v, var Expr egv, var Expr egnv)
    if (v ∉ Support(e))
        egv ← egnv ← e
        return
    if (e is marked)
        egv ← cached value
        egnv ← cached value
        return
    mark e as visited
    if (e = v or e = v')
        egv ← (e = v) ? TRUE : FALSE
        egnv ← (e = v) ? FALSE : TRUE
        cache both egv and egnv with e
        return
    <e_i, e_t, e_e> ← <if-part(e),then-part(e),else-part(e)>
    Expr egv_i,egnv_i
    GivenThat(e_i,v,egv_i,egnv_i)
    Expr egv_t,egnv_t
    GivenThat(e_t,v,egv_t,egnv_t)
    Expr egv_e,egnv_e
    GivenThat(e_e,v,egv_e,egnv_e)
    egv ← IfThenElse(egv_i,egv_t,egv_e)
    egnv ← IfThenElse(egnv_i,egnv_t,egnv_e)
    cache both egv and egnv with e
    return
```

Figure 6.7: Computing $e|_v$ and $e|_{v'}$ in one traversal of e. IfThenElse(i,t,e) returns an expression representing the function (**if** i **then** t **else** e)

.

Proof: Follows from Lemma 2.

□

Lemma 5: *There exist functions for which the number of nodes in a corresponding* OBDD *is exponential in the number of input variables regardless of chosen variable order.*

Proof: Bryant [Bry91, Theorem 4] showed that any OBDD representation for the Boolean function representing the middle output of an integer multiplier for word size n is exponential in the number of nodes.

□

Lemmas 4 and 5 prove that the complexity of the algorithm shown in Figure 6.6 is ex-

ponential in the worst-case, since it must construct all the nodes in the OBDD corresponding to the order it computes. Lemma 2 guarantees that at the kth recursive call to *SplitOrder* the cardinality of *exprSet* is greater than or equal to the number of nodes at level k in the OBDD representation for e, thus we are ensured that $|exprSet|$ grows exponentially in the worst-case, meaning that m, the number of nodes in the multiply-rooted DAG with roots in *exprSet*, also grows exponentially in the worst-case.

With the exception of integer multiplication, most Boolean functions in digital logic design applications have reasonable size OBDDs. Thus the average complexity of SplitOrder is more likely to be a function of how well GivenThat manages to propagate simplifications once variables are replaced with constants. From Lemma 3 we can see that using no simplifications could result in an almost doubling of *exprSet* at each recursive call to SplitOrder, which in turn would make SplitOrder exponential even when the resulting OBDD is polynomial.

6.4.4 Inefficiencies in computing given that

As mentioned in the proof of Lemma 2 we can not assure that $|exprSet|$ is equal to the number of nodes at the corresponding level in the OBDD representation of an expressions e, since the expressions in *exprSet* are not in canonical form. Thus, many of the expressions passed to *SplitOrder* as *exprSet* may be different representations of the same Boolean function.

To see how bad things were we took the ISCAS benchmark `C432` and passed all 7 outputs to SplitOrder. We then counted the number of expressions in *exprSet* ($|exprSet|$) at each recursive call to SplitOrder and compared it with the minimum achievable. The minimum achievable can be computed by converting each expression in *exprSet* to canonical form and removing duplicate expressions (expressions with identical pointers) from the set.

In Table 6.7 we show the results. The first column shows the number of variables in *support* at each recursive call to SplitOrder. The second column shows the cardinality of *exprSet* in each call. We see that even though *exprSet* does not grow exponentially it still

gets quite big. The set sizes are even more disappointing when compared to the minimum achievable shown in column 3 in Table 6.7.

The huge difference between the actual cardinalities encountered by SplitOrder and the minimum achievable, indicates that the *given that* operator is not very good at propagating the consequences of setting a variable to TRUE and FALSE. For example, when 7 variables remain in *order*, SplitOrder had to call GivenThat for 10207 expressions which represented only 46 different Boolean functions.

The main problem is illustrated in Figure 6.8. Assume the function f to be represented as shown in Figure 6.8 (a), and assume that we want to compute $f|_a$. The result as it would be returned by *GivenThat* is shown in Figure 6.8 (b). The *IfThenElse* operator has no easy way of detecting that the **if**-part of some triple doesn't appear in the **then**- and **else**-parts (the only condition that would detect this situation is the *Variable order condition*). Since *GivenThat* repeatedly sets variables to TRUE and FALSE the situation appearing in Figure 6.8 (b) is very common.

The transformation that would be required to go from Figure 6.8 (b) – (c) is an $O(m)$ operation if the DAG is a tree (m is then the number of nodes in the tree), but when nodes have multiple fanout the transformation can not be carried out in one traversal of the DAG.

It is still an open problem to improve the *given that* operator so that it will propagate simplifications more efficiently.

6.4.5 Results

The result of applying the SplitOrder heuristic to examples from the benchmark set for the 1989 International WorkShop on Logic Synthesis [Lis88] are summarized in Table 6.8. The examples used in this table are the same as those used in Table 6.1 and Table 6.2. The results reported in this table use the same starting point as the depth-first ordering heuristics, that is, we apply SplitOrder without doing any higher level transformations that normally result in more structured networks and hence better orders.

Remaining variables	$\lvert exprSet \rvert$	Minimum
36	7	7
35	12	12
34	22	21
33	34	33
32	55	53
31	84	77
30	123	103
29	130	91
28	111	62
27	217	117
26	323	156
25	324	124
24	279	62
23	541	119
22	541	110
21	732	93
20	652	62
19	1242	121
18	1242	114
17	1844	101
16	2772	77
15	3894	117
14	3866	106
13	5752	85
12	3141	47
11	5216	75
10	5045	68
9	7668	55
8	6419	32
7	10207	46
6	9633	41
5	9616	44
4	42	21
3	29	13
2	9	5
1	2	2

Table 6.7: Computing a variable order for the 7 primary output of `C432` using SplitOrder as shown Figure 6.6. First column indicates how many variables there are in *support* at each recursive call to SplitOrder. Second column is the cardinality of *exprSet* at that recursive call to SplitOrder. The third column shows the minimum achievable cardinalities of *exprSet* which are computed by converting the expressions in *exprSet* to canonical form and removing duplicate pointers.

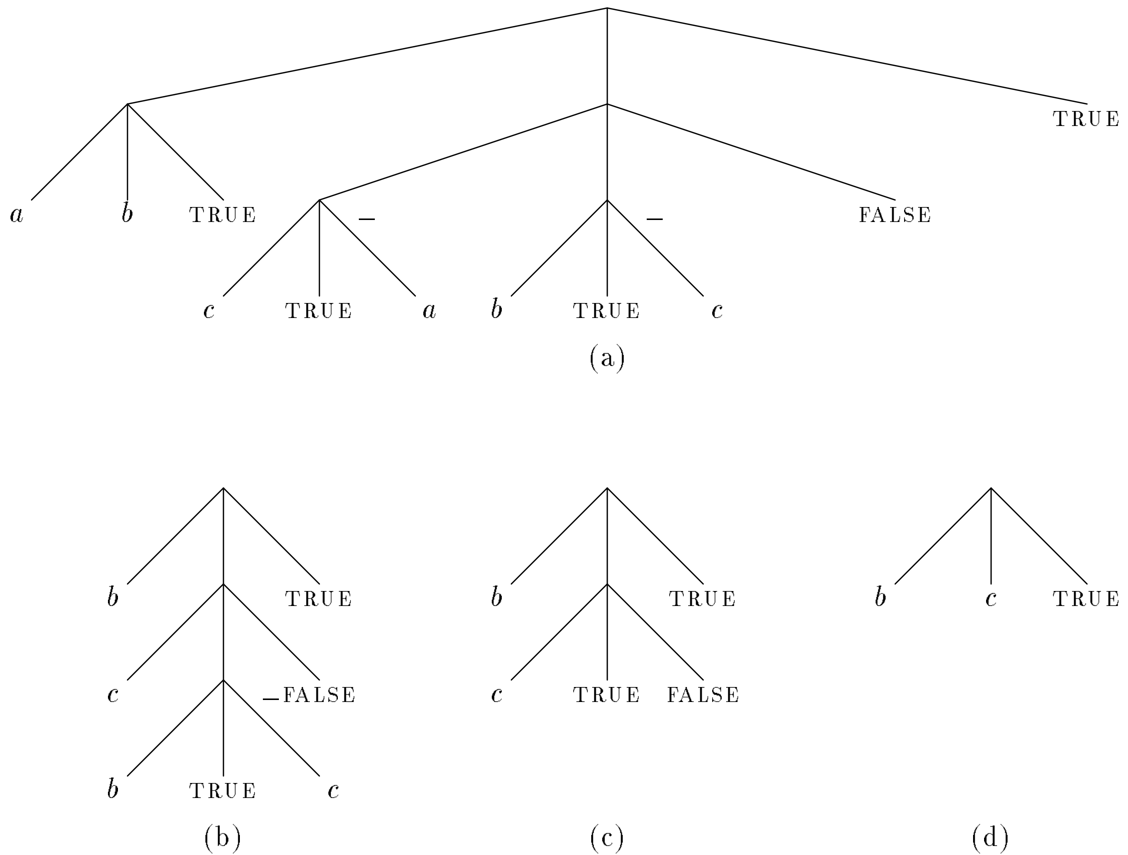

Figure 6.8: (a): An if-then-else DAG representation for the function $f = ab' + (c + a')(b + c')$. (b): The result of computing $f|_a$, notice how *GivenThat* fails to propagate changes in **if**-branches to **then** and **else**-branches. (c): Propagating the changes of the **if**-branches. (d): Final result, obtained directly from (c) but not from (b) by using the *No-Two-Constants* condition.

All numbers in the table report the number of nodes and the height of the canonical if-then-else DAG and ordered binary decision diagram. For canonical if-then-else DAGs we use `size` to count nodes and for OBDDs we use `siscount` (refer to Section 6.3.4). The first column is the name of the benchmark example. Columns 2 through 4 report results when converting to OBDDs. Column 2, *depth*, contains the sizes and height of the OBDDs using the order acquired by the best depth-first ordering heuristic; these numbers are taken from column *best* of Table 6.2. Column 3, *split*, reports the same measures using the variable order obtained by SplitOrder. Column 4, *rev(s)*, reports the size and height of OBDDs when using the reverse of the order obtained by the SplitOrder heuristic. Finally, columns 5

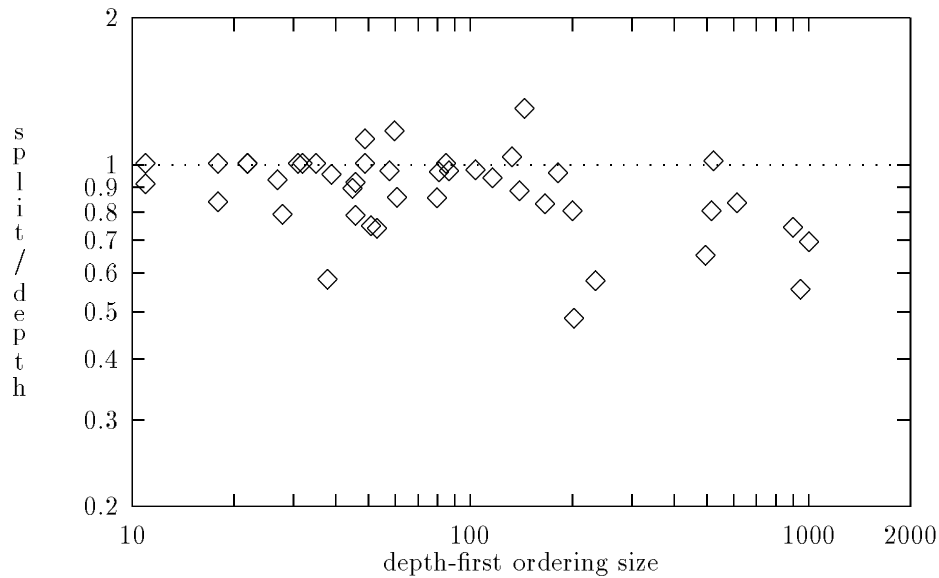

Figure 6.9: The size of OBDD when using SplitOrder divided by the size when using the best of the depth-first ordering heuristics. Squares below 1 indicate that SplitOrder wins.

through 7 are the results when converting to a canonical if-then-else DAG. All columns are summed up, and the total can be taken as an estimate of the area needed to implement all examples using one particular strategy.

The data from Table 6.8 is summarized in Figure 6.9 and Figure 6.10, which shows scatter diagrams for all 39 examples comparing the depth-first heuristic to the split order heuristic when converting to respectively an OBDD and a canonical if-then-else DAG.

When converting to an OBDD, the new split order heuristic wins in all but a few examples. In 39 examples SplitOrder won in 28 examples and lost in only 6 examples. In the winning examples the size reductions were anywhere from 2% to 59% averaging 20%. In the 6 losses the worst size increase was 52% and the average was 18%. Over all 39 examples, SplitOrder resulted in a 25% decrease in size of OBDDs. Comparing SplitOrder to the best single depth-first heuristic (*reconvergent count*) we see that SplitOrder results in a 37% reduction in number of nodes.

When converting to a canonical if-then-else DAG, the new split order heuristics wins in 16 examples and looses in 20 examples. In the winning examples the size reductions were

Example	OBDD depth	OBDD split	OBDD rev(s)	canonical ited depth	canonical ited split	canonical ited rev(s)
alu2	176:9	176:9	267:9	*152:9*	160:9	277:9
alu4	446:13	*370:13*	941:13	*302:11*	352:13	958:13
apex6	886:19	*643:19*	2126:23	689:19	*550:14*	2295:19
apex7	453:18	*319:18*	1053:23	*238:17*	292:13	528:18
b9	177:12	*148:12*	203:13	164:9	*153:9*	173:11
c8	*98:9*	106:10	248:12	102:9	*96:10*	179:11
cc	78:5	*65:5*	52:6	60:4	*54:5*	90:6
cht	*131:4*	136:4	118:5	*127:4*	139:4	189:4
cm138a	19:5	19:5	39:5	*22:5*	42:5	22:5
cm151a	22:5	*20:6*	74:9	28:5	*25:6*	72:9
cm152a	17:3	*17:3*	384:10	18:3	18:3	384:10
cm162a	45:9	*39:9*	56:10	43:7	*40:7*	58:7
cm163a	41:7	*37:7*	42:8	40:7	*36:7*	51:8
cm42a	21:3	21:3	25:3	*22:3*	26:3	22:3
cm85a	40:9	*31:9*	41:9	45:9	*37:6*	45:8
cmb	37:11	*35:11*	37:11	*46:11*	49:11	48:11
count	202:18	*82:18*	205:19	*98:18*	203:18	130:18
cu	*41:10*	58:9	79:12	*48:7*	57:8	98:11
decod	33:4	33:4	47:4	*35:4*	49:4	35:4
example2	514:13	*355:13*	746:15	*334:12*	537:12	494:12
f51m	48:7	*40:7*	75:7	57:7	*49:6*	68:7
frg1	*128:22*	195:22	199:24	*144:19*	195:17	220:20
frg2	2103:20	*1420:22*	2563:23	*862:13*	1512:15	1247:18
lal	101:12	*98:12*	117:12	101:10	*100:10*	97:11
ldd	*74:7*	75:7	111:8	*76:6*	78:7	99:8
pcle	54:10	*44:10*	114:11	*66:10*	81:10	82:10
pcler8	133:11	*131:11*	152:12	113:10	*111:10*	148:11
pm1	*49:8*	51:8	46:8	57:7	*52:6*	61:7
sct	75:12	*58:13*	95:12	*79:6*	86:11	68:9
tcon	26:1	26:1	35:2	25:1	25:1	41:2
term1	208:19	*113:19*	200:19	209:14	*120:10*	140:13
ttt2	157:13	*139:13*	219:13	*149:10*	163:9	184:11
unreg	86:4	*83:3*	121:5	87:4	*84:3*	196:5
vda	522:12	*511:12*	5082:16	*541:12*	547:12	5089:16
x1	935:22	*481:21*	612:22	726:19	*468:18*	605:19
x2	41:6	*37:6*	75:9	*44:6*	45:6	71:7
x3	1019:19	*700:19*	1419:22	847:18	*687:14*	1407:15
x4	596:14	*434:14*	570:14	*396:9*	480:11	377:12
z4ml	36:6	*25:6*	28:6	27:6	27:6	22:6
TOTAL	9868:411	7371:413	18616:464	7219:360	7825:349	16370:404

Table 6.8: The size and height after applying the various ordering heuristics to an initial if-then-else DAG. The first column is the name of the example. The next three columns are size (`siscount`) and height when converting the network to an OBDD using the best of the depth-first ordering heuristics, the split order heuristic, and the reverse of the split order. The last three columns are the size (`size`) and height when converting to a canonical if-then-else DAG. The row *Total* shows the total size of implementing all examples using the given heuristic.

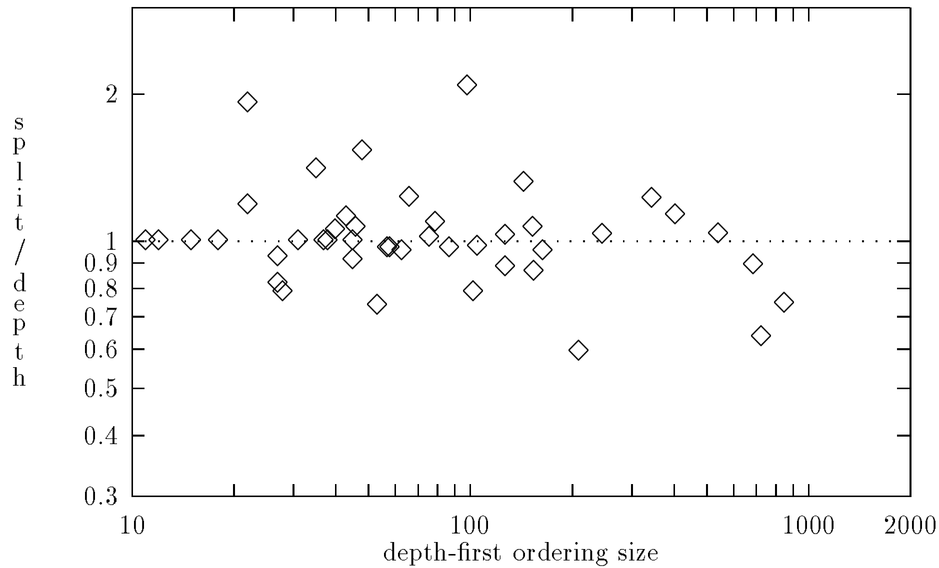

Figure 6.10: The size of canonical if-then-else DAG when using SplitOrder divided by the size when using the best of the depth-first ordering heuristics.

anywhere from 1% to 43% averaging 13%. In the losing examples the worst size increase was 107% (`count`) averaging 28%. Over all 39 examples, SplitOrder resulted in a 8.3% increase in size of if-then-else DAGs. Comparing SplitOrder to the best single depth-first heuristic for if-then-else DAGs (*reconvergent count*) we see that SplitOrder results in a 11% reduction in number of nodes, which still makes SplitOrder the single best variable ordering heuristic for if-then-else DAGs.

The results are impressive when converting to OBDDs, but quite disappointing when converting to if-then-else DAGs. However, it does appear as if canonical if-then-else DAGs are less sensitive to the variable order than OBDDs, hence the room for improvement may be greater for OBDDs than for canonical if-then-else DAGs. It comes as no surprise that SplitOrder performs best on OBDDs, since it actually constructs the OBDD and not the canonical if-then-else DAG as it goes along. The "split" variable is based on how much it would cost to factor an expression e into $ve|_v + v'e|_{v'}$, and this is exactly the function represented by one of the nodes with label v in the OBDD.

If the split order heuristic constructed the canonical if-then-else DAG as it went along, the choice of a "split" wouldn't be a simple variable, but a general expression. If the expression

we are converting to a canonical if-then-else DAG has n variables and the order is decided for k of these, then the search would be among 2^{n-k} variable sets instead of $n-k$ variables.

Even though there is no improvement when converting to canonical if-then-else DAG using SplitOrder (remember though, that SplitOrder was compared against 9 depth-first ordering heuristics), it still seems that a good variable order for an OBDD is generally also a good variable order for a canonical if-then-else DAG. In the cases where the blow-up is significant, as in examples `cm138a` and `count`, part of the reason is that canonical if-then-else DAG are left-associative, whereas OBDDs are right-associative, so a simple reversal of the order will improve the result. However, as can be seen from the table it is not generally the case that reversing the order for an OBDD will result in a good order for an if-then-else DAG, in fact it most often results in a worse order. It is interesting to observe, that the only examples where a reversal of the order helps are those for which SplitOrder lost over the depth-first order.

Applying SplitOrder to optimized examples

The previously presented results all used unoptimized circuits as starting point. Normally optimized circuits are more structured than unoptimized and hence we would expect SplitOrder to find better orderings if applied to former. Here we present results using SplitOrder on the same examples as before, but prior to running SplitOrder we optimize each example with SIS using the standard script coming with SIS.

Table 6.8 compares the results of applying SplitOrder to the SIS optimized examples and the non-optimized examples. The columns marked *noopt* are the result of SplitOrder applied to the non-optimized examples—these columns are copied from Table 6.8. The columns marked *opt* are the result of applying SplitOrder to the SIS optimized examples. The table also includes columns *best*, which shows the best over all the results presented in this chapter. The columns best are used later when we compare the sizes of OBDDs against those of canonical if-then-else DAGs.

Example	OBDD noopt	OBDD opt	OBDD best	canonical ited noopt	canonical ited opt	canonical ited best
alu2	176:9	179:9	176:9	160:9	183:9	152:9
alu4	370:13	441:13	370:13	352:13	446:13	302:11
apex6	643:19	741:19	643:19	550:14	690:14	550:14
apex7	319:18	299:18	299:18	292:13	243:15	238:17
b9	148:12	148:12	148:12	153:9	169:10	153:9
c8	106:10	98:10	98:10	96:10	85:10	85:10
cc	65:5	62:6	52:6	54:5	70:6	54:5
cht	136:4	127:3	118:5	139:4	122:3	122:3
cm138a	19:5	19:5	19:5	42:5	42:5	22:5
cm151a	20:6	19:4	19:4	25:6	21:4	21:4
cm152a	17:3	17:3	17:3	18:3	18:3	18:3
cm162a	39:9	37:9	37:9	40:7	55:8	40:7
cm163a	37:7	27:7	27:7	36:7	42:7	36:7
cm42a	21:3	21:3	21:3	26:3	26:3	22:3
cm85a	31:9	30:9	30:9	37:6	37:6	37:6
cmb	35:11	32:11	32:11	49:11	49:11	46:11
count	82:18	82:18	82:18	203:18	203:18	98:18
cu	58:9	59:9	41:10	57:8	61:7	48:7
decod	33:4	33:4	33:4	49:4	49:4	35:4
example2	355:13	496:13	355:13	537:12	367:11	334:12
f51m	40:7	40:7	40:7	49:6	46:7	46:7
frg1	195:22	130:22	128:22	195:17	148:18	144:19
frg2	1420:22	1142:20	1142:20	1512:15	1273:15	862:13
lal	98:12	91:12	91:12	100:10	105:10	97:11
ldd	75:7	77:7	74:7	78:7	77:7	76:6
pcle	44:10	45:10	44:10	81:10	85:10	66:10
pcler8	131:11	114:11	114:11	111:10	134:10	111:10
pm1	51:8	49:8	46:8	52:6	55:6	52:6
sct	58:13	60:13	58:13	86:11	88:9	68:9
tcon	26:1	26:1	26:1	25:1	25:1	25:1
term1	113:19	108:19	108:19	120:10	123:10	120:10
ttt2	139:13	136:13	136:13	163:9	158:9	149:10
unreg	83:3	110:4	83:3	84:3	111:4	84:3
vda	511:12	512:12	511:12	547:12	530:12	530:12
x1	481:21	508:21	481:21	468:18	518:19	468:18
x2	37:6	34:6	34:6	45:6	42:6	42:6
x3	700:19	669:19	669:19	687:14	636:15	636:15
x4	434:14	420:14	420:14	480:11	498:10	377:12
z4ml	25:6	18:6	18:6	27:6	22:6	22:6
TOTAL	7371:413	7256:410	6840:412	7825:349	7652:351	6388:349

Table 6.9: This table compares the result of applying SplitOrder to unoptimized examples and examples that have been optimized with SIS using the standard script. Columns named *noopt* are copied from Table 6.8. The columns headed with *opt* are the results of applying SplitOrder to the SIS optimized examples. The columns headed with *best* shows the best results presented in this chapter for a particular example, and can be used to compare the sizes of OBDDs against canonical if-then-else DAGs.

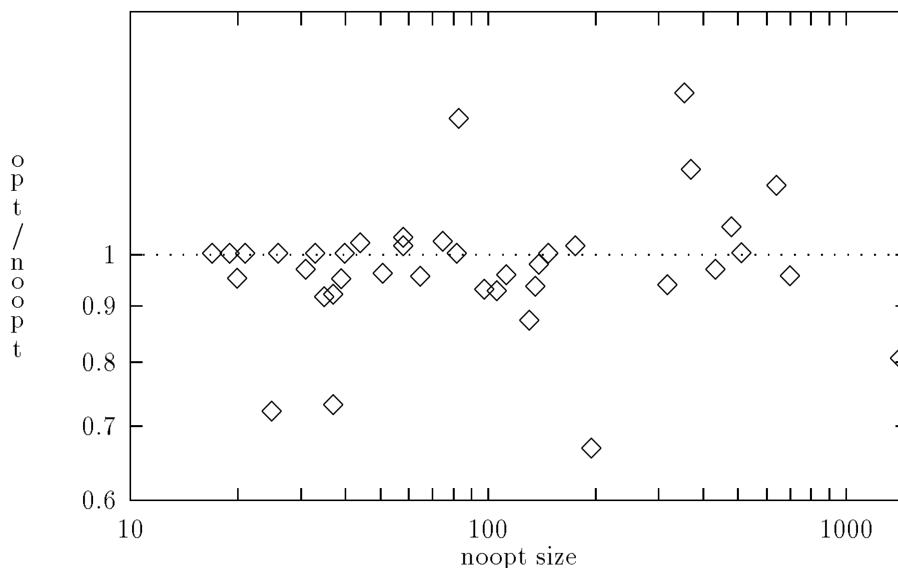

Figure 6.11: The size of OBDDs when using SplitOrder on unoptimized examples divided by the size when using the SIS optimized examples. Squares below 1 indicate that the optimized examples resulted in better variable order.

As we see from the table, the total improvement is not significant—for OBDDs we got a 1.5% decrease in size and for canonical if-then-else DAGs the decrease was 2.2%. However, if look at the number of examples that were improved, we see that for OBDDs 20 examples got better, 11 worse and 8 remained unchanged. For canonical if-then-else DAGs 14 examples got better, 17 worse and 8 remained unchanged. The data from Table 6.9 is plotted in Figure 6.11 and 6.12 which shows scatter diagrams for all examples comparing the result of SplitOrder applied to unoptimized and optimized examples.

Comparing the sizes of canonical if-then-else DAGs to the sizes of OBDDs

From Table 6.9 it can be seen the total size of canonical if-then-else DAGs is only 6.6% smaller than the total size of OBDDs. It can also be seen that in general OBDDs have fewer nodes than canonical if-then-else DAGs. Figure 6.13 shows the ratio of the canonical if-then-else DAG size over OBDD size for the examples in Table 6.9. For each example we have used the smallest canonical if-then-else DAG and smallest OBDD found during the experiments in this chapter—this corresponds to plotting the ratio of the columns *best* in Table 6.9.

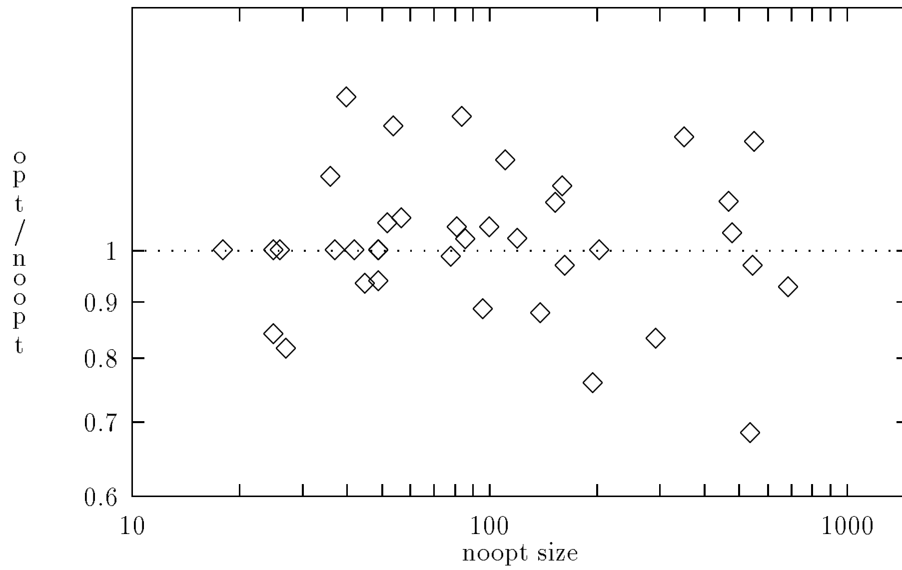

Figure 6.12: The size of canonical if-then-else DAGs when using SplitOrder on unoptimized examples divided by the size when using the SIS optimized examples. Squares below 1 indicate that the optimized examples resulted in better variable order.

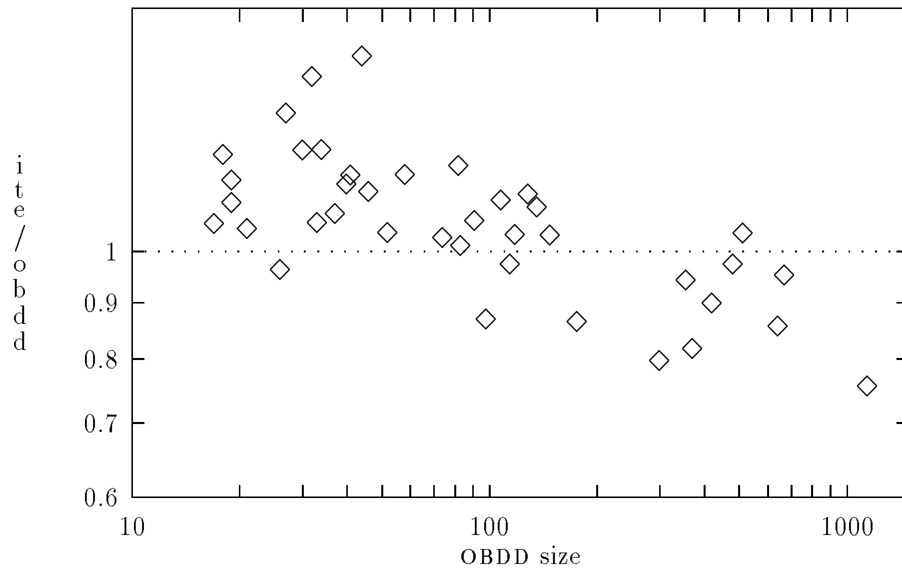

Figure 6.13: The number of nodes (`size`) in canonical if-then-else DAGs divided by the the number of nodes (`siscount`) in OBDDs. Squares below 1 indicate that the if-then-else DAG is smaller than the OBDD. For each example we used the smallest canonical if-then-else DAG and smallest OBDD obtained through all the experiments in this chapter.

It is interesting to observe that for small examples it appears as if the OBDD is winning, whereas the larger examples have smaller canonical if-then-else DAGs. This is believed to be more a coincidence than a general rule. Looking at Table 6.3 and 6.4 we see that for a different set of large examples, OBDDs and canonical if-then-else DAGs are almost the same size with no leaning towards one or the other.

We finally compare OBDDs and canonical if-then-else DAGs towards the total best of both canonical representations. In Table 6.10 we summarize the results by normalizing the total of each heuristic (all the heuristics used in this chapter) to the total best of both canonical representations. Let *BestSize* denote the smallest number of nodes in any of the canonical forms (the smallest of the columns *best* in Table 6.9) for a particular example. Table 6.10 then reports the ratio between the total for each heuristic and the total *BestSize*. In the table we report the ratios for both canonical if-then-else DAGs (ItemCanonical) and OBDDs, and thus the table also compares canonical if-then-else DAGs sizes with OBDD sizes. Each row identifies a given heuristic. The row *best* is the result of normalizing the columns *best* in Table 6.9 to *BestSize*. The row *Original* is the result of using the variable order as it appears in the input file. The row *Random* is the result of using a randomly generated variable order.

From Table 6.10 we see that the canonical if-then-else DAGs are only 3% larger than the best known canonical form whereas OBDDs are 11% larger than the best known canonical form. We also see that if-then-else DAGs are not quite as sensitive to the variable order as OBDDs. Again, the latter is not surprising since allowing general expressions in the **if**-part makes an if-then-else DAG more flexible than an OBDD. We see that SplitOrder is the only variable ordering heuristic that performs better on OBDDs than on if-then-else DAGs, and that it is the single best heuristic for both canonical if-then-else DAGs and OBDDs.

Applying SplitOrder to larger examples

In Table 6.3 and 6.4 we showed results of applying the depth-first ordering heuristics to examples from the ISCAS benchmark set for testing [Lis88]. We repeat the same examples

Total Size/Total BestSize				
	ItemCanonical		OBDD	
Method	Incremental	Reconvergent	Incremental	Reconvergent
Simple Depth	1.69	1.64	2.30	2.23
Fanout	1.64	1.63	2.32	2.26
Height	2.29	1.51	2.73	2.02
Count	1.59	1.42	2.12	1.90
SIS	1.66		2.13	
Split	1.26		1.19	
best	1.03		1.11	
Original	3.36		4.15	
Random	3.01		3.14	

Table 6.10: The result of normalizing each heuristic to the best of all heuristics. This table shows the ratio between the total number of nodes using one heuristic and the total BestSize. BestSize, for a particular example, is the smallest number of nodes seen with any of the variable orders tried in the canonical if-then-else DAG or OBDD whichever is the smallest. The row *best* is the result of normalizing the columns *best* in table 6.9 to *BestSize*.

using SplitOrder and the results are shown in Table 6.11.

For some of the examples, we were unable to complete SplitOrder on all the outputs at once, and had to run it on just the largest output. In Table 6.11 we indicate this by giving the name of the output we passed to SplitOrder. For many of the ISCAS examples, a good order for the largest output is also a reasonable order for the other outputs, and hence converting all the outputs to canonical form with the order computed for the largest output works well.

We see that with SplitOrder we are now able to compute an order for which we can construct the canonical forms. This was not the case with the depth-first ordering heuristics (see Table 6.3 and 6.4).

Iterating SplitOrder

We finally show the results of iterating SplitOrder on examples from the ISCAS benchmark set [Lis88]. Table 6.12 contains the results for the examples for which we could find previous results in the literature [MWBS88, FFM93, MKR92].

Problem	Outputs	OBDD	ItemCan
C432	all	1570:35	1307:35
C499	all	42162:40	42181:40
C880	all	9021:42	7188:37
C1355	1324GAT(583)	77746:40	77765:40
C1908	75(866)	10356:30	9876:30
C2670	all	54895:70	40754:42
C3540	all	61054:37	58659:37
C5315	all	3327:46	2639:26

Table 6.11: Result of SplitOrder on examples from the ISCAS benchmark set. The second column indicates which outputs were used to obtain the order; *all* refers to using SplitOrder on all the outputs at once. The third column is the *total* number of nodes (`siscount`) and height in the multiply-rooted OBDD with roots in the primary outputs. The fourth column is the total number of nodes (`size`) and height in corresponding canonical if-then-else DAG.

SplitOrder is iterated according to Figure 6.2. We start with a non-canonical if-then-else DAG and compute a variable order for all the primary outputs at once (this is done by passing all primary output expressions to SplitOrder). Then the canonical form transformation is used to convert each primary output to ordered binary decision diagrams, which in the next iteration are passed to SplitOrder to compute a new variable order.

For some of the examples, we were unable to complete SplitOrder on all the outputs at once, and had to run it on just the largest output. In Table 6.12 we indicate this by giving the name of the output we passed to SplitOrder. For many of the ISCAS examples, a good order for the largest output is also a reasonable order for the other outputs, and hence converting all the outputs to OBDDs with the order computed for the largest output works well. Furthermore, after converting all the outputs to OBDDs using the order computed for the largest output, we are often able to iterate SplitOrder on all the outputs at once.

For all the examples, we report both the total size over all outputs, and the size for the largest output alone. Total size of a network, where all the outputs have been converted to one multiply-rooted OBDD, has received little consideration in the literature. It is important to realize that just finding an order that results in size k for the largest output, and sizes less than k for the other outputs, doesn't necessarily mean that the total number of different

nodes for all the outputs have been minimized, since a multiply-rooted BDD can share varying amounts of the subDAGs. This is illustrated most clearly in the iteration for C1908.

Table 6.12 contains a set of rows for each example. The first row for each example is the result of applying SplitOrder and OBDD conversion to the initial DAG, which is not in canonical form, but merely a multiply-rooted if-then-else DAG. If we were unable to run SplitOrder on all the outputs at once, the output named was chosen among all the outputs as the largest of the non-canonical roots in the multiply-rooted if-then-else DAG. The remaining rows correspond to iterating on the result of the previous row. In the column *Total* we report the total size over all outputs, that is, we count the number of different nodes in the multiply-rooted OBDD with roots in the primary outputs. For the column *Largest output* we report the size of the output that is largest with respect to the order computed. It should be noted that this output may change from iteration to iteration.

For all the examples, we achieve a significant reduction in size when SplitOrder is iterated. In all but 3 examples we beat the previous best order for the largest output by as much as 60%. We do poorly on C2670, which is the only example we were unable to iterate on. C1908 illustrates how minimizing the largest output does not reduce the total size—here we achieve further reduction in the total size while increasing the size of the largest output. The most unstable example was C3540, where total size fluctuates from 36200 to 42400 if iteration is continued beyond the last row. All other examples were stable within less than 1% of the result in the last row.

C1355 is logically equivalent to C499, but SplitOrder came up with different orders for the two examples. Generally the starting point for the iteration has a big influence on the final result. We tried converting C1355 to an OBDD using the input order (the order in which the primary inputs occur in the input file) and then iterating from that point. This gave us a smaller total size for C1355 (32169) but the largest output had increased to 2308. For both C499 and C1355 we were unable to iterate SplitOrder on all the outputs at once, thus making it impossible to optimize for total size in the most logical way.

Problem	Method	Total	Largest output	Best previous largest output
C432	all;obdd	1570	454	
	all;obdd	1326	388	
	all;obdd	*1288*	370	*369* [MKR92]
C499	all;obdd	42162	2088	
	OD0(242);obdd	41074	2055	
	OD0(242);obdd	*40794*	*2047*	4283 [MKR92]
C880	all;obdd	9021	3057	
	all;obdd	5709	2172	
	all;obdd	5537	1969	
	all;obdd	*4829*	1853	*1248* [MKR92]
C1355	1324GAT(583);obdd	77746	3330	
	1324GAT(583);obdd	37706	2063	
	all;obdd	35842	1991	
	all;obdd	*35298*	*1959*	4283 [MKR92]
C1908	75(866);obdd	10356	1357	
	75(866);obdd	9541	*1264*	1606 [MKR92]
	75(866);obdd	9515	1265	
	all;obdd	8863	1408	
	all;obdd	*8805*	1403	
C2670	311(1278);obdd	*54895*	30560	*14763* [FFM93]
C3540	405(1717);obdd	61054	21791	
	405(1717);obdd	43163	11269	
	405(1717);obdd	*36155*	10334	
	all;obdd	42436	12462	
	all;obdd	36274	*10305*	17747 [MKR92]
C5315	all;obdd	3327	780	
	all;obdd	2381	492	
	all;obdd	*2289*	*472*	1296 [MKR92]
	all;obdd	2320	472	

Table 6.12: Result of iterating SplitOrder. The second column is the method used to obtain the order, *all* refers to using SplitOrder on all the outputs at once, *obdd* is converting all the outputs to OBDD using the computed order. If *obdd* is preceded by *x(y)* then this is the name of a primary output and means that we ran SplitOrder only on this output, but used that order for all the other outputs as well when converting to OBDD. The third column is the *total* number of nodes (`siscount` in the multiply-rooted OBDD with roots in the primary outputs. The fourth column is the total number of nodes in the OBDD rooted at the largest output. Finally the last column gives the previous best known size of the largest output.

6.5 The effect of variable order on other transformations

In the previous sections we focused on finding variable orders that result in small canonical forms. In this section we will show that not only the canonical form transformations rely on variable order.

As has already pointed out in Chapter 4, LocalFactor gets most of its power from converting small parts of the DAG to canonical form. Using the strong canonical form described in Section 3.3 it is clear that identical subDAGs will result in explicit sharing. But as has become clear from this chapter different orderings result in very different sharing, and hence it is important to find the ordering that results in the most explicit sharing.

Encouraged by the results of iterating SplitOrder (see Table 6.12) we decided to make the experiment shown in Figure 6.14 (these are the commands used by ITEM). We first construct the initial if-then-else DAG, which we optimize using LocalFactor (`transform -m local`) and two-column rectangle replacement (`bcov`). The first time through LocalFactor uses default variable order constructed as it occurred in the input file. LocalFactor and two-column rectangle replacement are iterated as this generally improves the results slightly. The optimized DAG is then used by SplitOrder (`order split -c siscount`) to compute a variable order from which we construct an OBDD (`obdd`). With the OBDD we iterate SplitOrder once and with the resulting order we iterate the entire script[2].

The ITEM script in Figure 6.14 was iterated 5 times. Even though there were some fluctuations we only report the results for the first and last run through the script. In Table 6.13 we show the results. We report the measures *count*, *edges*, *size*, and *height*, all of which (except *edges* which counts the number of edges in the DAG) we have used in other sections. For each example and each cost measure we report the initial result, the last result, and the difference (last−initial). The last row sums up the difference for each cost measure.

[2] ITEM caches logically equivalent expressions in equivalent rings and during optimization of a function it always checks the equivalent ring to see if it contains a "better" implementation. Hence, before iterating the script we must clear all equivalent rings so that the starting point is the same for each run through the script.

Example	Count			Edges			Size			Height		
	first	last	diff	first	last	diff	first	last	diff	first	last	diff
alu2	393	374	-19	447	424	-23	175	166	-9	11	9	-2
alu4	728	763	35	1085	894	-191	492	353	-139	34	14	-20
apex6	781	775	-6	1184	1061	-123	631	551	-80	20	12	-8
apex7	245	239	-6	399	375	-24	221	207	-14	14	14	0
b9	101	101	0	172	172	0	115	115	0	9	9	0
c8	133	132	-1	152	147	-5	79	76	-3	9	9	0
cc	64	64	0	87	87	0	51	51	0	5	5	0
cht	184	184	0	221	221	0	121	121	0	3	3	0
cm138a	24	24	0	40	40	0	22	22	0	5	5	0
cm151a	27	24	-3	40	25	-15	29	20	-9	9	4	-5
cm152a	22	22	0	22	22	0	18	18	0	3	3	0
cm162a	40	42	2	63	64	1	41	41	0	8	8	0
cm163a	39	40	1	61	60	-1	42	41	-1	7	7	0
cm42a	27	27	0	44	44	0	21	21	0	3	3	0
cm85a	43	49	6	71	65	-6	43	37	-6	10	6	-4
cmb	40	40	0	67	67	0	46	46	0	11	11	0
count	143	143	0	174	174	0	98	98	0	18	18	0
cu	51	51	0	79	79	0	46	46	0	6	6	0
decod	42	42	0	68	68	0	31	31	0	3	3	0
example2	304	309	5	464	465	1	271	270	-1	12	12	0
f51m	88	84	-4	108	112	4	48	52	4	7	8	1
frg1	174	144	-30	264	204	-60	145	115	-30	14	18	4
frg2	979	964	-15	1690	1648	-42	897	874	-23	15	14	-1
lal	92	91	-1	156	145	-11	93	86	-7	10	10	0
ldd	113	103	-10	156	151	-5	69	69	0	6	10	4
pcle	64	64	0	98	98	0	60	60	0	9	9	0
pcler8	88	88	0	138	138	0	84	84	0	11	11	0
pm1	49	44	-5	79	72	-7	48	45	-3	6	6	0
sct	70	71	1	107	106	-1	62	61	-1	8	8	0
tcon	40	40	0	40	40	0	25	25	0	1	1	0
term1	209	117	-92	354	158	-196	197	97	-100	21	10	-11
ttt2	240	200	-40	330	274	-56	157	133	-24	8	10	2
unreg	128	128	0	144	144	0	84	84	0	3	3	0
vda	1074	1114	40	1473	1361	-112	628	540	-88	24	16	-8
x1	300	305	5	523	521	-2	288	285	-3	14	15	1
x2	51	51	0	80	77	-3	44	42	-2	7	6	-1
x3	865	792	-73	1214	1005	-209	623	508	-115	17	14	-3
x4	390	395	5	625	530	-95	357	292	-65	10	8	-2
z4ml	61	37	-24	88	43	-45	44	22	-22	9	5	-4
TOTAL	8506	8277	-229	12607	11381	-1226	6546	5805	-741	400	343	-57

Table 6.13: Results extracted from iterating the script in Figure 6.14. The first column is the name of the example, the following 4 columns report the specified cost metric after the *first* and *last* run through the script. The number *diff* reports the difference in the cost metric between the first and last run.

```
# optimize the network using LocalFactor and bcov
            transform -m local
            bcov
            transform -m local
            bcov
            transform -m local
            bcov
            print count() edges() size() height()
# compute a variable order
            order split -c siscount
# convert the outputs to OBDDs
            obdd
# recompute a variable order
            order split -c siscount
# iterate the script
```

Figure 6.14: Optimization script used with ITEM to show the effect of variable order on other optimizations by LocalFactor. The script was iterated 5 times.

From the table we see that overall there is a significant difference between the first and last run through the script for all the metrics. The most significant differences were in the *size*, *edges*, and *height* metrics. This makes sense, since LocalFactor specifically tries to reduce the number of edges and SplitOrder attempts to find an order that reduces the size of the canonical DAG. The results demonstrate that a good variable order is important when applying local transformations to the DAG. Finally, the *size* column of Table 6.13 should be compared to the *best* column for canonical if-then-else DAGs in Table 6.9. The difference (totally 9.1%) can be taken as an estimate for the cost of converting a non-canonical if-then-else DAG to canonical form.

6.6 Conclusion

In this chapter we have presented several depth-first ordering heuristics and the new SplitOrder heuristic for finding variable orders for canonical if-then-else DAGs and OBDDs.

Among the depth-first ordering heuristics the new *reconvergent* merging scheme improves the results while maintaining the $O(n)$ complexity associated with traversal-based heuristics.

The new SplitOrder heuristic results in significant improvement over the depth-first heuristics when converting to OBDDs. When converting to canonical if-then-else DAGs, SplitOrder is still the single best heuristic, but only about as good as trying 8 different depth-first ordering heuristics and choosing the best for each example. For canonical if-then-else DAGs, trying all the different heuristics presented in this chapter, results on the average in 14% fewer nodes than using SplitOrder alone (2% OBDDs).

An open problem is to find ordering heuristics that are targeted for canonical if-then-else DAGs rather than OBDDs.

It appears as if OBDDs in general have fewer nodes than canonical if-then-else DAGs. The height of canonical if-then-else DAGs is almost always significantly less than the height of OBDDs. The results also demonstrate that OBDDs are more sensitive to the variable order than canonical if-then-else DAGs.

We also showed that iterating SplitOrder on the ISCAS benchmark set greatly improves the resulting variable order for OBDDs. We have experimented with iteration of the depth-first ordering heuristics, but here the results did not converge and didn't seem to fluctuate between any fixed sizes.

Finally we showed that we could improve the results of logic optimization by iterating through combinations of paths in Figure 6.2, and in particular we showed how the variable order can affect the final result of logic optimization.

7. Conclusions and future research

This thesis presented two important aspects of logic optimization: common subexpression extraction using *two-column rectangle replacement* and variable ordering for Boolean functions represented as if-then-else DAGs and binary decision diagrams.

7.1 Two-column rectangle replacement

Two-column rectangle replacement is a general sub-expression extraction technique that does not rely on the underlying representation of Boolean expressions. Two-column rectangle replacement recognizes commonality between commutative and associative Boolean expressions.

We showed how Boolean matrices could be used to represent Boolean expressions and how rectangles of the matrices corresponded to common subexpressions. Specifically we constructed an XOR-matrix and an OR-matrix, where the first represents the XOR and XNOR expressions in a set of expressions and the latter represents the AND, NAND, OR, and NOR-expressions.

In two-column rectangle replacement, rectangles of exactly two columns are replaced sequentially with a new single column until each row uses exactly one column. The two-column rectangle-replacement problem is to replace rectangles in such an order that the area or delay of the final circuit is minimized.

When optimizing for area the order of replacement is determined based on the value of a rectangle. The value of a rectangle is the difference in the number of 1's in the matrix before and after a replacement. The results demonstrated a 10% improvement in our area estimate when applying two-column rectangle replacement to expressions minimized using LocalFactor (an optimization technique in ITEM, which relies on canonical form to detect sharing).

When optimizing for delay we showed that two-column rectangle replacement could effectively be used in balancing any cost measure of DAG. Typically, we use height as

our delay estimate and by replacing the two-column rectangle in which the new column will have the lowest height, we balance the DAG and thus minimize the delay estimate. We demonstrated that this technique was very effective in balancing the DAG, while keeping the area under control. Especially the latter is important—there are easier and more efficient techniques (DMIG) for balancing a network of commutative and associative operators, but these techniques completely ignore other cost measures such as area of the final circuit. We compared two-column rectangle replacement for delay with DMIG, which ensures minimum height, and found that two-column rectangle replacement for delay totally resulted in height that was only 2.5% from the optimum. When compared to LocalFactor alone, DMIG increased the area estimated by more than 13%, while two-column rectangle didn't change the total area estimate significantly.

The power of two-column rectangle replacement comes from having to consider only rectangles of two columns rather than rectangle spanning multiple columns. In two-column rectangle replacement we can afford to enumerate all possible rectangles and replace the best.

7.2 Variable ordering

The second topic addressed by this thesis was a problem specific to if-then-else DAGs and binary decision diagrams. We showed techniques for finding variable orders that result in small canonical if-then-else DAGs and small ordered binary decision diagrams. Two different approaches to the problem were investigated: *depth-first ordering heuristics* and the *SplitOrder* heuristic.

The depth-first ordering heuristics all compute a total variable order based on a single depth-first traversal of an initial non-canonical if-then-else DAG. The heuristics merge total orders into a new total order using two different variants: *the incremental* merging strategy, which orders the variables in the order they are visisted, and the new *reconvergent* merging strategy, which is based on reconvergent fanout and tend to order shared variables first. Among the depth-first ordering heuristics the *reconvergent* merging strategy is superior; it

produces variable orders that on the average result in 11% smaller canonical forms than the best incremental ordering heuristic. Still, no single traversal-based heuristic is particular effective, and trying combinations of several will almost always improve the results.

The second approach to variable ordering was based on the SplitOrder heuristic, which is especially well suited for finding good variable orders for ordered binary decision diagrams. SplitOrder constructs the variable order by building an OBDD top-down one level at a time, choosing the next variable such that the corresponding level in the OBDD has few nodes and represents expressions that are as small as possible. When converting to OBDDs SplitOrder averages 25% fewer nodes than taking the best of 8 depth-first techniques (36% better than the best single depth-first technique). A nice property of the SplitOrder heuristic (not found with the depth-first ordering heuristics) is that it can be iterated to improve the resulting variable order. By iterating SplitOrder we achieved variable orders that result in up to 50% smaller OBDDs than what other researchers have reported using depth-first traversal or simulated annealing.

Finally, we showed that variable ordering is important for more than just the canonical form transformation of if-then-else DAGs. Especially we showed that our general transformations for technology-independent logic optimization (LocalFactor together with two-column rectangle replacement) performed much better if they started from an order computed by SplitOrder rather than a default order.

7.3 Future work

The thesis leaves open the problem of finding variable ordering heuristics that are target for canonical if-then-else DAGs rather than OBDDs. Even though SplitOrder is the single best ordering heuristics for canonical if-then-else DAGs, it still, on the average, results in 20% larger DAGs than what could be achieved by trying all the heuristics presented in this thesis.

The *given that* operator presented in Section 6.4.2 should be improved to propagate simplifications more efficiently. This would greatly improve the accuracy of the cost function

used by SplitOrder.

There are many more areas of interest that still need work.

- Better local transformations for factoring including testability-preserving transformations. The current transformations are rather ad hoc and part of this research would involve getting them into a theoretical background that will make them easier to explain. We also want to modify LocalFactor to preserve path-delay-fault testability.
- Improve synthesis for testability by finding better testability preserving transformations for factoring and sub-expression extraction.
- Better delay and area estimates are important for technology-independent minimization. When optimizing for delay using two-column rectangle replacement we have used the height of the DAG to estimate delay. Height is the most widely used estimate for delay, but unfortunately it is not very accurate. When targeting FPGA look-up table architectures we often use look-up table height as the delay estimate, but even then, this is inaccurate since it does not take into account routing delay.
- Techniques for minimizing sequential circuits. ITEM has only recently been extended to handle sequential logic as well as combinational logic, but so far all our optimization techniques are for combinational logic only.
- Adaptations of other logic minimization work to if-then-else DAGs. We have had great success with the rectangle covering techniques and also see use for other techniques such as global flow algorithms [BT88].
- New technology mappers to cell libraries or cell generators. We are also looking for mappers for sequential logic.
- Don't-care information is used by the current factoring transformations, but not very effectively. There are several ways in which the don't-care usage could be increased, and we need to determine which of these are effective and inexpensive. Also, the algorithms that apply don't-care information need to be sped up.

References

[BCGH86] Karen Bartlett, William Cohen, Aart De Geus, and Gary Hachtel. Synthesis and optimization of multilevel logic under timing constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-5(4):582–596, October 1986.

[Ber91] C. Leonard Berman. Circuit width, register allocation, and reduced function graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-10(8):1059–1066, August 1991.

[BHJ+87] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. H. Moceyunas, C. R. Morrison, and D. Ravenscroft. The boulder optimal logic desing system. In *IEEE International Conference on Computer-Aided Design ICCAD-87*, pages 62–65, Santa Clara, CA, 9–12 November 1987.

[BHMS84] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[BHS90] R. K. Brayton, G. D. Hachtel, and A. L. Alberto Sangiovanni-Vincentelli. Multilevel logic synthesis. *Proceedings of the IEEE*, 78(2):264–300, February 1990.

[Bra87a] Robert K. Brayton. Algorithms for multi-level logic synthesis and optimization. In G. De Micheli, Alberto Sangiovanni-Vincentelli, and P. Antognetti, editors, *Design Systems for VLSI Circuits—Logic Synthesis and Silicon Compilation*, pages 197–247. Martinus Nijhoff Publishers, 1987.

[Bra87b] Robert K. Brayton. Factoring logic functions. *IBM Journal of research and development*, 31(2):187–198, March 1987.

[BRKM91] Kenneth M. Butler, Don E. Ross, Rohit Kapur, and M. Ray Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *ACM IEEE* 28th *Design Automation Conference Proceedings*, pages 417–420, San Francisco, California, June 17–21 1991.

[BRSW87a] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-6(6):1062–1081, November 1987.

[BRSW87b] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert Wang. Multi-level logic optimization and the rectangle covering problem. In *IEEE International Conference on Computer-Aided Design ICCAD-87*, pages 66–69, Santa Clara, CA, November 1987.

[Bry85] Randal Everitt Bryant. Symbolic verification of MOS circuits. In Henry Fuchs, editor, *1985 Chapel Hill Conference on Very Large Scale Integration*, pages 419–438. Computer Science Press, 1985.

[Bry86] Randal Everitt Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[Bry91] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, February 1991.

[BT88] Leonard Berman and Louise Trevillyan. A global approach to circuit size reduction. In Jonathan Allen and F. Thomson Leighton, editors, *Proceeding of the 5th MIT Conference on Advanced Research in VLSI*, pages 203–214, Cambridge, MA, March 1988.

[CCD+92] Kuang-Chien Chen, Jason Cong, Yuzheng Ding, Andrew B. Kahng, and Peter Trajmar. DAG-Map: Graph-based FPGA technology mapping for delay optimization. *IEEE Design and Test of Computers*, pages 7–20, September 1992.

[DBG+84] John A. Darringer, Daniel Brand, John V. Gerbi, Jr. William H Joyner, and Louise Trevillyan. LSS: A system for production logic synthesis. *IBM Journal of research and development*, 28(5):537–545, September 1984.

[FFK88] Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *IEEE International Conference on Computer-Aided Design ICCAD-88*, pages 2–5, Santa Clara, CA, 7–10 November 1988.

[FFM93] Masahiro Fujita, Hisanori Fujisawa, and Yusuke Matsunaga. Variable ordering algorithms for ordered binary decision diagrams and their evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-12(1):6–12, January 1993.

[FS87] Steven J. Friedman and Kenneth J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *ACM IEEE 24th Design Automation Conference Proceedings*, pages 348–355, Miami Beach, FL, 28 June–1 July 1987.

[GDP86] D.D. Gajski, N.D. Dutt, and B.M. Pangrle. *Silicon Compilation.* Addison-Wesley Publishing Company, 1986.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, CA, 1979.

[HJKM89] G. Hachtel, R. Jacoby, K. Keutzer, and C. Morrison. On the relationship between area optimization and multifault testability of multilevel logic. In *IEEE International Conference on Computer-Aided Design ICCAD-89*, pages 422–425, Santa Clara, CA, November 1989.

[Joh79] Dave Johannsen. Bristle blocks: A silicon compiler. In *Proceedings of 16th Design Automation Conference*, pages 310–313, 1979.

[Kar88] Kevin Karplus. Representing Boolean functions with If-Then-Else DAGs. Technical Report UCSC-CRL-88-28, Board of Studies in Computer Engineering, University of California at Santa Cruz, Santa Cruz, CA 95064, December 1988.

[Kar89] Kevin Karplus. Using if-then-else dags for multi-level logic minimization. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 101–118, Pasadena, CA, 20-22 March 1989.

[Kar90] Kevin Karplus. Discussions in daily meetings, Fall 1990.

[Kar91a] Kevin Karplus. Amap: a technology mapper for selector-based field-programmable gate arrays. In *ACM IEEE 28th Design Automation Conference Proceedings*, pages 244–247, San Francisco, California, June 17–21 1991.

[Kar91b] Kevin Karplus. Canonical forms of if-then-else dags are robustly path-delay-fault testable, April 1991. Unpublished paper.

[Kar91c] Kevin Karplus. Item: an if-then-else minimizer for logic synthesis, April 1991. Unpublished paper.

[Kar91d] Kevin Karplus. Xmap: a technology mapper for table-lookup field-programmable gate arrays. In *ACM IEEE 28th Design Automation Conference Proceedings*, pages 240–243, San Francisco, California, June 17–21 1991.

[Kar93] Kevin Karplus. Xtmap: a generate-and-test mapper for table-lookup gate arrays. In *Compcon 1993*, pages 391–399, 22–26 Feb 1993.

[Keu87] Kurt Keutzer. DAGON: Technology binding and local optmization by DAG matching. In *ACM IEEE 24th Design Automation Conference Proceedings*, pages 341–347, Miami Beach, FL, 28 June–1 July 1987.

[Law64] Eugene L. Lawler. An approach to multilevel Boolean minimization. *Journal of the Association for Computing Machinery*, 11(3):283–295, July 1964.

[Lis88] Robert Lisanke. Logic synthesis and optimization benchmarks. Technical report, Microelectronics Center of North Carolina, P.O. Box 12889, Research Triangle Park, NC 27709, 16 December 1988.

[MBSS91] Patrick C. McGeer, Robert K. Brayton, Alberto Sangiovanni-Vincentelli, and Sartaj K. Sahni. Performance enhancement through the generalized bypass transform. In *IEEE International Conference on Computer-Aided Design ICCAD-91*, pages 184–187, Santa Clara, CA, 11–14 November 1991.

[MF89] Y. Matsunaga and Masahiro Fujita. Multi-level logic optimization using binary decision diagrams. In *IEEE International Conference on Computer-Aided Design ICCAD-89*, pages 556–559, Santa Clara, CA, November 1989.

[MKR92] M. Ray Mercer, Rohit Kapur, and Don E. Ross. Functional approaches to generating orderings for efficient symbolic representations. In *ACM IEEE 29th Design Automation Conference Proceedings*, pages 624–627, Anaheim, California, June 1992.

[MWBS88] Sharad Malik, Albert R. Wang, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *IEEE International Conference on Computer-Aided Design ICCAD-88*, pages 6–9, Santa Clara, CA, 7–10 November 1988.

[NB86] Ravi Nair and Daniel Brand. Construction of optimal DCVS trees. Technical Report RC 11863, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 19 March 1986.

[Ros90] Don E. Ross. *Functional calculations using ordered partial multi decision diagrams*. PhD thesis, The University of Texas at Austin, Austin, Texas, August 1990.

[Rud89] Richard L. Rudell. *Logic Synthesis for VLSI Design.* PhD thesis, Department of Electrical and Computer Science, University of California at Berkeley, Berkeley, CA, May 1989.

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Workshop on Logic Synthesis*, Lake Tahoe, May 1993.

[SK91] Søren Søe and Kevin Karplus. Logic minimization using two-column rectangle replacement. In *ACM IEEE* 28th *Design Automation Conference Proceedings*, San Francisco, CA, 17–21 June 1991.

[SK93] Søren Søe and Kevin Karplus. Ordering heuristics for ordered binary decision diagrams and canonical if-then-else dags. In *International Workshop on Logic Synthesis*, Lake Tahoe, CA, May 1993.

[SU70] Ravi Sethi and J. D. Ullman. The generation of optimal code for arithmetic expressions. *Journals of the ACM*, 17(4):715–728, 1970.

[Wan89] Albert Ren Rui Wang. *Algorithms for Multilevel Logic Optimization.* PhD thesis, Department of Electrical and Computer Science, University of California at Berkeley, Berkeley, CA, May 1989.