

IMPROVEMENTS TO FIELD-PROGRAMMABLE GATE ARRAY DESIGN
EFFICIENCY USING LOGIC SYNTHESIS

by

Andrew C. Ling

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy in Engineering
Graduate Department of Electrical and Computer Engineering
University of Toronto

© Copyright by Andrew C. Ling 2009

IMPROVEMENTS TO FIELD-PROGRAMMABLE GATE ARRAY DESIGN EFFICIENCY USING LOGIC SYNTHESIS

Andrew C. Ling

Doctor of Philosophy, 2009

Graduate Department of Electrical and Computer Engineering

University of Toronto

Abstract

As Field-Programmable Gate Array (FPGA) capacity can now support several processors on a single device, the scalability of FPGA design tools and methods has emerged as a major obstacle for the wider use of FPGAs. For example, logic synthesis, which has traditionally been the fastest step in the FPGA Computer-Aided Design (CAD) flow, now takes several hours to complete in a typical FPGA compile. In this work, we address this problem by focusing on two areas. First, we revisit FPGA logic synthesis and attempt to improve its scalability. Specifically, we look at a binary decision diagram (BDD) based logic synthesis flow, referred to as *FBDD*, where we improve its runtime by several fold with a marginal impact to the resulting circuit area. We do so by speeding up the classical cut generation problem by an order-of-magnitude which enables its application directly at the logic synthesis level. Following this, we introduce a guided partitioning technique using a fast global budgeting formulation, which enables us to optimize individual “pockets” within the circuit without degrading the overall circuit performance. By using partitioning we can significantly reduce the solution space of the logic synthesis problem and, furthermore, open up the possibility of parallelizing the logic synthesis step.

The second area we look at is the area of Engineering Change Orders (ECOs). ECOs are incremental modifications to a design late in the design flow. This is beneficial since it is minimally disruptive to the existing circuit which preserves much of the engineering effort invested previously in the design. In a design flow where most of the steps are fully automated, ECOs still remain largely a manual process. This can often tie up a designer for weeks leading to missed project deadlines which is very detrimental to products whose life-cycle can span only a few months. As a solution to this, we show how we can leverage existing logic synthesis techniques to automatically modify a circuit in a minimally disruptive manner. This can significantly reduce the turn-around time when applying ECOs.

Acknowledgments

I would like to gratefully acknowledge the enthusiastic supervision of my advisors Professor Jianwen Zhu and Professor Stephen D. Brown for their continuous guidance and inspiration. Their technical leadership and coaching ability is something I aspire to achieve and their influence has made me into a better research scientist, mentor, and person.

I would also like to thank Professor Andreas Veneris and Professor Zvonko Vranesic for their guidance throughout this process. They have always shown interest in my development and are always there to lend some advice when it is needed. They have definitely contributed to the completion and success of this dissertation and my development.

I would like to thank Dr. Sean Safarpour and Terry Yang for their friendship and fruitful discussions; particularly for the work presented in Chapter 5 of this dissertation. They have always made time to meet me which has often led to several insights discussed in this work.

I would like to thank Dr. Tomasz Czakwoski, Rami Beidas, and Franjo Plavec for the many years of constructive criticisms they have provided. Through their feedback, this work has definitely been made stronger.

I would like to acknowledge the friends I have made during this process from the LP392 and SF2206 labs (Jason Luu, Navid Toosizadeh, Nahi Abdul Ghani, Dr. Ian Kwon, Sari Onaissi, Dr. Imad Ferzli, Dr. Peter Yiannacouras, Khaled Heloue, Ivan Matosevic, Davor Capalija, Cedimir Segulja, and Ahmed Abdelkhalek). They have always kept the boredom away and have been a great support in this phase of my life.

I would like to thank my parents for their continued guidance and advice throughout the years, who have always been there for me, far beyond the last few years of my life.

Finally, I would like to thank my wife, Julie Sit, for being there with me during this journey. You are my best friend and companion, and I would not be the same without you.

Contents

| | |
|--|-------------|
| List of Figures | viii |
| List of Tables | xii |
| 1 Introduction | 1 |
| 1.1 Introduction to Field-Programmable Gate Arrays | 1 |
| 1.2 Motivation and Overview | 2 |
| 1.3 Objective and Contributions | 6 |
| 1.4 Dissertation Organization | 7 |
| 2 Background | 8 |
| 2.1 Terminology | 8 |
| 2.2 Field-Programmable Gate Array (FPGA) Architecture | 10 |
| 2.2.1 Programmable Logic | 10 |
| 2.2.2 Commercial Architectures | 12 |
| 2.3 FPGA Computer-Aided Design (CAD) | 13 |
| 2.3.1 Logic Synthesis and Technology Mapping | 14 |
| 2.3.2 Clustering, Placement, and Routing | 17 |
| 2.3.3 Physically-Driven Synthesis | 19 |
| 2.3.4 Engineering Change Orders (ECOs) | 19 |
| 2.3.5 Timing Analysis | 21 |
| 2.4 Introduction to Binary Decision Diagrams | 24 |
| 2.4.1 Zero-Suppressed Binary Decision Diagrams | 31 |
| 2.5 Introduction to Boolean Satisfiability | 33 |
| 2.6 Solving the Boolean Satisfiability Problem | 34 |
| 2.6.1 Heuristics To Solve the Boolean Satisfiability Problem | 36 |
| 2.6.2 Circuits and Boolean Satisfiability | 39 |
| 2.7 Relationship between Binary Decision Diagrams and Boolean Satisfiability | 43 |
| 2.8 Summary | 44 |
| 3 Improving BDD-based Logic Synthesis through Elimination and Cut-Compression | 45 |
| 3.1 Introduction to Logic Synthesis | 46 |
| 3.2 Motivation | 46 |
| 3.3 Adaptation of Covering Problem to Elimination | 50 |
| 3.3.1 Covering Problem | 50 |

| | | |
|----------|--|------------|
| 3.3.2 | Binary Decision Diagram (BDD)-based Cut-Compression | 54 |
| 3.3.3 | Symbolic Cut Generation Algorithm | 60 |
| 3.3.4 | Ensuring K -Feasibility | 61 |
| 3.3.5 | Finding the Minimum Cost Cut | 63 |
| 3.3.6 | Using Zero-Suppressed Binary Decision Diagrams (ZDDs) | 65 |
| 3.3.7 | Reconvergence and Edge-Flow | 65 |
| 3.4 | Results | 68 |
| 3.4.1 | BddCut: Scalable Cut Generation | 68 |
| 3.4.2 | Edge Flow Heuristic | 72 |
| 3.4.3 | Covering based Elimination | 73 |
| 3.4.4 | Comparison with Structural Synthesis | 75 |
| 3.5 | Summary | 76 |
| 4 | Budget Management for Partitioning | 78 |
| 4.1 | Background and Previous Work | 83 |
| 4.1.1 | Budget Management | 83 |
| 4.2 | Partitioning with Delay Budgeting | 85 |
| 4.2.1 | Partitioning and And-Inverter Graph (AIG) construction | 86 |
| 4.2.2 | Budget Management | 87 |
| 4.2.3 | Resynthesis with Delay Budgets | 107 |
| 4.3 | Results | 107 |
| 4.3.1 | Dual Problem Performance | 108 |
| 4.3.2 | Budget Management for Partitioned Logic Synthesis | 109 |
| 4.4 | Summary | 111 |
| 5 | Automation of ECOs | 112 |
| 5.1 | Background and Related Work | 115 |
| 5.1.1 | Terminology | 115 |
| 5.1.2 | Related Work | 115 |
| 5.1.3 | Boolean Satisfiability (SAT) | 116 |
| 5.2 | Automated ECOs: General Technique | 118 |
| 5.2.1 | Netlist Localization | 120 |
| 5.2.2 | Netlist Modification | 122 |
| 5.3 | Results | 136 |
| 5.3.1 | Specification Changes | 136 |
| 5.3.2 | Error Correction | 140 |
| 5.3.3 | Generation of Model Netlist | 143 |
| 5.4 | Summary | 143 |
| 6 | Conclusion and Future Work | 145 |
| 6.1 | Summary of Contributions | 145 |
| 6.2 | Future Work | 146 |

| | |
|--|------------|
| 7 Appendix: ZDD Prune Algorithm | 149 |
| References | 152 |

List of Figures

| | | |
|------|---|----|
| 1.1 | A traditional island-style FPGA architecture. | 1 |
| 1.2 | Initial costs for fabricating an Application-Specific Integrated Circuit (ASIC) as measured by the mask set costs for each technology node from 1994 to 2007[Yan01, RMM ⁺ 03, Lam05]. | 2 |
| 1.3 | A normalized cost comparison between a Xilinx FPGA versus a Texas Instruments Digital Signal Processing/Processor (DSP)[Alt05a, Bie07]. | 3 |
| 2.1 | A Basic Logic Element (BLE) consisting of a Lookup Table (LUT) and a configurable register. | 11 |
| 2.2 | The hierarchical structure of an FPGA. | 12 |
| 2.3 | Altera’s Stratix commercial FPGA [LBJ ⁺ 03]. | 13 |
| 2.4 | A multiply-accumulate “hard” block on the commercial Stratix FPGA [Alt05b]. | 14 |
| 2.5 | A generic CAD flow for FPGAs. | 15 |
| 2.6 | An illustration of the logic synthesis process and technology mapping. (a) An unoptimized netlist. (b) An optimized netlist. (c) Identification of nodes to pack into a LUT for technology mapping. (d) Technology mapped circuit to 4-input LUTs. | 16 |
| 2.7 | The half-perimeter wirelength of a net is defined as half of the rectangle perimeter which encompasses all terminals of the net. This rectangle is termed as a <i>bounding box</i> | 18 |
| 2.8 | Illustration of physically-driven synthesis applying retiming. (a) Current placement with long distance between register d and logic element c. (b) Post-placement layout-driven optimization using retiming. (c) Incremental placement process for legalization. (d) Final legal placement of optimized netlist. | 20 |
| 2.9 | An illustration of a circuit graph with static timing values assigned to each node and edge. | 22 |
| 2.10 | Graphical representation of Shannon’s expansion. | 25 |
| 2.11 | Truth-table, cube, and Binary Decision Diagram (BDD) representation of function $f = \overline{x_1}x_3\overline{x_4} + x_1x_2\overline{x_3} + x_1x_2x_4$ | 26 |
| 2.12 | An illustration of reduction rule 1: removing redundant assignments. | 27 |
| 2.13 | An illustration applying reduction rule 1 to the entire graph: removing all redundant assignments. | 28 |
| 2.14 | An illustration of reduction rule 2: removing duplicate nodes. | 29 |
| 2.15 | An illustration of using inverted edges. | 30 |
| 2.16 | An illustration of two resulting BDDs if alternate variable orders are used. | 31 |

| | | |
|------|---|----|
| 2.17 | BDD and ZDD representation of $f = x_1x_2 + x_3x_4$ [Mis01]. | 32 |
| 2.18 | BDD and ZDD representation of the characteristic function of $\{x_1x_2, x_1x_3, x_3\}$ [Mis01]. | 32 |
| 2.19 | A Boolean formula in Conjunctive Normal Form. | 33 |
| 2.20 | An example of a unit clause, given that $x_1x_2x_3 = 110$ and x_4 is free. | 36 |
| 2.21 | A conflict-driven analysis implication graph. | 38 |
| 2.22 | Backtracking due to a conflict in Figure 2.21. | 40 |
| 2.23 | A characteristic equation derivation for 2-input AND gate. | 40 |
| 2.24 | A cascaded gate characteristic function. | 41 |
| | | |
| 3.1 | An illustration of the generic logic synthesis flow. | 47 |
| 3.2 | An illustration of an elimination operation followed by a decomposition. | 49 |
| 3.3 | Illustration of the covering problem when applied to K -LUT technology mapping. (a) Initial network. (b) A covering of the network. (c) Conversion of the covering into 4-LUTs. | 51 |
| 3.4 | High-level overview of network covering. | 51 |
| 3.5 | High-level overview of forward traversal. | 52 |
| 3.6 | High-level overview of backward traversal. | 52 |
| 3.7 | Example of two cuts in a netlist for node v_5 where c_2 dominates c_1 ($K = 3$). | 54 |
| 3.8 | Example of generating cut sets through Cartesian product operation of fanin cutsets. | 55 |
| 3.9 | The BDD representation of the cut set in Figure 3.10. | 56 |
| 3.10 | A Boolean expression based representation of cut sets. | 57 |
| 3.11 | Illustration of reusing BDDs to generate larger BDDs. (a) Small BDDs representing cut set function f_b and f_c . (b) Reusing BDDs in (a) as cofactors within cut set function f_a | 58 |
| 3.12 | BDD representation of node a cuts c_1 , c_2 , and c_3 ($K = 3$). | 59 |
| 3.13 | BDD representation of node a cuts c_1 and c_3 ($K = 3$). | 60 |
| 3.14 | High-level overview of symbolic cut generation algorithm. | 61 |
| 3.15 | High-level overview of BDD AND operation with pruning for K | 62 |
| 3.16 | Construction of BDD cut set given f_z if given positive and negative cofactors. | 63 |
| 3.17 | Find the minimum cost cut in a given cut set. | 64 |
| 3.18 | Illustration of cones with no reconvergent paths (a) and with reconvergent paths (b). | 66 |
| 3.19 | Illustration of fanout dependency on the covering. | 67 |
| 3.20 | Equation for estimating a node's fanout size. | 68 |
| | | |
| 4.1 | Partitioning of circuit for resynthesis. | 78 |
| 4.2 | Retransformation optimization to shorten the critical path using a localized view of each partition. Original depth of circuit in Figure 4.2 is 12, final depth is 10 along shaded gates. | 80 |
| 4.3 | Retransformation optimization to shorten the critical path using budget constraints to guide optimizations in each partition. Final depth is 8. | 81 |
| 4.4 | Illustration of numeric <i>depth budget</i> assignments to each partition input. | 82 |

| | | |
|------|--|-----|
| 4.5 | Alternate implementations of the same circuit, each node with a different circuit latency allocated to it. | 85 |
| 4.6 | Clustering of a simple netlist. (a) Original netlist (b) Partitioning of original netlist (c) Representing each partition and PI as a single node | 87 |
| 4.7 | Example of an And-Inverter Graph (AIG) where each node is a 2-input AND gate and each edge can be inverted. | 87 |
| 4.8 | Illustration of circuit optimization for 4.8(a) area (gate-count=4, logic-levels=4) and 4.8(b) depth (gate-count=7, logic-levels=3). | 88 |
| 4.9 | Annotated partition inputs after delay budgeting. | 89 |
| 4.10 | Simplified inverse relationship between delay budget, b_{ij} , and area estimation, $F_{ij}(b_{ij})$, defined over variable b_{ij} | 90 |
| 4.11 | Graph to ILP formulation. | 92 |
| 4.12 | Illustration of dependency of depth between inputs. | 92 |
| 4.13 | Graphical illustration of transforming the budget management problem into its dual network flow problem. (a) Partitioned graph. (b) Upper and lower bound added as edges to create network flow graph. | 100 |
| 4.14 | Illustration of proof for Proposition 4.2.3. (a) Total cost $M = -L_{ij} \times \rho_{ij} + U_{ij} \times \lambda_{ij}$. (b) Alternate feasible flow that does not violate the flow conservation of node i or j . Total cost $M' = -L_{ij} \times (\rho_{ij} - 1) + U_{ij} \times (\lambda_{ij} - 1) = M + L_{ij} - U_{ij}$, since $L_{ij} < U_{ij}$ then $M' < M$ | 102 |
| 4.15 | Graphical illustration of finding a_i^* values from the residual network flow graph. (a) Dual network flow graph. (b) Residual graph formed after solving dual problem. (c) a_i^* values found for each node i along with resulting b_{ij} values for each input edge. | 103 |
| 4.16 | Illustration of area penalty with depth reduction and inverted edges. | 104 |
| 4.17 | Illustration of how reducing the depth of a path impacts decisions along other paths and partitions. | 105 |
| 4.18 | Illustration of area-depth relationship for function $F(b_{ij})$ | 106 |
| 4.19 | Depth assignments to inputs after delay budget assignments. (a) Delay budgets assigned to each partition input. (b) Depth adjustment found from Equation 4.18. Assignments to each partition input are used to drive the resynthesis engine. | 108 |
| 5.1 | Generalized flow using ECOs. | 113 |
| 5.2 | Characteristic function derivation for 2-input AND gate. | 117 |
| 5.3 | Cascaded gate characteristic function: top clauses from AND gate; bottom clauses from OR gate. | 118 |
| 5.4 | Automated ECO flow: a MAX-SAT based netlist localization step followed by a SAT based Netlist Modification step. | 119 |
| 5.5 | Maximum satisfiability solution to localization. | 120 |
| 5.6 | Illustration of localized node ξ . Modification can be applied by replacing $H(\mathbf{X}_p)$ with some arbitrary function $F_c(\mathbf{X}_s)$ | 123 |
| 5.7 | Circuit to be modified | 125 |

5.8 Functional dependency example 128

5.9 Deriving global function $F = h(g_1, g_2)$ using dependency check construct. . 129

5.10 Searching for basis functions flow. 132

5.11 Random simulation example. 132

5.12 Circuit reduction. 134

5.13 Example changes in HDL code. 137

5.14 Impact of output removal on SAT solver runtime for circuit b15. 142

7.1 High-level overview of ZDD Unate Product operation with pruning for K . . 150

7.2 Illustration of dominator cut removal in BDD versus ZDD. 151

7.3 Larger example of illustration of dominator cut removal in BDD versus
ZDD. Picture taken from the CUDD BDD/ZDD package [Som98]. 151

List of Tables

| | | |
|------|---|-----|
| 2.1 | Conversion rules for CNF Construction. | 34 |
| 2.2 | Characteristic functions for basic logic elements [Smi04, ch.2]. | 42 |
| 3.1 | Detailed comparison of BddCut cut generation time against IMap and ABC. IMap could not run for $K \geq 8$ | 69 |
| 3.2 | Detailed comparison of BddCut cut generation time against ABC for $K \geq 8$ | 70 |
| 3.3 | Average ratio of $\frac{IMap}{BddCut}$ cut generation times. IMap could not be run for $K \geq 8$ | 70 |
| 3.4 | Average ratio of $\frac{ABC}{BddCut}$ cut generation times. | 70 |
| 3.5 | Average ratio of $\frac{ABC}{BddCut}$ memory usage. | 70 |
| 3.6 | Runtime comparison of BddCut with ABC on circuit leon2 (contains 278,292 4-LUTs). | 71 |
| 3.7 | FPGA channel-width reduction using EdgeFlow with Cut-Height (<i>Delay</i> values in nanoseconds). | 72 |
| 3.8 | Detailed comparison of area and runtime of $FBDD_{new}$ against FBDD and SIS for $K = 8$. $\frac{FBDD_{new}}{FBDD \text{ or } SIS}$ | 74 |
| 3.9 | Comparison of area and runtime of FBDD against $FBDD_{new}$ for various values of K , $\frac{FBDD}{FBDD_{new}}$ | 75 |
| 3.10 | Comparison of area and runtime of FBDD ($K = 8$) against AIG Rewrit- ing/Refactoring. | 76 |
| 4.1 | Infimum of several functions | 97 |
| 4.2 | Runtime comparison of ILP and NF formulation, over 100 circuits ran, only largest circuits shown. Run on a Pentium 4, 2.80GHz with 2GB of RAM109 | |
| 4.3 | Impact of budget management framework on area and depth for the IWLS set, only larger circuits shown in detail. | 110 |
| 5.1 | Impact of modifications on circuit performance on a Stratix chip. | 138 |
| 5.2 | Automated correction results on a Stratix chip. | 141 |

List of Acronyms

- AIG** And-Inverter Graph
- ASIC** Application-Specific Integrated Circuit
- BDD** Binary Decision Diagram
- BLE** Basic Logic Element
- CAD** Computer-Aided Design
- CLB** Clustered Logic Block
- CNF** Conjunctive Normal Form
- DAG** Directed Acyclic Graph
- DSP** Digital Signal Processing/Processor
- ECO** Engineering Change Order
- EDA** Electronic-Design Automation
- FPGA** Field-Programmable Gate Array
- GPU** Graphical Processing Unit
- HDL** Hardware Description Language
- HPC** High-Performance Computing
- ILP** Integer-Linear Program
- IWLS** International Workshop on Logic and Synthesis
- LAB** Logic-Array Block
- LUT** Lookup Table
- MAX-SAT** Maximum Boolean Satisfiability
- QoR** Quality of Result
- ROBDD** Reduced Ordered Binary Decision Diagram

SAT Boolean Satisfiability

SOP Sum of Products

SOPC System on Programmable Chip

ZDD Zero-Suppressed Binary Decision Diagram

1 Introduction

1.1 Introduction to Field-Programmable Gate Arrays

Since their introduction in 1985, Field-Programmable Gate Arrays (FPGAs) have become an effective medium to quickly implement a wide range of digital applications. An FPGA is a regular array of programmable logic and routing structures, as illustrated in Figure 1.1. Digital logic is implemented on an FPGA by programming their logic blocks and routing fabric to various configurations.

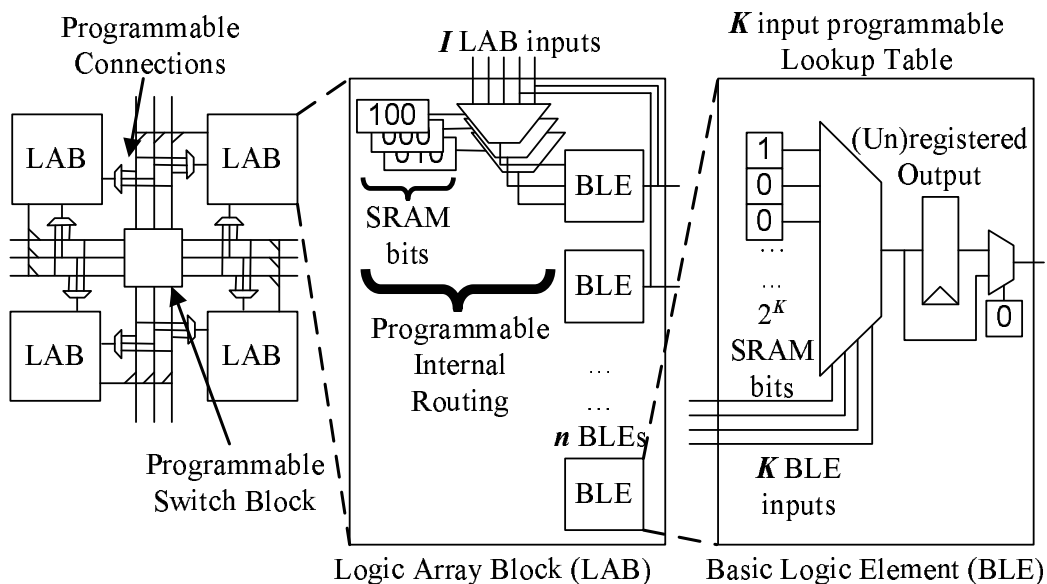


Figure 1.1: A traditional island-style FPGA architecture.

1.2 Motivation and Overview

The programmable nature of FPGAs make them an extremely cost-effective solution for designing digital applications. For example, for a few thousand dollars, one can purchase an FPGA to implement an entire System on Programmable Chip (SOPC) [ter09]. Contrast this to the initial development and fabrication costs for an ASIC which currently costs upwards of a million dollars [ICK04, Lam03]. As this trend has worsened over time, as shown in Figure 1.2, it is expected that FPGA use over ASICs will grow.

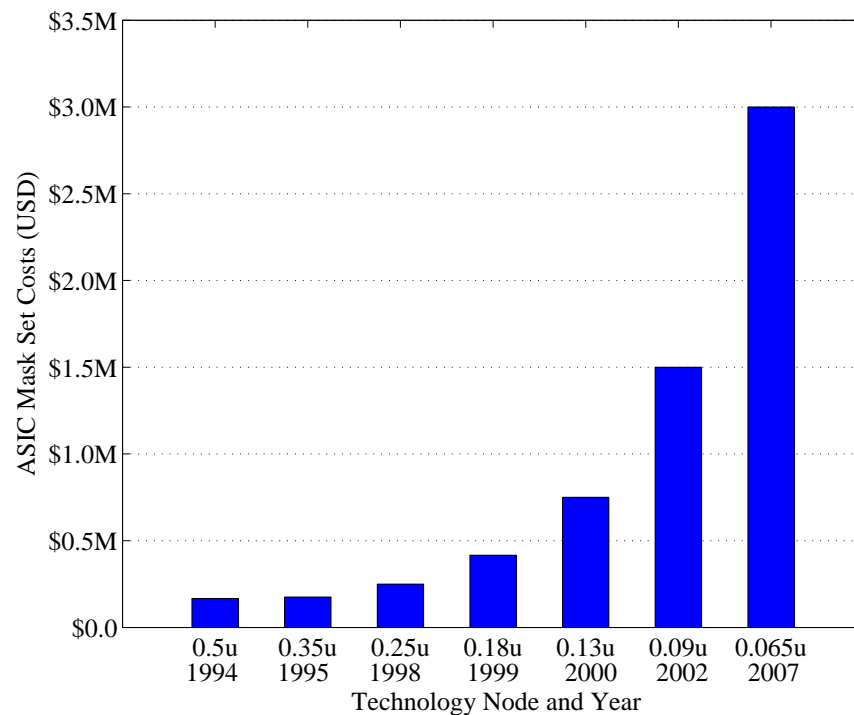


Figure 1.2: Initial costs for fabricating an ASIC as measured by the mask set costs for each technology node from 1994 to 2007[Yan01, RMM⁺03, Lam05].

The cost advantage of FPGAs, however, comes with a penalty in terms of circuit area, power, and performance. Fortunately, a large body of work in the last 20 years has significantly improved both the costs and performance of FPGAs. These can be broken down into architectural enhancements, which explore programmable fabric improvements on

the FPGA [WRV96, RBM99, AR00, LBJ⁺03, LAB⁺05, Mor06b]; and CAD enhancements, which explore algorithms that map digital applications onto the programmable fabric [BRV90, ME95, BR97b, BR97a, MBR99, LSB05b, MSB05, SMB05a, MBV06, MCB06a, MCB06b]. As a result of these improvements, FPGA costs and performance have improved by several fold. Furthermore, in several application domains FPGAs have a significant cost and performance advantage. For example, Figure 1.3 illustrates the costs, measured in terms of silicon area, of an FPGA and DSP implementation of a well known DSP application ¹. This clearly shows that for equal performance, the FPGA is an *order-of-magnitude* cheaper than the DSP [Alt05a, Bie07]. However, even with these compelling advantages of FPGAs, they currently take up *less than 2%* of the semiconductor market [McM08, Wor06].

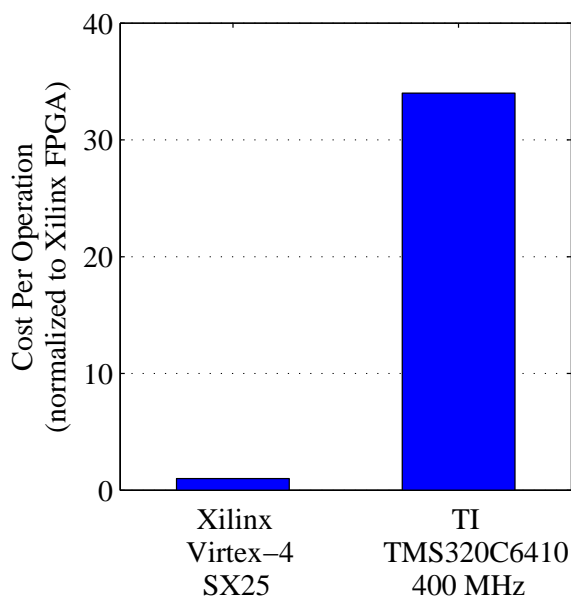


Figure 1.3: A normalized cost comparison between a Xilinx FPGA versus a Texas Instruments DSP [Alt05a, Bie07].

One reason that FPGA growth has been limited is due to the cumbersome nature and high-learning curve of the FPGA design flow. The FPGA CAD flow has generally been

¹The DSP application was an orthogonal frequency-division multiplexing (OFDM) unit, each operation is defined as the computation required to process a single channel [Alt05a, Bie07].

adopted from the ASIC domain, and as a result, FPGA CAD flows suffer from scalability problems typically found in by ASIC CAD tools. In particular, the compile time of the entire FPGA CAD flow commonly takes on the order of hours to complete. Compare this to the typical compile times for DSPs which take on the order of minutes to complete. A second issue is that manual intervention is often required by the user to alter the design and meet constraints: something that requires extensive knowledge of hardware design and the underlying FPGA architecture. Both these factors significantly degrade FPGA design productivity, which deters new entrants from using FPGAs to leverage their benefits; particularly those who are unfamiliar with the digital design flow. This dissertation attempts to address this issue by focusing on two areas. First, we demonstrate how we can improve the productivity of the FPGA CAD flow by reducing its runtime through several fast global optimization techniques. Second, we remove the manual nature of the backend FPGA CAD flow.

To reduce the runtime of the FPGA CAD flow, we focus on logic synthesis. Logic synthesis has a large impact on the final implementation of a circuit and remains an important problem in the FPGA CAD flow. Furthermore, it occupies a significant portion of runtime in a circuit compile ². In this dissertation, we will look at a synthesis flow that leverages Binary Decision Diagrams (BDDs). BDD-based synthesis flows are a powerful means to optimize a circuit for area, power, and delay [YCS00, VKT02, MSB05]. However, one of the primary bottlenecks in BDD-based synthesis flows is their clustering and elimination step. During elimination, redundancies in the circuit are removed and resynthesis regions are defined. Current methods for elimination accomplish these tasks through trial-and-error and, hence, will not scale to modern designs containing hundreds of thousands of logic blocks. This issue could be solved by treating elimination as a covering problem. Doing so would convert the elimination problem to an extremely fast global optimization

²Commercial numbers have not been publicly disclosed by informally have listed anywhere from 30 to 50% of the overall CAD runtime [Alt07, Xil00].

problem rather than a greedy-based heuristic. In order to ensure that the covering problem scales to elimination, we introduce a novel compression technique using BDDs to help compress the cuts necessary when solving the covering problem. This ultimately leads to an order-of-magnitude speedup in the cut generation process and a several fold speedup in the overall synthesis flow with negligible impact on circuit area.

Another technique to improve the runtime of CAD is through partitioning for parallelization on multi-core processors. During partitioning, a circuit is split into several independent subcircuits where each subcircuit is optimized individually. Although this can significantly reduce the runtime of the optimization process, each partition only provides a localized view of the entire circuit and, as a result, optimizing each partition does not guarantee a satisfactory global result. As a solution to this, we formulate an Integer-Linear Program (ILP) that derives partition constraints during optimization. By following these constraints, the localized optimizer will produce a solution with superior quality to that of one with only a localized view of the partition. Unfortunately, our ILP formulation is NP-complete and does not scale to large circuits. To avoid this issue, we show how we can reduce our ILP to polynomial complexity by leveraging the concept of duality. Doing so improves the problem runtime by over 100x. Furthermore, although our reduced problem theoretically has a polynomial complexity with respect to circuit size, empirically we find that it runs in linear time on average. When run on the IWLS benchmark set (the largest academic benchmark set used in logic synthesis), we show that our ILP formulation can improve circuit depth by 11% on average when compared against partitioning-based flows that do not use our technique. Furthermore, our reduction in depth comes with a less than 1% penalty to circuit area.

Finally, in an effort to improve the FPGA design flow, we investigate ECOs. ECOs are an essential methodology to apply late-stage specification changes and bug fixes. ECOs are beneficial since they are applied directly to a place-and-routed netlist which preserves

most of the engineering effort invested previously. In a design flow where almost all tasks are automated, ECOs remain a primarily manual and expensive process. As a solution, we introduce an automated method to tackle the ECO problem. Specifically, we introduce a resynthesis technique using Boolean Satisfiability (SAT) which can automatically update the functionality of a circuit by leveraging the existing logic within the design; thereby removing the inefficient manual effort required by a designer. By using this technique, we show how we can automatically update a circuit implemented on an FPGA while keeping over 90% of the placement unchanged.

1.3 Objective and Contributions

The objective of this research is to enhance the user experience of FPGA design tools through two goals as follows:

1. Enhancing the scalability to the general logic synthesis flow.
2. Removing the manual nature of ECOs in the FPGA CAD flow.

Achieving these goals led to five primary contributions as follows:

1. A novel BDD-based compression technique for cut generation to reduce its runtime and memory use by an order of magnitude.
2. A novel edge-flow heuristic that reduces the number of routing wires by 7% when used during logic synthesis ³.
3. A global covering based approach to solve the elimination problem in logic synthesis reducing the elimination runtime by an order of magnitude reduction.

³As technology scales down beyond 65nm, routing wires begin to dominate the overall circuit area, thus reducing routing wires significantly reduces the cost and improves the delay of the circuit [ZLM06].

4. A formulation of the slack-budget management problem as an Integer-Linear Program followed by a reduction method leveraging duality to reduce our Integer-Linear Program to a network flow algorithm with polynomial complexity. This reduces the slack-budget management problem runtime by two orders of magnitude on average.
5. An automated approach to the FPGA ECO flow that uses Boolean Satisfiability to isolate and resynthesize logic.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 gives a brief overview of FPGA architecture and CAD flow. A brief tutorial on Binary Decision Diagrams (BDDs) and Boolean Satisfiability (SAT) is also be provided. Chapter 3 introduces our novel approach to elimination for logic synthesis. We also describe how we use BDDs to significantly reduce the memory and computational requirements of generating cuts during elimination. Chapter 4 outlines our slack-budget management formulation as an Integer-Linear Program and show how it can be used in conjunction with partitioning to improve circuit depth. Chapter 5 covers our automated approach for ECOs using Boolean Satisfiability. Finally, we conclude in Chapter 6 with a summary and directions for future work.

2 Background

This chapter provides a brief overview of FPGA architecture and CAD. Also, a description of Binary Decision Diagrams (BDDs) and the Boolean Satisfiability (SAT) problem is given. Each section is generally self contained, thus, the reader can skip to sections that they are unfamiliar with to give a better foundation on knowledge when reading the succeeding chapters in this dissertation.

2.1 Terminology

The following section describes some basic terminology used throughout this dissertation. The combinational portion of a Boolean circuit can be represented as a directed acyclic graph (DAG) $G = (V(G), E(G))$. A node in the graph $v \in V(G)$ represents a logic gate, primary input or primary output, and a directed edge in the graph $\langle u, v \rangle \in E(G)$ represents a signal in the logic circuit that is an output of gate u and an input of gate v . For a given edge $\langle u, v \rangle$, u is known as the tail and v is known as the head. A *fanin* for node v , $fanin(v)$, is defined as a tail node for an edge with head v . Similarly, a *fanout* for node v is defined as a head node for an edge with tail v . A *primary input* (PI) node has no fanins and a *primary output* (PO) node has no fanouts. An *internal* node has both fanins and fanouts. Registers can be represented in the DAG if the inputs of the registers are modelled as POs and the outputs of the registers are modelled as PIs. Since registers can form a cycle in a graph, they should not be treated as nodes, as cycles are very difficult to handle in many circuit

optimization algorithms.

A node v is K -feasible if $|fanin(v)| \leq K$. If every node in a graph is K -feasible then the graph is K -bounded. A path, is defined as a sequence of nodes starting at node s and ending at node t such that for each adjacent two nodes, u and v , in a sequence, there exists a directed edge $\langle u, v \rangle \in E(G)$. For any given path, s is known as a transitive fanin for node t and t is known as the transitive fanout for node s . Here, we assume parallel edges do not exist within the graph, since our definitions of edges and paths cannot disambiguate parallel edges and paths (e.g. two edges connecting the same source u and sink v). The length of a path is the sum of the delays of the edges and nodes along the path. If we assume that the delay of each edge are equal, at a node v , the depth, $depth(v)$, is the length of the longest path from a primary input to v and the height, $height(v)$, is the length of the longest path from v to a primary output. Both the depth for a PI node and the height for a PO node are zero. The depth or height of a graph is the length of the longest path in the graph.

When visiting nodes in the graph, they are often visited in *topological order*. In topological order, nodes are visited if and only if all of their fanins have been already visited. This implies that the node traversal occurs from PIs to POs. Reverse topological order is the opposite case where nodes are visited if and only if all of their fanout nodes have been visited already.

A *cone* of v , C_v , is a subgraph consisting of v and some of its nonPI predecessors such that any node $u \in C_v$ has a path to v that lies entirely in C_v . Node v is referred to as the *root* of the cone. The size of a cone is the number of nodes plus edges in the cone. This parameter often determines the computational complexity of operations on the cone since many optimization on a cone of logic require traversing all the nodes or edges within the cone. At a cone C_v , the set of fanins, $fanin(C_v)$, are the set of nodes that are the tail nodes of edges with a head in C_v and the set of fanouts, $fanout(C_v)$, are the set of nodes that are the head nodes of edges with v as a tail. With fanins and fanouts so defined, a cone can

be viewed as a node, and notions that were previously defined for nodes can be extended to handle cones. Notions such as $depth(\cdot)$, $height(\cdot)$ and K -feasibility all have similar meanings for cones as they do for nodes.

A *cut* of a node v is the set of fanin nodes to a cone whose root is v . Thus, every cone defines a cut and there is a one-to-one correspondence between each cone and cut. Note that we are using the term cut in a different manner than what is traditionally used in graph theory, where a cut typically is defined as a set of edges, which separates two sections of a graph. In our case, we are using the source nodes of the edges to define our cut. A cut is K -feasible if it contains at most K distinct nodes. Assuming a given node v has only 2 fanin nodes, u and w . A cut for node v can be created by concatenating two cuts from the fanin nodes u and w . The concatenation operation can be represented using the $*$ symbol where if c_v , c_u , and c_w are cuts for nodes v , u , and w respectively, $c_v = c_u * c_w$.

A *net*, n , is defined as a set of edges with a common tail u . Here, u is known as the *driver* or *source* node of net n and the set of head nodes v are the *sinks* of net n . A net can be thought as a set of wires which connect a source node to a set of sink nodes. A *netlist* is a set of nets which define all of the connections and nodes within a circuit.

2.2 FPGA Architecture

2.2.1 Programmable Logic

The fundamental building block of an FPGA is the Basic Logic Element (BLE), as illustrated in Figure 2.1. This traditionally has consisted of a Lookup Table (LUT) and a register that can be by-passed via a programmable multiplexer. By programming the SRAM bits in the LUT, any logic function of up to K variables can be implemented. Determining an ideal value of K for the LUT is the main focus when designing a BLE. Having a large value of K is beneficial since this increases the amount of logic that can be packed in the

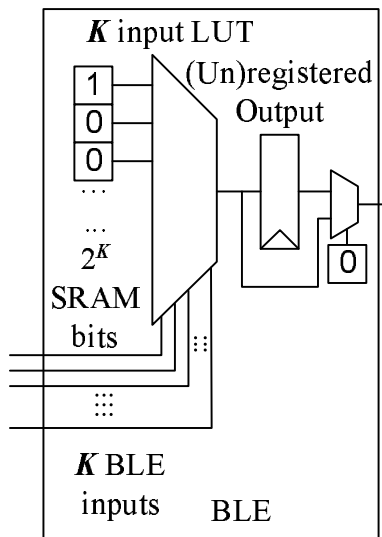


Figure 2.1: A Basic Logic Element (BLE) consisting of a Lookup Table (LUT) and a configurable register.

BLE and reduces the number of BLEs along the critical path of the circuit. However, each additional input to the LUT doubles its size thus finding a good balance between delay and area is necessary. Previous work has shown that BLEs containing 4-input LUTs result in the best area-delay balance [AR00]; though more recently, LUTs with 5 or 6 inputs have been favoured to improve circuit delay [Mor06b, LAB⁺05] at the cost of some area.

BLEs are connected together via wire segments and programmable switches. However, there is a significant delay penalty associated with each connection switch a signal has to pass through. To mitigate this problem, FPGA architectures have adopted a hierarchical structure, where BLEs are clustered together into larger blocks known as Logic-Array Blocks (LABs) (the term Clustered Logic Block (CLB) is also commonly used). The connection fabric within a LAB is often an order of magnitude faster than the interconnect between LABs [RBM99]. Thus, by packing critical portions of a circuit into a small number of LABs, circuit delay improves dramatically. An example of the hierarchical LAB structure is illustrated in Figure 2.2 where each LAB contains n BLEs and I inputs. Previous work has shown that the number of required cluster

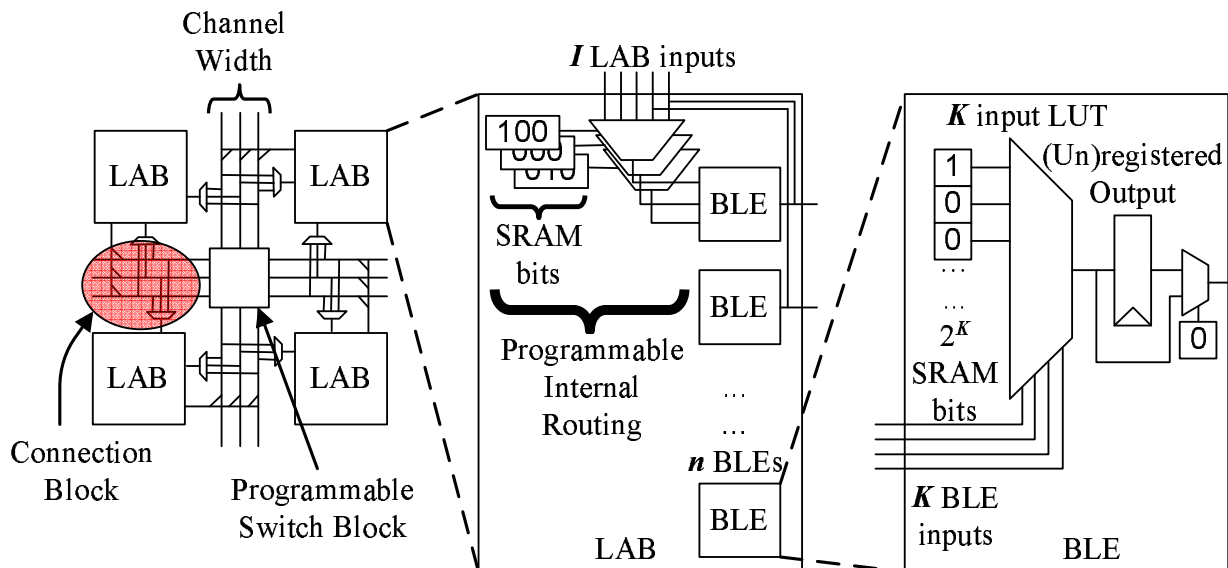


Figure 2.2: The hierarchical structure of an FPGA.

inputs I can be significantly lower than the maximum value of $K \times n$ for full logic utilization within a LAB [BR97a, RBM99, AR00]. In particular, for BLEs containing 4-input LUTs, previous studies has shown that a good number of LAB inputs is $I = 2n + 2$ [BR97a, RBM99, AR00].

As Figure 2.2 shows, LABs are connected together with discrete routing segments where each segment can host at most one inter-LAB signal. Connections between segments are made using programmable multiplexers to determine which segments attach to a given LAB. The number of segments found in a single channel is known as the *channel-width* and the channel-width together with the programmable switch blocks form the FPGA routing fabric. Previous work has explored both the routing segment structure [BR99, LBJ⁺03] and connections blocks [CFK96, Wil97] which yield good delay at a reasonable cost.

2.2.2 Commercial Architectures

In order to improve the versatility and performance of FPGAs, commercial FPGA architectures often add specialized blocks onto the FPGA which works in conjunction with the

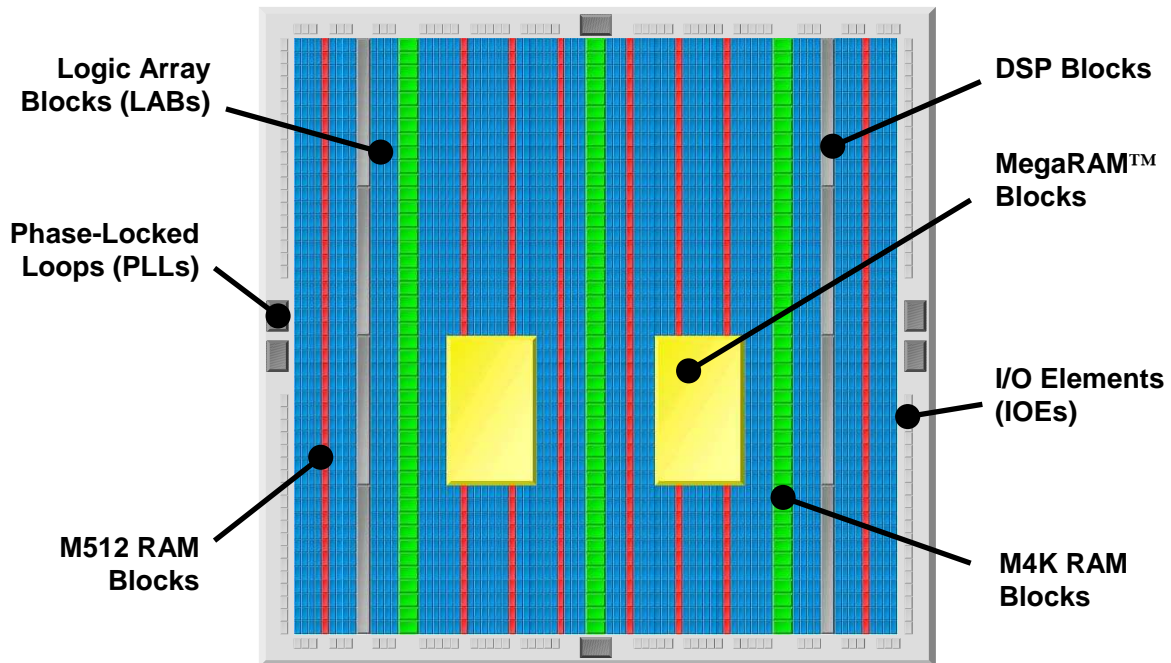


Figure 2.3: Altera's Stratix commercial FPGA [LBJ⁺03].

programmable logic. For example, Figure 2.3 has a high-level overview of Altera's Stratix FPGA [LBJ⁺03]. This consists of BLEs and programmable interconnect, along with memory and dedicated "hard" blocks. These hard blocks are high-performance structures which can perform various arithmetic operations. An example of a typical hard block is shown in Figure 2.4 which implements a multiply-accumulate operation. Since these structures lack any programmable logic, they are much faster and smaller than a fully programmable alternative. This both reduces the costs and enhances the performance of the FPGA. This has led to more studies to both quantify the benefits of using hard blocks on FPGAs and explore new structures that may be beneficial [JR05, LKJ⁺09].

2.3 FPGA CAD

The FPGA CAD flow has a major impact on performance of applications running on FPGAs and each step is coupled very tightly to the underlying FPGA architecture. These

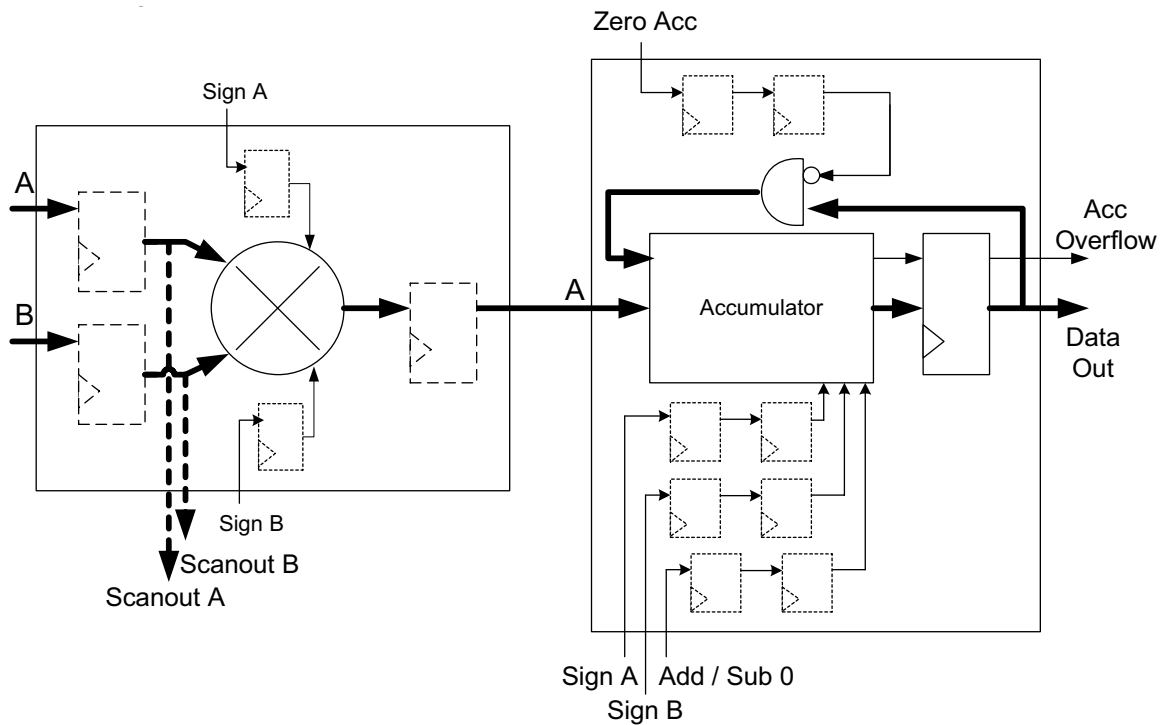


Figure 2.4: A multiply-accumulate “hard” block on the commercial Stratix FPGA [Alt05b].

steps are highlighted in Figure 2.5 and determine how efficiently a design gets maps onto the programmable fabric of the FPGA.

2.3.1 Logic Synthesis and Technology Mapping

Logic synthesis is the first step in the CAD flow after a high-level description is converted to a *technology independent* netlist. We define a technology independent netlist as a circuit representation where each node represents a Boolean function that is not tied to any specific technological implementation. The goal of logic synthesis is to create an optimized netlist consisting of basic gates, such as 2-input AND and OR gates [ESV92, MCB06a]. An example of this is shown in Figure 2.6. In Figure 2.6(a), an unoptimized netlist is passed to logic synthesis resulting in a less costly implementation as shown in Figure 2.6(b). Several common techniques are used during logic synthesis to improve the area and delay of

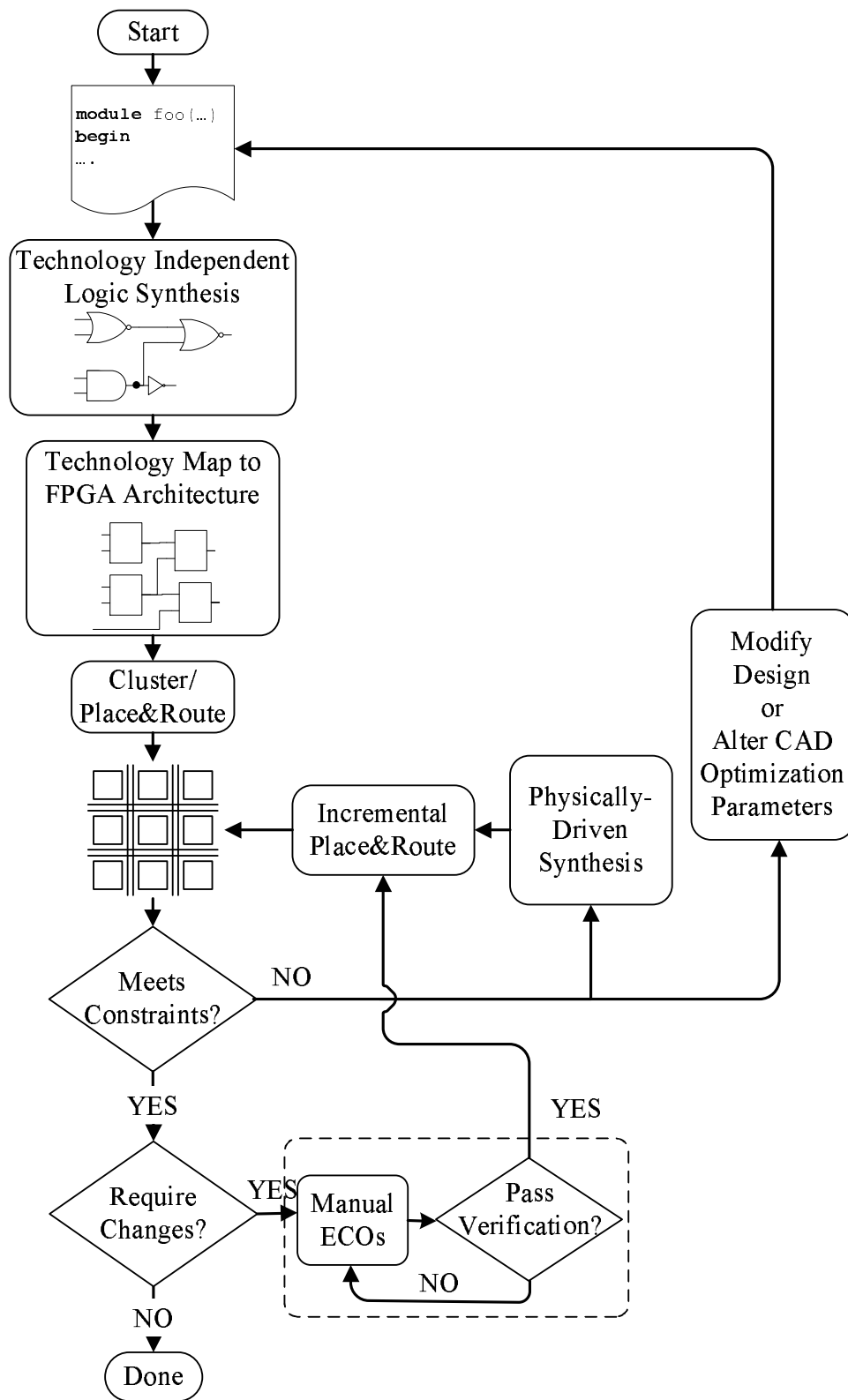


Figure 2.5: A generic CAD flow for FPGAs.

a circuit including two-level minimization [BHMSV84], algebraic division [BM82], functional decomposition [Ash59, RK62, JJHW97, YSC99, YCS00], rewiring [SB98, CLL02, LJHM07], and rewriting [LSB05b, MCB06a].

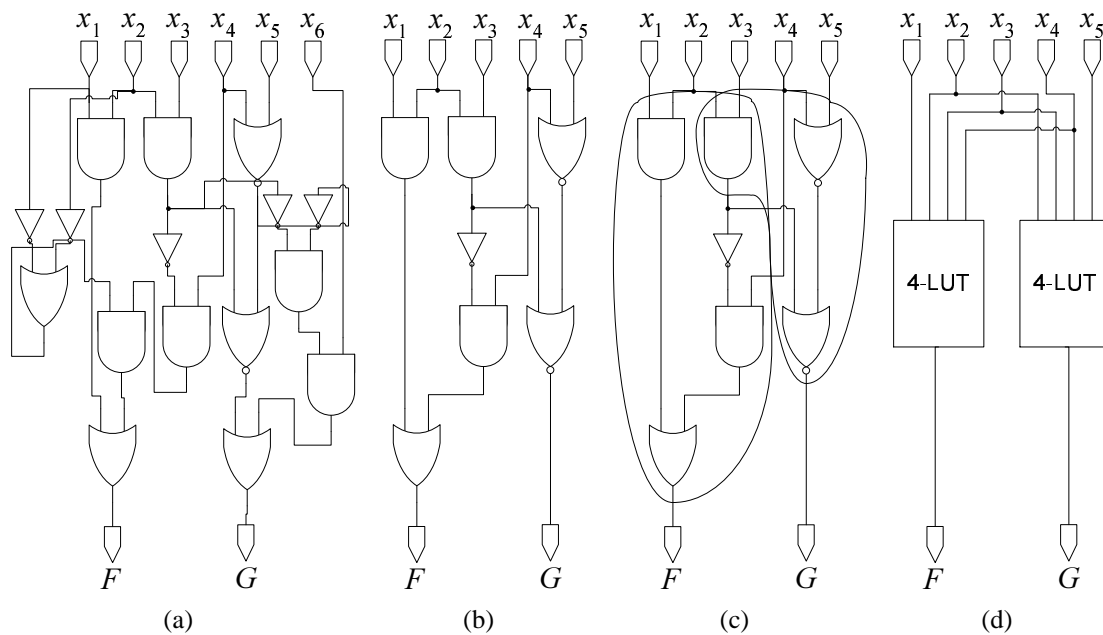


Figure 2.6: An illustration of the logic synthesis process and technology mapping. (a) An unoptimized netlist. (b) An optimized netlist. (c) Identification of nodes to pack into a LUT for technology mapping. (d) Technology mapped circuit to 4-input LUTs.

Once optimized, the netlist is then passed to the technology mapper which maps the basic gates into technology specific nodes, such as 4-input LUTs [MBV06] as illustrated in Figure 2.6(d). When mapping to LUTs, the goal is to pack as much logic as possible into each LUT. This is beneficial since it minimizes the number of LUTs required to implement the circuit. Other metrics such as delay and power should also be taken into consideration during technology mapping. For example, in [AN06] the authors prove that static power consumption of storing logic “1” versus a logic “0” is asymmetric and this can be leveraged during technology mapping to reduce the FPGA power use.

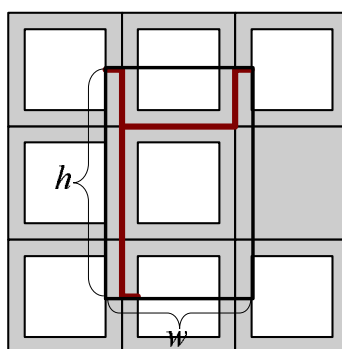
2.3.2 Clustering, Placement, and Routing

Clustering involves packing each BLE produced after technology mapping into a set of LABs. One of the first works to effectively accomplish this was the VPACK tool [BR97a]. In VPACK, a greedy approach is used where BLEs are clustered one at a time. This starts off with a seed BLE as the initial cluster. Next, additional BLEs are added to it based on an attraction function. In VPACK, BLEs with more common connections to the current seed cluster are chosen over other BLEs. This continues until the current cluster utilizes all of the logic or inputs into the LAB. Next, a new cluster is started until all BLEs are clustered into LABs. VPACK was later improved in T-VPACK to include timing information into its attraction function [MBR99]. More recently, routing considerations were taken into account [SMS02, BMYS04] which was shown to reduce the overall area and power use of the resulting netlist. A useful side-effect of clustering is that it can significantly reduce the number of placeable objects in the circuits. This in turn reduces the solution space of the placement problem.

During placement, each LAB is assigned to a single location on the FPGA chip such that the overall interconnect use and circuit timing is minimized. Although several placement techniques have been explored in the past, simulated annealing, as exemplified by VPR [BR97b], has become the de facto standard for FPGA placement. During simulated annealing based placement, individual LABs are randomly swapped between locations. The swaps are accepted if circuit metrics such as wirelength or timing is improved¹. During placement, wirelength is estimated as the summation of the *half-perimeter* length of all nets within the design. An example of the half-perimeter of a net is shown in in Figure 2.7. For timing-driven placement, the delay between two LABs is estimated using a lookup method where the delay between two locations is precalculated empirically and cached in a

¹To avoid getting "trapped" in local minima, simulated annealing will often accept moves which hurt timing or wirelength, so long as this improves the timing and wirelength in succeeding swaps [BR97b].

lookup table [MBR00]. Along with several heuristics such as hill-climbing, simulated annealing based placement has proved to be an extremely effective means to solve the FPGA placement problem. One drawback of simulated annealing is its computational complexity. As a result, placement has traditionally consumed the majority of the runtime in the FPGA CAD flow; however recent advancement in parallelization [LBP08] and partitioning [SR99] has significantly reduced the runtime of FPGA placement to manageable levels.



$$\text{half-perimeter} = h + w$$

Figure 2.7: The half-perimeter wirelength of a net is defined as half of the rectangle perimeter which encompasses all terminals of the net. This rectangle is termed as a *bounding box*.

Once placement completes, routing begins. During routing, LABs are connected together via programmable connections and wire segments. Due to the discrete nature of the FPGA routing problem, FPGA routing is significantly more difficult than the general routing problem found in ASICs. One of the more general approaches to FPGA routing is PathFinder [ME95]. PathFinder connects LABs together such that the wire delay is minimized between connections. During this initial step, wire segments can be used more than once to create connections between LABs. Once all required connections are made, a second routing iteration starts which “rips out” overused wire segments. The cost of the overused segments is incremented to deter the router from overusing them again. This process continues until no more wire segments are overused.

2.3.3 Physically-Driven Synthesis

Physically-driven synthesis, or physical synthesis for short, has become an important step in achieving timing closure after placement. During physical synthesis, logic transformations are applied in conjunction with timing information derived from the placement. An example of this is shown in Figure 2.8. Here we show a basic retiming operation where registers are “pushed” across the circuit to shorten the register-to-register delay of the circuit [LS91, SMB05b]. Retiming is a popular means to optimize circuit delay since it does not change the cycle behaviour of the circuit, thus external interfaces to the circuit do not need to be changed. In Figure 2.8(a), we show an unclustered netlist at the top and the clustered and placed netlist below it. In this figure, each LAB is represented by a large block and can pack at most four BLEs. Assuming that the delay between two BLEs is proportional to their distance, shortening the length of the critical path will improve the circuit delay. This is highlighted in Figure 2.8(b) where register d is pushed ahead of logic element c. Although this shortens the path between logic element c and register d, the current placement is now illegal since the pushed register is assigned to a full LAB. In order to create a legal placement, the cells surrounding register d must be displaced such that the resulting placement is legal as shown in Figure 2.8(c) and Figure 2.8(d). This iterative process between logic transformations and incremental placement continues until the circuit delay converges to a desired value.

2.3.4 Engineering Change Orders (ECOs)

Engineering Change Orders (ECOs) cover a wide range of work which is either used to incrementally improve the delay of a design [CS00] or help modify the behaviour of a design such that circuit delay is maintained [MCB89, CMB08, YSVB07, HCC99, Men05, Xil08]. The work presented in this dissertations falls in the latter category where we focus on late-

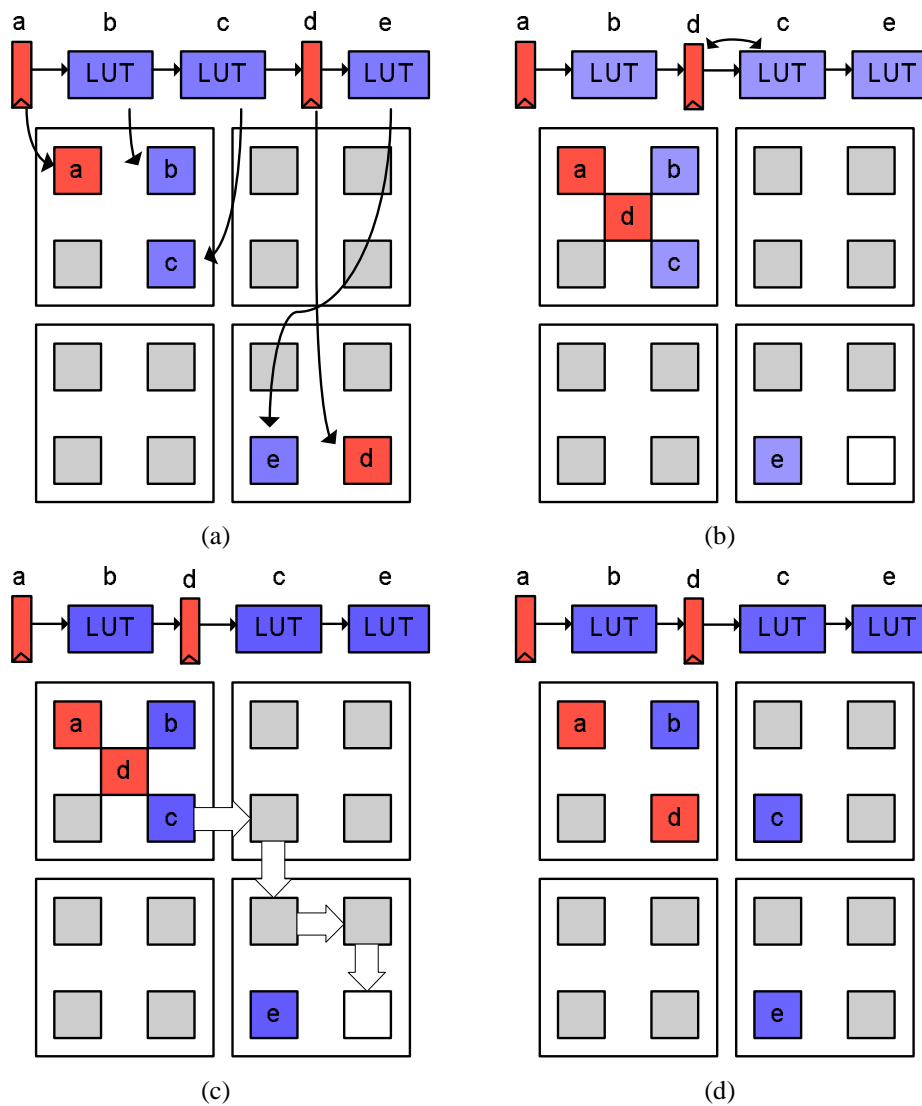


Figure 2.8: Illustration of physically-driven synthesis applying retiming. (a) Current placement with long distance between register *d* and logic element *c*. (b) Post-placement layout-driven optimization using retiming. (c) Incremental placement process for legalization. (d) Final legal placement of optimized netlist.

stage ECOs that are applied directly to a place-and-routed netlist. Late-stage functional changes often occur due to last minute feature changes or due to bugs which have been missed in previous verification phases. The most recent steps toward the automation of the ECO experience include [CMB08] and [YSVB07]. Here, using formal methods and random simulation, the authors in [CMB08, YSVB07] show how netlist modifications can be automated. To apply modifications, the authors use random simulation vectors to stimulate the circuit. Using the resulting vectors at each circuit node, they are able to find suggested alterations to their design to match a specified behaviour. Following their modifications, they require a formal verification step to ensure that their modification is correct. The results of their work is promising where they can automatically apply ECOs in more than 70% of the cases they present.

The technique in [YSVB07, CMB08] requires an explicit representation of any modification, which does not scale to large changes. This is not a problem in ASICs since ECOs requiring major changes are not desired since they are difficult to implement; however in FPGAs, where we can reprogram individual logic cells, large changes can be implemented while maintaining circuit delay. Our approach improves on this where we can handle much larger changes by using a SAT-based approach shown in Chapter 5.

2.3.5 Timing Analysis

Timing analysis occurs at every level of the FPGA CAD flow and is a main driver for circuit optimizations aimed at improving the delay of the design. During timing analysis, every node and edge within the circuit graph is assigned a delay value and critical portions of the circuit are found. An example of this is shown in Figure 2.9 where each node and edge is annotated with a delay value and the longest path delay of the circuit, $Delay_{max}$, is shown. This path is known as the *critical path* of the circuit and determines the minimum clock period (or maximum clock frequency) of the design.

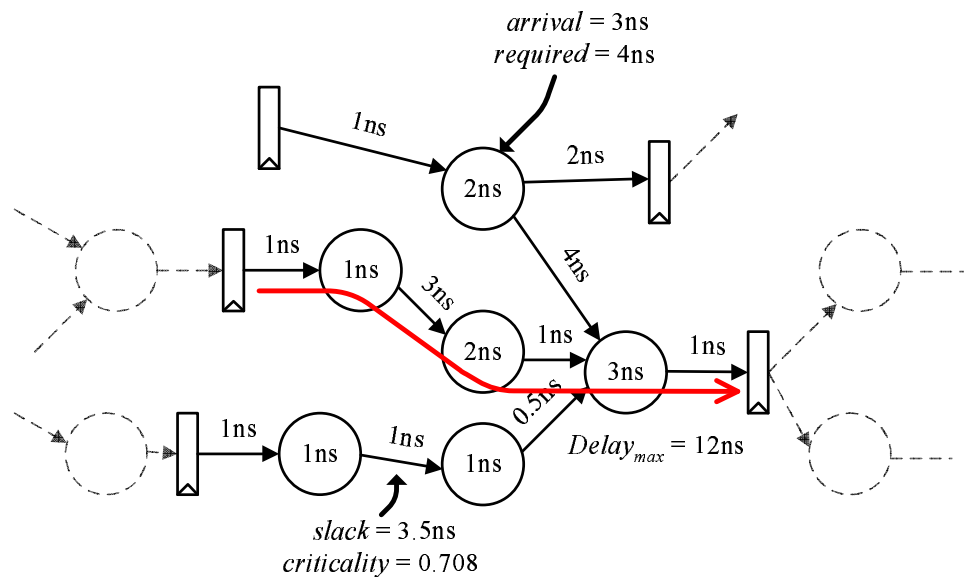


Figure 2.9: An illustration of a circuit graph with static timing values assigned to each node and edge.

One simple method to timing analysis is *static timing analysis*. Static timing analysis uses the static delay values of nodes and edges in the circuit graph to calculate the *criticality* of each edge. Paths that contain many critical edges are optimized for delay.

The *criticality* of an edge inversely relates to the amount of flexibility an edge has to increase its delay where critical edges have very little flexibility to increase its delay without harming the clock frequency of the circuit. This flexibility is based on the notion of *arrival time* and *required time*. The arrival time is defined by Equation 2.1. In Equation 2.1, the arrival time of a node v is defined as the maximal value of the arrival time of a fanin to node v plus the delay of the fanin node and its edge connection, $delay(u) + delay(\langle u, v \rangle)$. Basically, the arrival time of node v is the longest path delay from a primary input to the

node v .

$$arrival(v) = \begin{cases} MAX_{u \in fanin(v)} \{delay(\langle u, v \rangle) + arrival(u)\} + delay(v), & \text{if } v \neq \{PI, register\} \\ 0, & \text{otherwise} \end{cases} \quad (2.1)$$

The required times of each node is defined by a similar relationship as defined in Equation 2.2 which is the longest path delay between node v and a primary output.

$$required(u) = \begin{cases} MIN_{v \in fanout(u)} \{required(v) - delay(\langle u, v \rangle)\} - delay(u), & \text{if } u \neq \{PO, register\} \\ Delay_{max}, & \text{otherwise} \end{cases} \quad (2.2)$$

Using the arrival time and required time, we can define *slack* as shown in Equation 2.3. To find the criticality of each edge, we first normalize the slack to the largest arrival time value, as shown in Equation 2.4. This is known as the slack ratio of the edge. Finally, using the slack ratio, we can define the criticality of each edge as defined by Equation 2.5. Here, the criticality of v is defined as 1 minus the slack ratio of v . The criticality of an edge takes on the value between 0 and 1. Nodes which are attached to critical edges with a value equal to or near 1 should be optimized for delay to help shorten the critical path.

$$slack(\langle u, v \rangle) = required(v) - arrival(u) - delay(\langle u, v \rangle) \quad (2.3)$$

$$slackratio(\langle u, v \rangle) = \frac{slack(\langle u, v \rangle)}{Delay_{max}} \quad (2.4)$$

$$criticality(\langle u, v \rangle) = 1 - slackratio(\langle u, v \rangle) \quad (2.5)$$

The primary difficulty in performing static timing analysis is assigning accurate delay

values to each node and edge. The most simplistic pre-placement delay model uses *unit delay*. In the unit delay model, the delay of each edge is assigned a value of 0 and nodes have a unit delay value. Various other more sophisticated delay estimation methods have been applied for logic synthesis such as *netrange* [LMS04]. The netrange of a net is defined as the difference between the minimum and maximum logic depth of any given node connected to a net where the delay of a net is proportional to its netrange. Unfortunately, effective pre-placement delay models are still lacking and require further work [MCSB06].

Post-placement delay estimation is much easier since the relative position of logic blocks, and hence the approximate length of interconnect segments is well defined. Thus, after placement and routing has completed estimations of delay values in the circuit is very accurate. Leveraging this information can be extremely powerful when trying to improve circuit clock frequency. This occurs during the physically-driven synthesis and Engineering Change Orders step where placement information is used to help guide logic transformations to incrementally increase the circuit clock frequency or modify its behaviour.

2.4 Introduction to Binary Decision Diagrams

A Binary Decision Diagram (BDD) is a directed acyclic graph that represents a Boolean function. A BDD can be understood in terms of Shannon's expansion [Sha38], as shown in definition 2.4.1.

Definition 2.4.1 *Shannon's expansion:*

$$F(x_0, x_1, \dots, x_n) = \overline{x_0} \cdot F(0, x_1, \dots, x_n) + x_0 \cdot F(1, x_1, \dots, x_n) \quad (2.6)$$

In definition 2.4.1, $F(0, x_1, \dots, x_n)$ and $F(1, x_1, \dots, x_n)$ are the negative and positive cofactors of function F with respect to variable x_0 respectively. The negative cofactor can be represented as $F|_{\overline{x_0}}$ and the positive cofactor can be represented as $F|_{x_0}$. Definition 2.4.1

states that any Boolean expression can be represented as a summation of a variable and its complement conjoined with its respective cofactor. Shannon's expansion can be represented graphically as shown in Figure 2.10. Here, the dotted line represents the negative assignment to x_0 , and the solid line represents a positive assignment.

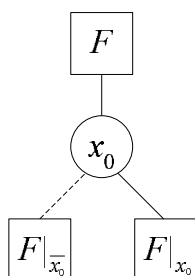


Figure 2.10: Graphical representation of Shannon's expansion.

A BDD extends the graph shown in Figure 2.10 to all variables in function F . This is illustrated in Figure 2.11 where we show the truth table, cube representation, and BDD representation for the Boolean expression $f = \bar{x}_1x_3\bar{x}_4 + x_1x_2\bar{x}_3 + x_1x_2x_4$. In the BDD shown in Figure 2.11, each path from the root node F to a terminal node 0 or 1 represents a minterm in function F . The terminal value of that path represents the value of F assuming the minterm along that path is set to true.

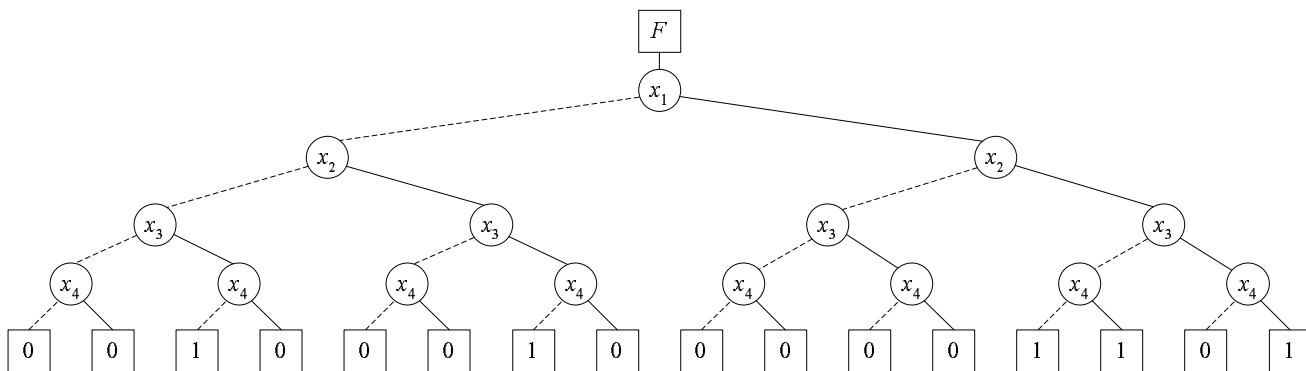
The fully expanded BDD shown in Figure 2.11 is not very useful since it grows exponentially with respect to the number of variables in the function. A more useful form is the Reduced Ordered Binary Decision Diagram (ROBDD). A ROBDD significantly reduces the size of a BDD by leveraging two reduction rules. The first one is illustrated in Figure 2.12. This removes nodes whose positive and negative edge point to the same node. In Figure 2.12(a), we identify a node whose positive and negative edge point to the same value 0. This is applied to all nodes resulting in the graph in Figure 2.12(d). Following this, we remove redundant terminal nodes as illustrated in Figure 2.13(a). When applied to all terminal nodes, the resulting graph is shown in Figure 2.13(b).

| $x_1x_2x_3x_4$ | F |
|----------------|-----|
| 0000 | 0 |
| 0001 | 0 |
| 0010 | 1 |
| 0011 | 0 |
| 0100 | 0 |
| 0101 | 0 |
| 0110 | 1 |
| 0111 | 0 |
| 1000 | 0 |
| 1001 | 0 |
| 1010 | 0 |
| 1011 | 0 |
| 1100 | 1 |
| 1101 | 1 |
| 1110 | 0 |
| 1111 | 1 |

(a) Truth-table

| $x_1x_2x_3x_4$ | F |
|----------------|-----|
| 0-10 | 1 |
| 110- | 1 |
| 11-1 | 1 |

(b) Cubes



(c) BDD

Figure 2.11: Truth-table, cube, and Binary Decision Diagram (BDD) representation of function $f = \bar{x}_1x_3\bar{x}_4 + x_1x_2\bar{x}_3 + x_1x_2x_4$.

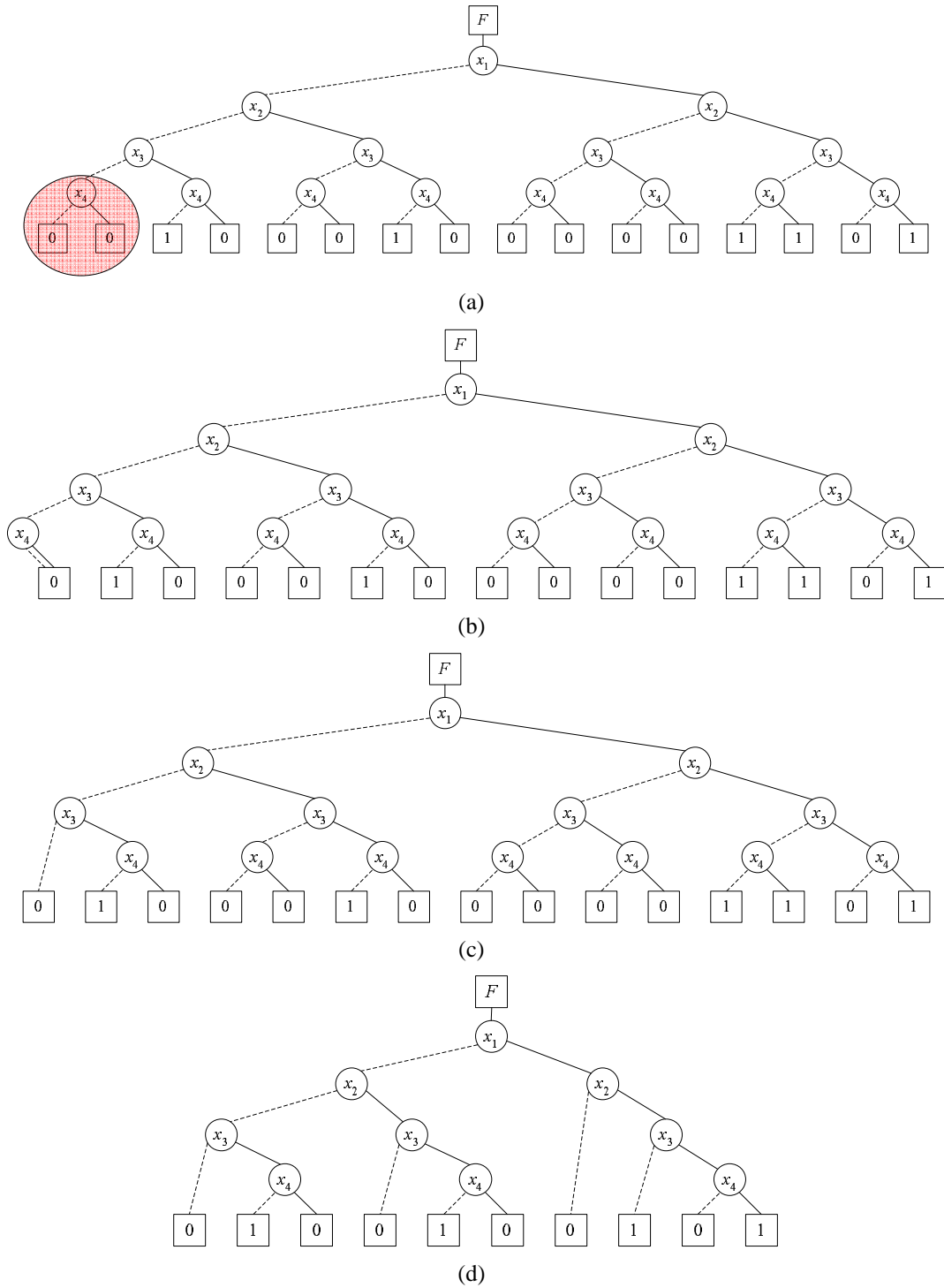


Figure 2.12: An illustration of reduction rule 1: removing redundant assignments.

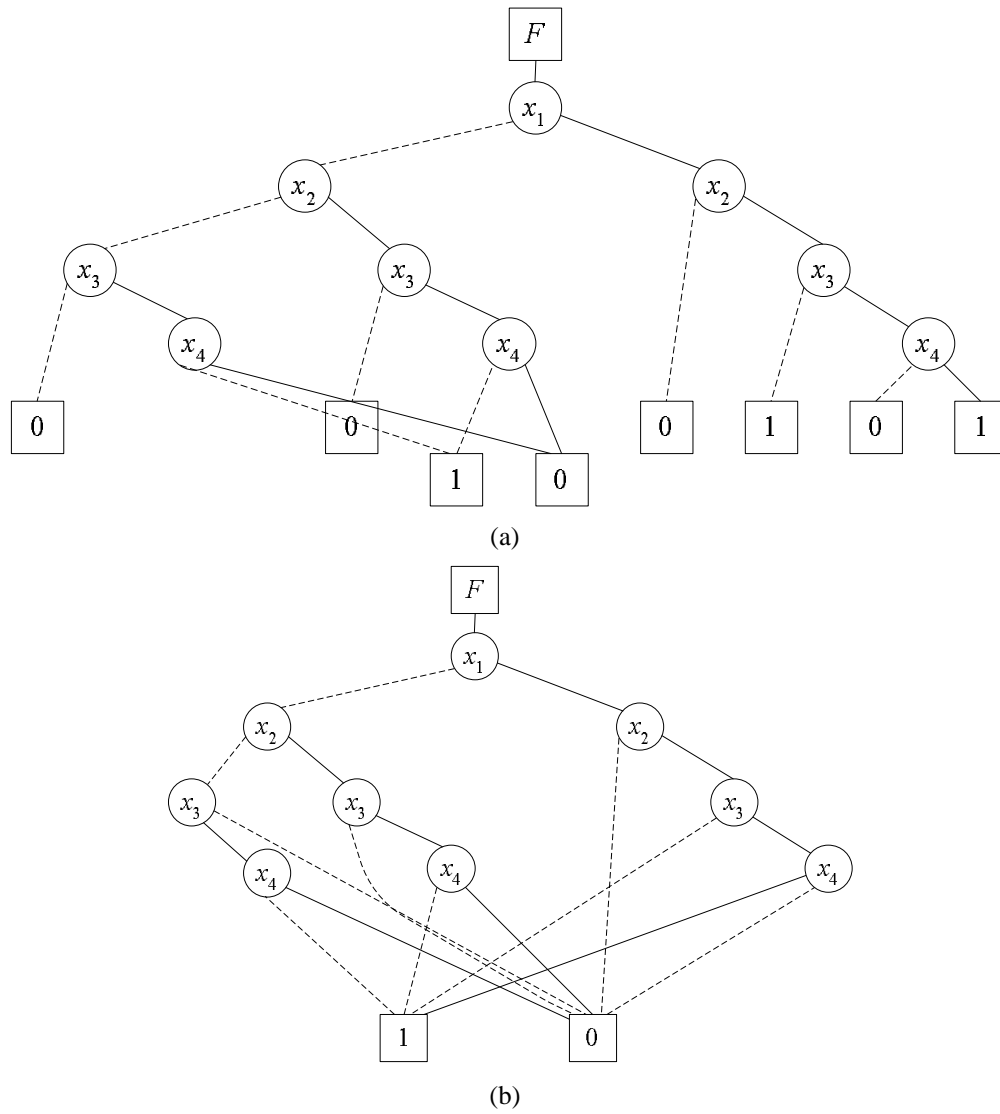


Figure 2.13: An illustration applying reduction rule 1 to the entire graph: removing all redundant assignments.

The second reduction rule is shown in Figure 2.14. Here, nodes that have the same variable label and whose positive and negative edge point to the same nodes can be replaced with a single node. The parent edge of all nodes that are removed are then reattached to the remaining node. This process is shown in figures 2.14(a) and 2.14(b). After applying both of the reduction rules, the resulting ROBDD is shown in Figure 2.14(c). Generally in literature, BDD is used synonymously with ROBDD and we will do the same in this dissertation.

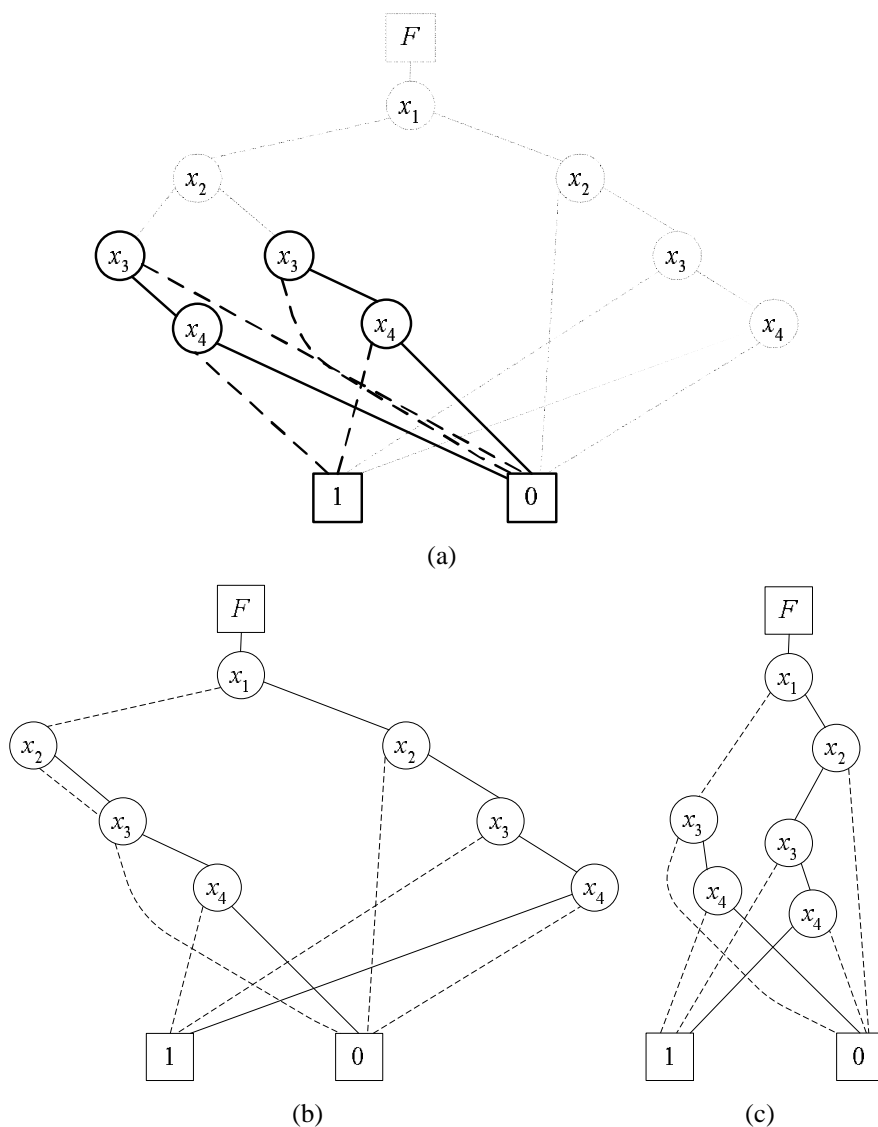


Figure 2.14: An illustration of reduction rule 2: removing duplicate nodes.

An interesting characteristic of BDDs is that every subgraph rooted at a single node and terminated by the 0 and 1 terminal nodes represents a Boolean expression. For example, Figure 2.15(a) identifies the two expressions rooted at each x_3 node. This fact makes BDD construction extremely efficient through dynamic programming. Here, subgraphs can be reused to represent larger BDDs; this saves a significant amount of time and storage space when manipulating BDDs [Som98]. We leverage this fact to improve the algorithms discussed in Chapter 3.

One additional reduction rule leverages the inverted representation of Boolean expressions. Since each subgraph represents a Boolean expression, there may exist subgraphs where are complements of each other. This is shown in Figure 2.15(a). We can reuse the complemented form of a subgraph if we introduce inverted edges into the BDD. If a subgraph in the BDD is pointed to by an inverted edge, this implies that the terminal nodes are inverted. This is shown in Figure 2.15(b) by the finely dotted line.

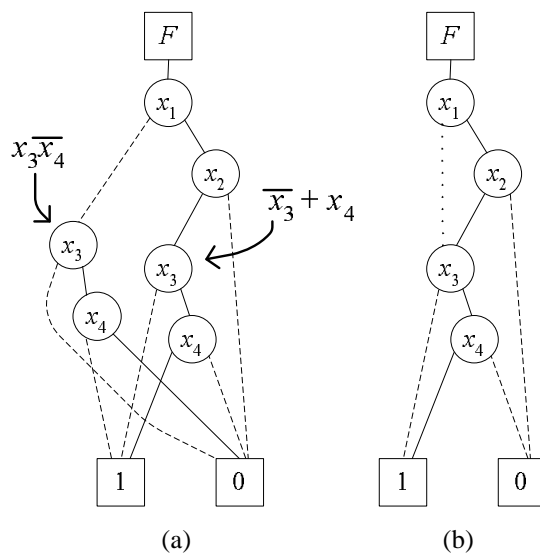


Figure 2.15: An illustration of using inverted edges.

BDDs have a canonical property where for a given variable ordering in the graph, and assuming all of the previous reduction rules have been applied, there exists exactly one representation for a given Boolean function. For different variable order, the canonical

property does not hold. Furthermore, the number of nodes in the BDD varies dramatically with variable order. For example, for the previous function $f = \overline{x_1}x_3\overline{x_4} + x_1x_2\overline{x_3} + x_1x_2x_4$, alternate variable orderings can lead to very different graphs as shown in Figure 2.16. Several heuristics have been used to find a “good” ordering such that the size of the BDD is minimized [Rud93].

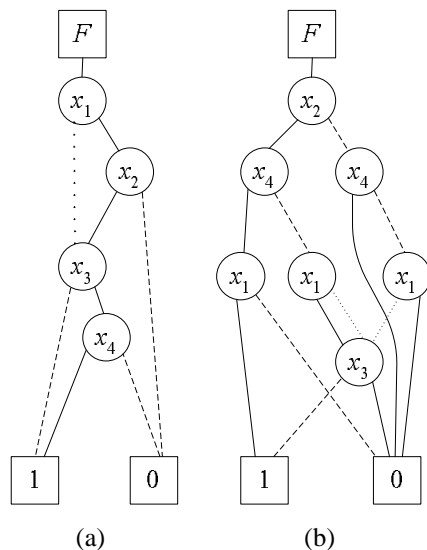


Figure 2.16: An illustration of two resulting BDDs if alternate variable orders are used.

2.4.1 Zero-Suppressed Binary Decision Diagrams

The BDD data structure has been used in a wide range of applications for logic synthesis. However, for some domains, BDDs are not well suited. As a result, there have been several variants to the BDD data structure which have proved to be very useful for various applications [BFG⁺93]. One variation is the Zero-Suppressed Binary Decision Diagram (ZDD). ZDDs differ from BDDs is one of its reduction rules. In BDDs, if the positive and negative edge of a node v point to the same node u , v is removed. In contrast, ZDDs will remove a node if its positive edge points to the 0 terminal node. This variation has major implications on representing Boolean expressions. For example, Figure 2.17 illustrates a Boolean

expression represented as a BDD and as a ZDD.

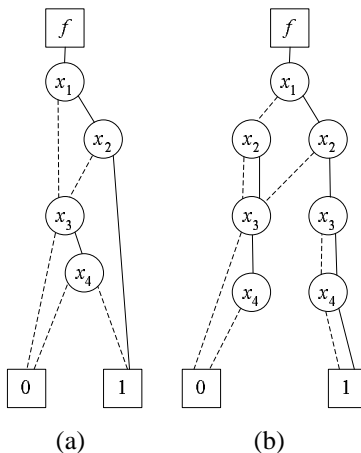


Figure 2.17: BDD and ZDD representation of $f = x_1x_2 + x_3x_4$ [Mis01].

Figure 2.17 highlights that ZDDs are not good at representing generic Boolean functions. However, when representing sets, ZDDs are extremely efficient. For example, consider the set $\{x_1x_2, x_1x_3, x_3\}$. When represented as its characteristic function [CM77], this is represented as $f = x_1x_2\bar{x}_3 + x_1\bar{x}_2x_3 + \bar{x}_1\bar{x}_2x_3$ whose BDD and ZDD representation is shown in Figure 2.18. Clearly, the ZDD representation is much more efficient and as a result, ZDDs have been widely used to quickly manipulate sets within CAD [Min93].

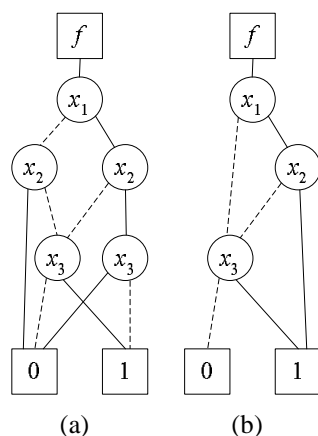


Figure 2.18: BDD and ZDD representation of the characteristic function of $\{x_1x_2, x_1x_3, x_3\}$ [Mis01].

2.5 Introduction to Boolean Satisfiability

The Boolean Satisfiability (SAT) problem was introduced in 1971 as the first problem classified as NP-Complete [Coo71]. Given that P class problems have a polynomial run time, it is generally thought that NP-Complete class problems are harder than P class problems. This statement, although not formally proven, is often described by $NP-C \neq P$ [CLRS01, ch.34]. As a result, efficient heuristics must be applied to reduce the problem space and time requirements for solving NP-Complete problems. Research in the area of SAT is an example of this where heuristics have demonstrated runtime improvements on a range of NP-complete problems in the order of $1000\times$ in comparison to brute force methods.

Given a Boolean formula $F(x_1, x_2, \dots, x_n)$, SAT asks if there is an assignment to the variables, x_1, x_2, \dots, x_n , such that F evaluates to 1. If such an assignment exists, F is said to be *satisfiable*, otherwise, it is *unsatisfiable*. A SAT solver serves to answer the SAT problem.

$$\underbrace{(\bar{A} + B + C)}_{\text{clause}} \cdot (A + B + \underbrace{\bar{C}}_{\text{literal}})$$

Figure 2.19: A Boolean formula in Conjunctive Normal Form.

For practical purposes, modern day SAT solvers work on Boolean formulae in Conjunctive Normal Form (CNF). Boolean formulae in CNF consist of a conjunction of clauses. A clause is a disjunction (logical OR) of literals and a literal is any Boolean variable, $x \in \{0, 1\}$, or its complement. An example Boolean expression in CNF is shown in Figure 2.19. Any Boolean formula can be converted to CNF using basic Boolean algebraic techniques such as those shown in Table 2.1.

In CNF, the problem of SAT can be rephrased to the following: Given a Boolean formula,

| Logic Operation | Symbolic Representation | CNF |
|-----------------|-----------------------------|---|
| De Morgan's Law | $\overline{(A + B)}$ | $(\overline{A} \cdot \overline{B})$ |
| | $\overline{(A \cdot B)}$ | $(\overline{A} + \overline{B})$ |
| Implication | $(A \longrightarrow B)$ | $(\overline{A} + B)$ |
| Equivalence | $(A \longleftrightarrow B)$ | $(\overline{A} + B) \cdot (A + \overline{B})$ |

Table 2.1: Conversion rules for CNF Construction.

$F(x_1, x_2, \dots, x_n)$, in Conjunctive Normal Form (CNF), seek an assignment to the variables, x_1, x_2, \dots, x_n , such that each clause has at least one literal evaluating to 1. Interestingly, when dealing with CNF Boolean formulae whose clauses all have fewer than three literals, SAT is a polynomial problem [CLRS01, ch.34]. Once there exists at least one clause with three or more literal terms, the problem becomes NP-Complete [CLRS01, ch.34].

2.6 Solving the Boolean Satisfiability Problem

One of the earliest works on SAT was done in 1960 by Davis and Putnam in [DP60]. This was refined a few years later by Davis, Logemann, and Loveland in [DLL62] to create the DPLL algorithm. The DPLL algorithm was originally applied to *theorem provers* which attempt to verify a set of propositional statements. The DPLL algorithm is based upon the *splitting rule* which can be understood in terms of Shannon's expansion [Sha38], as shown in Definitions 2.6.1 and 2.6.2.

Definition 2.6.1 *Shannon's expansion:*

$$F(x_0, x_1, \dots, x_n) = \overline{x_0} \cdot F(0, x_1, \dots, x_n) + x_0 \cdot F(1, x_1, \dots, x_n) \quad (2.7)$$

Definition 2.6.2 *Splitting Rule:* $F(x_0, x_1, \dots, x_n)$ is satisfiable if and only if

$F(0, x_1, \dots, x_n)$ is satisfiable or $F(1, x_1, \dots, x_n)$ is satisfiable.

The splitting rule naturally defines a simple recursive algorithm for SAT shown in Algorithm 2.1. Algorithm `sat_solve` accepts three parameters: the function F , a set of variables

```

sat_solve( $F, V, A$ )
begin
  if ( $F(A) \equiv 1$ )
    return satisfiable
  if ( $F(A) \equiv 0$ )
    return unsatisfiable
  // select a variable and assign it to 0
  // to check if  $F(0, \dots)$  is satisfiable
  select a variable  $p \in V$ 
  assign  $V \leftarrow (V - p)$ 
  assign  $p \leftarrow 0$ 
  assign  $A \leftarrow (A \cup p)$ 
  if (sat_solve( $F, V, A$ )  $\equiv$  satisfiable)
    return satisfiable
  // 0 assignment failed, assign it to 1
  // to check if  $F(1, \dots)$  is satisfiable
  assign  $A \leftarrow (A - p)$ 
  assign  $p \leftarrow 1$ 
  assign  $A \leftarrow (A \cup p)$ 
  if (sat_solve( $F, V, A$ )  $\equiv$  satisfiable)
    return satisfiable
  // Formula is not satisfiable under the current assignment.
  unassign  $p$ 
  assign  $A \leftarrow (A - p)$ 
  assign  $V \leftarrow (V \cup p)$ 
  return unsatisfiable
end

```

Algorithm 2.1: A recursive algorithm to solve SAT.

V that define F , and a set of assignments A to the variables in V . First `sat_solve` checks if the current assignment leads to a satisfiable or unsatisfiable solution. If F is in an indeterminate state with respect to the current set of assignments, `sat_solve` selects an unassigned variable (referred to as a *free* variable) and assigns it the value 0. `sat_solve` then recursively checks if the remainder of the formula is satisfiable; if so, it returns satisfiable. If not, it toggles the last variable assignment to 1 and repeats the recursive check. If both assignments lead to an unsatisfiable solution, `sat_solve` returns unsatisfiable.

Algorithm 2.1 is horribly inefficient; however, it is the core of all modern day SAT solvers, which have come a long way since the first DPLL algorithm. Some popular ones include Chaff, Grasp, and SATO [MMZ⁺01, MSS99, Zha97]. The reason for their success stems from heuristics which can drastically reduce the search space of the SAT solver. These heuristics include, but are not limited to, Boolean Constant Propagation, conflict-driven learning, and non-chronological backtracking. For the interested reader, the following sections explain these concepts in detail, but is not necessary to understand the successive chapters in this dissertation.

2.6.1 Heuristics To Solve the Boolean Satisfiability Problem

Boolean Constant Propagation

Boolean Constant Propagation (BCP) reduces the search space by ignoring decisions that will create an obvious unsatisfiable state. BCP works by taking advantage of *unit clauses* to force variable assignments in an *implication* process. A unit clause is a clause with only one free literal where all other literals in the clause evaluate to 0. Thus to satisfy a unit clause, the implication process forces the free literal to evaluate to 1. For example, given the assignment shown in Figure 2.20, x_4 must be assigned to 0 to satisfy the expression.

$$(\overline{x_1} + x_2) \cdot \underbrace{(\overline{x_2} + x_3 + \overline{x_4})}_{\text{unit clause}}$$

Figure 2.20: An example of a unit clause, given that $x_1x_2x_3 = 110$ and x_4 is free.

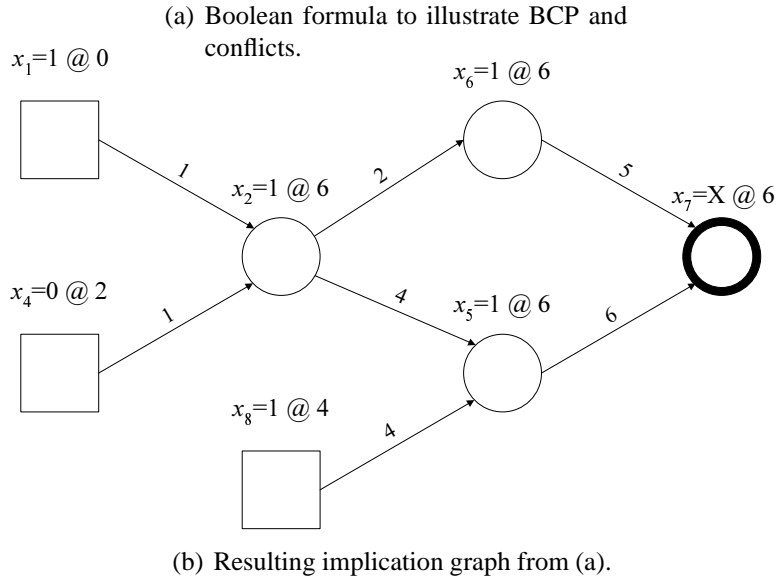
The BCP process can be illustrated through an *implication graph*, which is a directed acyclic graph where nodes represent variable assignments and directed edges represent

implications due to BCP. At each node, the notation $x_i = b @ l$ implies x_i is assigned at time l to value b . For example, in Figure 2.21b variables x_1 , x_4 , and x_8 are assigned at time 0, 2, and 4 respectively. The variables assigned at time 1, 3, and 5 are not shown for simplicity. Each implication edge in Figure 2.21b is labeled to reference the clause subscripts shown in Figure 2.21a. The referenced clause indicates the cause of the implication. For example, the edges labeled 1 imply an implication occurred due to clause 1 ($(\overline{x_1} + x_4 + x_2)_1$) and variable assignments $x_1 = 1 @ 0$ and $x_4 = 0 @ 2$, with the resulting assignment, $x_2 = 1 @ 6$. An interesting feature of BCP is that they may form a chain of implications. For example, referring back to Figure 2.21b, when clause 1 becomes a unit clause, x_2 is forced to 1, which in turn causes two more implications due to clause 2 and 4. This chain of assignments are directly related through implication and it is impossible to satisfy the expression without following the implications to completion where no more unit clauses remain. Thus, variables involved in a chained implication process are grouped and all share the same time label. This is shown in Figure 2.21b where several nodes are given a time value of 6. It is possible for BCP to create inconsistent implications. The node labeled $x_7 = X @ 6$ is an example of this and shows the two inconsistent implications leading to that node. Inconsistent implications are known as conflicts. When a conflict occurs, previous assignments must be undone until the conflict is removed. This process, referred as backtracking, is very costly and heuristics are used such as conflict-driven analysis to minimize the time spent backtracking.

Conflict Analysis

Conflict-driven learning is a process where *conflict clauses* are added to the Boolean formula to remove solution space regions that always lead to an unsatisfiable solution. A detailed algorithm of this can be found in [MSS99] and is not discussed here, but referring back to Figure 2.21, a quick explanation can be presented through an example. As stated

$$F = \dots \cdot (x_1 + x_2)_0 \cdot (\overline{x_1} + x_4 + x_2)_1 \cdot (\overline{x_2} + x_6)_2 \cdot (\overline{x_4} + x_2 + \overline{x_1})_3 \cdot (\overline{x_2} + x_5 + \overline{x_8})_4 \cdot (x_4 + \overline{x_6} + \overline{x_7})_5 \cdot (x_4 + \overline{x_5} + x_7)_6 \cdot \dots \quad (2.8)$$



$$F = \dots \cdot (\overline{x_1} + x_4 + \overline{x_8}) \quad (2.9)$$

(c) Conflict clause added due to conflict in (b).

Figure 2.21: A conflict-driven analysis implication graph.

previously, node $x_7 = X @ 6$ is in conflict due to the conflicting implication edges 5 and 6, thus backtracking must occur to undo this conflict. However, without learning it is possible that this conflict arrangement may occur again; thus, it is beneficial to add information to the Boolean formula to prevent this. Closer inspection of the implication graph in Figure 2.21 shows that the conflict originates at edges 1 and 4 due to assignments $x_1 = 1$, $x_4 = 0$, and $x_8 = 1$. Further analysis of Equation 2.8 shows that assignment $x_1 = 1$, $x_4 = 0$, and $x_8 = 1$ will always cause a conflict. Thus, to prevent this in the future, a learnt clause is added which is shown in Equation 2.9. The learnt clause is a redundant clause and

does not change the meaning of the original Boolean formulae, but it makes it impossible for the assignment $x_1 = 1$, $x_4 = 0$, and $x_8 = 1$ to occur again.

Non-Chronological Backtracking

The process of reversing the most recent assignments due to a conflict is called chronological backtracking (i.e. in-order backtracking). However, often several assignments can be reversed out of order if the source of the conflict can be identified. This is known as non-chronological backtracking. Like conflict analysis, non-chronological backtracking uses conflict information to determine the original source of the conflict and reverse all the decisions to that source. Continuing with the example in Figure 2.21, the original source of the conflict was due to node $x_8 = 1 @ 4$. Thus, the backtracking process should jump to decision level 4 to fix the conflict. This is depicted in the *decision tree* shown in Figure 2.22b which contrasts with chronological backtracking shown in Figure 2.22a. A decision tree represents the variable assignment process where each branch represents an assignment decision and each leaf represents a satisfiable or unsatisfiable solution. In the case of unsatisfiable solutions, the decision tree backtracks up the tree to remove the conflict.

2.6.2 Circuits and Boolean Satisfiability

The benefit of using SAT for problems in CAD is that circuit properties can be quickly evaluated using any efficient SAT solver [McM03, SVV04, LSB05b, SMV⁺07, LJHM07]. However, before a circuit CAD problem can be formulated as a SAT problem, the circuit must be converted to a Boolean formula in CNF form. This can be achieved using a technique first presented in [Lar92], which derives a *characteristic function* of a logic circuit. This CNF representation contain variables representing the primary input, primary output, and intermediate circuit signals and evaluates to 1 if these signals are consistent. For ex-

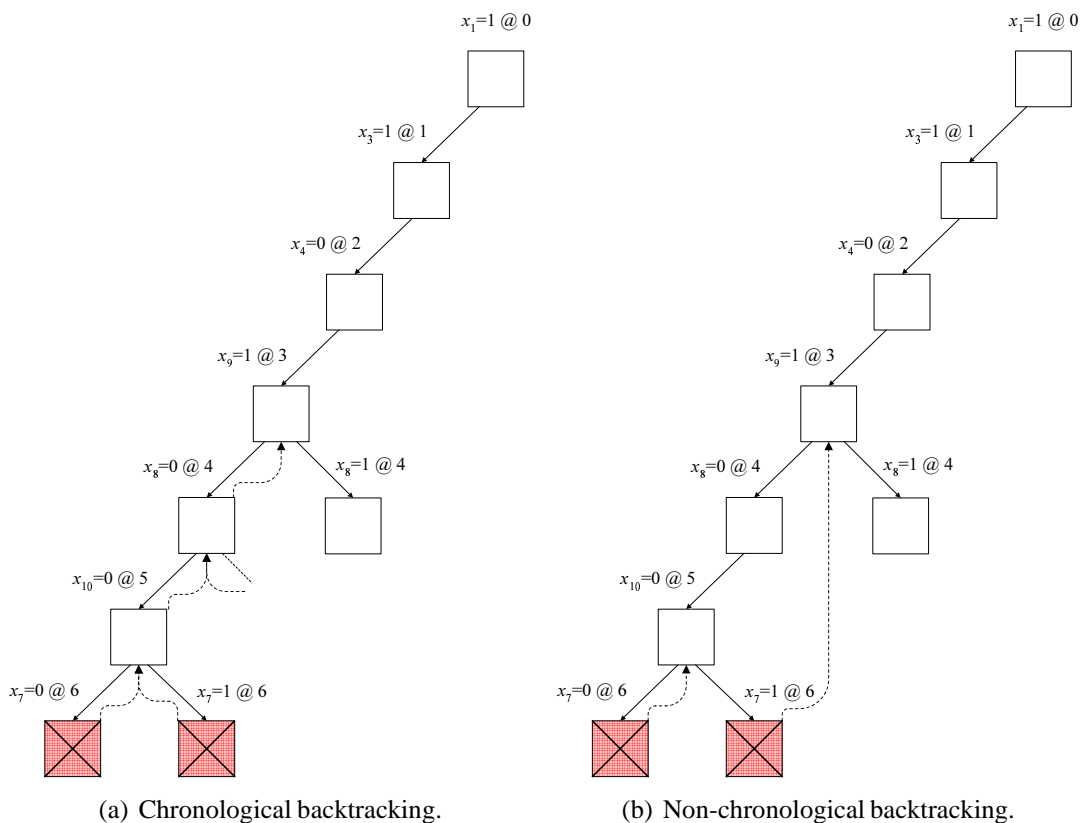


Figure 2.22: Backtracking due to a conflict in Figure 2.21.

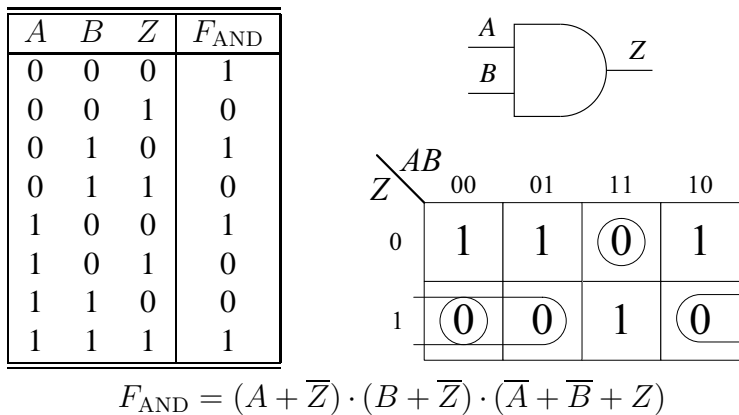


Figure 2.23: A characteristic equation derivation for 2-input AND gate.

ample, consider the AND gate shown in Figure 2.23. The table to the left gives the truth table for the AND gate characteristic function where the onset contains all valid input-output

relations of an AND gate. This can be converted to CNF using any standard minimization technique, such as a Karnaugh map as shown. Table 2.2 lists several other common gates and digital functions with their associated characteristic function [Smi04, ch.2]. Notice FULL-ADD has two outputs to the circuit (i.e. s, c_{out}). The technique shown in Figure 2.23 is directly applicable to multiple output circuits. One simply adds all output variables to the characteristic function truth table where its onset still defines valid input-output vectors.

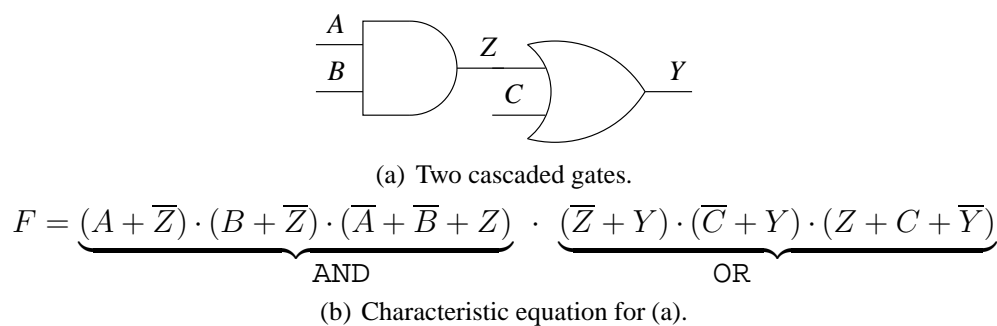


Figure 2.24: A cascaded gate characteristic function.

Deriving CNF functions directly from the circuit input-output relation is only practical for primitive gates and logic blocks where the number of inputs and outputs is small. Fortunately, characteristic equations for larger circuits can be derived iteratively from the conjunction of its subcircuit characteristic functions. For example, Figure 2.24a shows two cascaded gates. Notice the wire connecting the AND gate to the OR gate is labeled with variable Z for CNF construction. The characteristic function of the cascaded circuit is simply the conjunction of the AND and OR gate characteristic functions with variable Z as the logical link between the two functions. The characteristic equation in Figure 2.24b evaluates to 1 if all wire signals are consistent. This includes the primary inputs and outputs as in Figure 2.23, plus any intermediate wire signals (i.e. Z).

| Gate | Function | CNF formula |
|----------|---|---|
| AND | $y = x_1 \cdot x_2 \cdots x_n$ | $\left(\prod_{i=1}^n (x_i + \bar{y}) \right) \cdot \left(\left(\sum_{i=1}^n \bar{x}_i \right) + y \right)$ |
| NAND | $y = \overline{x_1 \cdot x_2 \cdots x_n}$ | $\left(\prod_{i=1}^n (x_i + y) \right) \cdot \left(\left(\sum_{i=1}^n \bar{x}_i \right) + \bar{y} \right)$ |
| OR | $y = x_1 + x_2 + \cdots + x_n$ | $\left(\prod_{i=1}^n (\bar{x}_i + y) \right) \cdot \left(\left(\sum_{i=1}^n x_i \right) + \bar{y} \right)$ |
| NOR | $y = \overline{x_1 + x_2 + \cdots + x_n}$ | $\left(\prod_{i=1}^n (\bar{x}_i + \bar{y}) \right) \cdot \left(\left(\sum_{i=1}^n x_i \right) + y \right)$ |
| XOR | $y = x_1 \oplus x_2$ | $(\bar{x}_1 + \bar{x}_2 + \bar{y}) \cdot (x_1 + x_2 + \bar{y}) \cdot$ $(\bar{x}_1 + x_2 + y) \cdot (x_1 + \bar{x}_2 + y)$ |
| NXOR | $y = \overline{x_1 \oplus x_2}$ | $(\bar{x}_1 + \bar{x}_2 + y) \cdot (x_1 + x_2 + y) \cdot$ $(\bar{x}_1 + x_2 + \bar{y}) \cdot (x_1 + \bar{x}_2 + \bar{y})$ |
| BUFFER | $y = x$ | $(x + \bar{y}) \cdot (\bar{x} + y)$ |
| NOT | $y = \bar{x}$ | $(x + y) \cdot (\bar{x} + \bar{y})$ |
| MUX 2:1 | $y = (s_0 \leftrightarrow 1) ? x_1 : x_0$ | $(s_0 + \bar{x}_0 + y) \cdot (s_0 + x_0 + \bar{y}) \cdot$ $(\bar{s}_0 + x_1 + \bar{y}) \cdot (\bar{s}_0 + \bar{x}_1 + y)$ |
| MUX 4:1 | $y = (s_1 \leftrightarrow 1) ?$ $[(s_0 \leftrightarrow 1) ? x_3 : x_2] :$ $[(s_0 \leftrightarrow 1) ? x_1 : x_0]$ | $(s_0 + s_1 + \bar{x}_0 + y) \cdot (s_0 + s_1 + x_0 + \bar{y}) \cdot$ $(s_0 + \bar{s}_1 + \bar{x}_1 + y) \cdot (s_0 + \bar{s}_1 + x_1 + \bar{y}) \cdot$ $(\bar{s}_0 + s_1 + \bar{x}_2 + y) \cdot (\bar{s}_0 + s_1 + x_2 + \bar{y}) \cdot$ $(\bar{s}_0 + \bar{s}_1 + \bar{x}_3 + y) \cdot (\bar{s}_0 + \bar{s}_1 + x_3 + \bar{y})$ |
| FULL-ADD | $s = a \oplus b \oplus c_{in}$ $c_{out} = (a \oplus b) \cdot c_{in} + ab$ | $(a + b + \bar{c}_{out}) \cdot (\bar{a} + \bar{b} + c_{out}) \cdot$ $(a + c_{in} + \bar{c}_{out}) \cdot (\bar{a} + \bar{c}_{in} + c_{out}) \cdot$ $(b + c_{in} + \bar{c}_{out}) \cdot (\bar{b} + \bar{c}_{in} + c_{out}) \cdot$ $(s + \bar{a} + c_{out}) \cdot (\bar{s} + a + \bar{c}_{out}) \cdot$ $(s + \bar{b} + c_{out}) \cdot (\bar{s} + b + \bar{c}_{out}) \cdot$ $(s + \bar{c}_{in} + c_{out}) \cdot (\bar{s} + c_{in} + \bar{c}_{out}) \cdot$ $(s + \bar{a} + \bar{b} + \bar{c}_{in}) \cdot (\bar{s} + a + b + c_{in})$ |

Table 2.2: Characteristic functions for basic logic elements [Smi04, ch.2].

2.7 Relationship between Binary Decision Diagrams and Boolean Satisfiability

In the previous two sections, we explored the properties of BDDs and SAT. We learnt that the BDD explores properties of a circuit by storing Boolean expressions as a graph; whereas SAT maps a circuit to a characteristic function and then applies a sequential algorithm to traverse the function. BDDs have had a long history in CAD spanning several decades. When applied to CAD, BDD graphs can be manipulated using fast BDD operators and through dynamic programming which can be exploited due to the DAG structure of the BDD. This had made BDDs extremely attractive in the past. However, the problem with BDDs is that they suffer from memory explosion. Here, a given BDD may require an exponentially large number of nodes to represent a given Boolean expression. This has created a wall in extending the application of BDDs to new areas.

Only in the last decade has SAT become popular in CAD. This has primarily been due to the recent innovations started by the zChaff project [MMZ⁺01] which incorporated several heuristics to significantly speed up the SAT process. Unlike BDDs, SAT does not suffer from memory explosion to the extent that BDDs do, since there is a one-to-one correspondence between the characteristic function of a circuit and the circuit size. The characteristic function allows SAT to traverse the CNF to explore various properties of the circuit. The drawback with this is that runtime becomes the limiting factor for SAT, where for some SAT instances, the runtime of solving a problem is intractable.

Thus, there exists a complementary relationship between BDDs and SAT where BDDs explore problems in space and SAT explore properties in time. For smaller problem instances, BDDs are in general a faster means to solve problems than SAT. However, more recently, more problems are finding a SAT-based solution which have been proved to be much more scalable. This does not imply that SAT-based solutions will subsume all BDD-

based solutions in the future, but simply illustrates that SAT and BDDs will continue to play an important and synergistic role within CAD.

2.8 Summary

An overview of the primary architectural features of an FPGA was given. First the BLE was introduced as the basic building block of an FPGA. Following this, the LAB was introduced and the benefits of using an hierarchical architecture was discussed followed by a brief overview of the programmable routing fabric.

Following an architectural overview, the FPGA CAD flow was described. This dissertation focuses primarily on improving the CAD flow where our goal is to improve the user experience and scalability of existing CAD algorithms; specifically in logic synthesis and ECOs.

The chapter then concludes with a brief overview of the Binary Decision Diagram data structure and Boolean Satisfiability problem. These two problem domains share a complementary relationship: BDDs solve Boolean expressions in space through a graph data structure whereas SATs solves Boolean expressions in time through several algorithmic approaches. In general, SAT based approaches to solving CAD problems has emerged as more scalable than BDD based approaches and its application in CAD has grown in recent years to many problem domains.

3 Improving BDD-based Logic Synthesis through Elimination and Cut-Compression

In a progression towards a more scalable and manageable FPGA CAD flow, we investigate methods to improve logic synthesis. We focus on logic synthesis since it has the most flexibility to modify the circuit structure and has a large overall impact on the final circuit performance. Furthermore, logic synthesis occupies a significant portion of runtime. In this chapter we present means to help reduce the runtime of logic synthesis without impacting the Quality of Result (QoR) of the optimized netlist.

The rest of the chapter will be organized as follows. Section 3.1 will provide a brief overview of the logic synthesis flow. Section 3.2 will describe some problems associated with the existing logic synthesis flow. Specifically, we show how existing techniques do not scale to large circuit designs. Section 3.3 will outline our solution to the problems we discuss in the proceeding sections. Here, we propose a scalable approach to a key step during logic synthesis: elimination. In doing so, we introduce a novel BDD-based cut generation method that is an order of magnitude faster than previous approaches. Also, we describe a new structural metric referred to as *edge flow* which helps reduce circuit area. Section 3.4 will give an overview of our results and the chapter will conclude in Section 3.5.

3.1 Introduction to Logic Synthesis

In the FPGA CAD flow, logic synthesis occurs after a high-level description is broken down into basic logic gates. During logic synthesis, the goal is to optimize the circuit netlist such that the resulting netlist has a more cost-effective implementation or has better performance. Although there have been several generic flows to the logic synthesis problem [ESV92, CJJ⁺03, MCB06a], in this work we focus on BDD-based synthesis flows. BDD-based logic synthesis has gained popularity due to its compact representation of logic function and scalable transformation techniques [YCS00, WZ05]. During BDD-based synthesis, several generic steps are generally taken as shown in Figure 3.1. The first step taken is *elimination*. The purpose of elimination is to remove redundancies within the circuit netlist and form resynthesis regions in which succeeding optimizations can be applied. After elimination, *decomposition* and *shared-extraction* can occur. During this step, the resynthesis regions formed during the elimination step are resynthesized such that the resulting circuit is optimized. This process continues until the circuit meets some cost metrics or no further improvement is achieved, after which the netlist is then passed onto technology mapping.

3.2 Motivation

There has been a large body of work which has improved the decomposition and shared-extraction step in logic synthesis; both to improve its QoR [BM82, Ash59, RK62, JJHW97] and to improve its scalability [YSC99, YCS00, WZ05, MCB06a]. However, little focus has been given to the elimination step. As a result, elimination has emerged as the primary bottleneck for scalability in BDD-based synthesis engines where it has been reported to take up to 70% of the runtime [WZ05]. During elimination, regions are grown from a seed node where its fanins are successively collapsed into the node in a greedy fashion.

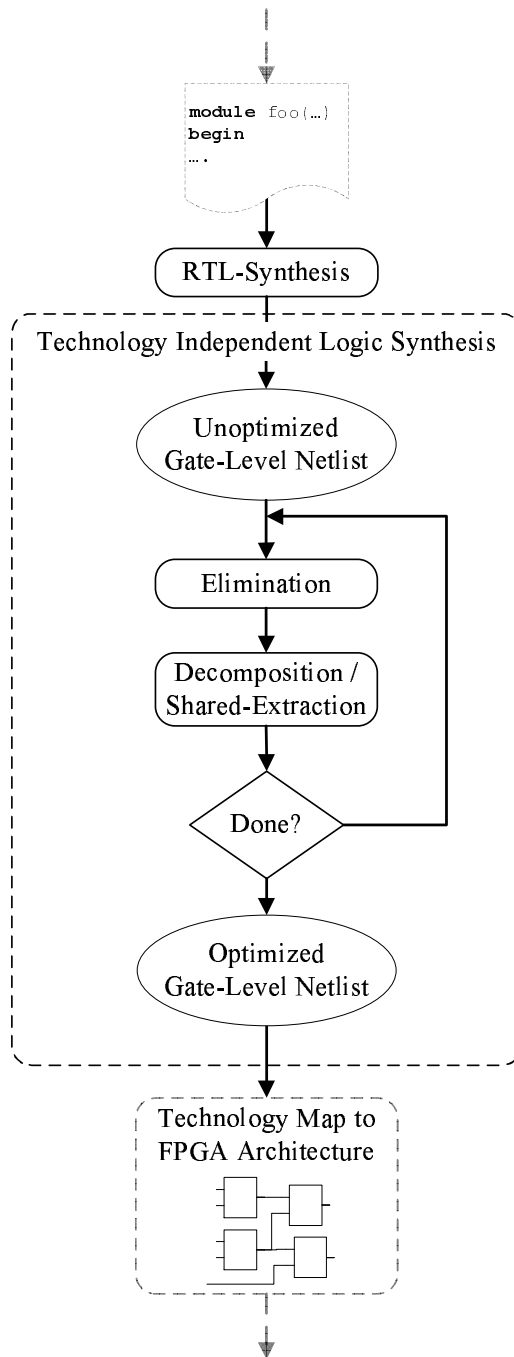


Figure 3.1: An illustration of the generic logic synthesis flow.

If the cost of the logic is reduced after the collapse, the collapse is committed into the netlist, otherwise the collapse is undone. We measure simplification in terms of BDD nodes, where we aim to reduce the number of BDD nodes after a collapse. This process is illustrated in Figure 3.2. Figure 3.2(a) shows an unoptimized gate-level netlist. This netlist is transformed to its BDD-based netlist representation in Figure 3.2(b) where each basic gate in the original netlist is mapped to a logic node, whose logic function is represented by a BDD. Next, a collapse operation is illustrated in Figure 3.2(c) and 3.2(d) starting from a randomly chosen seed node. Notice that after the collapse operation completes, the number of resulting BDD nodes is reduced from 12 to 9 (terminal '1' and '0' nodes are not included in the count). In situations where there is an increase in BDD nodes, the collapse would be undone. After several collapse operations are done, the resulting netlist is shown in Figure 3.2(e) resulting in 4 BDD nodes contained in a single elimination region (for larger circuits, several elimination regions would result). Following this, a decomposition is applied to decompose the netlist back into basic gates as shown in Figure 3.2(f) and 3.2(g).

In order to get to the simplified netlist in Figure 3.2(e), a total of five collapse operations need to be applied. In general, the number of collapse operations during elimination can be very large which is costly, particularly when a collapse needs to be undone. Ideally, forming each elimination region should occur as one large collapse operation consisting of several nodes, without the need to undo any collapse operations. For example, being able to go directly from Figure 3.2(b) to Figure 3.2(e) would significantly reduce the runtime of the overall elimination step. One way to accomplish this is to solve elimination as a covering problem. As a covering problem, elimination regions are created by covering the netlist. Following this, each cover is collapsed into a single node, which is significantly faster than collapsing several nodes individually.

The covering problem has traditionally not been used to solve elimination since there are

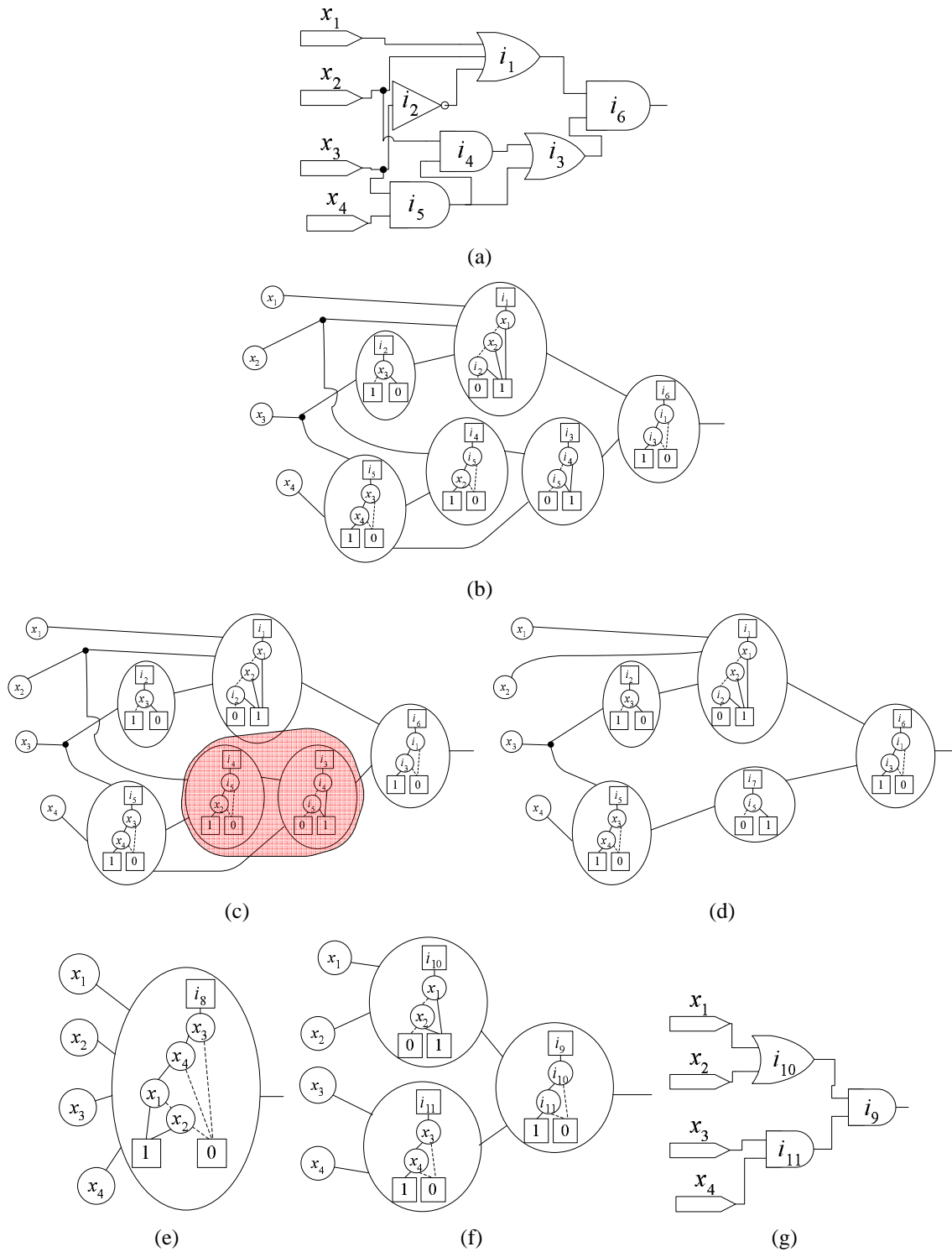


Figure 3.2: An illustration of an elimination operation followed by a decomposition.

several problems associated with it. The first problem is related to scalability. The covering problem relies on a well known cut-generation step to generate its covers. However, cut generation does not scale for cuts beyond a size of 6. For example, to generate all 7-input cuts for circuits in the IWLS benchmark set can take several hours and requires several hundred megabytes of memory. The second problem is that current methods to define covers are not applicable to elimination. If the covering problem can be used for elimination, each cover must be chosen such that it can eliminate redundancies when collapsed and define regions ideal for resynthesis. In the following sections we will overview the covering problem in detail and describe how we tackle both of these problems. First we illustrate a compression technique for cut generation using BDDs. As we will show, this compression technique leads to an order of magnitude reduction in both runtime and memory use when generating cuts. Second, we introduce a new heuristic referred to as *edge flow* which helps to quickly identify covers ideal for elimination.

3.3 Adaptation of Covering Problem to Elimination

3.3.1 Covering Problem

The covering problem seeks to find a set of covers for a graph such that a given characteristic of the final covered graph is optimized. For example, when applied to K -LUT technology mapping, the covering problem returns a covered graph such that the number of resulting LUTs in the graph is minimized (each K -input cover in the covered graph is mapped directly to a K -LUT). This is illustrated in Figure 3.3.

A common framework to solve the covering problem is shown in Figure 3.4. The covering problem starts by generating all K -feasible cuts in the graph (line 1). This is followed by a set of forward and backward traversals (line 3-4) which attempt to find a subset of cuts to cover the graph such that a given cost function is minimized. Iteration is necessary

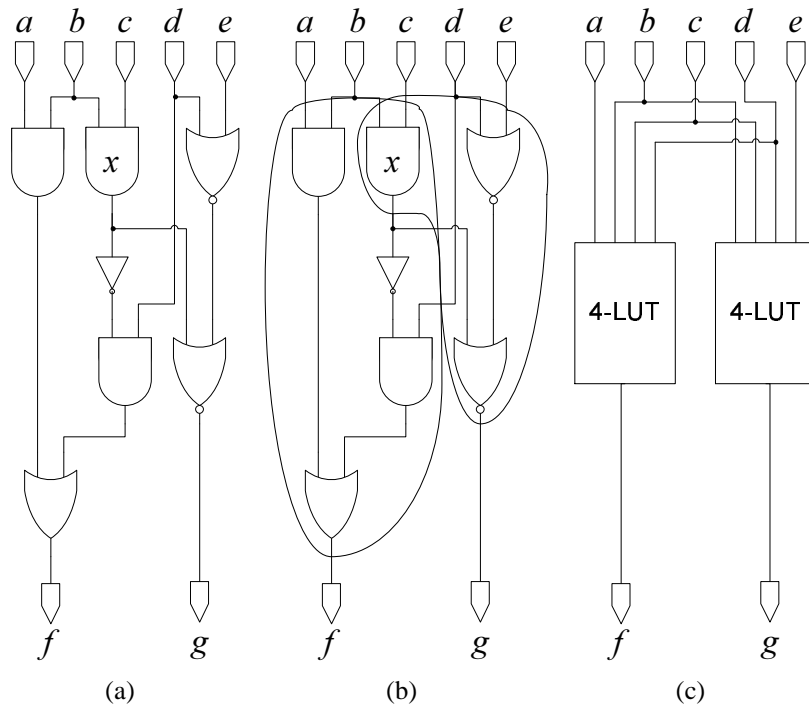


Figure 3.3: Illustration of the covering problem when applied to K -LUT technology mapping. (a) Initial network. (b) A covering of the network. (c) Conversion of the covering into 4-LUTs.

```

1  GENERATECUTS( $K$ )
2  for  $i \leftarrow 1$  upto  $MaxI$ 
3      TRAVERSEFWD()
4      TRAVERSEBWD()
5  end for

```

Figure 3.4: High-level overview of network covering.

($MaxI > 1$) if the covering found in TRAVERSEBWD() influences the cost function used in TRAVERSEFWD(). A detailed description of this algorithm when applied to technology mapping can be found in [MBV06].

Forward Traversal

```

1  foreach  $v \in \text{TSORT}(G(V, E))$ 
2     $cut_v \leftarrow \text{MINCOSTCUT}(v)$ 
3     $cost_v \leftarrow \text{COST}(cut_v)$ 
4  end foreach

```

Figure 3.5: High-level overview of forward traversal.

Figure 3.5 illustrates the high-level overview of the forward traversal. Here, each node is visited in topological order from PIs to POs. For each node, the minimum cost cut is found (line 2). After the minimum cost cut is found, the cost of the root node v is assigned the cost of the cut (line 3). Note that MINCOSTCUT is dependent on the goal of the algorithm. In later sections, we will describe the cost function used when we apply the covering problem to elimination.

Backward Traversal

```

1  MARKPOASVISIBLE()
2  foreach  $v \in \text{RTSORT}(G(V, E))$ 
3    if VISIBLE( $v$ )
4      foreach  $u \in fanin(cut_v)$ 
5        MARKASVISIBLE( $u$ )
6      end if
7  end foreach

```

Figure 3.6: High-level overview of backward traversal.

Figure 3.6 illustrates the high-level overview of the backward traversal. First, all POs are marked as visible (line 1). Next, the graph is traversed in reverse topological order. If

a node is visible, its minimum cost cut found in the preceding forward traversal, cut_v , is selected and all of its fanins are marked as visible (line 4-5). After the backward traversal completes, the minimum cost cuts of all visible nodes in the graph are converted to cones to cover the network.

Cut Generation

The primary bottleneck of the covering problem is its cut-generation process. Thus, overcoming this problem is essential if the covering problem can be migrated to elimination.

While cut generation has been traditionally applied to iterative FPGA technology mappers, such as DAOmap [CC04] and IMap [MBV06], there has been a renewed interest in the cut generation problem [MCB06b, CMB06] due to its growing use in several other CAD problems including:

- Boolean matching of PLBs [CH98, LSB05a]
- resynthesis of LUTs [LSB05b]
- synthesis rewriting [MCB06a]

In contrast to network flow methods to generate cuts [CD94, CH95], one of the first pieces of work to generate all K -feasible cuts in a circuit graph was in [CD93]. Here, cuts are generated using the recursive set relation shown in Equation 3.1.

$$\begin{aligned} \Phi(v) = \{c_u * c_w \mid c_u \in \{\{u\} \cup \Phi(u) \mid u \in fanin(v)\}, \\ c_w \in \{\{w\} \cup \Phi(w) \mid w \in fanin(v)\}, u \neq w, |c_u * c_w| \leq K\} \end{aligned} \quad (3.1)$$

In Equation 3.1, $\Phi(v)$ represents the cut set for node v ; $\{u\}$ represent the trivial cut (contains u only); c_u represents a cut from the cut set $\{\{u\} \cup \Phi(u)\}$; and $\Phi(u)$ represents

the cut set for fanin node u . A cut set, $\Phi(v)$, is formed by visiting each node in topological order from PIs to POs and merging cut sets as defined by Equation 3.1. Two cut sets are merged by performing a concatenation ($c_u * c_w$) of all cuts found in each fanin cut set, and removing any newly formed cuts that are no longer K -feasible ($|c_u * c_w| \leq K$). For example, referring to Figure 3.7, cut c_2 is generated by combining the cut c_1 with the trivial cut v_4 ($c_2 = c_1 * v_4 = v_1v_2v_4$).

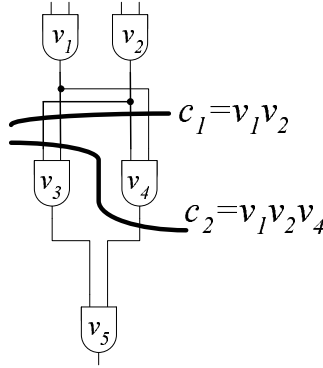


Figure 3.7: Example of two cuts in a netlist for node v_5 where c_2 dominates c_1 ($K = 3$).

In terms of set operations, Equation 3.1 defines the Cartesian product operation. In other words, generating cuts for a given node v is equivalent to taking the Cartesian product of the fanin cut sets of v , and removing any elements which have more than K circuit nodes. This is illustrated in Figure 3.8.

3.3.2 BDD-based Cut-Compression

The primary problem in generating cuts through Equation 3.1 is that cuts must be duplicated and stored every time a new cut is generated. For example, referring back to Figure 3.8, the cuts de and eg are duplicated and create the cuts in set $\Phi(a)$. Furthermore, Equation 3.1 generates redundant cuts. A cut, c_2 , is redundant if it completely contains all the input nodes of another cut, c_1 , in which case c_2 is known as a *dominator* cut. Figure 3.7 illustrates this relation. We term these cuts as redundant since their removal does not impact the final

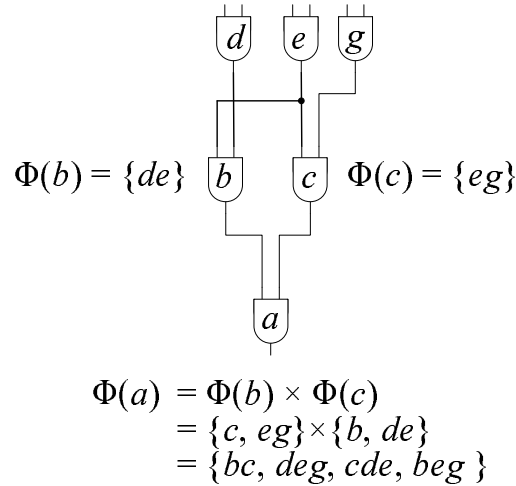


Figure 3.8: Example of generating cut sets through Cartesian product operation of fanin cutsets.

quality of our application ¹. These factors lead to a large amount of redundant data and operations when generating cuts and thus cut generation still does not scale beyond a cut size of 7. Although this is not a problem for commercial FPGAs that restrict their LUT size to 6 or less [Alt04], migrating the covering problem to larger problems such as elimination requires a more scalable cut generation solution.

Redundant data could be removed if a “factored” representation of the cuts was possible. For example, in Figure 3.8, cut set $\Phi(a)$ in factored form would be $\Phi(a) = \{(de, b)(eg, c)\}$. This factorization operation has an analogy to Boolean algebra where Boolean expressions are factored to share common subexpressions. If we could represent our cut sets as a Boolean expression, we could leverage existing factorization representations and potentially reduce storage requirements of maintaining large cut sets. This is possible if we represent a cut as a cube of positive literals and a cut set as a Boolean disjunction of these cubes to form a Sum of Products (SOP). For example, in SOP form, the cut set for $\Phi(a)$ would be equivalent to $f_a = bc + cde + beg + deg$. If we define a *subcut* as a subset of fanin nodes of a cut, it is possible to share subcuts to represent an entire set of cuts. This

¹It should be noted, that some applications do not consider dominator cuts redundant and can be chosen in the final solution

requires a factorization of the cut set SOP representation to extract common subexpressions. Instead of explicitly applying a factorization operation which can be expensive if applied numerous times, we can use a BDD which implicitly stores Boolean expressions in factored form [Som99]. For example, the BDD used to represent the cut set f_a is shown in Figure 3.9.

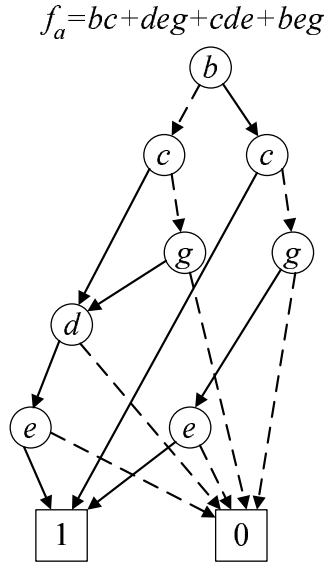


Figure 3.9: The BDD representation of the cut set in Figure 3.10.

In order to fully extend our cut set representation into a Boolean space, we must provide a Boolean operation to apply the Cartesian product operation used in Equation 3.1. Fortunately, if each cut set is represented as a SOP, the Cartesian product of these sets is equivalent to a logical AND operation. This is shown in Equation 3.2.

$$f_v = \prod_{u \in fanin(v)} (u + f_u) \quad (3.2)$$

Equation 3.2 is structurally very similar to the set relation shown in Equation 3.1; however, Equation 3.2 is a Boolean operation applied directly to Boolean expressions representing each cut set. Here, we map a unique Boolean variable to each node v found in our netlist and represent cuts by the conjunction of the fanin node variables. To join cut sets, we

replace the set union operation (\cup) with a logic OR. Furthermore, the Π operation can be thought as the logical AND of all clauses $(u + f_u)$, though this operation must be modified to take care of a few special cases as described later in this chapter. An example of the Boolean approach to cut generation is shown in Figure 3.10. This shows how each node is represented by a Boolean variable and each product term in the function represents a cut. Also, notice that the cut set function f_a is the conjunction between the clauses $(c + f_c)$ and $(b + f_b)$.

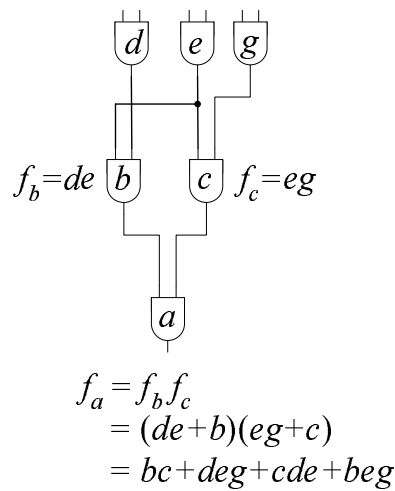


Figure 3.10: A Boolean expression based representation of cut sets.

There are three primary benefits of using BDDs to represent our cut sets: 1) Cuts sets can be implicitly shared when constructing larger cuts sets, thereby removing redundant storage and computation; 2) Subcuts are shared within a cut set as cofactors within the BDD; 3) Dominator cuts are automatically removed due to the BDD reduction rules presented in Chapter 2. Examples of each of these benefits are shown in the following sections.

1) Cut Set Sharing: Figure 3.11 illustrates an example where cut sets are reused to form larger cuts. In Figure 3.11a, two smaller cut sets are represented as BDDs. In Figure 3.11b, the BDDs shown in Figure 3.11a are reused as cofactors to build the BDD for function f_a . We have effectively saved time and storage by avoiding the duplication operation of the smaller cut sets where we have effectively used a single BDD to represent three different

cut sets.

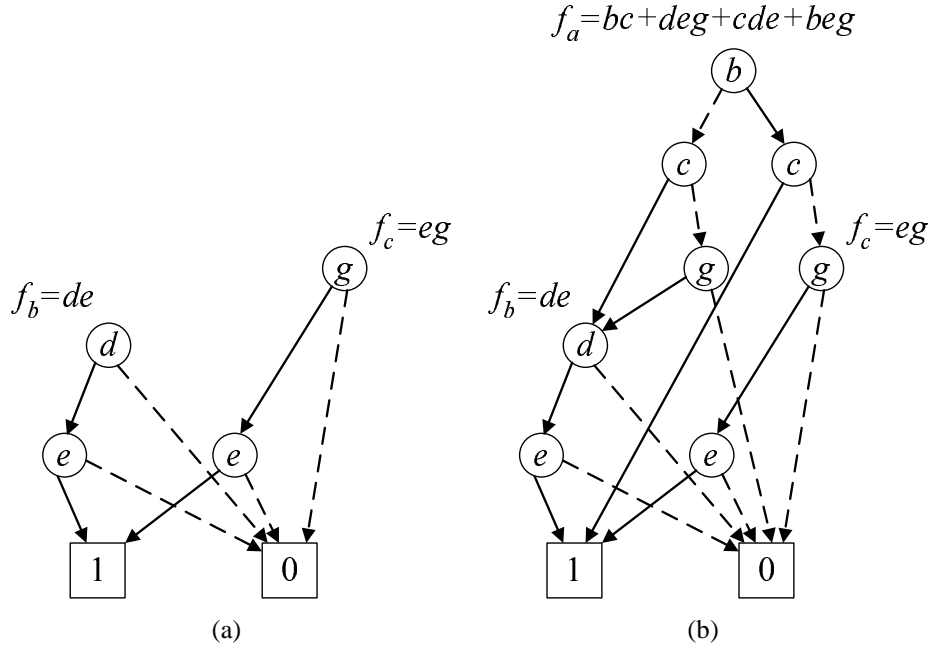


Figure 3.11: Illustration of reusing BDDs to generate larger BDDs. (a) Small BDDs representing cut set function f_b and f_c . (b) Reusing BDDs in (a) as cofactors within cut set function f_a .

2) Subcut Sharing as Cofactors: An example of subcut sharing is shown in Figure 3.12. Notice that in the BDD representation, the subcut $c_1 = de$ is a positive cofactor for variable c and g , and is shared by two larger cuts $c_3 = cde$ and $c_2 = deg$. The benefit of subcut sharing is very sensitive to variable ordering. For example, in the previous example, c_1 could not be shared if variables d and e were found at the top of the BDD. Hence, to ensure that subcut sharing is maximized, we assign BDD variables to nodes such that fanin node variables are always found below their fanout node variables in the BDD cut set. This is stated formally in Lemma 3.3.1 and Proposition 3.3.2.

Lemma 3.3.1 Consider two functions f_1 and f_2 represented as BDDs where f_1 is composed of f_2 and some other variables (i.e. $f_1 = g(f_2, x_0, \dots, x_n)$). Also, let θ be the set of variables found in f_1 which are not in f_2 . The BDD graph f_2 can exist as a subgraph in f_1 if and only if all the variables in f_2 are below all variables in θ .

An intuitive explanation to Lemma 3.3.1 can be seen in Figure 3.11 where the BDD representing f_b is a subgraph in f_a , which would not be possible if variables d and e were not at the bottom of the BDD in f_a .

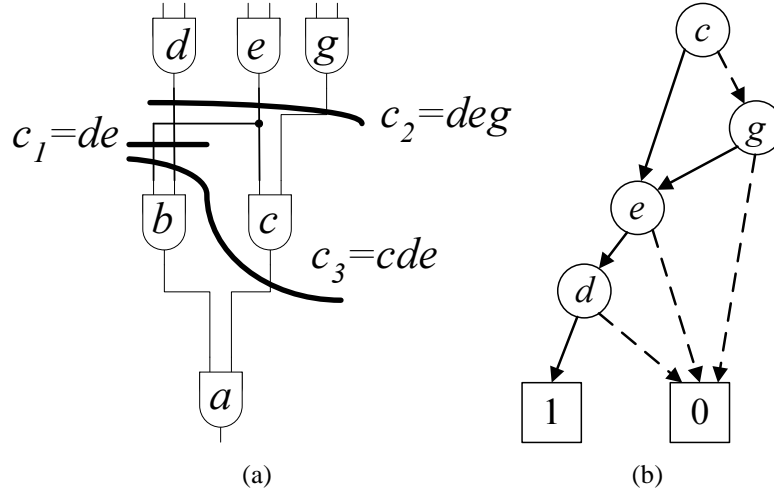


Figure 3.12: BDD representation of node a cuts c_1 , c_2 , and c_3 ($K = 3$).

Cut sharing will only occur if BDD variables are assigned nodes in the following manner. If the variable assigned to node v and the variable assigned to node u appear in the same BDD, f , and if $u \in fanin(v)$, the variable assigned to node v should appear above the variable assigned to node u in f . For example, this relationship is shown in Figure 3.12. Here, the node labeled c in Figure 3.12a has fanins labeled e and g . This translates to the BDD shown in Figure 3.12b where BDD variable c appears above variables e and g . This variable ordering condition is stated formally in Proposition 3.3.2.

Proposition 3.3.2 *Cut set sharing can only occur if the variable assigned to nodes is such that variables assigned to a node will appear above variables assigned to their transitive fanins if those variables appear in the same BDD cut set.*

Proof: Proof by contradiction. Recall from Equation 3.2 that the Boolean expression to generate the cut set for a given node v is $f_v = \prod_{u \in fanin(v)} (u + f_u)$. First we will look at a single fanin to v . Let us assign $g = (u + f_u)$ and assume the BDD f_u exists as a subgraph

in g and hence is shared. Also assume that u is not the top variable in g . However, if u is not the top variable in g , f_u cannot exist in g by Lemma 3.3.2. Thus by contradiction, if f_u is shared in g , u must be the top variable in g . Since all variables in f_u are assigned to the transitive the fanin nodes to the node u , the variables assigned to the transitive fanin nodes appear below u in function g . A same argument can be applied to all fanin nodes $u \in \text{fanin}(v)$. ■

3) Removal of Dominator Cuts: The third benefit of using BDDs for cut generation is shown in Figure 3.13 where dominator cuts are automatically removed. Figure 3.13a illustrates the cut c_1 and the dominator cut c_2 . As a BDD, c_1 and c_2 are shown in Figure 3.13b. Since BDD node c is now redundant (Chapter 2), it can be removed as in Figure 3.13c which removes the dominator cut c_2 .

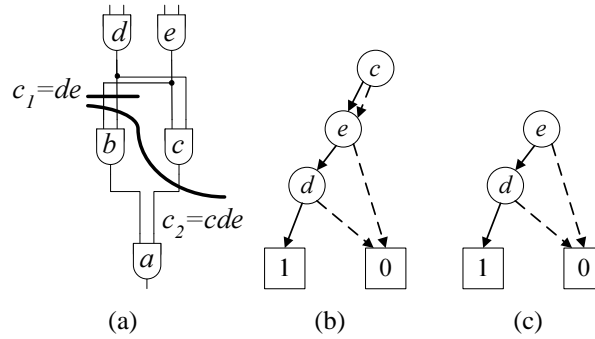


Figure 3.13: BDD representation of node a cuts c_1 and c_3 ($K = 3$).

3.3.3 Symbolic Cut Generation Algorithm

Figure 3.14 illustrates our cut generation algorithm. First, the netlist is sorted in topological order (line 1). Next, the cut set function, f_v , for each node in the graph is initialized to a constant 1 and assigned a unique variable (line 2-5). Finally, for each node, v , its cut set is formed following Equation 3.2 (line 7-10). When forming the cut set for node v , each fanin node, u , is visited (line 7) and a temporary cut set is formed by the logical OR of the trivial cut u and its cut set f_u . Next, the temporary cut set is conjoined to the cut set of v using the

```
CutGeneration()  
1   $G(V, E) \leftarrow \text{TSORT}()$   
2  foreach  $v \in G(V, E)$   
3     $f_v \leftarrow 1$   
4     $b_v \leftarrow \text{CREATENEWBDDVARIABLE}()$   
5  end foreach  
6  foreach  $v \in G(V, E)$   
7    foreach  $u \in \text{fanin}(v)$   
8       $f_x \leftarrow \text{BDDOR}(b_u, f_u)$   
9       $f_v \leftarrow \text{BDDANDPRUNE}(f_v, f_x, K)$   
10   end foreach  
11 end foreach
```

Figure 3.14: High-level overview of symbolic cut generation algorithm.

logical AND operation (line 9). When forming larger cuts with the logical AND operation, it is possible to form cuts larger than K , thus `BDDANDPRUNE` is also responsible for pruning cuts that are not K -feasible. It does so by removing all cubes that contain more than K positive literals which will be explained in the detail in the following section.

3.3.4 Ensuring K -Feasibility

When conjoining two cut sets together using the logical AND operation, we must ensure that all cuts remaining in the new cut set are K -feasible. We achieve this by modifying the BDD AND operation to remove cubes with more than K literals. This recursive algorithm is illustrated in Figure 3.15. Notice that the only difference in this algorithm compared to the recursive definition of a BDD AND operation is the check in line 9. The algorithm starts off by checking the trivial case where both BDD cut sets are constant functions (line 1). If not the trivial case, the top most variable of both cut sets is retrieved (line 3). Next, the cofactors relative to the variable b are found for the cut sets f_x and f_y (line 4-8). This is followed by recursive calls to find the negative and positive cofactors of the new cut set f_z (line 9-12). When constructing the positive cofactor, we make sure that the number of

```

< fz > BddAndPruneRecur(fx, fy, K, n)
1  if ISCONSTANT(fx) AND ISCONSTANT(fy)
2    return < fxANDfy >
3  b ← GETTOPVAR(fx, fy)
4  fnx ← fx(b = 0)
5  fny ← fy(b = 0)
6  fpx ← fx(b = 1)
7  fpy ← fy(b = 1)
8  fnb ← BDDANDPRUNERECUR(fnx, fny, K, n)
9  if n ≤ K
10   fpb ← BDDANDPRUNERECUR(fpx, fpy, K, n + 1)
11 else
12   fpb ← 0
13 SETCOFACTORS(fz, b)
14 SETCOFACTORS(fz, fnb, fpb)
15 return < fz >

```

Figure 3.15: High-level overview of BDD AND operation with pruning for K .

positive edges seen is less than or equal to K (line 9-10). If not, we prune out all cubes that form due to that branch in the BDD. This works since our cut sets, f_x and f_y , only contain positive literals and n is initialized to zero in the first call to `BDDANDPRUNERECUR`. Thus, we can assume n is equivalent to the size of the cube in the current branch of the BDD. Finally, we join the cofactors and form a new cut set, f_z , and return (line 13-15).

The function `SETCOFACTORS` must also be altered to suppress BDD nodes whose positive edge points to zero (line 3). If this occurs, this implies that there exists a cube with a negative literal. Since our cut sets should only contain literals in positive form, we eliminate them. This is similar to the reduction rule used to create a ZDD, though the resulting graph is not a ZDD since complemented edges are allowed and the common edge reduction rule is still used as described in Chapter 2.

```

SetCofactors( $f_z, f_{n_b}, f_{p_b}$ )
1  if  $f_{n_b} \equiv f_{p_b}$ 
2     $f_z \leftarrow f_{p_b}$ 
3  else if  $f_{p_b} \equiv 0$ 
4     $f_z \leftarrow f_{n_b}$ 
5  else
6    SETPOSITIVECOFACTOR( $f_z, f_{p_b}$ )
7    SETNEGATIVECOFACTOR( $f_z, f_{n_b}$ )
8  return

```

Figure 3.16: Construction of BDD cut set given f_z if given positive and negative cofactors.

3.3.5 Finding the Minimum Cost Cut

In general, the cost of a given cut is usually defined recursively with the form as shown in Equation 3.3.

$$cost_c = \sum_{u \in fanin(c)} cost_{min}(c_u) \quad (3.3)$$

In Equation 3.3, u is a fanin of cut c , c_u is the minimum cost cut associated with node u , and $cost_{min}(c_u)$ is the cost of the cut c_u . For traditional cut generation methods where subcuts are not shared, each cut has to be traversed independently to determine the minimum cost cut. Conversely, since we represent our cut set as a BDD where we share subcuts, we can leverage dynamic programming to calculate the cut cost and find the minimum cut cost. This is illustrated in the recursive algorithm in Figure 3.17. In **MINCUTCOSTRECUR**, the minimum cost cut, c_{min} , and its cost, $cost$, from the cut set f_v is returned. Notice that c_{min} is returned as a cube where each positive literal in the cube represents a fanin node to the cut. First, if the cut set is trivial ($f_v \equiv 1$), the algorithm returns an empty cube (const 1) with zero cost (line 1-2). If the cut set is empty, an invalid cube is returned (line 3-4). If the cut set is not an empty set, the algorithm checks if this cut set has been visited already, and if so, returns the cached information (line 6-7). If the cut set has not been visited previously, two recursive calls are done to find the minimum cost cut and cost for the cofactors (line 9-12). Next, the positive cofactor cost is modified with the node cost of the current variable

```

<  $c_{min}, cost$  > MinCutCostRecur( $f_v$ )
1  if  $f_v \equiv 1$ 
2    return < 1, 0 >
3  else if  $f_v \equiv 0$ 
4    return <  $\phi, \phi$  >
5  // dynamic programming step
6  if CACHED( $f_v$ )
7    return < LOOKUP( $f_v$ ) >
8   $b \leftarrow \mathbf{TOPVAR}(f_v)$ 
9   $fn_v \leftarrow f_v(b = 0)$ 
10  $fp_v \leftarrow f_v(b = 1)$ 
11 <  $cn_{min}, cost_n$  >  $\leftarrow \mathbf{MINCUTCOSTRECUR}(fn_v)$ 
12 <  $cp_{min}, cost_p$  >  $\leftarrow \mathbf{MINCUTCOSTRECUR}(fp_v)$ 
13  $cost_p \leftarrow cost_p + \mathbf{GETNODECOST}(b)$ 
14 if VALID( $cn_{min}$ ) AND VALID( $cp_{min}$ )
15   if  $cost_n < cost_p$ 
16     CACHE( $f_v, < cn_{min}, cost_n >$ )
17   else
18      $f_x \leftarrow \mathbf{BDDAND}(cp_{min}, b)$ 
19     CACHE( $f_v, < f_x, cost_p >$ )
20   end else
21 else if VALID( $cp_{min}$ )
22    $f_x \leftarrow \mathbf{BDDAND}(cp_{min}, b)$ 
23   CACHE( $f_v, < f_x, cost_p >$ )
24 else
25   CACHE( $f_v, < cn_{min}, cost_n >$ )
26 return < LOOKUP( $f_v$ ) >

```

Figure 3.17: Find the minimum cost cut in a given cut set.

(line 13). Finally, the minimum cost cut set and cost are returned (line 14-26). Note that when the minimum cost cut is found, it is cached for future reference (line 15-25).

3.3.6 Using Zero-Suppressed Binary Decision Diagrams (ZDDs)

Zero-Suppressed Binary Decision Diagrams (ZDDs) are well known to be efficient in storing and applying set operations for the same reasons as our BDD-based approach. The algorithms listed previously can be extended to well known ZDD set operations [Min93, Mis01] as illustrated in the appendix. However, in contrast to our method, ZDDs have two main limitations. First, ZDDs do not implicitly remove dominator cuts, as shown in Figure 3.13. In a ZDD representation, the dominator cuts are treated as unique set elements and will not be removed. Second, ZDD cannot use inverted edges. As a result, the amount of sub-cut sharing is much more limited. We will explore these two points further in the results section.

3.3.7 Reconvergence and Edge-Flow

In the previous section, we addressed the scalability issues faced by the covering problem. As a final step in migrating the covering problem to elimination, we need to formulate an effective cost function for the covering problem that is in line with elimination's goals. The key purpose of elimination is to define regions which contain redundancies and can be removed during a collapse or decomposition operation. Thus, a covering based approach to elimination should accomplish this by creating covers that contain redundancies. As a solution, we propose a structural method which attempts to identify redundancies by directly leveraging the structural characteristics of the netlist.

In our analysis, we will focus on reconvergent paths to help isolate redundancies. A reconvergent path is defined as a path that branches into two distinct paths which later merge

into a single path. Figure 3.18 illustrates two cones of logic: one without any reconvergent paths, and one with reconvergent paths.

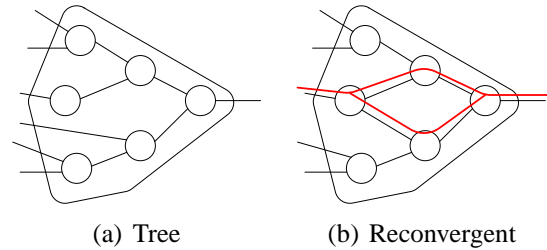


Figure 3.18: Illustration of cones with no reconvergent paths (a) and with reconvergent paths (b).

Proposition 3.3.3 *A cone of logic will have redundancies, only if it contains reconvergent paths.*

Proof: Proof by contradiction. Let's assume that a cone of logic has redundancies, but contains no reconvergent paths. If a cone of logic contains no reconvergent paths, it forms a tree structure. This implies that each node in the cone has exactly one path to a fanin to the cone. If the cone has redundancies, this implies that one of the nodes can be removed or reduced (removal of one of the node inputs). However, if a node is removed or reduced, one of the fanins of the cone will no longer have a path to the root of the cone, thereby changing the function output from the root of the cone. Thus, the circuit has no redundancies and we have a contradiction. ■

By Proposition 3.3.3, when defining our covers we should isolate cones with reconvergent paths. In order to accomplish this, we use a heuristic cost function which attempts to minimize the number of edges found within the final covered graph. This is ideal since covers that contain reconvergent paths have less input edges than covers with no reconvergent paths. Our cost function is referred to as edge flow, and by minimizing the overall edge flow during covering selection, the total cut size of the final covering is minimized.

$$ef(C_v) = \sum_{u \in fanin(C_v)} \frac{ef(u) + W_{\langle u, * \rangle}}{|fanout(u)|} \quad (3.4)$$

$$ef(v) = MIN\{ef(C_v) \mid C_v \in \Phi(v)\} \quad (3.5)$$

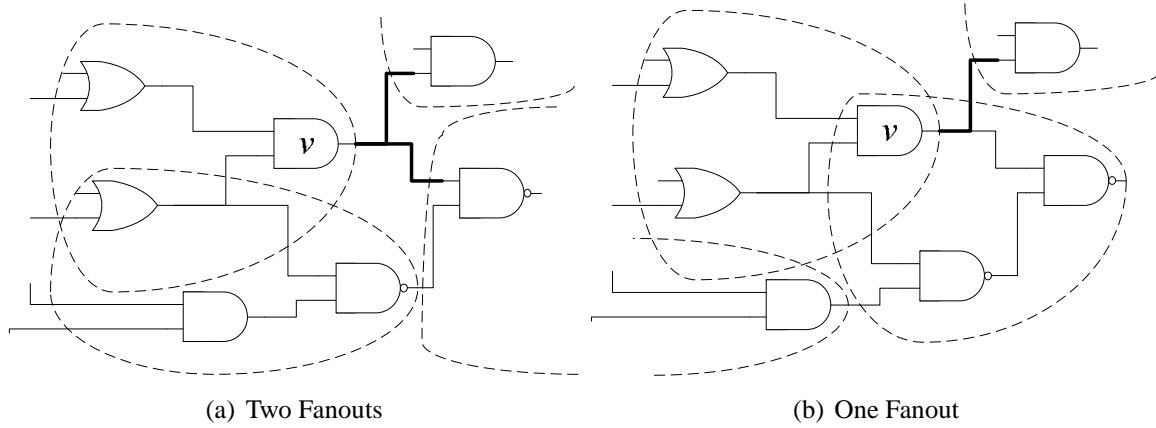


Figure 3.19: Illustration of fanout dependency on the covering.

Edge flow is defined recursively in Equation 3.4 and is denoted $ef(\cdot)$. The edge flow of cover C_v is defined as the sum of the edge flows of the fanin nodes to C_v plus the weight associated with each fanin edge $\langle u, * \rangle$, $W_{\langle u, * \rangle}$. Here $*$ is the head node for the fanin edge of C_v . Also, for each fanin, we divide its edge flow summation by the size of its fanout set. The edge flow for a node is defined as the minimum cost edge flow of any cover rooted at node v . If v is a primary input, its edge flow is defined as 0.

Under the general framework for the covering problem, the fanout size of a node will not be known until the covering problem completes. For example, consider Figure 3.19. If the covering chosen in the backward traversal is Figure 3.19a, node v has a fanout of 2; however, if the covering chosen is Figure 3.19b, the node v has a fanout of 1. If the fanout size can be estimated, edge flow can still be a good predictor of the resulting number of edges in a given covering. The fanout size estimated can be found by taking a weighted average of previous fanout sizes per node as shown in Equation 3.6 where $|fanout(v)'|_{est}$

$$|fanout(v)|_{est} = \frac{|fanout(v)'|_{est} + \alpha |fanout(v)|}{1 + \alpha} \quad (3.6)$$

Figure 3.20: Equation for estimating a node’s fanout size.

is the estimate found in the previous iteration and $|fanout(v)|$ is the actual number of edges found in the last backward traversal. In cases where v was not a cone root node in the last covering chosen, $|fanout(v)|$ is set to 1. On the first traversal, $|fanout(v)|_{est}$ is set to the number of output edges of the node v in the uncovered netlist. Previous reports showed that setting α between 1.5 and 2.5 was best for iterations of 8 or less [MBV06].

3.4 Results

We empirically validate our analysis in the previous sections here. First, we create a BDD-based cut generation algorithm, which we refer to as *BddCut*, and compare it to current techniques. Next, we evaluate our edge flow heuristic when used to drive the covering problem. Our last evaluation attempts to measure the benefits of the proposed covering method under the context of a complete logic synthesis flow. To this end, we embed *BddCut* as a replacement of the elimination procedure in FBDD, a BDD-based synthesis engine, and evaluate its impact on runtime and area.

3.4.1 BddCut: Scalable Cut Generation

To evaluate our cut generation technique, we look at two aspects. Our first evaluation focuses on the scalability of the cut generation approach. For evaluation, we compare our approach against two representative, state-of-the-art mappers: IMap, one of the earliest mappers to use cut-generation for technology mapping; and ABC, the most recently reported synthesis flow that employs a state-of-the-art cut generation algorithm for many of

its algorithms.

To investigate our symbolic approach to cut generation, we compare the cut generation time of BddCut against IMap’s [MBV06] and ABC’s [MCB06b] cut generation time. Note that during all flows, we generate all possible cuts (i.e. no pruning) of size K or less. Table 3.1 and Table 3.2 shows detailed results for select circuits, followed by Table 3.3 and 3.4 with summarized results for the entire ITC and ISCAS89 benchmark suite. We also show a peak memory use comparison between ABC and BddCut as shown in Table 3.5. Finally, we compared BddCut with ABC for one of the largest IWLS circuits which is shown in Table 3.6. In cases that the technology mapper ran out of memory, the circuit time is marked as n/a.

| <i>Circuit</i> | $K=6$ (sec) | | | $K=7$ (sec) | | |
|----------------------|---------------|-------------|------------|---------------|-------------|------------|
| | <i>BddCut</i> | <i>IMap</i> | <i>ABC</i> | <i>BddCut</i> | <i>IMap</i> | <i>ABC</i> |
| C6288 | 0.20 | 40.64 | 0.52 | 0.67 | 660.76 | 5.66 |
| des | 0.36 | 10.46 | 0.19 | 0.70 | 294.05 | 3.34 |
| i10 | 0.22 | 14.27 | 0.25 | 1.58 | 98.27 | 2.00 |
| b20 | 1.84 | 81.89 | 0.88 | 8.27 | 890.67 | 8.69 |
| b21 | 1.91 | 86.84 | 0.94 | 8.59 | 929.90 | 8.66 |
| b22_1 | 2.17 | 107.16 | 1.38 | 8.81 | n/a | 10.3 |
| s9234.1 | 0.11 | 3.96 | 0.13 | 0.33 | 38.32 | 0.75 |
| s38417 | 0.45 | 13.68 | 0.31 | 1.39 | 133.83 | 0.72 |
| s4863 | 0.11 | 19.27 | 0.11 | 0.36 | 269.07 | 0.84 |
| s1494 | 0.11 | 15.73 | 0.09 | 0.33 | 197.76 | 0.63 |
| Ratio Geomean | | 63x | 0.83 | | 225x | 1.8x |

Table 3.1: Detailed comparison of BddCut cut generation time against IMap and ABC. IMap could not run for $K \geq 8$.

The results in the previous table clearly indicate that due to subcut sharing and redundant cut removal, our symbolic approach scales better than traditional techniques where IMap is more than an order of magnitude slower. When compared against ABC, our technique scales much better where our average speedup and relative reduction in memory use improves as K gets larger. Unlike runtime, the improvement in memory does not occur until

| <i>Circuit</i> | $K=8$ (sec) | | $K=9$ (sec) | | $K=10$ (sec) | |
|----------------------|---------------|------------|---------------|------------|---------------|------------|
| | <i>BddCut</i> | <i>ABC</i> | <i>BddCut</i> | <i>ABC</i> | <i>BddCut</i> | <i>ABC</i> |
| c6288 | 2.48 | 14.49 | 9.91 | 150.13 | 41.86 | 1758.44 |
| des | 9.05 | 10.70 | 74.66 | 105.16 | 828.44 | 1126.50 |
| i10 | 2.83 | 6.06 | 11.41 | 57.17 | 50.78 | 581.09 |
| b20 | 42.01 | 73.53 | 200.27 | 889.92 | 895.63 | n/a |
| b21 | 44.03 | 80.34 | 205.25 | 942.88 | 920.22 | n/a |
| b22_1 | 41.22 | 84.36 | 180.58 | 924.38 | 766.63 | n/a |
| s9234.1 | 1.08 | 7.61 | 4.11 | 16.69 | 17.94 | 192.72 |
| s38417 | 4.31 | 6.19 | 14.19 | 58.09 | 47.97 | 536.84 |
| s4863 | 1.45 | 4.99 | 6.53 | 50.66 | 30.77 | 555.59 |
| s1494 | 1.20 | 3.53 | 5.88 | 32.63 | 31.61 | 295.38 |
| Ratio Geomean | | 2.5x | | 4.9x | | 10x |

 Table 3.2: Detailed comparison of BddCut cut generation time against ABC for $K \geq 8$.

| <i>Benchmark</i> | $K=6$ | $K=7$ |
|------------------|-------|-------|
| ITC | 27.8x | 46.5x |
| ISCAS89 | 12.2x | 26.5x |

 Table 3.3: Average ratio of $\frac{IMap}{BddCut}$ cut generation times. IMap could not be run for $K \geq 8$.

| <i>Benchmark</i> | $K=6$ | $K=7$ | $K=8$ | $K=9$ | $K=10$ |
|------------------|--------|-------|-------|-------|--------|
| ITC | 0.512x | 1.07x | 1.77x | 4.25x | 11.2x |
| ISCAS89 | 0.781x | 1.08x | 1.59x | 2.39x | 4.87x |

 Table 3.4: Average ratio of $\frac{ABC}{BddCut}$ cut generation times.

| <i>Benchmark</i> | $K=6$ | $K=7$ | $K=8$ | $K=9$ | $K=10$ |
|------------------|-------|-------|-------|-------|--------|
| ITC | 0.22x | 0.60x | 1.58x | 2.12x | 6.13x |
| ISCAS89 | 0.14x | 0.24x | 0.73x | 1.55x | 2.71x |

 Table 3.5: Average ratio of $\frac{ABC}{BddCut}$ memory usage.

K gets large ($K > 7$). We found that because there is a fixed overhead in the CUDD BDD manager [Som98], and therefore only at the larger cut sizes was this overhead amortized through subcut sharing or if the benchmark is large as in the case of leon2 shown

| <i>Cut Size</i> | <i>runtime (sec)</i> | | <i>memory (GB)</i> | |
|-----------------|----------------------|------------|--------------------|------------|
| | <i>BddCut</i> | <i>ABC</i> | <i>BddCut</i> | <i>ABC</i> |
| 6 | 23.3 | 77.9 | 0.43 | 1.41 |
| 7 | 58.9 | n/a | 0.45 | n/a |
| 8 | 152.9 | n/a | 0.60 | n/a |
| 9 | 547.6 | n/a | 0.81 | n/a |

Table 3.6: Runtime comparison of BddCut with ABC on circuit `leon2` (contains 278,292 4-LUTs).

in Table 3.6. Without subcut sharing, memory runs out for a few of the larger benchmark circuits when $K = 10$. This is also true for extremely large benchmark circuits as shown in Table 3.6 where ABC runs out of memory in circuit `leon2` for $K > 6$. Fortunately, ABC supports cut dropping which has proven to reduce the memory usage by several fold, but, from our experience, cut dropping increases the cut computation time so we did not turn on this feature. For example, with cut dropping enabled, ABC took more than 12 hours to generate 10-input cuts for circuit `b20`, whereas BddCut takes less than 15 minutes.

Although ABC outperforms BddCut for small cut sizes, the longest 6-input cut generation time in BddCut was 2.8 seconds. For small cut sizes, the overhead in storing and generating BDDs is not amortized when generating cut sets symbolically, thus ABC is still the better approach for smaller values of K . The exception to this trend occurs for circuits with a high degree of reconvergence such as for circuit `C6288` (`C6288` is a multiplier). For these circuits, our relative speedup is much larger for all values of K because reconvergent paths dramatically increase the number of cut duplications in conventional cut generation methods.

A concern one could raise with our symbolic approach is the effect of BDD representation of cuts on the cache. Since the CUDD package represents BDDs as a set of pointers, the nodes in each BDD may potentially be scattered throughout memory. Thus, any BDD traversal would lead to cache thrashing, which would dramatically hurt the performance of our algorithm. However, CUDD allocates BDD nodes from a continuous memory pool

leading to BDDs that exhibit good spatial locality. Our competitive results support this claim and indicate that good cache behaviour is maintained with CUDD.

3.4.2 Edge Flow Heuristic

The new edge flow heuristic is used in the context of the covering problem for elimination. Since in elimination attempts to capture reconvergent regions as described in Section 3.3.7, our goal is to ensure that edge flow results in a graph with less edges; however, we must ensure that other metrics correlated to overall area and delay are not degraded. To do so, we compare our edge flow based covering algorithm against a robust area driven covering algorithm [MBV06]. The metrics we use to evaluate the resulting cover are the number of resulting covers, the longest path of the resulting covered graph, and the resulting number of edges in the graph. The resulting number of covers is important since this directly correlates to the final area of the circuit. In a similar sense, the longest path correlates to the final delay of the circuit.

| <i>circuit</i> | <i># Covers</i> | | <i>Longest Path</i> | | <i># Edges</i> | |
|----------------------|-----------------|-----------|---------------------|-----------|----------------|-----------|
| | <i>af</i> | <i>ef</i> | <i>af</i> | <i>ef</i> | <i>af</i> | <i>ef</i> |
| c6288 | 498 | 492 | 25 | 24 | 1634 | 1589 |
| des | 1248 | 1250 | 7 | 7 | 4725 | 4621 |
| i10 | 793 | 799 | 14 | 14 | 3021 | 2913 |
| b20 | 4525 | 4501 | 29 | 29 | 11240 | 10824 |
| b21 | 3700 | 3689 | 28 | 27 | 12350 | 11235 |
| b22_1 | 5825 | 5801 | 28 | 28 | 16440 | 15111 |
| s9234.1 | 392 | 381 | 8 | 8 | 1182 | 1038 |
| s38417 | 3660 | 3620 | 9 | 9 | 10423 | 9593 |
| s4863 | 406 | 401 | 15 | 15 | 1434 | 1362 |
| s1494 | 266 | 265 | 5 | 5 | 937 | 875 |
| <i>Ratio Geomean</i> | 0.99 | | 1.00 | | 0.93 | |

Table 3.7: FPGA channel-width reduction using EdgeFlow with Cut-Height (*Delay* values in nanoseconds).

Table 3.7 shows our results on the ITC and ISCAS89 benchmark set where only a few

circuits have been shown in detail. Three groups of columns are shown that compare the number of covers (*# Covers*), longest path (*Longest Path*), and number of edges (*# Edges*) in the resulting covered graph. The first column (*af*) in each group lists the results of the area driven flow, and the second column (*ef*) in each group lists the results of the edge driven flow. The edge flow heuristic has a clear impact on the final covered graph. Overall, edge flow can reduce the overall number of edge in the final covered graph by 7% on average with no impact to the area or delay of the final covered circuit. Interestingly these results were also found successively in an independent study published in [JCCM08].

3.4.3 Covering based Elimination

After ensuring our symbolic cut generation approach was scalable and evaluating our edge flow heuristic, we migrated the covering problem to elimination and evaluated our elimination scheme against greedy based elimination schemes. Using our approach should significantly improve the runtime of the elimination step while ensuring that the resulting logic regions formed during elimination yield area results which are comparable to the original greedy based elimination approach. In our first set of experiments, we use a cut size of 8 to generate our cones for elimination. In following experiments, we will explore changing this parameter to see its impact on area and runtime.

To compare the two approaches, we replaced the folded elimination step in a BDD-based synthesis engine with our covering-based elimination algorithm and compared both the area and runtime of the original flow against our new flow. The synthesis engine we chose in our evaluation was FBDD [WZ05]. FBDD improved the BDD synthesis flow by leveraging shared-extraction [WZ05] where common divisors were extracted from multiple nodes and logic folding. Logic folding exploits the inherit regularity of logic circuits by sharing transformations between equivalent logic structures. This has a huge impact on runtime where it has been shown to reduce the number of elimination operations by 60% on aver-

age. We also compare against SIS for a common reference point. For ease of readability, we will refer to our flow which uses covering-based elimination as $FBDD_{new}$. Starting with unoptimized benchmark circuits, we optimized the circuits with $FBDD_{new}$, FBDD, and SIS. To compare their area results, we technology mapped our optimized circuits to 4-LUTs using the technology mapping algorithm described in [MBV06]. When optimizing the circuits in SIS, we used *script.rugged* [ESV92]. Table 3.8 illustrates detailed results for a few benchmark circuits. Column *Circuit* lists the circuit name, column *Time* lists the total runtime in seconds, and column *4-LUT Area* lists the 4-LUT count. Note a few circuits caused SIS to run out of memory and are marked as n/a. The final row lists the geometric mean of the ratio when compared against $FBDD_{new}$.

| <i>Circuit</i> | <i>Time (sec)</i> | | | <i>4-LUT Area</i> | | |
|----------------------|-------------------|-------|--------|-------------------|-------|-------|
| | $FBDD_{new}$ | FBDD | SIS | $FBDD_{new}$ | FBDD | SIS |
| b20 | 5.5 | 44.8 | 154.5 | 4514 | 4324 | 4773 |
| b21 | 7.9 | 35.8 | 185.1 | 3700 | 3689 | 3721 |
| b22_1 | 6.2 | 38.4 | 202.4 | 5788 | 6505 | 5664 |
| s38417 | 1.9 | 7.2 | 58.0 | 3560 | 3559 | 4052 |
| s38584 | 3.0 | 13.7 | 3927.3 | 4289 | 4152 | 4174 |
| s35932 | 3.9 | 4.1 | n/a | 3264 | 3360 | n/a |
| s15850 | 0.8 | 9.1 | 68.8 | 1282 | 1270 | 1329 |
| systemcdes | 3.1 | 11.3 | 123.1 | 1152 | 1207 | 1143 |
| vga_lcd | 38.9 | 585.2 | n/a | 40680 | 40676 | n/a |
| wb_conmax | 18.6 | 104.2 | 1313.5 | 19135 | 19479 | 19726 |
| Ratio Geomean | | 5.7x | 70x | | 1.00 | 1.03 |

Table 3.8: Detailed comparison of area and runtime of $FBDD_{new}$ against FBDD and SIS for $K = 8$. $\frac{FBDD_{new}}{FBDD \text{ or } SIS}$

For the circuits shown in Table 3.8, our new flow is significantly faster than the original FBDD with an average speedup of over 5x and an order of magnitude speedup over SIS. The results also show that this speedup comes with no area penalty.

To explore the impact of cut size on our elimination algorithm, we varied the cut size from 4 to 10 and compared the area and runtime against the original FBDD flow. This is

shown in Table 3.9 where we applied our new flow to the entire ITC, ISCAS89, and select IWLS benchmarks and took the geometric mean ratio of the FBDD result over $FBDD_{new}$. Column K lists the cut size used in $FBDD_{new}$ when generating resynthesis regions, column $Time$ is the time ratio, and column $4-LUT$ is the final 4-LUT area ratio. Each ratio column is given a benchmark heading indicating the benchmark suite used. As Table 3.9

| K | ITC Ratios | | ISCAS89 Ratios | | IWLS Ratios | |
|-----|------------|---------|----------------|---------|-------------|---------|
| | $Time$ | $4-LUT$ | $Time$ | $4-LUT$ | $Time$ | $4-LUT$ |
| 4 | 12.4x | 1.001 | 11.9x | 0.982 | 12.8x | 0.913 |
| 6 | 8.76x | 1.00 | 9.26x | 0.984 | 8.72x | 0.921 |
| 8 | 6.16x | 1.00 | 6.24x | 0.987 | 6.84x | 0.971 |
| 10 | 2.55x | 0.991 | 2.62x | 0.984 | 2.76x | 0.964 |

Table 3.9: Comparison of area and runtime of FBDD against $FBDD_{new}$ for various values of K , $\frac{FBDD}{FBDD_{new}}$.

shows, it appears that using a cut size of 4 or 6 has a substantial speedup of more than 10x in many cases; however, this comes with an area penalty, particularly in the IWLS benchmarks. This implies that the elimination regions created with these cut sizes are too small and does not capture large enough resynthesis regions in a single cone. In contrast, a cut size of 8 still maintains a significant average speedup of more than 6x for all benchmarks with negligible impact on the final area when compared to the original FBDD.

3.4.4 Comparison with Structural Synthesis

Structural synthesis methods have proven to be significantly faster than traditional synthesis techniques [MCB06a]. This has largely been due to the adaptation of And-Inverter Graphs (AIGs) for logic synthesis and fast hashing-schemes to “rewrite” the netlist. AIGs are circuit representations which only contain AND gates and inverters. Such a representation makes transformations on the netlist extremely fast where the netlist is locally updated using rewriting which replaces 4-input logic functions within the netlist with an AIG repre-

sensation that either uses fewer AND gates or reduces the number of logic levels. Thus it is generally accepted that structural AIG transformations are the fastest methods to perform logic synthesis.

Table 3.10 shows a comparison of runtime and area between the ABC logic synthesis engine, a robust AIG-based synthesis flow, and our improved BDD-based synthesis flow that uses a cut size of 8 during elimination. In the comparison, we use the resynthesis script `resyn2.rc` to perform AIG rewriting and refactoring on the netlist and technology map all circuits to 4-input LUTs.

| ITC Ratios | | ISCAS89 Ratios | | IWLS Ratios | |
|-------------|--------------|----------------|--------------|-------------|--------------|
| <i>Time</i> | <i>4-LUT</i> | <i>Time</i> | <i>4-LUT</i> | <i>Time</i> | <i>4-LUT</i> |
| 1.31x | 1.04 | 1.43x | 1.01 | 1.28x | 0.963 |

Table 3.10: Comparison of area and runtime of FBDD ($K = 8$) against AIG Rewriting/Refactoring.

Overall, our results suggest that BDD-based synthesis flows still cannot beat AIG-based flows using structural logic synthesis in terms of runtime. However, for larger benchmarks found in the IWLS benchmark suite, we have a slight area advantage. Since we have shown that we can reduce the synthesis flow runtime to an order of minutes, a 28% overhead in terms of runtime translates to a few minutes, which in many cases is an acceptable amount of runtime penalty when improving area. This shows that BDDs and structural techniques can play a complementary role in optimizing netlists for logic synthesis.

3.5 Summary

In this chapter we described a methodology for improving the runtime of BDD-based logic synthesis flows by 6x with a negligible impact to the final circuit area. This speedup was achieved by adapting the covering problem to elimination by introducing a novel edge flow

heuristic to reduce the number of edges found in a logic circuit along with a BDD-based compression technique to store and process cuts. Our results show that our edge flow heuristic can reduce the number of edges within a circuit by 7% on average, while our BDD-based compression technique was shown to reduce the cut generation runtime and memory use by an order of magnitude.

4 Budget Management for Partitioning

Partitioning has become a popular means to manage scalability problems of existing CAD algorithms. This is attractive since partitioning a circuit can dramatically reduce the solution space of the original problem, and hence improve its runtime. In the context of logic synthesis, this involves partitioning the netlist followed by applying logic optimizations to each partition. An example of a partitioned circuit is shown in Figure 4.1 which shows six partitions highlighted in the dashed lines.

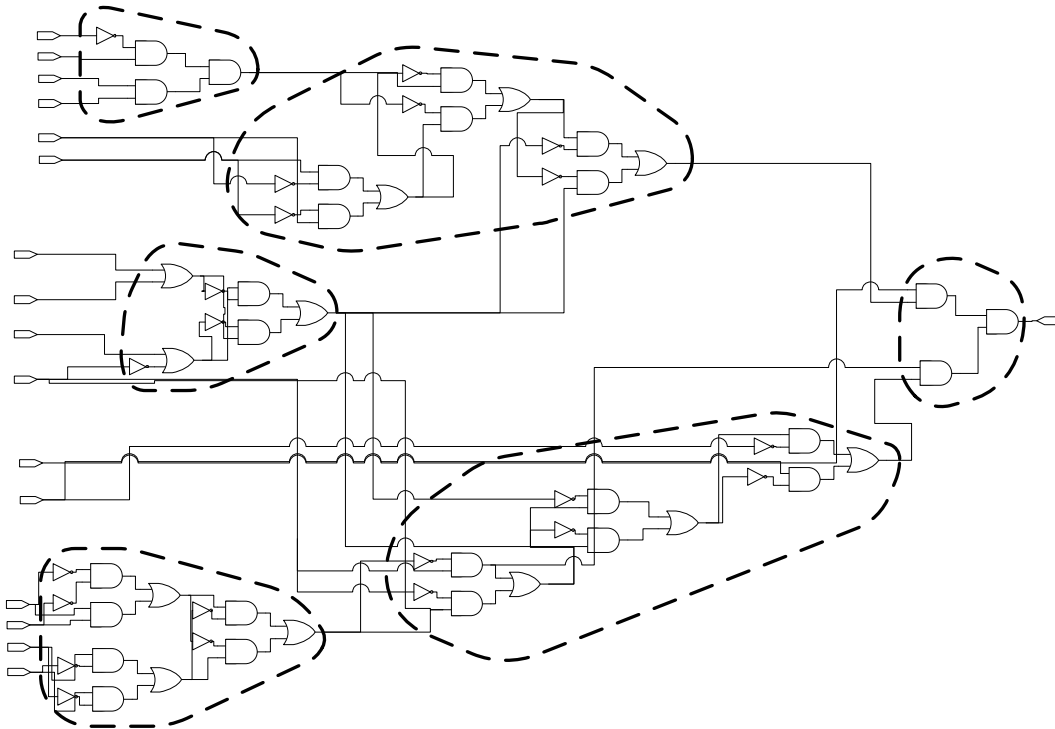


Figure 4.1: Partitioning of circuit for resynthesis.

In a partitioned circuit, each partition is optimized individually. For example, Fig-

Figure 4.2(a) illustrates a depth optimization on an individual partition where the partition depth is reduced. This optimized partition is then inserted back into the original circuit as shown in Figure 4.2(b). By optimizing the depth of each partition, the overall circuit depth has been reduced from 12 to 10 as highlighted by the shaded basic gates.

In the previous example, we were able to reduce the depth of the overall circuit by optimizing each individual partition optimally. However, an optimal decomposition for a circuit partition may not necessarily be the best decomposition when looking at the entire circuit. For example, in the previous figure, the optimal depth decomposition of a partition is shown in Figure 4.2(a). However, if a non-optimal skewed decomposition is used, as shown in Figure 4.3(a), an improved global circuit depth of 8 is achievable as shown in Figure 4.3(b).

The previous example highlights that optimizations which may be beneficial from a sub-circuit's perspective may actually lead to a poor global result. One possible solution to this is to apply incremental updates to the circuit, where partitions are individually optimized in an iterative fashion. However, this requires several static timing iterations, which can significantly hurt the runtime benefits of partitioning. Furthermore, a “ping-pong” problem can occur where paths are consistently over or under-optimized between iterations. To avoid these problems, circuit constraints should be applied to all the partitions in a single shot. In addition to removing the iterative nature of incremental approaches, this allows the resynthesis engine to produce skewed decompositions as shown in Figure 4.3(a).

In this chapter we create partition constraints to improve the circuit depth with minimal impact to the final circuit area. When applied to depth optimizations, the circuit constraint we seek is known as the depth budget, which is a numeric value assigned to each partition input. The depth budget for each input lists the maximum depth allowable between the given input and root node in order to achieve some global depth value. This is beneficial for two reasons. First, it indicates which input paths in a given partition must be “shortened”

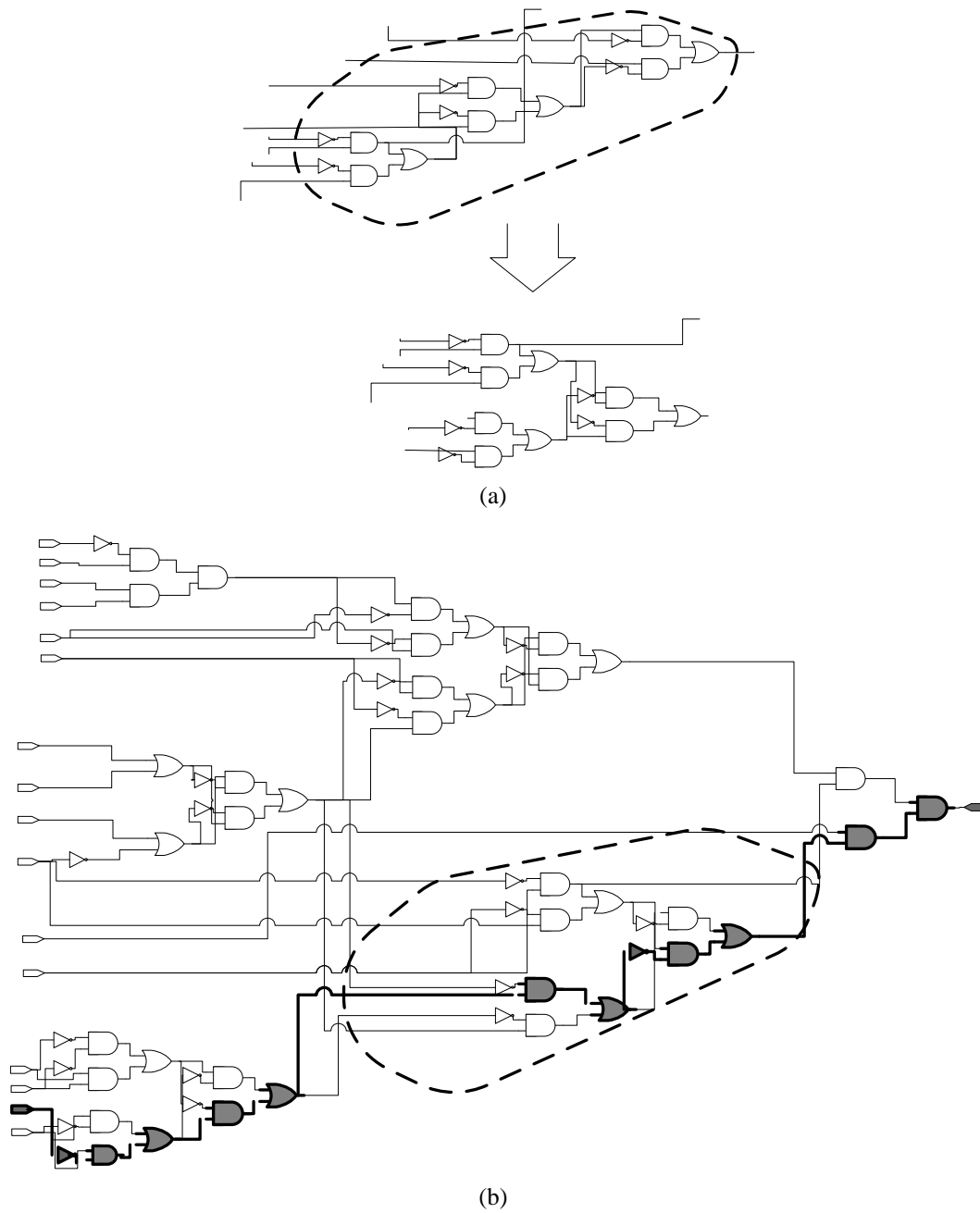


Figure 4.2: Retransformation optimization to shorten the critical path using a localized view of each partition. Original depth of circuit in Figure 4.2 is 12, final depth is 10 along shaded gates.

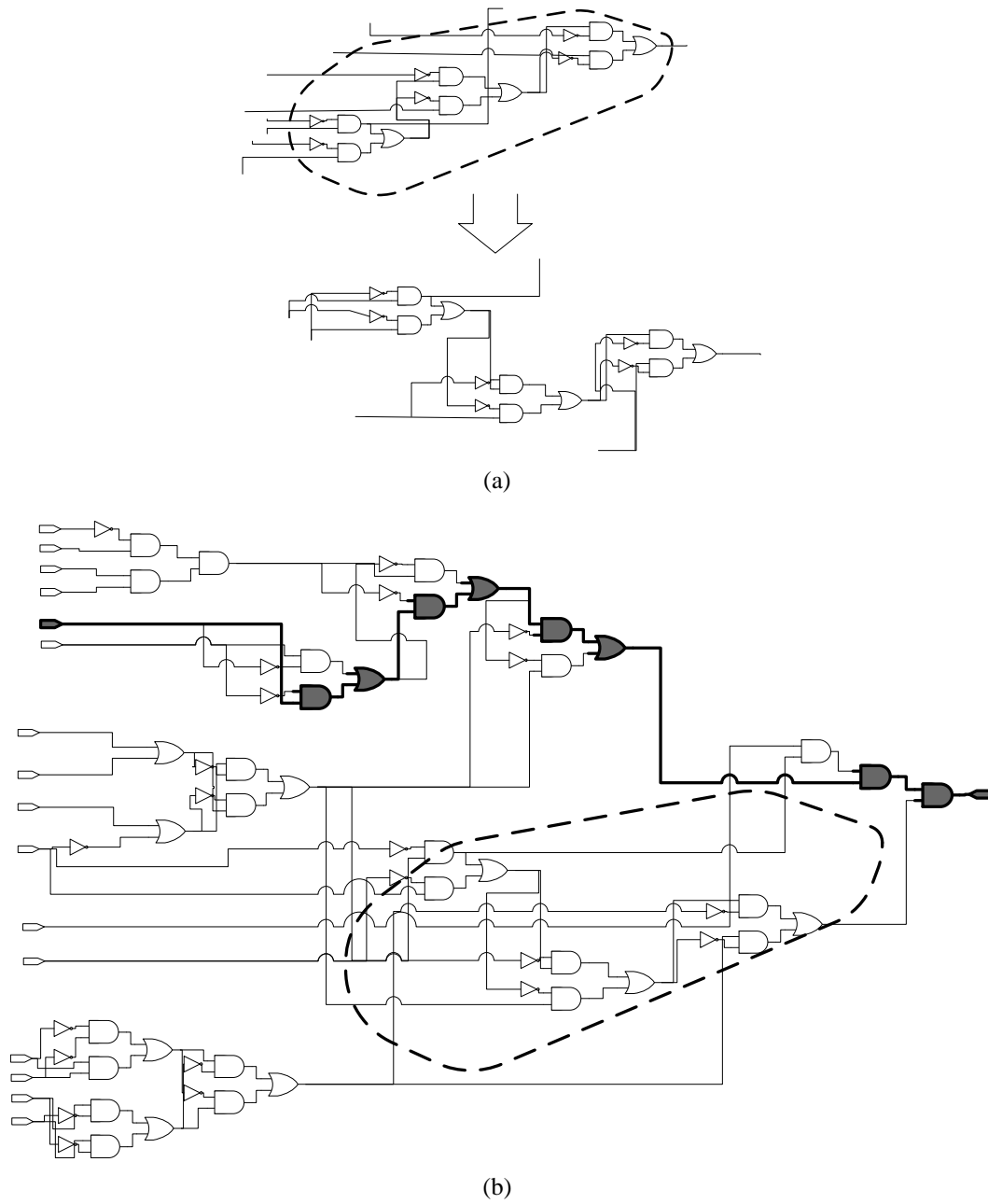


Figure 4.3: Retransformation optimization to shorten the critical path using budget constraints to guide optimizations in each partition. Final depth is 8.

in order to improve the overall circuit depth. Second, it gives flexibility to input paths to focus its circuit optimizations on area, as opposed to circuit depth. An illustration of this is shown in Figure 4.4, which shows how we can achieve the skewed decomposition shown previously using depth budgets assigned to each partition inputs. In Figure 4.4, input paths that are assigned a relatively large depth budget (6 in this case) can be optimized for area, while input paths assigned small depth budgets (2 in this case) must be optimized for depth.

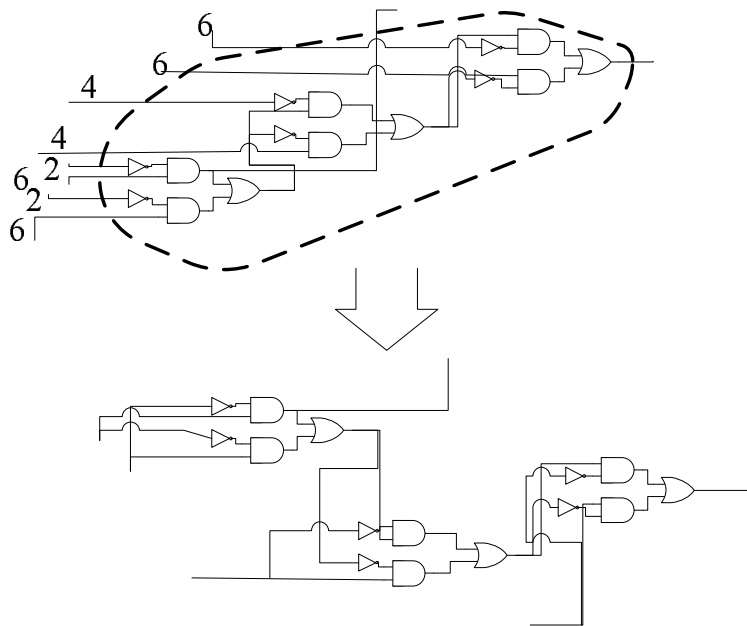


Figure 4.4: Illustration of numeric *depth budget* assignments to each partition input.

In this chapter, we show that the problem of deriving effective budget constraints as highlighted in Figure 4.4 can be formulated as an Integer-Linear Program (ILP). By following the constraints found from the ILP, the resulting optimized circuit has a superior Quality of Result (QoR) to that of one with only a localized view of the partition.

It is well known that the ILP problem is NP-complete [CLRS01] and, thus, solving our budgeting problem as an ILP will not scale to large circuits requiring a large number of constraints. To resolve this issue, we show later in Section 4.2.2 how we can reduce

our ILP to polynomial complexity by leveraging the concept of *duality*. Note that in this chapter, the term duality is used in the context of convex optimization theory, and has no relation to the definition of the *dual of a logic function* found in Boolean algebra. Doing so reduces the problem runtime by over 100x, which is a necessary condition for our technique to be practical for large circuits. Furthermore, although our reduced problem theoretically has a polynomial complexity with respect to circuit size, empirically we find that it runs in linear time on average. When run on the IWLS benchmark set, we show that our ILP formulation can assist partition based logic synthesis which improves circuit depth by 11% on average with a less than 1% penalty to area when compared against partitioning-based flows without budgeting.

The rest of the chapter is organized as follows: Section 4.1 gives some background information on the budget management problem; Section 4.2 illustrates our budget management scheme in the context of logic synthesis and how we reduce its complexity to polynomial time using duality; Section 4.3 highlights our results; and Section 4.4 concludes the chapter.

4.1 Background and Previous Work

4.1.1 Budget Management

Budget management is a common problem found in all steps of the FPGA CAD flow. It is the problem of judiciously setting circuit constraints on local regions within a circuit while providing enough freedom to optimize for other characteristics of the circuit [BGTS04, LXZ07]. Furthermore, these local constraints must be set to ensure that the global constraints of the entire circuit are also met. The difficulty of budget management arises when various circuit characteristics work in opposition to each other. For example, circuit area and power have an inverse relationship to circuit depth and performance and choosing between alternate design implementations is often a trade-off between these characteristics.

An example of the budget management problem is found in [GBCS04] which works on improving the resulting area of a circuit during high-level synthesis. During high-level synthesis, the circuit is represented as a data-flow graph (DFG) where each node represents a functional operation (e.g. addition or multiplication). Choosing between various implementations of each node involves a trade-off between clock-cycle latency and area. In [GBCS04], clock-cycle latency is defined as the number of clock cycles required to complete a computation; this has a big impact on the final area of the circuit. For example, for a given clock period, the area of a single-cycle multiplier is significantly higher than to that of a multi-cycle multiplier. The problem of budget management deals with how to choose between such alternatives of each DFG node such that the overall area is minimized while maintaining the latency, in terms of clock cycles, of the entire circuit. An example of this is shown in Figure 4.5 which shows two implementations of the same sequential circuit. In Figure 4.5, each node represents a functional unit such as a multiplier and is labeled with two numbers. The top number lists the latency of the node and the bottom lists its area units. For the two DFGs shown, the latency is 7 (i.e. there are 7 registers along every path from any input to output). In Figure 4.5(a), the total area is 84. Assuming that the bottom most node will have a significant area savings if replaced with a 2-cycle unit, the total circuit area can be reduced if an additional cycle is allocated to the bottom most node. However, in order to maintain a circuit latency of 7, we must obtain this additional cycle from one node along each path which the bottom node belongs to as shown in Figure 4.5(b). In our case, we moved one cycle from the two nodes feeding into the top most node (4-cycle unit replaced by a 3-cycle unit, and a 3-cycle unit is replaced with a 2-cycle unit).

As the previous example illustrate, clock cycles should be assigned to nodes which can best leverage the additional clock cycles to reduce its area. This is difficult since each DFG node has a unique clock latency versus area trade-off curve. Thus, greedy heuristic

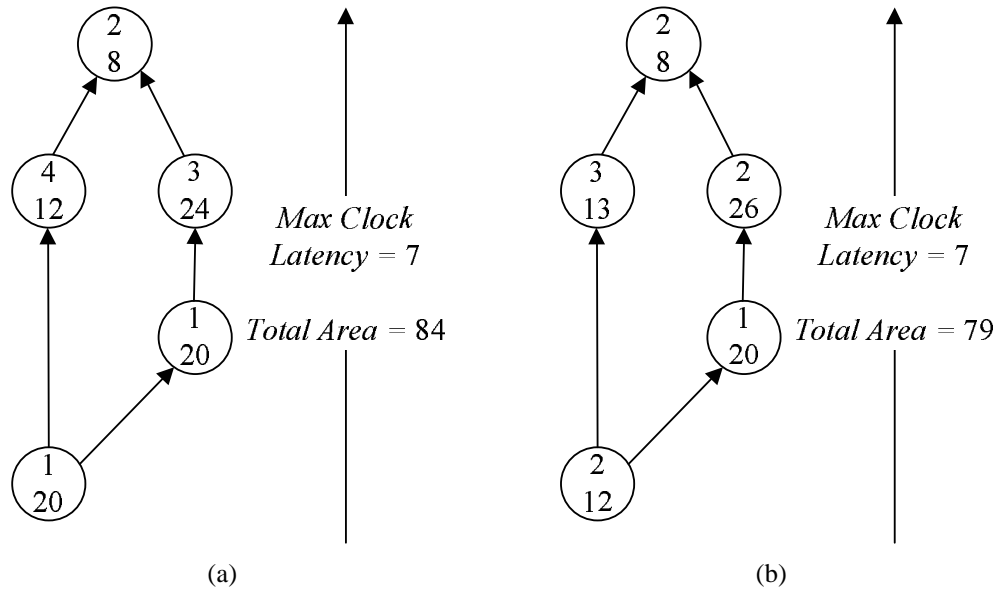


Figure 4.5: Alternate implementations of the same circuit, each node with a different circuit latency allocated to it.

approaches often lead to missed opportunities to reduce circuit area. As an alternative, the authors in [GBCS04] formulate this problem as a convex optimization problem. The results of their budget management is significant where they are able to reduce the overall FPGA LUT usage of their design by 29% without changing the latency of the final circuit.

From a budgeting perspective, our work is similar. However, in our case, we are dealing with logic depth as our main budgeting constraint, as opposed to clock cycles.

4.2 Partitioning with Delay Budgeting

During logic synthesis, the primary goal is to minimize the circuit area, while reducing the depth of the circuit. Achieving these goals after partitioning is significantly more difficult since the critical path cannot be identified by looking at each partition individually. Furthermore, even if the critical path information is annotated prior to partitioning, optimizing each path in isolation can cause problems where paths are “over-optimized” for depth while “under-optimized” for area. This chapter addresses these problems through budget man-

agement which annotates each partition with information to help guide the optimizer. This approach has four primary steps as follows:

- A clustering algorithm to partition the netlist into disjoint partitions.
- A fast conversion of the circuit to an And-Inverter Graph (AIG), which is necessary to help model the area-delay relationship of each partition used in the budget management formulation.
- A budgeting algorithm to allocate a fair distribution of budget to the partition inputs.
- A resynthesis algorithm that can utilize the budgeting information to drive a delay driven resynthesis engine.

4.2.1 Partitioning and AIG construction

To partition the circuit, we apply a variant of the covering problem which attempts to minimize the number of edges found between partitions. The resulting circuit will be a set of covers where each cover encapsulates a localized region of the circuit. A simplified illustration of the partitioning phase is shown in Figure 4.6 and is described in detail in Chapter 3. In Figure 4.6(b), partition regions are highlighted, which are then represented as a single node in Figure 4.6(c). Although only small partitions are shown in this example, larger partitions which can have more than 10 inputs are used in this work.

After partitioning, each partition is converted to an And-Inverter Graph (AIG). An AIG is a circuit representation which consists of only 2-input AND gates and inverters as illustrated in Figure 4.7. Here, we represent inverters as “bubbles” on a given graph edge. The benefit of this construction is that it unifies the representation of the entire circuit, which makes resynthesis extremely fast and simple [MCB06a]. To generate the AIG, we first convert the logic function for each node in the circuit into an irredundant sum-of-product

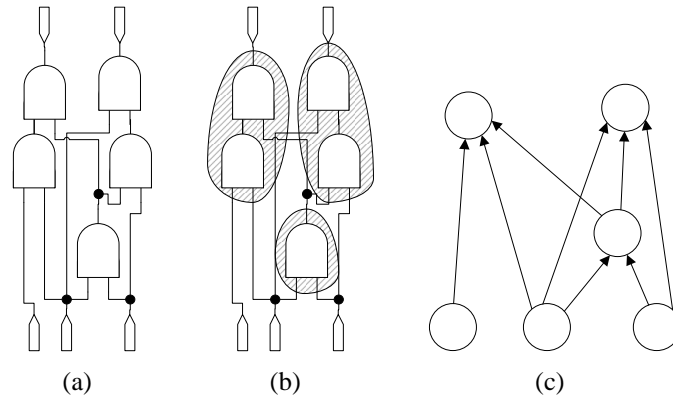


Figure 4.6: Clustering of a simple netlist. (a) Original netlist (b) Partitioning of original netlist (c) Representing each partition and PI as a single node

form (ISOP). An ISOP is a Boolean expression that contains no redundant cubes in the expression. Following this, the ISOP for each node is factored to reduce its cost. This simplified Boolean expression is then converted directly into a network of AND gates and inverters [MCB06a]. We will show in later sections that the AIG structure will help us express the area-delay relationship of each partition used in our budgeting formulation.

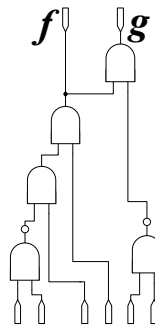


Figure 4.7: Example of an And-Inverter Graph (AIG) where each node is a 2-input AND gate and each edge can be inverted.

4.2.2 Budget Management

Following partitioning and AIG construction, each partition must be annotated with information to guide the succeeding resynthesis step. During resynthesis the two primary

optimization metrics are area and depth where we seek to minimize both. However, since in general area and depth are conflicting optimization parameters there exists a trade-off decision where partitions that are optimized for area usually increase in depth. Thus, the question that must be answered when resynthesizing each partition is as follows: what is the maximum tolerable increase in depth for each partition without violating a given overall circuit depth constraint? We seek to maximize the depth of each partition since this allows for more opportunities in area reduction. The inverse relationship between area and depth is illustrated in Figure 4.8 where Figure 4.8(a) illustrates the circuit implementation when area, in terms of gate count, is minimized while Figure 4.8(b) illustrates the circuit implementation when delay, in terms of logic levels, is minimized at the cost of area.

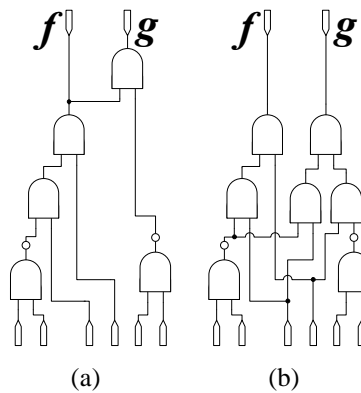


Figure 4.8: Illustration of circuit optimization for 4.8(a) area (gate-count=4, logic-levels=4) and 4.8(b) depth (gate-count=7, logic-levels=3).

The question raised previously can be rephrased more formally as a budget management problem stated as follows:

Problem 4.2.1 *Delay Budgeting Problem.* Given a partitioned circuit and a required delay D_{req} of the circuit measured in terms of circuit depth, allocate a delay budget on each partition input such that the resulting delay on any given path is less than or equal to D_{req} and the estimated total circuit area is minimized.

By solving Problem 4.2.1, a delay budget will be assigned to each partition input to give

guidance on resynthesis optimization. For example, referring to Figure 4.9, a delay budget assignment is given to each partition input. These assignments act as an upper-bound on depth for the resynthesis engine to follow. This results in a final circuit implementation, as shown previously in Figure 4.3, that has improved depth over a non-budgeted flow, even though locally the resulting circuit appears to be sub-optimal in terms of circuit depth. Note that we use depth as a measure of delay since at the logic synthesis level, accurate delay information of each edge is not available.

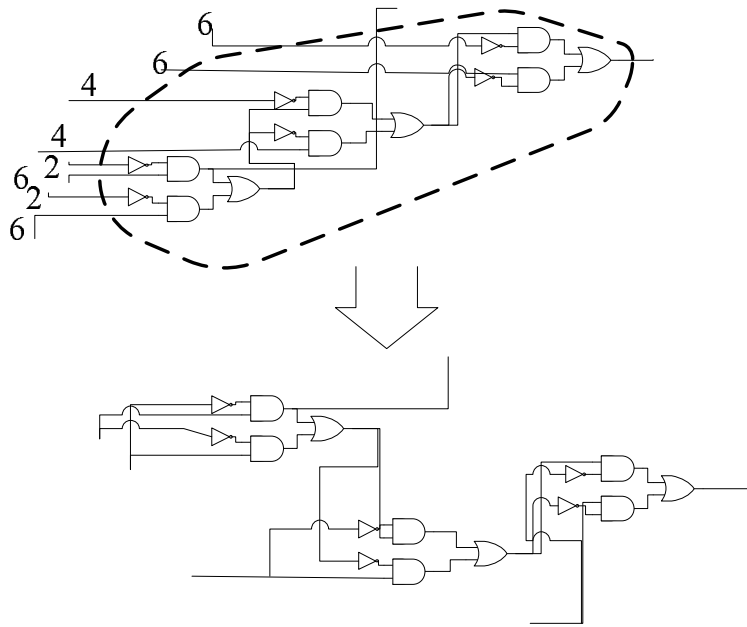


Figure 4.9: Annotated partition inputs after delay budgeting.

In order to model Problem 4.2.1, we must first model the delay-area relationship of each partition. For simplicity, we model the delay-area relationship as a piecewise-linear convex function with respect to the depth budget, b_{ij} . Here i is the label assigned to the partition whose output feeds into the partition j . For every partition input, there is a depth budget variable, b_{ij} . In the convex function, we measure delay in terms of logic levels, and area in terms of gate count. Finally, we apply a lower and upper bound on the delay constraint ($[L, U]$), since for any given partition, there will exist a minimum and maximum total depth

achievable by that partition. This relationship is illustrated in Figure 4.10 which shows the piecewise-linear convex function $F_{ij}(b_{ij})$. Here, b_{ij} is the depth budget, measured in terms of logic levels, and $F_{ij}(b_{ij})$ is the estimate of area along the given path. Later, in Section 4.2.2 we will illustrate how we derive the area-delay relationship, $F_{ij}(b_{ij})$, in more detail.

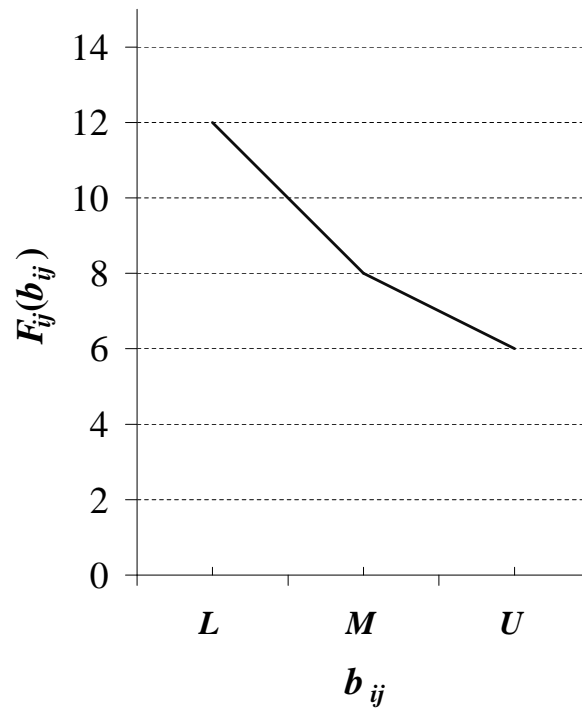


Figure 4.10: Simplified inverse relationship between delay budget, b_{ij} , and area estimation, $F_{ij}(b_{ij})$, defined over variable b_{ij} .

If we assume that the area and delay values take on integer values and that there exists an area-delay relationship along each input of the partition, we can represent Problem 4.2.1

as an ILP as shown in Equation 4.1.

$$\begin{aligned}
 \text{Min } & \sum_{\forall \langle i, j \rangle \in E} F_{ij}(b_{ij}) & (4.1) \\
 & b_{ij} = a_j - a_i \\
 & L_{ij} \leq b_{ij} \leq U_{ij} & \forall \langle i, j \rangle \in E \\
 & L_i \leq a_i \leq U_i & \forall i \in V
 \end{aligned}$$

In Equation 4.1, $G(V, E)$ represents the partitioned graph, where each node in the graph represents a primary input, primary output, or partition, and each edge represents a signal output from node i to node j . The variable b_{ij} represents the delay budget assigned to each partition input edge $\langle i, j \rangle$ and $F_{ij}(b_{ij})$ represents the estimated area for a given value b_{ij} where our goal is to minimize the overall estimated area of the circuit. In later sections we will show how we model the area-delay relationship to formulate $F_{ij}(b_{ij})$ for every edge $\langle i, j \rangle$. For each delay budget there is an upper and lower bound represented with variable L_{ij} and U_{ij} respectively. The delay budget on each input edge, $\langle i, j \rangle$, is equivalent to the arrival time, a_j , of partition node j (sink node) minus the arrival time, a_i , of partition node i (source node). For each arrival time a_i , we ensure that it is bounded by an upper and lower bound U_i and L_i . Here, the largest value set for U_i should be equal to the required circuit depth, D_{req} , and L_i should be greater or equal to zero. For illustration, Figure 4.11 shows the relationship between the partition graph and Equation 4.1.

One issue with Equation 4.1 is that it assumes that the delay-area relationship for each partition input are independent from each other. However, in general this is not true. For example, consider Figure 4.12. Assume that in Figure 4.12(a), the leftmost input has a depth budget of b_{ba} and its neighbouring input has a depth budget of b_{ca} . If we place $F_{ba}(b_{ba})$ and $F_{ca}(b_{ca})$ into Equation 4.1, assigning values to b_{ba} and b_{ca} occurs independently. However, in reality, if we decrease the depth for the leftmost input, the depth of

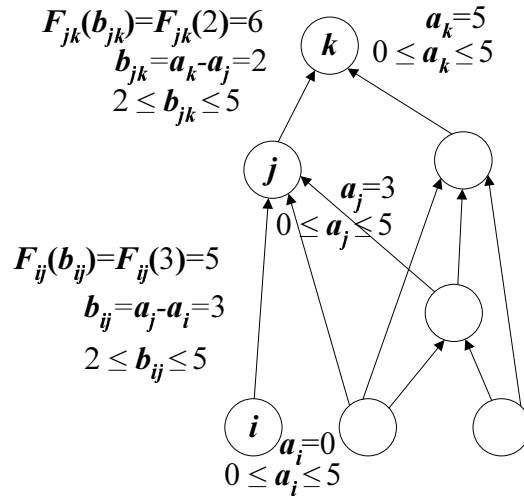


Figure 4.11: Graph to ILP formulation.

its neighbouring input must also decrease. Thus variables b_{ba} and b_{ca} are related. Adding this dependency to our problem significantly increases its complexity and for simplicity we ignore this dependency. In later sections we empirically evaluate this simplification and show in practice that it is not an issue.

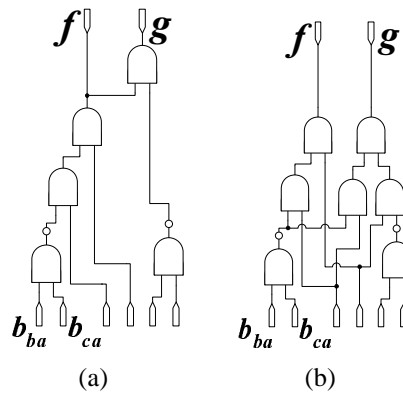


Figure 4.12: Illustration of dependency of depth between inputs.

Reduction of ILP using Duality

Once we formulate Equation 4.1, we can use any standard ILP solver to derive a depth budget, b_{ij} , for every edge $\langle i, j \rangle$ in the graph. However, solving our ILP problem directly

is not scalable to large designs. To avoid solving our objective function directly, we should attempt to reduce the complexity of our problem. Previous works [BGTS04, LXZ07] have shown that in certain cases, an ILP can be reduced to a form which has a much simpler complexity. These techniques are often based around the notion of convexity and duality where a *dual function* of the original objective function is used as the primary objective function. If it turns out that the original objective function is convex, finding the optimal solution to the dual function yields a result which is equivalent to the optimal value of the original function. Since in Equation 4.1 we assumed that $F_{ij}(b_{ij})$ is convex (linear or piecewise linear convex) for all edges $\langle i, j \rangle$, it can be shown that $\sum_{\forall \langle i, j \rangle \in E} F_{ij}(b_{ij})$ is also convex [BV04]. An example of convex dual problems whose optimal solutions are equivalent is the min-cut max-flow problem. The min-cut problem finds the capacity along the minimum cut within a network of pipes and is equivalent to the max-flow problem that finds the maximum flow along the same network of pipes. This min-max relationship is a common characteristic of dual functions where a minimization problem for a given function $f(a)$ is converted to a maximization problem for a dual function $g(\lambda)$, and $Min f(a) \equiv Max g(\lambda)$. Also, in general there does not exist a function G where $\lambda = G(a)$ which implies that a solution in a generally does not have an relationship to a solution in λ [BV04]. It should be noted that in general, the dual function of a given function may not be simpler to solve, but we will show in this section that for our specific problem, using the dual function proves to be significantly more scalable than solving the problem in its native ILP form.

To solve a problem using duality, several steps must be taken. First, the original convex function must be converted to its dual function. When converting a convex function to its dual, the original optimization variables are “removed” where a new set of optimization variables are introduced. These new variables are known as *Lagrangian multipliers*. Once the dual function is derived, the optimal values of the Lagrangian multipliers are found. Finally, the dual solution must be mapped back to the original problem space to derive

the optimal values on the original optimization variables. For example, referring back to the min-cut max-flow problem, optimizing for the max-flow problem results in a solution λ^* , which can be mapped to a solution a^* in the min-cut solution space. The remainder of this section will describe these steps in detail, though the reader may skip directly to Section 4.2.3 if not interested in the mathematical derivation. For those not familiar with the concept of duality in the context of convex optimizations, it is recommended to review duality in [BV04, ch.5].

When converting Equation 4.1 to its dual, we can assume that all our problem variables and constraints are integers as shown in Equation 4.2.

$$\begin{aligned} b_{ij}, U_{ij}, L_{ij} &\in \mathbb{Z} & \forall \langle i, j \rangle \in E \\ a_i, U_i, L_j &\in \mathbb{Z} & \forall i \in V \end{aligned} \quad (4.2)$$

Also, to simplify the derivation, we will assume that each function $F_{ij}(b_{ij})$ is a linear function $W_{ij}b_{ij}$ as shown in Equation 4.3. Later on, we will show how we will break this assumption to accommodate more general forms for $F_{ij}(b_{ij})$.

$$\begin{aligned} \text{Min} \quad & \sum_{\langle i, j \rangle \in E} W_{ij} b_{ij} & (4.3) \\ & b_{ij} = a_j - a_i \\ & L_{ij} \leq b_{ij} \leq U_{ij} & \forall \langle i, j \rangle \in E \\ & L_i \leq a_i \leq U_i & \forall i \in V \\ & W_{ij} \in \mathbb{Z} & \forall \langle i, j \rangle \in E \end{aligned}$$

In order to formulate the dual of Equation 4.3, we must first reformulate the function to remove the a_i constraints on the second last line and represent the objective function

$\sum_{\forall \langle i,j \rangle \in E} W_{ij} b_{ij}$ in terms of node variables (currently it is a summation of the edge variables b_{ij}). We can remove the constraint $L_i \leq a_i \leq U_i$ by creating a dummy node 0 and attaching each node i to this dummy node. Next, we will map every a_i variable to a b_{0i} variable. Thus, the resulting formulation remains the same, where $W_{0i} = 0$ for all $i \in V$. Following this, we will remove all variables b_{ij} and replace them with their equality constraint $b_{ij} = a_j - a_i$. This leads to Equation 4.4.

$$\begin{aligned} \text{Min } & \sum_{\forall \langle i,j \rangle \in E} W_{ij}(a_j - a_i) & (4.4) \\ & a_j - a_i \leq U_{ij} & \forall \langle i,j \rangle \in E \\ & a_j - a_i \geq L_{ij} & \forall \langle i,j \rangle \in E \end{aligned}$$

Following the removal of the b_{ij} variables, $\sum_{\forall \langle i,j \rangle \in E} W_{ij} b_{ij}$ can be reexpressed in terms of node variables only as shown in Equation 4.5 [LS91].

$$\sum_{\forall \langle i,j \rangle \in E} W_{ij}(a_j - a_i) = \sum_{\forall i \in V} \left(\left(\sum_{\forall \langle k,i \rangle \in E} W_{ki} - \sum_{\forall \langle i,j \rangle \in E} W_{ij} \right) a_i \right) \quad (4.5)$$

$$= \sum_{\forall i \in V} \sigma_i a_i \quad (4.6)$$

$$\sigma_i = \sum_{\forall \langle k,i \rangle \in E} W_{ki} - \sum_{\forall \langle i,j \rangle \in E} W_{ij}$$

Substituting Equation 4.5 into Equation 4.4 leads to Equation 4.7. Simplifying the con-

straints to a standardized form is shown in Equation 4.8

$$\begin{aligned}
 \text{Min } \sum_{\forall i \in V} \sigma_i a_i & & (4.7) \\
 a_j - a_i &\leq U_{ij} & \forall \langle i, j \rangle \in E \\
 a_j - a_i &\geq L_{ij} & \forall \langle i, j \rangle \in E
 \end{aligned}$$

$$\begin{aligned}
 \text{Min } \sum_{\forall i \in V} \sigma_i a_i & & (4.8) \\
 a_j - a_i - U_{ij} &\leq 0 & \forall \langle i, j \rangle \in E \\
 a_i - a_j + L_{ij} &\leq 0 & \forall \langle i, j \rangle \in E
 \end{aligned}$$

Once Equation 4.8 is formed, we are ready to find its dual function. First, we remove the constraints on a_i by using Lagrangian multipliers. Lagrangian multipliers remove constraints on an objective function by introducing new variables, known as Lagrangian multipliers, and multiplying these variables by the constraints. In our case, our Lagrangian multipliers are λ_{ij} and ρ_{ij} which creates a new objective function $L(a_i, \lambda_{ij}, \rho_{ij})$. Assuming that the Lagrangian multipliers are positive integers, they can be thought as a penalty factor which penalizes the objective function $L(a_i, \lambda_{ij}, \rho_{ij})$ whenever the constraints on the original objective function are not met. As a result, finding a valid $a_i, \lambda_{ij}, \rho_{ij}$ assignment that minimizes $L(a_i, \lambda_{ij}, \rho_{ij})$ is equivalent to finding a valid a_i that minimizes the original objective function.

$$L(a_i, \lambda_{ij}, \rho_{ij}) = \sum_{\forall i \in V} \sigma_i a_i + \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij} (a_j - a_i - U_{ij}) + \sum_{\forall \langle i, j \rangle \in E} \rho_{ij} (a_i - a_j + L_{ij}) \quad (4.9)$$

Although minimizing $L(a_i, \lambda_{ij}, \rho_{ij})$ will find us a valid a_i assignment, this is not useful since the complexity of minimizing $L(a_i, \lambda_{ij}, \rho_{ij})$ is equally as difficult as solving the

original objective function. Instead, we will *infimize* out the a_i variables in $L(a_i, \lambda_{ij}, \rho_{ij})$. This significantly reduces the complexity of our problem since the resulting function will be dependent only on variables λ_{ij} and ρ_{ij} and we no longer have to search for a solution in the domain of a_i .

To infimize a_i from $L(a_i, \lambda_{ij}, \rho_{ij})$, we will take the *infimum* of $L(a_i, \lambda_{ij}, \rho_{ij})$ with respect to variable a_i . The infimum of a function with respect to a variable x is defined as the greatest lower bound over the function over the domain of x . For example, Table 4.1 shows the infimum of several well known functions. Note that the infimum of the third and fourth function is also another function dependent on the remaining variables.

| $f(x)$ | $\inf_{x \in D} f(x)$ |
|--|-----------------------|
| e^x | 0 |
| $x^2 + 2$ | 2 |
| $x^2 + y$ | y |
| y | y |
| $mx + b, \{m, b\} \in \text{constant}, m \neq 0$ | $-\infty$ |
| $\sum_{i=0}^n m_i x_i + b_i, \forall i \{m_i, b_i\} \in \text{constant}, m_i \neq 0$ | $-\infty$ |

Table 4.1: Infimum of several functions

To derive the infimum of $L(a_i, \lambda_{ij}, \rho_{ij})$ with respect to a_i , we first separate out parts of $L(a_i, \lambda_{ij}, \rho_{ij})$ that are not dependent on a_i as shown in Equation 4.10. In Equation 4.11, we only show the terms dependent on a_i . These form a summation of linear functions $ma_i + b$, where m is equivalent to ρ_i or λ_{ij} , and $b = 0$. From Table 4.1, the infimum of a summation of linear functions is $-\infty$ ¹. However, as indicated in Table 4.1, there is one condition where the infimum is well defined (i.e. not $-\infty$). This occurs if $\sum_{\forall i \in V} \sigma_i a_i + \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij} (a_j - a_i) + \sum_{\forall \langle i, j \rangle \in E} \rho_{ij} (a_i - a_j)$ is constant zero (this is equivalent to the condition $\forall i m_i = 0$ in the last entry in Table 4.1), which in turn implies that Equation 4.11 evaluates to constant

¹The sum of a set of linear functions is known as an *affine* function

zero. Thus the infimum of $L(a_i, \lambda_{ij}, \rho_{ij})$ is defined on two cases as shown in Equation 4.12.

$$g(\lambda_{ij}, \rho_{ij}) = \inf_{a_i \in D} L(a_i, \lambda_{ij}, \rho_{ij}) \quad (4.10)$$

$$= \inf_{a_i \in D} \left(\sum_{\forall i \in V} \sigma_i a_i + \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij} (a_j - a_i - U_{ij}) + \sum_{\forall \langle i, j \rangle \in E} \rho_{ij} (a_i - a_j + L_{ij}) \right)$$

$$= \sum_{\forall \langle i, j \rangle \in E} \rho_{ij} L_{ij} - \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij} U_{ij}$$

$$+ \inf_{a_i \in D} \left(\sum_{\forall i \in V} \sigma_i a_i + \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij} (a_j - a_i) + \sum_{\forall \langle i, j \rangle \in E} \rho_{ij} (a_i - a_j) \right)$$

$$\inf_{a_i \in D} \left(\sum_{\forall i \in V} \sigma_i a_i + \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij} (a_j - a_i) + \sum_{\forall \langle i, j \rangle \in E} \rho_{ij} (a_i - a_j) \right) \quad (4.11)$$

$$g(\lambda_{ij}, \rho_{ij}) = \begin{cases} \sum_{\forall \langle i, j \rangle \in E} \rho_{ij} L_{ij} - \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij} U_{ij}, \\ \quad \text{if } \sum_{\forall \langle k, i \rangle \in E} \rho_{ki} - \sum_{\forall \langle i, j \rangle \in E} \rho_{ij} + \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij} - \sum_{\forall \langle k, i \rangle \in E} \lambda_{ki} = \sigma_i \quad \forall i \in V \\ -\infty, \quad \text{otherwise} \end{cases} \quad (4.12)$$

The result of the infimum of $L(a_i, \lambda_{ij}, \rho_{ij})$ is the dual function $g(\lambda_{ij}, \rho_{ij})$, which has two conditions. Since we are only interested in the well defined condition, we can form our

dual problem of the original ILP in Equation 4.1 as follows:

$$\begin{aligned}
 \text{Max} \quad & \sum_{\forall \langle i,j \rangle \in E} \rho_{ij} L_{ij} - \sum_{\forall \langle i,j \rangle \in E} \lambda_{ij} U_{ij} & (4.13) \\
 & \sum_{\forall \langle k,i \rangle \in E} \rho_{ki} - \sum_{\forall \langle i,j \rangle \in E} \rho_{ij} + \sum_{\forall \langle i,j \rangle \in E} \lambda_{ij} - \sum_{\forall \langle k,i \rangle \in E} \lambda_{ki} = \sigma_i \quad \forall i \in V
 \end{aligned}$$

If we negate the objective function in Equation 4.13, the problem becomes a cost minimization network flow algorithm where the dual variables λ_{ij} and ρ_{ij} are equivalent to the flow along each edge $\langle i, j \rangle$, U_{ij} and $-L_{ij}$ represents the cost per unit flow on each edge, and σ_i represents the flow demand on each node i . This is shown in 4.14 and whose optimal solution can be found in polynomial time [GT86, Gol97].

$$\begin{aligned}
 \text{Min} \quad & \sum_{\forall \langle i,j \rangle \in E} \lambda_{ij} U_{ij} - \sum_{\forall \langle i,j \rangle \in E} \rho_{ij} L_{ij} & (4.14) \\
 & \sum_{\forall \langle k,i \rangle \in E} \rho_{ki} - \sum_{\forall \langle i,j \rangle \in E} \rho_{ij} + \sum_{\forall \langle i,j \rangle \in E} \lambda_{ij} - \sum_{\forall \langle k,i \rangle \in E} \lambda_{ki} = \sigma_i \quad \forall i \in V
 \end{aligned}$$

An illustration of the mathematical transformation described previously is shown in Figure 4.13. Figure 4.13(a) shows the original partition graph where each node represents a partition and each edge represents connections between partitions. Figure 4.13(b) illustrates the converted graph with upper and lower bound constraints. Also, we show the addition of the dummy node v_0 which is necessary to handle the $a_i \leq U_i$ constraints. For clarity, we have shown this for only the top node.

Mapping dual variables to a_i solution

Solving Equation 4.14 finds an optimal assignments to variables λ_{ij} and ρ_{ij} . However, our original objective function seeks an optimal assignment to variables a_i . Thus, we must map the optimal solution λ_{ij}^* and ρ_{ij}^* in the dual solution space to the optimal solution a_i^* of the

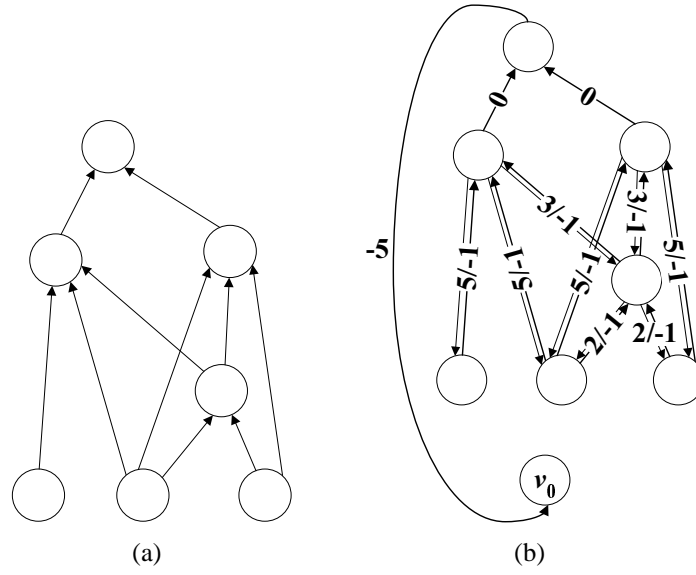


Figure 4.13: Graphical illustration of transforming the budget management problem into its dual network flow problem. (a) Partitioned graph. (b) Upper and lower bound added as edges to create network flow graph.

original objective function. In order to do this, we leverage the concept of complementary slackness. This is stated in Proposition 4.2.2.

Proposition 4.2.2 *Complementary slackness implies that for an optimal assignment to λ_{ij}^* and ρ_{ij}^* Equation 4.15 must hold.*

$$\lambda_{ij}^*(a_j^* - a_i^* - U_{ij}) = 0, \quad \forall \langle i, j \rangle \in E \quad (4.15)$$

$$\rho_{ij}^*(a_i^* - a_j^* + L_{ij}) = 0, \quad \forall \langle i, j \rangle \in E$$

Proof: Equation 4.15 is proven in Equation 4.16 [BV04, ch.5]. The first line states that the optimal value of the original objective function is equal to the optimal value of its dual. This is already assumed to be true from understanding duality for convex functions. The second line is simply the definition of the dual function in expanded form, which is defined as the infimum of the Lagrangian function with respect to a_i . The next inequality states that the infimum will always be less than or equal to any value of the Lagrangian. This is

true since the infimum by definition is always less than its function (i.e. the greatest lower bound). Finally, the last line states that the Lagrangian terms will always be less than or equal to zero, and as a result, the Lagrangian will always be less than or equal to the original objective function. Since the last line is equivalent to the original objective function, this implies that $\lambda_{ij}^*(a_j^* - a_i^* - U_{ij}) = 0$ and $\rho_{ij}^*(a_i^* - a_j^* + L_{ij}) = 0$ for all $\langle i, j \rangle \in E$.

$$\begin{aligned}
 \sum_{\forall i \in V} \sigma_i a_i^* &= g(\lambda_{ij}^*, \rho_{ij}^*) & (4.16) \\
 &= \inf_{a_i \in D} \left(\sum_{\forall i \in V} \sigma_i a_i + \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij}^*(a_j - a_i - U_{ij}) + \sum_{\forall \langle i, j \rangle \in E} \rho_{ij}^*(a_i - a_j + L_{ij}) \right) \\
 &\leq \sum_{\forall i \in V} \sigma_i a_i^* + \sum_{\forall \langle i, j \rangle \in E} \lambda_{ij}^*(a_j^* - a_i^* - U_{ij}) + \sum_{\forall \langle i, j \rangle \in E} \rho_{ij}^*(a_i^* - a_j^* + L_{ij}) \\
 &\leq \sum_{\forall i \in V} \sigma_i a_i^*
 \end{aligned}$$

■

The complementary slackness condition in Equation 4.15 is useful since it gives a relationship of each a_i variable as shown in Equation 4.17.

$$\forall \langle i, j \rangle \in E \quad a_j^* - a_i^* = \begin{cases} U_{ij}, & \text{if } \lambda_{ij}^* > 0 \\ L_{ij}, & \text{if } \rho_{ij}^* > 0 \end{cases} \quad (4.17)$$

A problem with Equation 4.17 is that if $\rho_{ij}^* > 0$ and $\lambda_{ij}^* > 0$, $a_j^* - a_i^*$ is undefined. However, we can prove that such a condition is impossible in the following:

Proposition 4.2.3 *If λ_{ij}^* and ρ_{ij}^* is the minimum cost solution to Equation 4.14, then $\lambda_{ij}^* \times \rho_{ij}^* = 0$, $\forall \langle i, j \rangle \in E$.*

Proof: We know that assignments λ_{ij}^* and ρ_{ij}^* provide the solution to objective function in Equation 4.14 with a minimum cost value of M . Now assume that for a given edge $\langle i, j \rangle$,

$\lambda_{ij}^* \times \rho_{ij}^* \neq 0$. This implies that there exists another feasible flow where $\lambda_{ij} = \lambda_{ij}^* - 1$ and $\rho_{ij} = \rho_{ij}^* - 1$ (this new flow will not violate flow conservation). This results in a new flow cost of $M' = M + (-U_{ij} + L_{ij})$. However, since $U_{ij} > L_{ij}$, this implies that $M' < M$. Thus, M is not the minimum cost flow, and we have a contradiction and $\lambda_{ij}^* \times \rho_{ij}^* = 0, \forall \langle i, j \rangle \in E$. This is illustrated in Figure 4.14. ■

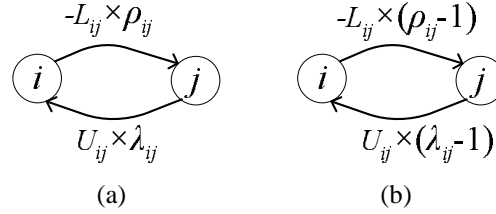


Figure 4.14: Illustration of proof for Proposition 4.2.3. (a) Total cost $M = -L_{ij} \times \rho_{ij} + U_{ij} \times \lambda_{ij}$. (b) Alternate feasible flow that does not violate the flow conservation of node i or j . Total cost $M' = -L_{ij} \times (\rho_{ij} - 1) + U_{ij} \times (\lambda_{ij} - 1) = M + L_{ij} - U_{ij}$, since $L_{ij} < U_{ij}$ then $M' < M$.

Equation 4.17 only gives a relationship between the solution a_i^* and its upper and lower bound values. To find the exact assignment to each a_i^* , we will formulate a relationship between each a_i variable and the shortest path between each node i and a primary input found in the network flow residual graph. A residual graph of a network flow graph can be thought as the “residual” flow capacity that exists for a feasible flow solution in the graph. It is created by adding a reverse edge for every edge that has flow along it and adding an edge for any edge that has any remaining flow capacity within it. For our problem, the residual graph is constructed by adding an additional backward edge for every edge which has flow through it. That is, for every $\lambda_{ij}^* > 0$ and $\rho_{ij}^* > 0$, add an edge $\langle j, i \rangle$. The cost of the backward edge is the negative cost of the forward edge $\langle i, j \rangle$. Once constructed, we can find the shortest path between a primary input to every node i . It turns out that the shortest path value, d_i , found for every node i is equivalent to $-1 \times a_i^*$. This is stated more formally in the following:

Proposition 4.2.4 *The shortest path value d_i from a primary input to a node i in the resid-*

ual graph is equivalent to $-1 \times a_i^*$ where a_i^* is the optimal value of the original objective function.

Proof: We will prove the case when $\rho_{ij}^* > 0$ where proving the case for $\lambda_{ij}^* > 0$ is similar. After solving the shortest path algorithm there exists a value d_i for every node i in the graph. If $\rho_{ij}^* > 0$, this implies that there exists a forward edge $\langle i, j \rangle$ and backward edge $\langle j, i \rangle$ with costs $-L_{ij}$ and L_{ij} respectively. Due to the shortest path definition, this implies that $d_j - d_i \leq -L_{ij}$ and $d_i - d_j \leq L_{ij}$. To satisfy both of these conditions, $d_i - d_j = L_{ij}$, which is equivalent to the relationship in Equation 4.17 where $d_i = -1 \times a_i^*$ and $d_j = -1 \times a_j^*$. ■

After calculating the a_i^* values, we can derive the budget for each edge as $b_{ij} = a_j^* - a_i^*$. An illustration of the mathematical transformation described previously is shown in Figure 4.15.

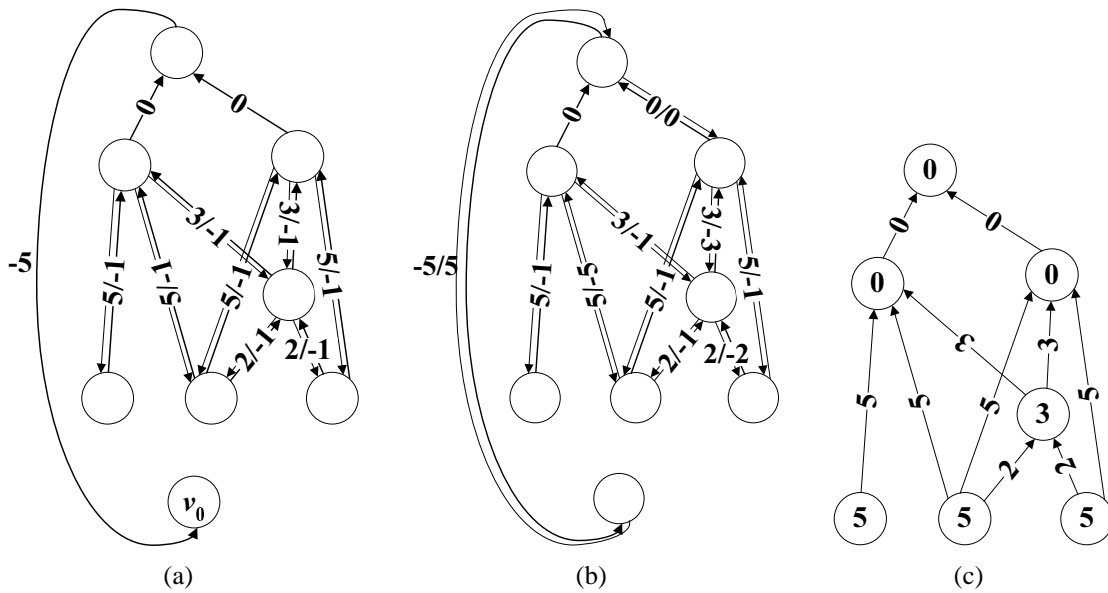


Figure 4.15: Graphical illustration of finding a_i^* values from the residual network flow graph. (a) Dual network flow graph. (b) Residual graph formed after solving dual problem. (c) a_i^* values found for each node i along with resulting b_{ij} values for each input edge.

Modelling area-delay relationship

In the previous sections, we introduced the area-delay relationship used for each partition as a piecewise-linear convex function. This allows us to change the delay-area relationship with respect to the delay, which often occurs when varying the depth of a given input path. For example, in Figure 4.16, reducing the depth of input a by 1 has no impact on area; however, reducing the depth of input a by 2 has an area penalty of 2 extra gates.

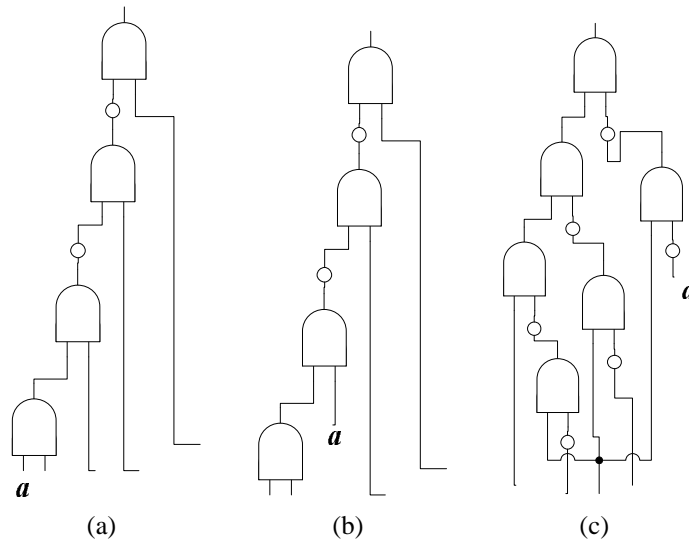


Figure 4.16: Illustration of area penalty with depth reduction and inverted edges.

To model the delay-area relationship, we leverage the information provided by the AIG, where we count the number inverted edges along the longest path between the input and the partition output. The key insight is that for a given input path, the depth of the path can be reduced with no area penalty until the depth is reduced beyond an inverted edge along the path. This phenomenon is shown in the previous example in Figure 4.16 where input a caused an area penalty only when its path depth was reduced by more than 1. Thus, to reduce the depth of a path to M_{ij} , has no area penalty. However, modelling the delay-area relationship as a constant does not model the fact that increasing the depth of an input gives more slack to other paths, which in turn may be optimized for area. For example,

Figure 4.17 illustrates two partitions. If the path depth along input a of the top partition is reduced, this forces the path depth of input b to increase as shown in Figure 4.17(b). This could be harmful since the bottom partition may now need to reduce its depth since input path b is now longer, which in turn could lead to an area increase.

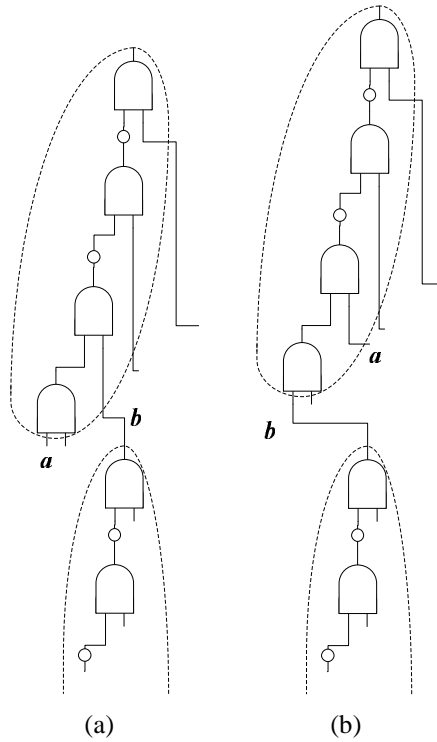


Figure 4.17: Illustration of how reducing the depth of a path impacts decisions along other paths and partitions.

To take into account the issue highlighted in the Figure 4.17, we model the area-delay relationship of $F_{ij}(b_{ij})$ in the interval $b_{ij} \in [M_{ij}, U_{ij}]$ as a line with a slope of -1. Here, U_{ij} is an upper bound assigned to b_{ij} , which basically states that path ij cannot expand beyond the depth U_{ij} . For the interval $b_{ij} \in [L_{ij}, M_{ij}]$, $F_{ij}(b_{ij})$ is modelled as a line with a slope of -2. This states that attempting to reduce the depth of the partition input beyond the number of inverted edges along the longest path has a stronger penalty. Here, L_{ij} is a lower bound assignment to b_{ij} which states that the depth of the input path ij cannot be reduced beyond L_{ij} .

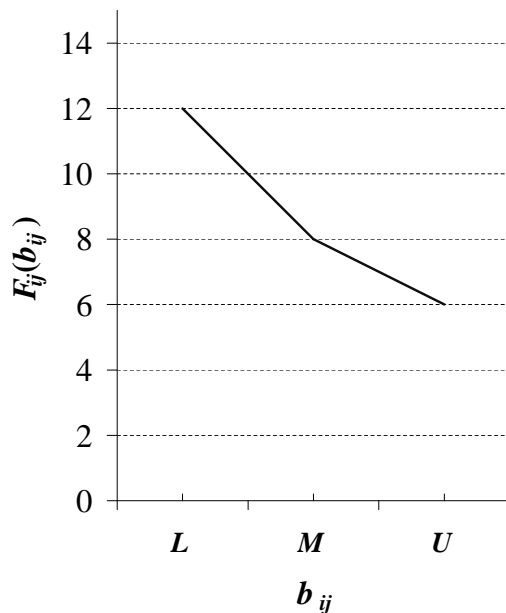


Figure 4.18: Illustration of area-depth relationship for function $F(b_{ij})$.

Assigning Upper and Lower Bound

For each b_{ij} variable, we have an upper and lower bound U_{ij} and L_{ij} . To assign the upper bound, we use the work presented in [BD97]. Here, the authors develop a disjoint decomposition algorithm based on the BDD data structure. Using a BDD representation of a function, [BD97] show how to decompose a BDD directly into basic gates and multiplexers. The authors show how to efficiently find a decomposition that minimizes area using the BDD data structure where most of the time a 2-input basic gate could be derived from each node in the BDD. Knowing that the maximum depth of a given BDD is N where N is the number of variables in the BDD and drawing from the work in [BD97] which shows how we can map a BDD node directly to a 2-input gate in the majority of cases, we set the upper bound for each edge to N . To set the lower bound, we take a look at two conditions. If there are no inverted edges along a input to the root node of the partition, the path must go through at least one AND gate, implying that the lower bound is 1. If there are inverted edges along this path, it must go through at least one additional inverted edge, thus in this

case we set the lower bound to 2.

4.2.3 Resynthesis with Delay Budgets

Once we have a delay budget assigned to each incoming edge of a partition, we can resynthesize the partition. To guide the resynthesis process, the delay budgets are used to alter the input depth of each partition input. Doing so penalizes input paths which should be optimized for depth, while giving freedom to input paths that can be optimized for area. We alter the depth of each partition input, such that nodes with a small delay budget are assigned a higher depth value. The depth assignment, δ_{ij} , for a partition i is defined as the maximum budget assigned to its partition inputs minus the budget assigned to edge $\langle i, j \rangle$ (Equation 4.18).

$$\delta_{ij} = \text{Max}\{b_{ij} \mid j \in \text{fanin}(i)\} - b_{ij} \quad (4.18)$$

An example of this assignment is shown in Figure 4.19. In Figure 4.19(a), each number assigned to the inputs represents the depth budget found after solving the dual problem described previously. After this, we derive depth assignments, δ_{ij} , for each input j , where in this case $\text{Max}\{b_{ij} \mid j \in \text{fanin}(i)\}$ equals 5. Looking at the depth assignments in Figure 4.19(b), the inputs with an assignment greater than zero are penalized to alter the input depth. This alters resynthesis decisions to attempt to reduce the path along penalized input paths more than the depth along non-penalized input paths.

4.3 Results

We empirically validate our analysis in the previous sections here. First, we do a comparison between solving the budget management problem as an ILP versus using the dual problem that was derived in Section 4.2.2. When solving budget management as an ILP, we use the commercial MOSEK ILP solver [MOS08]. For the dual problem, we create our

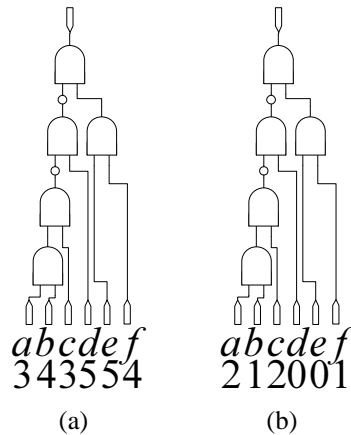


Figure 4.19: Depth assignments to inputs after delay budget assignments. (a) Delay budgets assigned to each partition input. (b) Depth adjustment found from Equation 4.18. Assignments to each partition input are used to drive the resynthesis engine.

dual solver with the assistance of Andrew Goldberg’s network flow solver [Gol97]. Following this, we evaluate our budget management framework in the context of a partition based logic synthesis flow.

4.3.1 Dual Problem Performance

Table 4.2 illustrates our performance results between the commercial ILP solver and our dual formulation where on average, the dual problem is two orders of magnitude faster than solving the budget management problem in its native ILP form. Since we use the notion of duality to achieve this speedup, optimality is preserved and the same result is found in both the ILP formulation and dual problem. We ran our problem on over 100 different IWLS circuits, where we show only the largest circuits in detail. In Table 4.2, the first column lists the circuit name, followed by its size in terms of 2-input AND gates. The last two columns lists the runtime in seconds of the MOSEK ILP solver followed by the dual problem runtime using Goldberg’s network flow solver. The speedup experienced by the dual solver is a necessary condition in order to apply our budget management problem in

| Circuit | nNodes | Time _{ILP} (sec) | Time _{NF} (sec) |
|-----------------------------|--------|---------------------------|--------------------------|
| b14_1 | 7196 | 27.72 | 0.40 |
| b14 | 8248 | 105.31 | 0.32 |
| b15_1 | 12874 | 91.52 | 1.18 |
| b15 | 14008 | 60.56 | 1.66 |
| b20_1 | 14124 | 128.11 | 0.96 |
| b21_1 | 14672 | 128.14 | 1.23 |
| b20 | 16452 | 154.31 | 1.02 |
| b21 | 17560 | 158.61 | 1.50 |
| b22_1 | 21764 | 107.14 | 1.58 |
| b22 | 25144 | 217.34 | 2.06 |
| b17_1 | 36692 | 1185.28 | 6.16 |
| b17 | 39956 | 3185.96 | 6.96 |
| des_perf | 88586 | 3185.96 | 6.96 |
| vga_lcd | 152594 | 8530.26 | 17.27 |
| <i>ILP/NF_{ave}</i> | | | 101x |

Table 4.2: Runtime comparison of ILP and NF formulation, over 100 circuits ran, only largest circuits shown. Run on a Pentium 4, 2.80GHz with 2GB of RAM

the context of logic synthesis.

4.3.2 Budget Management for Partitioned Logic Synthesis

After verifying that our dual formulation is indeed scalable to large designs, we applied our budget management framework to logic synthesis. We compared two flows: a partitioned based flow without budget management, and a partitioned flow with budget management. To synthesize and technology map each partition, we used the ABC logic synthesis engine running the script `resyn2.rc`. After applying each logic synthesis flow, we measured the area and depth of the resulting circuits in terms of 4-input LUTs.

Table 4.3 highlights our results when applied to the IWLS benchmark set (only larger circuits shown in detail). In Table 4.3, the first column lists the circuit name, the next two columns show the area results, and the final two columns show the depth results. For the area results, we record the number of 4-input LUTs and show two columns of data listing

| <i>Circuit</i> | <i>Area (4-LUTs)</i> | | <i>Depth</i> | |
|--------------------------------------|----------------------|---------------|-----------------|---------------|
| | <i>NoBudget</i> | <i>Budget</i> | <i>NoBudget</i> | <i>Budget</i> |
| b14_1 | 1799 | 1791 | 21 | 19 |
| b14 | 2062 | 2068 | 22 | 20 |
| b15_1 | 3005 | 3112 | 18 | 16 |
| b15 | 3189 | 3360 | 25 | 23 |
| b20_1 | 3531 | 3546 | 26 | 22 |
| b21_1 | 3668 | 3670 | 26 | 21 |
| b20 | 4113 | 4135 | 27 | 23 |
| b21 | 4390 | 4359 | 27 | 23 |
| b22_1 | 5441 | 5441 | 26 | 22 |
| b22 | 6286 | 6278 | 26 | 23 |
| b17_1 | 9173 | 9181 | 18 | 16 |
| b17 | 9989 | 10230 | 38 | 32 |
| des_perf | 31977 | 31759 | 6 | 6 |
| vg_lcd | 33165 | 34739 | 8 | 8 |
| Ratio Geomean_{ALL} | | 1.00 | | 0.95 |
| Ratio Geomean_{LARGE} | | 1.01 | | 0.89 |

Table 4.3: Impact of budget management framework on area and depth for the IWLS set, only larger circuits shown in detail.

the non-budgeted results followed by the budgeted results. Similarly, the depth results record the number of logic levels and show two columns listing the non-budgeted results followed by the budgeted results. The second last line in Table 4.3 shows the average comparison of the entire benchmark set. This shows that budgeting overall only improves the circuit depth by 5%. However, after examining many circuits, it turned out that if a partition could capture most of the critical paths in the circuit, budgeting was not necessary. Budgeting is not necessary in such cases since a single partition has the true critical path of the entire circuit does not need external information to optimally reduce the critical path of the circuit. This phenomenon generally occurred in small circuits with a very low number of logic levels. Ignoring these circuits, our improvement to circuit depth increases to 11% with little impact to the final area. The improvements we achieve in Table 4.3 almost comes for free as the overhead to run the budget management flow is negligible where it takes less

than 2 minutes for the larger IWLS circuits.

4.4 Summary

We have described a budget management framework which is empirically shown to be scalable and applicable to improve a partitioned based logic synthesis flow. By converting our budget management problem to its dual form, we show how we can improve the runtime of the application by two orders of magnitude. Furthermore, when applied in the context of a logic synthesis flow, we improve the logic depth by 11% with negligible impact to area.

As partitioning becomes more common in existing CAD flows, budget management is a necessary condition to ensure that QoR is maintained while improving the scalability of the FPGA CAD flow.

5 Automation of ECOs

As FPGA design complexity increases, achieving tight timing and area constraints is becoming very challenging and often requires several design iterations as shown at the top of Figure 5.1. Here, a design described in a Hardware Description Language (HDL) is passed to an FPGA CAD flow such as Quartus II. If design constraints are not met after the CAD flow finishes, the designer must modify their existing HDL code and rerun the CAD flow. To complicate things, even if performance constraints are met, bug fixes or feature changes may still be required. Applying these changes directly at the HDL and using a “from scratch” FPGA recompile is often not an option since this does not guarantee that circuit performance will be maintained. To avoid this problem, designers typically handle late-stage changes through a process known as engineering change orders (ECOs). ECOs are small functional changes applied directly to a place-and-routed netlist such as the rewiring of LUTs or changing LUT implementations. As a result, ECOs have a very predictable impact on the final performance of the circuit and preserve much of the engineering effort previously invested in the design.

In a design flow where almost all tasks are automated, ECOs remain a primarily manual process. As a result, applying ECOs is very error-prone and can require several iterations to correctly modify a design. This makes FPGA very difficult to use as a digital design medium. The ECO process is illustrated in Figure 5.1 through the path labelled as *Original Flow*. The feedback process shown in *Original Flow* can often tie up a designer for several months [Mor06a, Gol04, Goe07, Pla05] which leads to missed project deadlines. To make

things worse, this methodology will not scale since FPGA designs double in size with each generation. With very short design cycles demanded today, this will be very detrimental to a product's success.

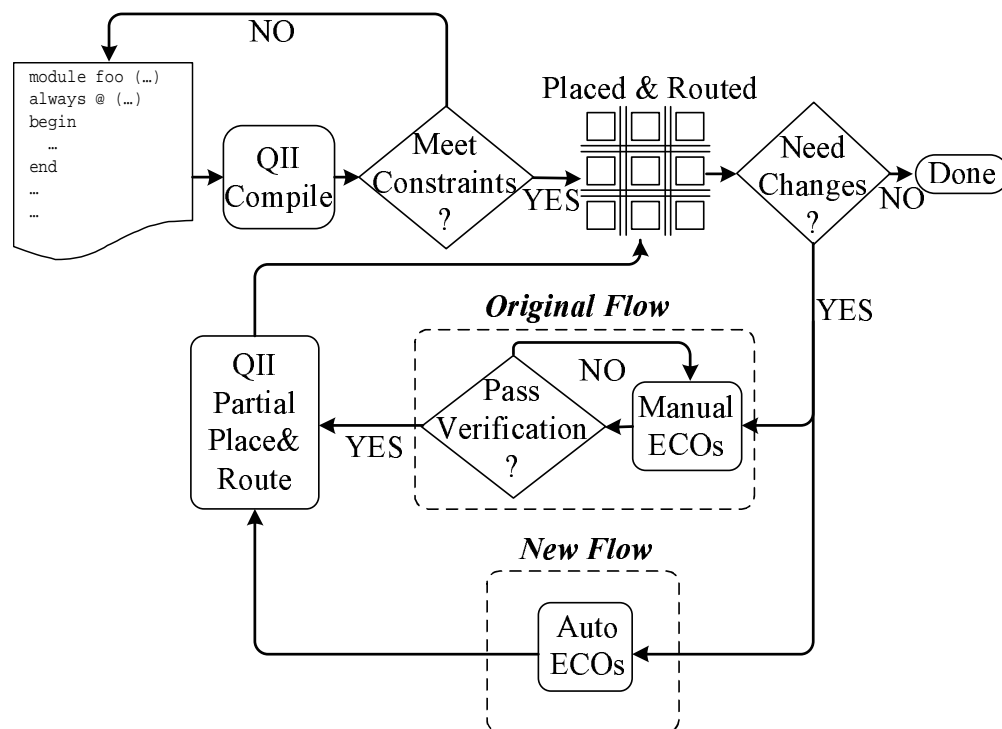


Figure 5.1: Generalized flow using ECOs.

To avoid the expensive and error-prone nature of ECOs, we introduce a resynthesis approach that is able to automatically update the behaviour of a circuit by using existing logic within the design. This is beneficial since it requires no manual intervention. This can potentially reduce the time spent on ECOs from weeks to a few hours.

The proposed flow is shown at the bottom of Figure 5.1 labelled as *New Flow*. Unlike the *Original Flow*, the *New Flow* does not require an iterative approach. During *New Flow*, the circuit behaviour is updated using two steps. First, the proposed algorithm applies a localization step which isolates nodes within the netlist which need to be changed. For example, during bug-fixes, the localization step will isolate nodes which are the source of

the erroneous behaviour of the circuit. Following this, the proposed algorithm applies a resynthesis step to alter the circuit. The resynthesis step first defines the function that will replace the existing nodes found in the localization step. Next, the proposed algorithm uses a SAT-based decomposition to break our necessary changes into a set of subfunctions. During decomposition, subfunctions must exist within the place-and-routed netlist. This ensures that the proposed algorithm has a predictable impact on the final performance of the circuit. By using a SAT-based approach, we will show that unlike previous approaches, we can handle multiple changes and do not require a formal verification step to ensure that our modifications are correct.

The ultimate goal of this work is to create an ECO flow which can automatically update the behaviour of a circuit in a manner which preserves as much as possible the placement and routing of the circuit while maintaining timing. This can be achieved by satisfying the following three criteria:

- be minimally disruptive to an existing place-and-routed circuit.
- have little or no impact to the performance of a circuit.
- be robust enough to handle a wide range of changes.

In later sections, we will show that our approach does indeed satisfy the previous three criteria. In particular, when applied to Altera's Stratix architecture using several benchmarks from the Altera QUIP [Alt08] and the ITC benchmark suites we will show our technique on average leaves over 90% of a place-and-routed circuit unchanged; has a marginal impact to circuit performance where we reduce the frequency by less than 3% on average; and is robust enough to handle over 80% of the changes presented. In all cases, we require no manual intervention by the user.

The rest of the chapter is organized as follows: Section 5.1 discusses the background and related work; Section 5.2 describes the proposed algorithm in detail; Section 5.3 shows

experimental results; and Section 5.4 concludes the chapter.

5.1 Background and Related Work

5.1.1 Terminology

The combinational portion of a LUT network can be represented as a directed acyclic graph (DAG). A node in the graph represents a LUT, primary input (PI), or primary output (PO). An *edge* in the graph with head v , and tail u , represents a signal in the logic circuit that is an output of node u and an input of node v . Primary inputs (PIs) are nodes with no inputs, and primary outputs (POs) are nodes with no outputs.

A node u is a *transitive fanout (fanin)* of v if there exists a path between an input (output) of u and an output (input) of v . In this chapter, we will often use the term transitive fanout or transitive fanin of a node v to refer to all nodes which are a transitive fanout or transitive fanin to node v .

When defining a function at a node v , it is termed a *global function* if the function support set consist of only PIs. Conversely, a *local function* is a function whose support set consists of variables which may not be PIs. A *support set* of a function is the set of variables that directly determine the output value of a function and the size of the support set is the number of variables in the set.

5.1.2 Related Work

ECOs covers a wide range of work which is either used to incrementally improve the performance of a design [CS00] or help modify the behaviour of a design such that circuit performance is maintained [MCB89, CMB08, YSVB07, HCC99, Men05, Xil08]. Our

work falls in the latter category where we focus on late-stage ECOs which are applied directly to a place-and-routed netlist. Late-stage functional changes often occur due to last minute feature changes or due to bugs which have been missed in previous verification phases. The most recent steps toward the automation of the ECO experience include [CMB08] and [YSVB07]. Here, using formal methods and random simulation, the authors in [CMB08, YSVB07] show how netlist modifications can be automated. To apply modifications, the authors use random simulation vectors to stimulate the circuit. Using the resulting vectors at each circuit node, they are able to find suggested alterations to their design to match a specified behaviour. Following their modifications, they require a formal verification step to ensure that their modification is correct. The results of their work is promising where they can automatically apply ECOs in more than 70% of the cases they present.

The technique in [YSVB07, CMB08] requires an explicit representation of any modification, which does not scale to large changes. This is not a problem in ASICs since ECOs requiring major changes are not desired since they are difficult to implement; however in FPGAs, where we can reprogram logic cells without disrupting the placement, large changes can be implemented while maintaining circuit performance. Our approach improves on this where we can handle much larger changes by using a SAT-based approach shown in the following sections.

5.1.3 Boolean Satisfiability (SAT)

For completeness, this section gives a brief overview of the SAT problem. A more detailed description of the SAT problem is found in Chapter 2, Section 2.5. Given a Boolean expression $F(x_1, x_2, \dots, x_n)$, Boolean satisfiability (SAT) seeks an assignment to the variables, x_1, x_2, \dots, x_n , such that F evaluates to true. If this is possible, F is said to be *satisfiable*, otherwise, it is *unsatisfiable*. SAT solvers are tools that serve to find if a Boolean formula is

satisfiable or not. For practical reasons, modern day SAT solvers work on Boolean expressions in Conjunctive Normal Form (CNF). An example of a Boolean expression is shown in equation F_{AND} in Figure 5.2

Recent advancement in SAT solvers [MMZ⁺01] have improved their runtime performance by an order of magnitude, thus several problems in EDA can be practically solved using SAT. In order to solve circuit problems with SAT, often the circuit needs to be converted into a Boolean expression which is then input into the SAT solver. This Boolean expression [Lar92] is known as a *characteristic function* for the circuit. The characteristic function of a circuit evaluates to true if all variable assignments of the wires, inputs, and outputs of the circuit have a feasible assignment. For example, consider the AND gate shown in Figure 5.2. The table to the left gives the truth table for the AND gate characteristic function which can be converted to CNF using any standard minimization technique. Figure 5.3 shows how to derive the characteristic function of larger circuits by conjoining the characteristic functions of individual gates within the circuit.

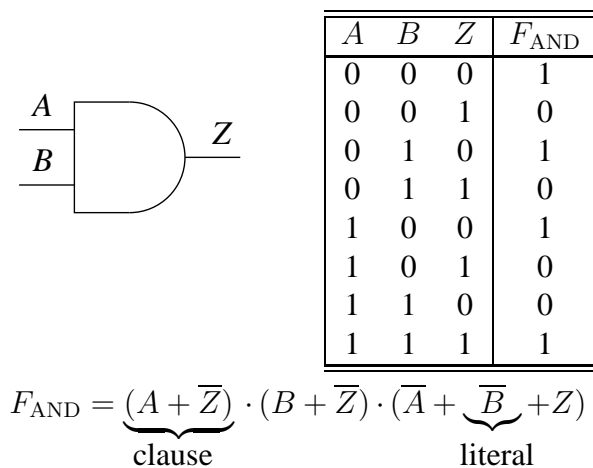


Figure 5.2: Characteristic function derivation for 2-input AND gate.

After deriving the characteristic function of a circuit, we can use it in conjunction with SAT solvers to examine logical properties of the circuit for various CAD problems. For example, in Section 5.2.1, we will show a SAT-based technique using the characteristic

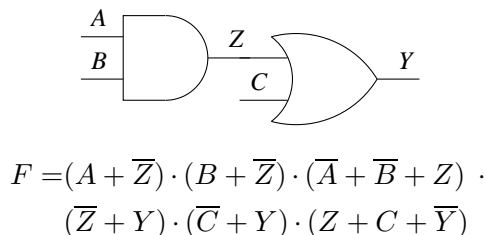


Figure 5.3: Cascaded gate characteristic function: top clauses from AND gate; bottom clauses from OR gate.

function to isolate errors within a design.

In this chapter, we will often refer to a circuit as being *satisfiable* without explicitly showing its characteristic function. We define a circuit as being satisfiable if there is a satisfying assignment to the characteristic function of the circuit. The circuit is unsatisfiable if no such assignment exists. Furthermore, a *satisfying assignment* to a circuit is an assignment which is consistent with the circuit's characteristic function. For example, one possible satisfying assignment to the circuit in Figure 5.3 is $ABCZY = 01101$.

5.2 Automated ECOs: General Technique

From a manual perspective, ECOs are applied by first isolating the logic that needs to be changed. Following this, the designer must determine what to map the existing logic to, that is, what new logic functions are required to correctly change the circuit. Finally, the designer must determine how to implement these new logic functions within the context of the existing place-and-routed circuit. In this work, we propose a method to automate all of these steps using Boolean satisfiability. We do so by breaking our problem up into two steps: netlist localization, followed by netlist modification. This flow is highlighted in Figure 5.4.

Before netlist localization or netlist modification can occur, the desired behaviour of the

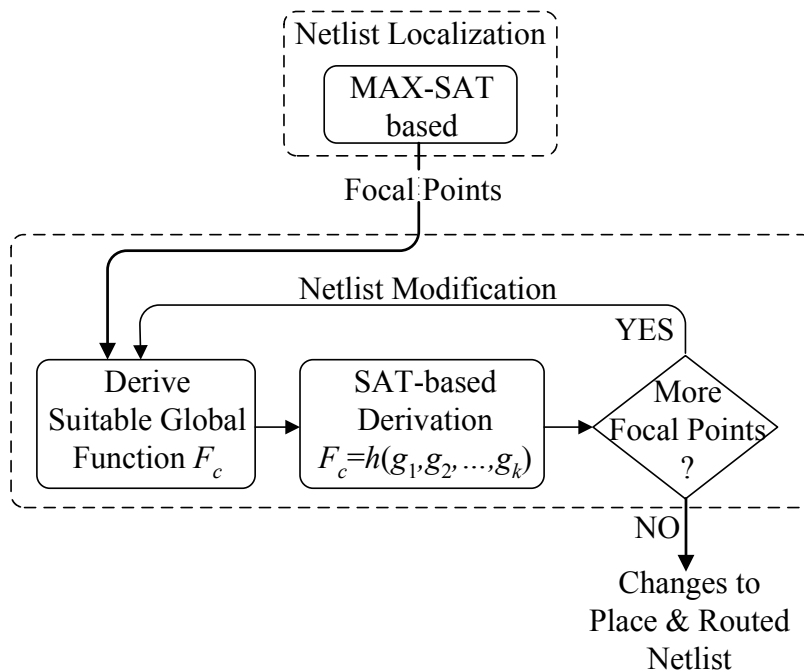


Figure 5.4: Automated ECO flow: a MAX-SAT based netlist localization step followed by a SAT based Netlist Modification step.

circuit must be well defined. This is the behaviour that we ultimately want our circuit to implement. To model this desired behaviour, we will use a *model netlist*. This model netlist is a logic netlist which defines the new input and output behaviour of the circuit. Once we have the model netlist, netlist localization and modification can occur.

Netlist localization finds a set of locations which we will refer to as *focal points*. These focal points help isolate regions that need to be modified. In this work, we propose using a MAX-SAT formulation first introduced in [SMV⁺07] to find our focal points. Following netlist localization, the *netlist modification* occurs. During this step, the proposed algorithm will leverage existing logic found in the design to help rewire the nodes at each focal point such that the resulting circuit will match the behaviour of the model netlist. This starts by first deriving the proper global function F_c to replace the logic at each focal point. After a new global function is found, a search of the netlist is conducted to find a proper set of subfunctions, $\{g_1, g_2, \dots, g_k\}$, to implement the new global function F_c such

that $F_c = h(g_1, g_2, \dots, g_k)$. Since these subfunctions, which we will refer to as *basis functions*, already exist in the place-and-routed netlist, the only required changes to the netlist revolves around the implementation of the *dependency function* h . As we will show in Section 5.2.2, implementing h requires much fewer changes than implementing F_c from scratch and in this work, we will show how we both select our basis functions g_i and derive the dependency function h using SAT. This makes the proposed algorithm well suited for ECOs which must preserve the existing place-and-routed circuit as much as possible.

5.2.1 Netlist Localization

Netlist localization is the process of identifying which nodes must be changed in order to change the behaviour of the circuit such that it matches the behaviour of the model netlist. The approach we use to find these nodes is based upon the work presented in [SMV⁺07]. Here the authors introduce a localization technique based upon Maximum Boolean Satisfiability (MAX-SAT). Like SAT, MAX-SAT seeks a satisfiable variable assignment to a CNF. However, in the case of unsatisfiability, MAX-SAT seeks an assignment which maximizes the number of satisfying clauses. Using the set of unsatisfiable clauses, we can isolate discrepancies between two netlists.

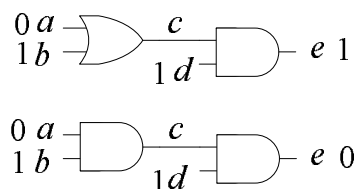


Figure 5.5: Maximum satisfiability solution to localization.

This process is illustrated in the Figure 5.5. Consider the two circuits in Figure 5.5. The top circuit is the model netlist while the bottom circuit is the implementation of the original circuit. Under the input stimulus $\{a = 0, b = 1, d = 1\}$ the original circuit has a response of $\{e = 0\}$ which conflicts with the model response of $\{e = 1\}$. The corresponding CNF

for the original circuit and the input/output vectors are shown below.

$$\begin{aligned}
 & (\bar{a}) \cdot (b) \cdot (d) \cdot (e) \cdot \\
 & (a + \bar{c}) \cdot (b + \bar{c}) \cdot (\bar{a} + \bar{b} + c) \cdot \\
 & (c + \bar{e}) \cdot (d + \bar{e}) \cdot (\bar{c} + \bar{d} + e)
 \end{aligned} \tag{5.1}$$

The above equation is unsatisfiable; however, the MAX-SAT problem will attempt to maximize the number of satisfied clauses. One possible solution to this is satisfying all clauses except for clause $(a + \bar{c})$. Since clause $(a + \bar{c})$ is derived from the AND-gate, this indicates that the function output from this gate (i.e. function on wire c) is the source of the discrepancy. Clause (\bar{a}) could have also been identified in the MAX-SAT solution which does not help identify the discrepancy in our circuit. To avoid fruitless clauses from being identified, such clauses can be forced a satisfying assignment prior to solving the MAX-SAT problem.

This type of localization is known as trace-based localization where a vector set is used to help identify discrepancies between two netlists. The main limitation with this approach is that if only a small set of vectors or traces are available, false locations will be identified. However, assuming that a large enough set of vectors are available, an iterative approach can be applied to the localization process where the number of potential locations are reduced.

In our case, we will often refer to discrepancy points isolated by the localization point as *focal points*. Also, in many examples, we will assume there only exists one focal point, which we will refer to as *single-location modifications*.

In the example shown in Figure 5.5, the original circuit and model netlist exist as a gate-level netlist. This is not necessary, since we are only interested in the characteristic function of the current circuit and IO response of the model netlist; not their implementation. This is important in our case where our circuit exists as a netlist of LUTs on Altera's Stratix

architecture and our model netlist consists of a netlist of AND and inverter gates (AIG) synthesized from an RTL netlist.

5.2.2 Netlist Modification

After netlist localization occurs, the netlist can be modified. Modifications are centered around the *focal points* that were identified during the localization phase. Referring back to Figure 5.4, the process of netlist modification requires a series of steps. First, we must define a global function F_c to replace the existing global function at each focal point. Following this, we must decompose the replacement function F_c such that it leverages existing logic within the place-and-routed netlist. We will show later on that we use a specialized circuit construct to define an implicit representation of the on and offset of F_c . This significantly enhances the scalability of our approach. Once we have found a suitable representation for F_c , we will then decompose F_c into a set of basis functions. Since our basis functions already exist within the place-and-routed netlist, we have significant control in how we decompose our function, which is a requirement if we want to minimally disrupt the existing place-and-routed circuit.

To understand why we need to find a suitable global function to replace the function at each focal point, consider the circuit illustrated in Figure 5.6. Assume node ξ is identified during the localization step. The transitive fanin and fanout for node ξ are labelled as TFI and TFO . The inputs to the circuit are labeled as the variable set \mathbf{X} , and the outputs of the circuit are labeled as the variable set \mathbf{Y} . Also, note that we illustrate some of the input branches to the transitive fanout with dotted lines which will be important later on in the following figure (Figure 5.7). Assuming we can change the global function $H(\mathbf{X}_p)$ to any arbitrary function $F_c(\mathbf{X}_s)$ where $\mathbf{X}_p \subseteq \mathbf{X}$ and $\mathbf{X}_s \subseteq \mathbf{X}$, we can alter the circuit behaviour to match our model netlist. Thus, the ECO problem becomes one of finding a suitable global function $F_c(\mathbf{X}_s)$ to replace the existing $H(\mathbf{X}_p)$. This leads to the following definition and

lemma.

Definition 5.2.1 *Suitable Global Function:* In the case of a single focal point, ξ , identified during the localization step, a suitable global function F_c is defined as a function which can be used to replace an existing global function H found at ξ such that the resulting circuit matches the input-output behaviour of a model netlist.

Lemma 5.2.2 For a single-location modification, such as the one shown in Figure 5.6, the circuit behaviour of the original circuit can be altered to match a model netlist if and only if the global function located at focal point ξ is replaced with another suitable global function $F_c(\mathbf{X}_s)$.

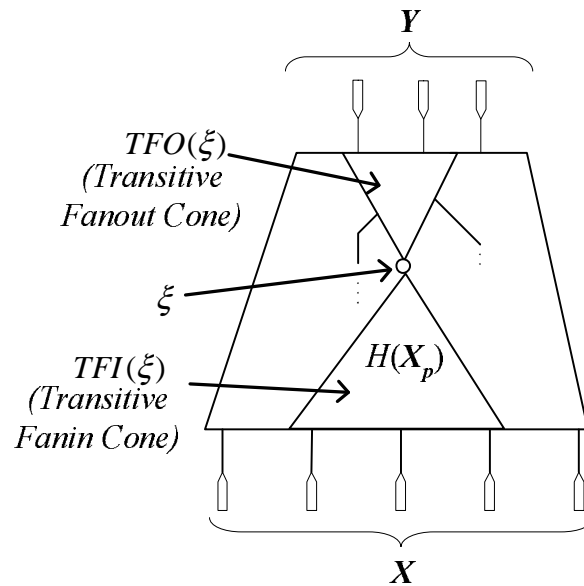


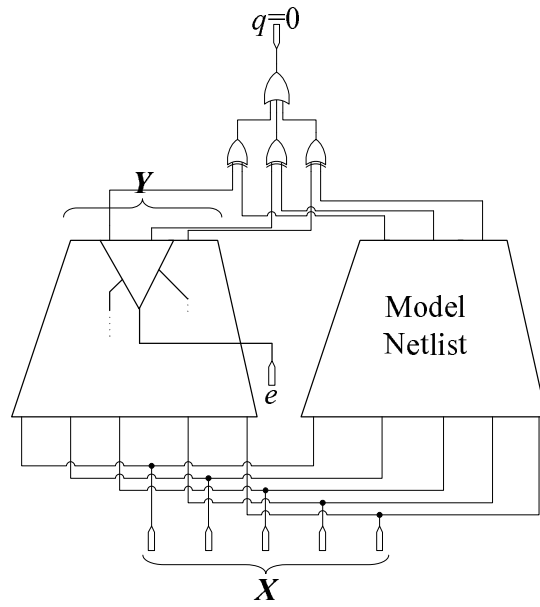
Figure 5.6: Illustration of localized node ξ . Modification can be applied by replacing $H(\mathbf{X}_p)$ with some arbitrary function $F_c(\mathbf{X}_s)$.

Our work follows Lemma 5.2.2 where given a set of focal points returned by the localization step, we attempt to replace the global function at each location with another global function. This requires us to first find a suitable global function, which we label as F_c . We do so by deriving an implicit representation of F_c using a specialized circuit construct.

Following this, we search for a set of basis functions $\{g_1, g_2, \dots, g_k\}$ from the existing netlist and see if they can be used to implement the new function F_c . This requires a SAT-based functional dependency check described later on in this section. If the basis functions pass the SAT test, we can generate the decomposition $F_c = h(g_1, g_2, \dots, g_k)$ where h is known as the dependency function between F_c and the basis functions $\{g_1, g_2, \dots, g_k\}$. To understand this in detail, we will first consider the simple case of a combinational circuit containing a single focal point returned from the localization step.

Deriving a suitable global function, F_c , can be done through brute force means where we explicitly create F_c . However, an explicit representation of F_c , such as a truth table or BDD, will not scale since F_c may depend on several variables. To avoid this problem we instead choose to construct an implicit representation of F_c . We start off with a construction of the circuit shown in Figure 5.7. On the left side of Figure 5.7(a) we show a circuit we intend to modify, which looks similar to Figure 5.6. However, in Figure 5.7(a), we replace the focal point ξ with an input pin e . Doing so gives us the freedom to search for a suitable F_c to replace pin e . Next, we have to constrain the input-output behaviour of the circuit to match the model netlist. We do this by attaching the model netlist to our circuit as shown on the left side of Figure 5.7(a). Here, we tie the circuit inputs together and connect the outputs with an XOR gate along with an OR gate at the top. Following this, we force the OR gate output to 0 ($q = 0$). This equivalence construction is also known as a Miter circuit [Bry86] and as a result of these constraints, any satisfying assignment to the circuit in Figure 5.7(a) will be consistent with the input-output behaviour of the model netlist.

After constraining the circuit to the model netlist behaviour, we seek to remove the “don’t care” space of the suitable function F_c . This gives us greater flexibility in choosing a proper F_c . We do this by restricting the input space, \mathbf{X} , to values where pin e is observable at the outputs. Informally, e , is observable at a PO, y , if for a given assignment, the value of e impacts the value of y . Otherwise, e is not observable at node y . This is appropriate since



(a) Attaching circuit to model netlist

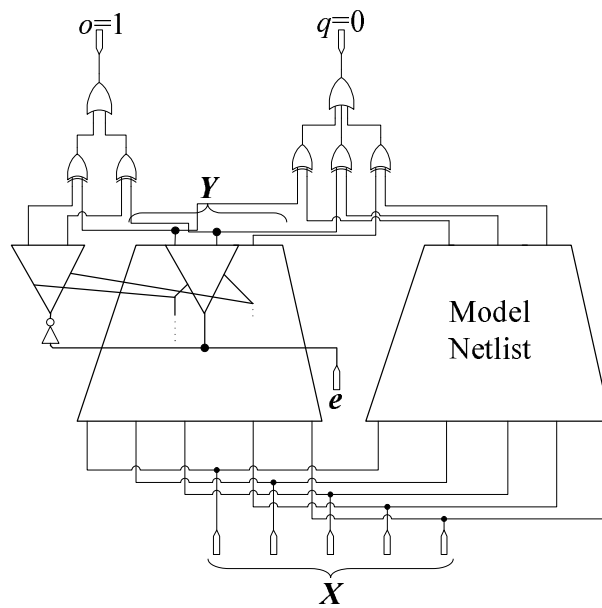
(b) Attaching observability constraints for *CircuitConstrain*

Figure 5.7: Circuit to be modified

pin e will ultimately be replaced with F_c . This is shown in Figure 5.7(b) which we will refer to as *CircuitConstrain*. In *CircuitConstrain*, we duplicate the transitive fanout of pin e . When connecting the duplicated transitive fanout to the original circuit, we complement input pin e , while all other input branches to the transitive fanout are kept the same. Also, for each output pair of the duplicated transitive fanout and original circuit, we add an XOR gate followed by an OR gate and set the output of the OR gate to 1 ($o = 1$). The circuitry added at the outputs along with the duplicated transitive fanout checks to see if the input pin e is observable at the outputs. For example, if the output of the OR gate is 0 ($o=0$), this implies that the value on pin e does not matter and will not affect the outputs. Based on the previous description of *CircuitConstrain*, we find the following lemma and proposition to be true.

Lemma 5.2.3 *CircuitConstrain is satisfiable if and only if the output of the current circuit matches the output of the model netlist and if pin e is observable at the outputs.*

Proposition 5.2.4 *CircuitConstrain is satisfiable if and only if the assignment to e and the assignment to the input vector \mathbf{X} is consistent with a suitable F_c . In other words, for all satisfying assignments to *CircuitConstrain*, $F_c(\mathbf{X}) = e$.*

Proof: Assume that if we have a satisfying assignment to *CircuitConstrain* then the values on e and \mathbf{X} do not match any suitable F_c . In other words, assume that for a given satisfiable assignment to e and \mathbf{X} then $e \neq F_c(\mathbf{X})$. Thus, this implies that the output of the current circuit does not match the model netlist by Lemma 5.2.2. However, this is not possible since any satisfying assignment to *CircuitConstrain* must match the input-output behaviour of the model netlist by Lemma 5.2.3. Thus, by contradiction any satisfying assignment to e and \mathbf{X} implies a match to a suitable F_c . Now let us assume that if the values on e and \mathbf{X} match a suitable F_c then we do not have a satisfying assignment to *CircuitConstrain*. In this case, if $e = F_c(\mathbf{X})$, then the output response of

our circuit will match the model netlist. Also, e must be observable since we can restrict the inputs space of X to the observable space of F_c . However, these two conditions must imply that we must have a satisfying assignment to *CircuitConstrain* by Lemma 5.2.3. Thus, by contradiction, if the values on e and X match a suitable F_c then we have a satisfying assignment to *Circuit-Constrain*. Hence we have proved that *CircuitConstrain* is satisfiable if and only if the assignment to e and the assignment to the input vector X is consistent with a suitable F_c . ■

By proposition 5.2.4, we can use the circuit in Figure 5.7(b) to find a suitable F_c by exploring all satisfying assignments to e and X . Doing so will effectively create the entire truth table for function F_c . For example, we would start by setting X to 000...00. We would then find a satisfying value for e . This value found for e would represent $F_c(000...00)$. Next, we would continue for $X = 000...01$ and so on.

The problem with the previously described approach is that this requires an exhaustive traversal of the input space X , which does not scale. Furthermore, implementing F_c directly is not desirable since this may require significant changes to the existing place-and-routed netlist. A more practical approach for ECOs would be to search for a set of basis functions, $\{g_1, g_2, \dots, g_k\}$, found from the existing netlist and check if this set of basis functions can be used to implement F_c such that $F_c = h(g_1, g_2, \dots, g_k)$. Informally, this will be possible if and only if there does not exist a situation where the output values of the basis functions $\{g_1, g_2, \dots, g_k\}$ are equivalent for a given pair of input vectors and F_c is not equivalent for the same pair of input vectors. For example, consider Figure 5.8. Here we show four functions, g_1, g_2, g_3 , and F_c . In this example, all functions are dependent on three variables x_1, x_2 , and x_3 . Looking at function g_1 and g_2 , we see that there exists a given input pair $\{000, 001\}$ where g_1 and g_2 are equivalent, but F_c is not equivalent. Thus, it is impossible to find a function h such that $F_c = h(g_1, g_2)$. In contrast, if we select functions g_1 and g_3 , it is impossible to find an input pair where g_1 and g_3 are equivalent and F_c is not equivalent.

Thus, in this case, there exists a dependency function h such that $F_c = h(g_1, g_3)$ (in this case $h = g_1 \cdot g_3 + \overline{g_1} \cdot \overline{g_3}$). This process is beneficial since only the dependency function h needs to be derived, which in general should have a smaller support set than the global function F_c . This is ideal for ECOs since implementing the dependency function h within the existing circuit has a much smaller and predictable impact than implementing the entire global function F_c from scratch.

| $x_1 x_2 x_3$ | $g_1 g_2 g_3$ | F_c |
|---------------|---------------|-------|
| 000 | 0 0 0 | 1 |
| 001 | 0 0 1 | 0 |
| 010 | 0 0 0 | 1 |
| 011 | 0 1 1 | 0 |
| 100 | 0 0 1 | 0 |
| 101 | 0 0 1 | 0 |
| 110 | 1 0 1 | 1 |
| 111 | 1 1 1 | 1 |

Figure 5.8: Functional dependency example

To formally check the condition described previously we will use a SAT-based approach [LJHM07]. This requires constructing a characteristic function that checks for the following condition: for a given set of basis functions $\{g_1, g_2, \dots, g_k\}$, does there exist a minterm pair where the basis functions are equivalent and F_c is not equivalent. To create this characteristic function, we construct the circuit shown in Figure 5.9. In Figure 5.9, we have duplicated *Circuit-Constrain* and labeled them as *CircuitConstrain₀* and *CircuitConstrain₁*. In each duplicated copy, we have simplified *CircuitConstrain* and have only shown their duplicated input and output pins.

To check if there exists a minterm pair where a set of basis functions output values are equivalent, we can extract a basis function set from each duplicated circuit and connect them together with an XOR-OR gate network and set the OR gate output to 0 ($d = 0$) as shown in Figure 5.9. Here, we have selected two wires g_1 and g_2 as the basis functions. The support set of the basis functions g_1 and g_2 in *CircuitConstrain₀* (*CircuitConstrain₁*)

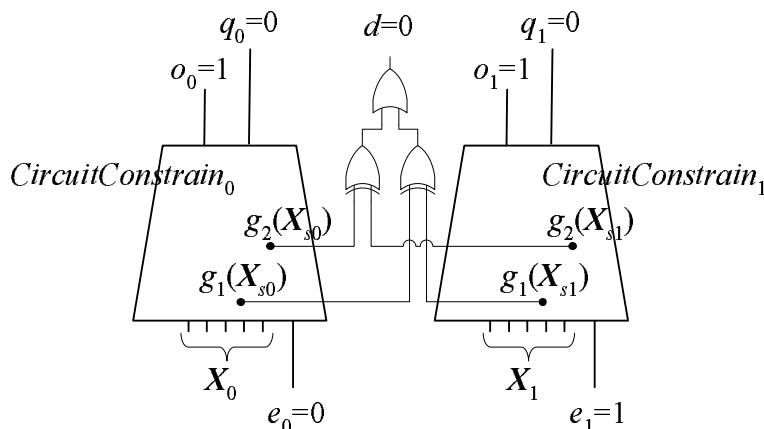


Figure 5.9: Deriving global function $F = h(g_1, g_2)$ using dependency check construct.

are labeled as X_{s0} (X_{s1}) which is a subset of the variable set X_0 (X_1). Due to these constraints, a satisfying assignment to Figure 5.9 will find an minterm pair assignment to variables X_0 and X_1 where the basis functions are equivalent.

To check if there exists a minterm pair where F_c is not equivalent, we can also use the circuit construct in Figure 5.9. Here, we will constrain each duplicated pin e . Since in *CircuitConstrain*, only satisfiable assignments to e and X are consistent with a suitable global function $F_c(X)$ by proposition 5.2.4, constraining e to 0 or 1 provides a means to constrain the input X to the offset and onset for $F_c(X)$. This is stated more formally in the following lemma.

Lemma 5.2.5 *In *CircuitConstrain* of Figure 5.7(b), if pin e is assigned a value 0, *CircuitConstrain* is satisfiable if and only if the input vector X is assigned a value found in the offset of the function $F_c(X)$. Conversely, if pin e is assigned a value 1, the circuit is satisfiable if and only if the input vector X is assigned a value found in the onset of the function $F_c(X)$.*

Following from Lemma 5.2.5, the circuit in Figure 5.9 constrains e_0 to 0 and e_1 to 1 to restrict the possible assignments to X_0 and X_1 . Specifically, setting e_0 to 0 constrains all possible assignments of X_0 to the offset of F_c , while setting e_1 to 1 constrains all possible

assignments of \mathbf{X}_1 to the onset of F_c . Thus, due to these constraints, a satisfying assignment to Figure 5.9 will find a minterm pair assignment to variables \mathbf{X}_0 and \mathbf{X}_1 that respectively map to the offset and onset of function F_c . Thus, for this minterm pair, F_c is not equivalent.

Using the previous circuit constraints in Figure 5.9, we can create the characteristic function shown in Equation 5.2.

$$\begin{aligned} \exists \mathbf{X}_0 \exists \mathbf{X}_1 & (CircuitConstrain_0_{CNF}) \cdot \\ & (CircuitConstrain_1_{CNF}) \cdot \\ & (e_0 \equiv 0) \cdot (e_1 \equiv 1) \cdot (g_1(\mathbf{X}_{s0}) \equiv g_1(\mathbf{X}_{s1})) \cdot \\ & (g_2(\mathbf{X}_{s0}) \equiv g_2(\mathbf{X}_{s1})) \cdot (\mathbf{X}_0 \neq \mathbf{X}_1) \end{aligned} \quad (5.2)$$

In Equation 5.2, assume that $CircuitConstrain_0_{CNF}$ ($CircuitConstrain_1_{CNF}$) represents the clauses derived from the characteristic function of $CircuitConstrain_0$ ($CircuitConstrain_1$). Equation 5.2 asks if there exists a minterm assignment pair to \mathbf{X}_0 and \mathbf{X}_1 which map to the off and onset of F_c respectively (i.e. $e_0 \equiv 0$ and $e_1 \equiv 1$), while the basis functions are equivalent. This leads to the following proposition.

Proposition 5.2.6 *If Equation 5.2 is satisfiable the basis functions, g_i , cannot be used to implement a suitable global function F_c . If Equation 5.2 is unsatisfiable, the basis functions, g_i , can implement a suitable global function F_c .*

Proof: Assume if Equation 5.2 is satisfiable then F_c has a decomposition $h(g_1, g_2, \dots, g_k)$. If Equation 5.2 is satisfiable, this implies that there is a vector assignment C_0 to \mathbf{X}_0 which maps to the offset of F_c and there exists a vector assignment C_1 to \mathbf{X}_1 which maps to the onset of F_c , which follows from Lemma 5.2.5. This also implies that vector C_0 and vector C_1 map to the same output vector, $K = k_1 k_2 \dots k_k$ in the output space of the basis set $\{g_1, g_2, \dots, g_k\}$ due to the g_i equality constraints in Equation 5.2. However, since we have assumed $F_c = h(g_1, g_2, \dots, g_k)$ in the first statement, this implies $F_c(C_0) = h(K) = 0$

and $F_c(C_1) = h(K) = 1$, thus we have a contradiction. Thus if Equation 5.2 is satisfiable then F_c cannot be decomposed into $h(g_1, g_2, \dots, g_k)$. To prove the second statement in proposition 5.2.6, we can use a similar argument and will not show it in detail here. ■

If Equation 5.2 is unsatisfiable, we can derive the correct dependency function h from the proof of unsatisfiability. This requires backtracing through the set of clauses that are the source of the unsatisfiable condition and creating a function based on these clauses. In our case, if the function h is small enough (its support set size is less than 20), we create a BDD representation of h . We limit the variable size to 20 since this minimizes the risk of BDD explosion [BS02]. In the cases where the BDD does get very large, we fail to create a proper implementation of h . Deriving h based on the proof of unsatisfiability is based on the theorem of Craig interpolation and will not be discussed here. For a detailed description on deriving h please refer to [McM03] and [LJHM07].

If the support size of h is less than the number of inputs to a LUT, we can implement h in a single LUT and the modification process is complete. However, if the support size of h is larger than the number of inputs to a LUT, we must decompose and technology map h to the circuit architecture. In our case, we decompose h into a network of LUTs for the Stratix architecture. However, one can modify their decomposition algorithm to map h into any technology. During the decomposition, since timing information is available, we skew the decomposition such that critical wires are placed near the top of the decomposition tree. This is similar to the work presented in [MSB05] and reduces the number of logic levels the critical wires have to go through in order to preserve timing.

Searching for Basis Functions

In the previous description, we assumed that the set of basis functions necessary in our functional dependency check were available. In practice, we must search the circuit netlist for an appropriate set of basis functions. This flow is illustrated in Figure 5.10. First the

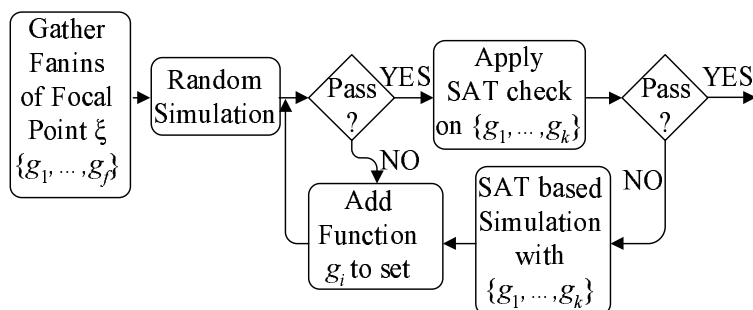


Figure 5.10: Searching for basis functions flow.

fanins to the focal point, ξ , are chosen as the basis functions. Next, we use a random simulation based check to see if this set can potentially be used as a dependency set to implement the corrective function F_c . If the check fails, we search for additional functions which exist within the circuit to add to the set of basis functions. Additional functions are added until our random simulation based check passes. When searching for additional basis functions, we first select functions within the transitive fanin of the focal point ξ . If this fails, we collect the primary outputs found within the transitive fanout of the focal point ξ and select functions found within the transitive fanins of those primary outputs. Once a set of basis functions pass the random simulation check, we apply our SAT-based decomposition technique. This process continues until we find a suitable set of basis functions to implement F_c such that $F_c = h(g_1, g_2, \dots, g_k)$. Although not shown in Figure 5.10, if the number of basis functions required grows to more than 20 functions or if the number random simulation calls is more than 10, the search will terminate.

| $x_1 x_2 x_3$ | e | $g_1 g_2 g_3$ |
|---------------|-----|---------------|
| 100 | 0 | 0 0 1 |
| 001 | 0 | 0 0 1 |
| 010 | 1 | 0 0 0 |
| 011 | 0 | 0 1 1 |
| 110 | 1 | 1 0 1 |
| 101 | 0 | 0 0 1 |

Figure 5.11: Random simulation example.

The random simulation check shown in Figure 5.10 is used as a quick check to see if the current basis function set can be used to implement F_c without going through the more expensive SAT-based check described in the previous section. Here, using *CircuitConstrain* in Figure 5.7(b), we randomly simulate the PIs X and pin e . Vectors which violate the observability ($o = 1$) and equality ($q = 0$) constraint are discarded. Following this, we search through the circuit netlist and find a set of nodes whose vectors do not satisfy Equation 5.2. For example, consider Figure 5.11. Assume that the circuit PIs are variables x_1 , x_2 , and x_3 and functions g_i are chosen from the existing circuit. Here we show the value of the functions, g_i , with respect to a random input vector assignment to the PIs and pin e . Using this sample set, we can see that the set $\{g_1, g_2\}$ cannot be used as our basis set since there exists a primary input pair, $\{001, 010\}$, which satisfies Equation 5.2. In contrast, basis set $\{g_1, g_3\}$ is a possible set since there does not exist any input pair which satisfies Equation 5.2. Since our random simulation set is not exhaustive, we must apply the SAT-based check described in the previous section to cover any input vectors missed during the random simulation.

If the current basis set fails the random simulation test, we must add additional functions to build a proper set of basis functions. To help guide the search, we use a SAT based vector generation to create input patterns that can find additional vector pairs that need to be *differentiated*. The SAT based vector generation works by duplicating the circuit and “locking” each pair of duplicated basis functions to equivalent values and setting the pair of duplicated input pins e to complemented values. Following this, a SAT solver is run 32 times to find a set of 32 different satisfying values on the other wire which generates a new vector set. Using these new vector sets, we search for functions that can *differentiate* minterms currently differentiated by pin e , but not differentiated by the current basis function set. Informally, a function set can *differentiate* a pair of minterms if their outputs are different for that minterm pair. For example, in Figure 5.11, $\{g_1, g_2\}$ cannot differentiate

minterms 001 and 010 since $g_1(001) \equiv g_1(010)$ and $g_2(001) \equiv g_2(010)$. Since pin e differentiates this pair, to create a valid basis set from $\{g_1, g_2\}$ for pin e , we would need to pick an additional function g_j that can differentiate vectors 001 and 010 (differentiating pairs $\{100, 010\}$ and/or $\{101, 010\}$ are also possible). Wires whose functions cover the most vectors not differentiated by the current basis set are chosen first. This greedy algorithm helps to minimize the size of the basis function set. The second criteria when choosing basis functions is their timing criticality: basis functions with non-critical wires are chosen over basis functions with critical wires.

Partitioning of Circuit and Treatment of Registers

In order to help reduce the size of Equation 5.2, we can remove portions of the circuit which do not impact the behaviour of the current node being modified. This is illustrated in Figure 5.12. Here, we are assuming that node ξ was localized in our netlist localiza-

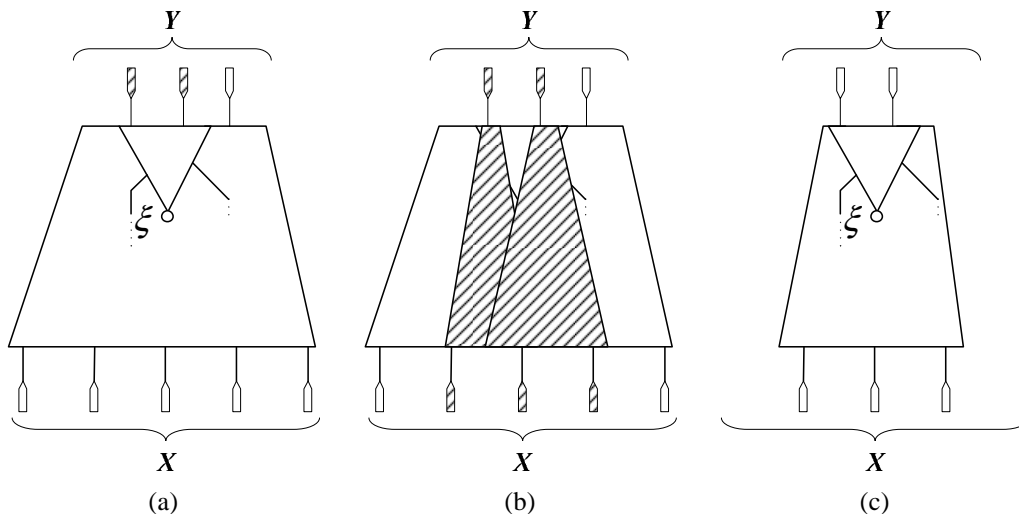


Figure 5.12: Circuit reduction.

tion step. Using node ξ , we collect the set of outputs within the transitive fanout of ξ , as shown in Figure 5.12(a). Following this, we collect the transitive fanin of each output, as shown in Figure 5.12(b). Since the POs of the implemented netlist may require that their

transitive fanin PIs be changed during the modification, the transitive fanin collected in Figure 5.12(b) is defined by the model netlist. Thus, this requires that our model netlist be a white-box implementation, such as an gate-level netlist derived from an RTL description. Any node which is not part of a transitive fanin of a given output can be removed, as shown in Figure 5.12(c). In conditions that the transitive fanout is excessively large and contains several hundred POs, we limit the number of POs kept to under 50. Thus, we are able to remove a significant portion of the circuit to speed up the SAT-based functional dependency check. This can potentially create an incomplete solution since this heuristics removes some information that may be needed to derive a corrective function F_c . However, empirically we rarely found this to be a problem where for our circuit set, 50 outputs were sufficient to find a valid correction.

In the case of sequential circuits, we treat registers as primary inputs and outputs where the input of a register becomes a primary output and the output of a register becomes a primary input. This reduces the complexity of our problem; however, limits the application of our algorithm to circuits which have not been retimed.

Multiple Focal Points

In the previous sections, we described the situation when only a single focal point is returned by the localization step. In the case of multiple focal points, where we need to find multiple suitable global functions, we apply a step-by-step approach where we modify each focal point one at a time. While modifying a given focal point, ξ , additional focal points are “locked down” such that they are set to fixed values. For example, one overly simplistic heuristic would be to set all focal points other than ξ to value 0 (in general, this would not work since this could make the focal point ξ unobservable at the outputs, such as in a case where ξ and another focal point enter into a single AND gate). This ensures that changes seen at the output are due to the focal point ξ , and not due to the other focal points found

within the netlist. The locking down of focal points has the possibility of not converging, particularly for complex changes requiring several modifications. However, in practice the step-by-step heuristic works in the majority of cases as shown empirically in Section 5.3 where we have at most four focal points into the design.

5.3 Results

To ensure that the resynthesis technique described in the previous sections can be applied to late-stage ECOs, we must ensure that our technique can automatically update the behaviour of a circuit while minimally disrupting the existing place-and-routed netlist and preserve timing. Thus, we evaluate our work based on three criteria: how disruptive our technique is on a place-and-routed netlist, how much impact our technique has on performance, and how robust our technique is in handling changes. Two experiments will be run to quantify these three criteria. The first experiment models a late-stage specification change at the RTL, while the second experiment models a late-stage bug fix. We run our algorithm on circuits from Altera's QUIP [Alt08] and the ITC benchmark suite.

All experiments are run on a PentiumD 3.2 GHz processor with 2MB of RAM. Circuits are synthesized using Quartus II 7.1 and for Altera's Stratix architecture. To communicate with Quartus II, we use Altera's QUIP TCL interface [Alt08].

5.3.1 Specification Changes

In our first set of experiments, we seek to see how disruptive our approach is on an existing place-and-routed circuit and its impact on performance. This is important for late-stage ECOs since we want to preserve as much as possible the engineering effort invested previously when applying our changes. During this experiment, we apply a late-stage specifica-

tion change both using a full recompile of the design and using the proposed ECO approach. During the experiment, we first implement a circuit described in Verilog or VHDL using Altera's Quartus II CAD flow. In the initial compile, we use Altera's design space explorer to achieve the best possible timing performance. Next, we modify the original HDL code and use this modified code as our new specification (this would be the *model netlist* as described in Section 5.2). Modifications are primarily constrained to control-path code or modifications to `case` and `if-else` statements where we change less than three lines of code. An example of such changes is shown in Figure 5.13. After our code is modified, we perform a full recompile of the modified HDL code and record the performance impact. For comparison, we also apply the required HDL changes directly to the optimized circuit using our ECO approach and record its performance impact and measure the disruption to the place-and-routed netlist.

```

//Original                                //Modification
...
if(busreg[j-1] == 1)                    if(busreg[j-1] == 1)
  begin                                  grant = j-1;
    grant = j-1;                          else
    lock = hlock[j-1];                    grant = inter1;
  end                                     if(busreg[j-1] == 0)
else                                     lock = hlock[j-1];
  begin                                  else
    grant = inter1;                        lock = inter2;
    lock = inter2;                          ...
  end                                     ...
...

```

Figure 5.13: Example changes in HDL code.

To quantify the disruption to the existing netlist due to a full recompile and from the ECO compile, we measure the percent change in timing and the percentage of LUTs and wires disrupted in the original place-and-routed circuit. These results are shown in Table 5.1. The first column shows the circuit name, *Circuit*, followed by three major column groups. The first group, *Before*, reports the circuit size in terms of LUTs and timing numbers during the

| Circuit | Before | | After _{FULL} | | After _{ECO} | | | |
|----------------|--------|---------------------|-----------------------|-------------------|----------------------|-------------------|----------------|----------------|
| | LUTs | CLK _(ns) | LUTs | CLK _{%Δ} | LUTs | CLK _{%Δ} | Δ _P | Δ _R |
| aes_core | 5359 | 7.96 | 5359 | -0.25% | 5359 | 2.26% | 0.08% | 0.16% |
| huffman_enc | 633 | 5.12 | 634 | -0.39% | 640 | 4.10% | 7.36% | 9.32% |
| usb_phy | 177 | 3.2 | 180 | -0.62% | 179 | 0.31% | 9.25% | 10.35% |
| fip_cordic_rca | 467 | 21.6 | 467 | 6.39% | 470 | 4.86% | 3.54% | 5.32% |
| mux64_16bit | 1277 | 5.86 | 1277 | 4.10% | 1277 | 1.37% | 2.23% | 3.52% |
| ahb_arbiter | 862 | 9.45 | 862 | 9.95% | 864 | 1.16% | 2.34% | 3.54% |
| barrel64 | 1043 | 8.59 | 1042 | -0.23% | 1043 | 0.81% | 1.30% | 2.50% |
| Average | | | | 2.70% | | 2.13% | 3.73% | 4.95% |

Table 5.1: Impact of modifications on circuit performance on a Stratix chip.

initial compile of the design (i.e. before applying any changes). Timing numbers report the minimum clock period in terms of nanoseconds. The next group of columns headed by *After_{FULL}* shows the number of LUTs and percent change in timing with respect to *Before* as a result of a full compile of the modified HDL code. The final group headed by *After_{ECO}* shows the impact of the proposed approach on circuit performance. Again, we show the number of LUTs and percent change in timing with respect to *Before* after modifying the design using the proposed ECO approach. Two additional columns, Δ_P and Δ_R , are shown which respectively show the percent disruption to the placement and routing after our changes are applied. This is not shown in the group *After_{FULL}* since in the full recompile of the circuit, all LUTs and wires will be modified. The last row shows the average ratio of the percentages shown.

As our results show, the results of a “from scratch” recompile shown in columns *After_{FULL}*, are fairly acceptable on average where performance decreases by 2.70%. However, when examining the individual performance impact of each circuit, we see that the results varies by a large amount from -0.62% to 9.95%. This illustrates the unpredictable nature of a full recompile on circuit performance. Furthermore, during a full recompile, the entire netlist is disrupted, thus all of the placement and routing effort invested previously is lost. In contrast, the proposed ECO approach has a much more predictable impact to perfor-

mance where the performance change varies between 0.31% to 4.86%. Also, the average disruption of our approach is relatively low where we keep over 95% of the placement and routing unchanged. Based on what is reported in industry [Mor06a], we feel that this number is acceptable to most designers, which is a necessary condition for the practical use of our approach. In terms of runtime, the proposed ECO approach is approximately 2x faster than performing a full compile. However, this discrepancy in runtime has more to do with the internal algorithms of Quartus II than our work and thus, detailed numbers are not shown. Furthermore, runtime comparisons would not reflect the reduction in time from going from a manual process, which often takes several days, to the proposed automatic ECO approach, which takes a few hours to complete.

During our experiments, we found that the class of changes we applied did not create drastic changes to the netlist. Specifically, during the localization process, all of our HDL changes would result in less than five focal points being returned. In cases where five or more focal points were returned, we found that our approach had difficulty finding a non-disruptive modification to the place-and-routed netlist.

We should note that we used Quartus II's TCL ECO interface to change and legalize the circuit. Unfortunately, the TCL ECO interface does not provide an incremental placement flow. Instead, Quartus II will apply a limited placement algorithm to new or modified nodes. Here, the ECO placer will only move nodes to existing free locations within the FPGA and will not displace pre-existing nodes within the netlist. As a result, this may hurt timing significantly since circuitous routes can be created by new nodes. To overcome this limitation, we manually unplace a small region surrounding each new node. This gives the placer much more flexibility in where to place nodes and reduces the impact of newly introduced nodes. However, if given access to a full incremental placement algorithm, we feel that our timing and disruptive impact would be less than what is shown in Table 5.1.

5.3.2 Error Correction

In our second set of experiments, we apply the proposed ECO technique to fix errors inserted into a design. Errors include rewiring of LUTs or changing the LUT functions of various nodes. To stress the approach, multiple errors are inserted into the design such that the errors interact with each other. Errors interact when they are inserted at nodes whose transitive fanouts overlap with each other. This is necessary to ensure that the complexity of the correction increases as more errors are added to the design. After errors are inserted into the design, Quartus II's place-and-route is rerun on the circuit to achieve the best performance possible.

Table 5.2 shows our results for various circuits when five errors are inserted into the design. The first column *Circuit* lists the circuit name. This is followed by two column groups: *Before* and *After_{ECO}*. *Before* lists the number of LUTs and timing numbers before the application of the ECO fix. Timing numbers report the minimum clock period in terms of nanoseconds. *After_{ECO}* lists the total number of LUTs after the ECO fix and the relative change in timing with respect to the the timing reported in the *Before* column. Also, additional columns Δ_P and Δ_R are shown which respectively show the percent disruption of the placement and wires after the ECO fix is applied. The final row shows the average of the percentages. Unlike the previous results shown in Table 5.1, comparisons with a full recompile is not done in this case since a designer would be required to fix the error manually by hand, which in many of the cases could not be practically solved by hand in an efficient manner. The proposed technique, however, can find a fix automatically.

Table 5.2 shows that the proposed ECO technique has a marginal impact to timing where we reduce timing on average by 0.18%. This is potentially misleading since for some circuits, such as b04 and b10, we improve timing by a large amount. This can occur in some rare cases since the decomposition of the dependency function h is timing driven and has the potential to shorten the critical path. After removing these outliers, the average

| Circuit | Before | | After _{ECO} | | | |
|----------------|--------|---------------------|----------------------|-------------------|----------------|----------------|
| | LUTs | CLK _(ns) | LUTs | CLK _{%Δ} | Δ _P | Δ _R |
| aes_core | 5359 | 7.95 | 5359 | 0.00% | 1.12% | 1.28% |
| huffman_enc | 633 | 5.08 | 633 | 0.20% | 3.16% | 3.61% |
| usb_phy | 177 | 3.13 | 177 | 0.00% | 13.56% | 10.33% |
| fip_cordic_rca | 467 | 21.6 | 471 | 2.31% | 12.85% | 14.68% |
| mux64_16bit | 1277 | 5.86 | 1277 | 0.51% | 4.70% | 5.37% |
| ahb_arbiter | 865 | 8.97 | 972 | 16.83% | 6.94% | 7.93% |
| barrel64 | 1041 | 8.58 | 1041 | -4.66% | 5.76% | 6.59% |
| b03 | 59 | 3.37 | 59 | 0.89% | 16.95% | 19.37% |
| b04 | 184 | 6.56 | 184 | -9.60% | 32.61% | 37.27% |
| b08 | 58 | 3.68 | 58 | 5.71% | 8.62% | 9.85% |
| b09 | 50 | 3.97 | 50 | -3.02% | 16.00% | 11.43% |
| b10 | 78 | 3.79 | 79 | -17.41% | 10.26% | 11.72% |
| b11 | 167 | 6.68 | 187 | 4.64% | 9.58% | 10.95% |
| b12 | 378 | 6.17 | 381 | 3.40% | 4.23% | 4.84% |
| b14 | 3014 | 8.17 | 3015 | 2.57% | 1.99% | 2.28% |
| b15 | 6012 | 9.17 | 6023 | 6.98% | 1.00% | 1.14% |
| Average | | | | 0.18% | 9.33% | 9.91% |

Table 5.2: Automated correction results on a Stratix chip.

impact to timing is 1.72%. Another outlier, `ahb_arbiter`, proved difficult to correct where we found that in this circuit, the proper alterations required more than 15 basis functions to correct it, which ultimately disrupted its timing significantly. Considering that the proposed technique applies ECO changes in an automated fashion within minutes, we feel that 1.72% is an acceptable tradeoff of timing for improved designer productivity where the designer would manually fix these problems in a span of several days. Also, we found that many of the fixes occurred on paths that were critical or near critical. Thus, without a timing driven decomposition of h and a timing driven search for basis functions, circuit performance would not be maintained.

In terms of average disruption to the existing netlist, we disturb approximately 10% of the place-and-routed netlist during our correction. We should note that this disruption is skewed by the fact that for smaller circuits, a large relative disruption cannot be avoided. For example, in the extreme case where a circuit contains only five LUTs, fixing five errors

in the circuit would disrupt 100% of the netlist. If circuits with less than 200 LUTs are removed from Table 5.2, the average disruption to the placement and routing drops to approximately 5%.

When inserting less than five errors into the design, we found we were able to correct over 80% of the circuits. However, as the number of errors increased, the circuit correction problem became significantly difficult. This was primarily due to the localization step. For complex fixes, the localization step often returns several focal points, which can be much more numerous than the number of errors injected into the design. This significantly complicates the correction process both in terms of deriving a suitable global function F_c and minimizing the disruption to the existing netlist. Fortunately, the likelihood that large errors are not detected early in the design flow is low and, as a result, applying large changes with late-stage ECOs is not a likely occurrence.

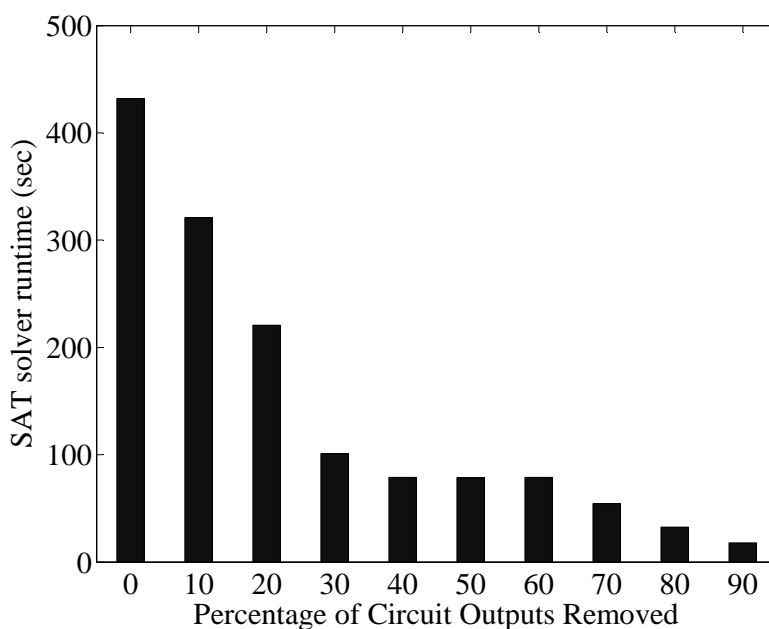


Figure 5.14: Impact of output removal on SAT solver runtime for circuit b15.

As a final experiment, we examined the impact of the the primary output removal heuristic described in Section 5.2.2. This is shown in Figure 5.14 where we show the percentage

of outputs removed versus the runtime of the SAT solver when correcting circuit b15. During this experiment, we inserted bugs into nodes whose transitive fanouts contained more than 300 primary outputs and registers. Following this, we removed a percentage of outputs or registers and attempted to rectify the circuit. After rectification, we recorded the runtime of the SAT solver. As Figure 5.14 shows, we are able to reduce the runtime of the SAT solver by more than 10x in some cases. Also, removing outputs usually did not prevent a valid correction from being found. Only until more than 95% of the outputs were removed from the circuit did the correction fail our verification check. This was due to a lack of information provided by the model netlist after pruning most of the outputs.

5.3.3 Generation of Model Netlist

In both experiments shown previously, we need to generate a model netlist to tie our input-output pins to the existing circuit. As stated at the beginning of Section 5.2, this model netlist is a logic netlist which defines the desired behaviour of the circuit. In the context of this work, this netlist is generated by using the Quartus II logic synthesis and technology mapping flow. Here, the model netlist is described using HDL code and compiled into a LUT netlist using Quartus II. Using this resulting LUT netlist, we tied the input and outputs into the equivalence relationship shown previously in Figure 5.7.

5.4 Summary

Improving the ECO experience through automation is necessary to reduce the complexity of using FPGAs. In this work, we have shown an approach to automate the ECO experience. We have shown that our technique is robust and applicable to commercial FPGAs where we apply our work to Altera's Stratix architecture. We have also shown how we can integrate our flow with Altera's commercial CAD suite Quartus II. All of these factors

provide a promising foundation toward the full automation of ECOs.

6 Conclusion and Future Work

6.1 Summary of Contributions

In this dissertation we improved the overall user experience of FPGA design tools through two means. First, we enhanced the scalability and runtime of existing logic synthesis flows. Secondly, we provided a solid foundation to automate the ECO process.

In Chapter 3, we focus on improvements to BDD-based logic synthesis. Since BDDs are a powerful means to manipulate and optimize circuits they are widely used in logic synthesis flows. Unfortunately, current BDD-based logic synthesis face scalability issues, particularly during its elimination step. As a solution, we demonstrated how we can solve elimination as a covering problem. In doing so, we applied two key innovations to the covering problem. First, we developed a scalable cut generation step based on BDD compression. Second, we developed a cost function termed edge flow to help isolate redundancies in the circuit. Our BDD-based compression technique for cut generation was able to reduce its memory and computation requirements by an order of magnitude. The edge flow heuristic leads to a 7% decrease in routing wires or edges in the final covered netlist, with no impact to area or delay of the circuit. Combining both these improvements to elimination yielded a 6x speedup in our BDD-based synthesis flow with a negligible impact to area.

We continue to improve the logic synthesis flow in Chapter 4 where we looked at improving partitioning based logic synthesis. Partitioning is a necessary step in many CAD

flows to improve scalability, however, the problem with partitioning is that optimizing each partition in isolation leads to suboptimal results. As a solution to this, we derived partition constraints to help during resynthesis through an Integer-Linear Program (ILP) formulation. By following these constraints, the localized optimizer will produce a solution with superior quality to that of one with only a localized view of the partition. Unfortunately, our ILP formulation is NP-complete and does not scale to large circuits. To avoid this issue, we showed how we can reduce our ILP to polynomial complexity by leveraging the concept of duality. Doing so improved the problem runtime by over 100x. When run on the IWLS benchmark set, we showed that our ILP formulation can improve circuit depth by 12% on average when compared against partitioning-based flows that do not use our technique. Furthermore, our reduction in depth came with a less than 1% penalty to circuit area.

Finally, in Chapter 5, we improved the FPGA ECO flow. In a design flow where almost all tasks are automated, ECOs remain a primarily manual and expensive process. As a solution, we introduced an automated method to tackle the ECO problem. Specifically, we introduced a resynthesis technique using Boolean Satisfiability (SAT) which can automatically update the functionality of a circuit by leveraging the existing logic within the design; thereby removing the inefficient manual effort required by a designer. By using this technique, we showed how we can automatically update a circuit implemented on an FPGA while keeping over 90% of the placement unchanged.

6.2 Future Work

Although we have significantly improved the scalability of existing BDD-based synthesis flows in Chapter 3, there still remains possibilities to improve the QoR. In particular, SAT based logic synthesis [LSB05b, MBJJ09] has been proven as a scalable and effective

means to significantly reduce circuit area. Since BDDs have a dual relationship with SAT, where BDDs solve problems in space, whereas SAT solve problems in time (Chapter 2, Section 2.7), there are further possibilities to map the extraction and decomposition algorithms in our BDD-based synthesis flows to SAT. Doing so may yield better results since using SAT could potentially expand the search window to resynthesize logic functions.

The budgeting work presented in Chapter 4 can also be extended to several other application domains. In particular, power budgeting is a very important problem for FPGAs, such as the Stratix III [LAC⁺09]. In the Stratix III, LABs can be configured to a low power mode, the drawback to this is that the performance of the LAB drops. Thus, only non-critical portions of the circuit can be set to low power. This significantly drops the overall power use of the FPGA with little or no impact to performance. The method they use to reduce the power in their circuit is incremental in nature, where post-placement, individual LABs are configured to low power in conjunction with timing analysis to ensure that the performance of the circuit has not degraded. There is potential to improve this process if the power configuration occurred in conjunction with placement or other steps in the CAD flow. This could be possible with the assistance of budget management. For example, during placement, a power budget could be assigned to each LAB. By following the budget of each LAB, the cost function of the placement algorithm could be altered to account for this to help increase the number of opportunities to set LABs to a low power configuration.

Finally, the ECO work presented in Chapter 5 can be extended to create a more robust ECO flow. In particular, extending the ECO work to handle retiming circuits, or if the latch behaviour is not well defined would be extremely powerful. Since retiming modifies the behaviour at each latch, our existing ECO approach is not applicable since it requires that the register behaviour be well defined in the model netlist. As a solution, a record of the steps taken during retiming can be made. This idea was first presented in [MB08] for sequential verification. Once a history is created, the retiming operations can be “undone”

during the ECO process, or the history can be encoded into our SAT formulation to obtain the proper modification for ECOs. The work done in [SKB01] may also be applicable to our problem. In [SKB01], the authors introduce the notion of *sequential SPFDs* which can be used to express “flexibility” within a logic network. Since in our ECO work, we ultimately want to find alternate implementations of a given Boolean function, quickly finding flexibility within the network may increase the number of opportunities to find valid modifications for our ECOs.

Overall, by improving the design efficiency of the FPGA design flow, we can significantly reduce the barriers of entry in using FPGA technology. This not only reduces the end costs of FPGA based products, but leads to wider use and easier access to this exciting and powerful technology.

7 Appendix: ZDD Prune Algorithm

ZDDs can also be used to compress our cut representations, however, there are a few drawbacks when using ZDDs in our case. First, since ZDDs cannot allow for complemented edges, a ZDD representations for each cut set is larger than the BDD representation. Also, dominator cuts are not removed in the ZDD representation. This has strong implications on the runtime of creating cutsets. When we embed ZDDs into our cut generation algorithm, it is 2x slower than the BDD based approach. Furthermore, it consumes approximately 10% more memory. The following lists the algorithms used to generate cuts when using a ZDD.

Figure 7.1 illustrates the ZDD algorithm which prunes the cut set. Note that this is analogous to `BDDANDPRUNERECUR` Chapter 3, though applied specifically to set manipulation in ZDDs.

Figure 7.2 illustrates the impact of removing dominator cut $cdefg$ from cut set in BDD representation versus ZDD representation. Without the removal of the dominator cuts, the ZDD has one extra BDD node. This problem is multiplies for larger cut sets as shown in Figure 7.3. In Figure 7.3, the BDD is approximately 30% smaller than the ZDD.

```

<  $f_z$  > ZddUnateProductPruneRecur( $f_x, f_y, K, n$ )
1  if ISCONSTANT( $f_x$ ) AND ISCONSTANT( $f_y$ )
2    return <  $f_x$  AND  $f_y$  >
3   $b \leftarrow$  GETTOPVAR( $f_x, f_y$ )
4   $fn_x \leftarrow f_x(b = 0)$ 
5   $fn_y \leftarrow f_y(b = 0)$ 
6   $fp_x \leftarrow f_x(b = 1)$ 
7   $fp_y \leftarrow f_y(b = 1)$ 
8   $fn_b \leftarrow$  ZDDUNATEPRODUCTPRUNERECUR( $fn_x, fn_y, K, n$ )
9  if  $n \leq K$ 
10    $f1_b \leftarrow$  ZDDUNATEPRODUCTPRUNERECUR( $fp_x, fp_y, K, n + 1$ )
11    $f2_b \leftarrow$  ZDDUNATEPRODUCTPRUNERECUR( $fp_x, fn_y, K, n + 1$ )
12    $f3_b \leftarrow$  ZDDUNATEPRODUCTPRUNERECUR( $fn_x, fp_y, K, n + 1$ )
13    $f4_b \leftarrow$  ZDDUNATEPRODUCTPRUNERECUR( $fn_x, fn_y, K, n$ )
14    $sum1 \leftarrow$  ZDDUNION( $f1_b, f2_b$ )
15    $sum2 \leftarrow$  ZDDUNION( $sum1, f3_b$ )
16   SETTOPVAR( $f_z, b$ )
17   SETCOFACTORS( $f_z, sum2, f4_b$ )
18 else
19    $f_z \leftarrow$  ZDDUNATEPRODUCTPRUNERECUR( $fn_x, fn_y, K, n$ )
20 return <  $f_z$  >

```

Figure 7.1: High-level overview of ZDD Unate Product operation with pruning for K .

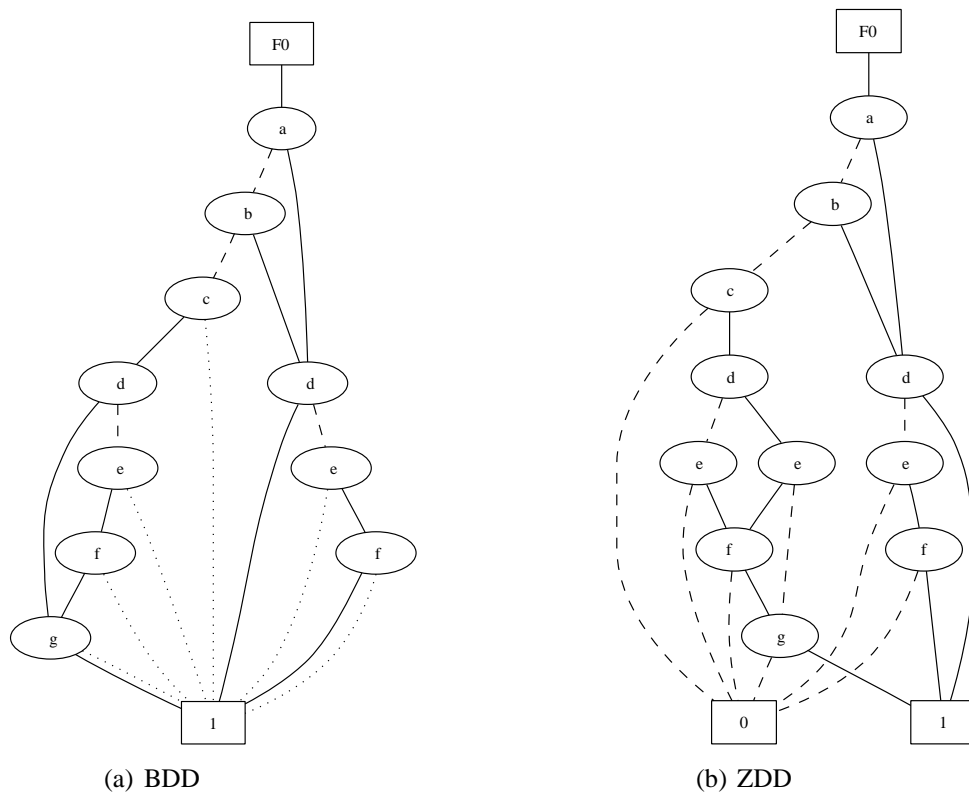


Figure 7.2: Illustration of dominator cut removal in BDD versus ZDD.

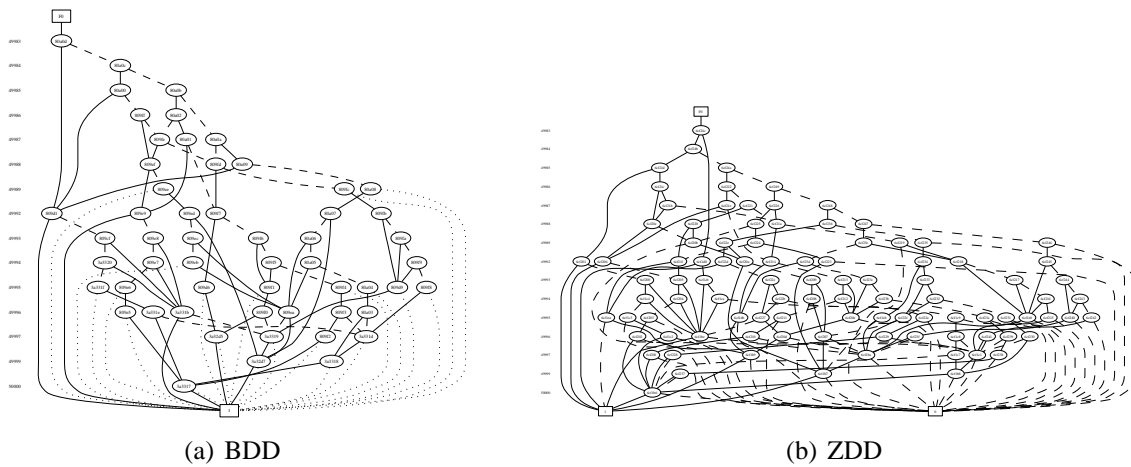


Figure 7.3: Larger example of illustration of dominator cut removal in BDD versus ZDD. Picture taken from the CUDD BDD/ZDD package [Som98].

References

- [Alt04] Altera Corporation. *Stratix II Device Handbook*, October 2004.
- [Alt05a] Altera Corporation. FPGAs for high performance DSP applications. Technical Report wp-01023-1.1, May 2005.
- [Alt05b] Altera Corporation. *Stratix MAC WYSIWYG Description*, May 2005.
- [Alt07] Altera Corporation. *Quartus II Handbook*, May 2007.
- [Alt08] Altera Corporation. *Quartus II University Interface Program*, 2008.
- [AN06] Jason H. Anderson and Farid N. Najm. Active leakage power optimization for FPGAs. *IEEE Journal on Technology in Computer Aided Design*, 25(3):423–437, 2006.
- [AR00] Elias Ahmed and Jonathan Rose. The effect of LUT and cluster size on deep-submicron FPGA performance and density. In *International Symposium on Field-Programmable Gate Arrays*, pages 3–12, 2000.
- [Ash59] R. L. Ashenurst. The decomposition of switching functions. In *International Symposium on Theory of Switching Functions*, pages 74–116, 1959.
- [BD97] Valeria Bertacco and Maurizio Damiani. The disjunctive decomposition of logic functions. In *International Conference on Computer-Aided Design*, pages 78–82, Washington, DC, USA, 1997. IEEE Computer Society.
- [BFG⁺93] Iris R. Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *International Conference on Computer-Aided Design*, pages 188–191, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [BGTS04] E. Bozorgzadeh, S. Ghiasi, A. Takahashi, and M. Sarrafzadeh. Optimal integer delay-budget assignment on directed acyclic graphs. *IEEE Journal on Technology in Computer Aided Design*, 23(8):1184–1199, August 2004.
- [BHMSV84] R. K. Brayton, G. D. Hatchel, C. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

-
- [Bie07] Jeff Bier. DSP performance of FPGAs revealed. *DSP Magazine*, pages 10–11, 2007.
- [BM82] R. K. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In *International Symposium on Circuits and Systems*, pages 49–54, May 1982.
- [BMYS04] E. Bozorgzadeh, S. Oğrenci Memik, X. Yang, and M. Sarrafzadeh. Routability-driven packing: Metrics and algorithms for cluster-based FPGAs. *Journal of Circuits Systems and Computers*, 13:77–100, 2004.
- [BR97a] Vaughn Betz and Jonathan Rose. Cluster-based logic blocks for FPGAs: Area-efficiency vs. input sharing and size. In *International Conference on Custom Integrated Circuits Conference*, pages 551–554, 1997.
- [BR97b] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *International Conference on Field-Programmable Logic and Applications*, pages 213–222, 1997.
- [BR99] Vaughn Betz and Jonathan Rose. FPGA routing architecture: Segmentation and buffering to optimize speed and density. In *International Symposium on Field-Programmable Gate Arrays*, pages 59–68, February 1999.
- [BRV90] Stephen D. Brown, Jonathan Rose, and Zvonko G. Vranesic. A detailed router for field-programmable gate arrays. In *International Conference on Computer-Aided Design*, pages 382–385, November 1990.
- [Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [BS02] Michael Baldamus and Klaus Schneider. The BDD space complexity of different forms of concurrency. *Journal of Fundamenta Informaticae*, 50(2):111–133, 2002.
- [BV04] Stephen Boyd and Lieven Vanderberghe. *Convex Optimizations*. Cambridge University Press, 2004.
- [CC04] D. Chen and J. Cong. DAomap: a depth-optimal area optimization mapping algorithm for FPGA designs. In *International Conference on Computer-Aided Design*, pages 752–759, Washington, DC, USA, 2004.
- [CD93] Jason Cong and Yuzheng Ding. On area/depth trade-off in LUT-based FPGA technology mapping. In *Proceedings of Design Automation Conference*, pages 213–218, 1993.
- [CD94] J. Cong and Y. Ding. FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Journal on Technology in Computer Aided Design*, 13(1):1–13, January 1994.

-
- [CFK96] Y.-W. Chang, D. F. Wong, and C. K. Wong. Universal switch modules for FPGA design. *Design Automation of Electronic Systems*, 1(1):80–101, January 1996.
- [CH95] Jason Cong and Yean-Yow Hwang. Simultaneous depth and area minimization in LUT-based FPGA mapping. In *International Symposium on Field-Programmable Gate Arrays*, pages 68–74, 1995.
- [CH98] Jason Cong and Yean-Yow Hwang. Boolean matching for complex PLBs in LUT-based FPGAs with application to architecture evaluation. In *International Symposium on Field-Programmable Gate Arrays*, pages 27–34, 1998.
- [CJJ⁺03] D. Chai, J.H. Jiang, Y. Jiang, Y. Li, A. Mishchenko, and R. Brayton. MVSIS 2.0 Programmer’s Manual, UC Berkeley. Technical report, Electrical Engineering and Computer Sciences, University of California, Berkeley, 2003.
- [CLL02] Jason Cong, Yizhou Lin, and Wangning Long. SPFD-based global rewiring. In *International Symposium on Field-Programmable Gate Arrays*, pages 77–84, New York, NY, USA, 2002. ACM.
- [CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2 edition, 2001.
- [CM77] E. Cerny and M.A. Marin. An approach to unified methodology of combinational switching circuits. *IEEE Transactions on Computers*, C-26(8):745–746, 1977.
- [CMB06] S. Chatterjee, A. Mishchenko, and R. Brayton. Factor Cuts. In *International Conference on Computer-Aided Design*, pages 143–150, 2006.
- [CMB08] K.H. Chang, I. L. Markov, and V. Bertacco. Fixing design errors with counterexamples and resynthesis. *IEEE Journal on Technology in Computer Aided Design*, 27(1):184–188, 2008.
- [Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC ’71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.
- [CS00] Jason Cong and M. Sarrafzadeh. Incremental physical design. In *International Symposium on Physical Design*, pages 84–93, 2000.
- [DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Journal of Communications of the ACM*, 5(7):394–397, 1962.

-
- [DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
- [ESV92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, Electrical Engineering and Computer Sciences, University of California, Berkeley, 1992.
- [GBCS04] S. Ghiasi, E. Bozorgzadeh, S. Choudhuri, and M. Sarrafzadeh. A unified theory of timing budget management. In *International Conference on Computer-Aided Design*, pages 653–659, Washington, DC, USA, 2004.
- [Goe07] Richard Goering. Post-silicon debugging worth a second look. *EETimes*, 2007.
- [Gol97] Andrew V. Goldberg. An efficient implementation of a scaling minimum-cost flow algorithm. *J. Algorithms*, 22(1):1–29, 1997.
- [Gol04] Steve Golson. The human ECO compiler. In *Synopsys User Group Conference (SNUG)*. Trilobyte Systems, 2004.
- [GT86] A V Goldberg and R E Tarjan. A new approach to the maximum flow problem. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 136–146, 1986.
- [HCC99] Shi-Yu Huang, Kuang-Chien Chen, and Kwang-Ting Cheng. AutoFix: a hybrid tool for automatic logic rectification. *IEEE Journal on Technology in Computer Aided Design*, 18(9):1376–1384, September 1999.
- [ICK04] Mask cost trends. *IC Knowledge LLC*, 2004.
- [JCCM08] Stephen Jang, Billy Chan, Kevin Chung, and Alan Mishchenko. WireMap: FPGA technology mapping for improved routability. In *International Symposium on Field-Programmable Gate Arrays*, pages 47–55, 2008.
- [JJHW97] Jie-Hong Jiang, Jing-Yang Jou, Juinn-Dar Huang, and Jung-Shian Wei. BDD based lambda set selection in roth-karp decomposition for LUT architecture. In *Proceedings of Asia South Pacific Design Automation Conference*, pages 259–264, January 1997.
- [JR05] Peter Jamieson and Jonathan Rose. A verilog RTL synthesis tool for heterogeneous FPGAs. In *International Conference on Field-Programmable Logic and Applications*, pages 305–310, August 2005.
- [LAB⁺05] David Lewis, Elias Ahmed, Gregg Baeckler, Vaughn Betz, Mark Bourgeault, David Cashman, David Galloway, Mike Hutton, Chris Lane, Andy Lee, Paul Leventis, Sandy Marquardt, Cameron McClintock, Ketan Padalia, Bruce

- Pedersen, Giles Powell, Boris Ratchev, Srinivas Reddy, Jay Schleicher, Kevin Stevens, Richard Yuan, Richard Cliff, and Jonathan Rose. The Stratix II logic and routing architecture. In *International Symposium on Field-Programmable Gate Arrays*, pages 14–20, New York, NY, USA, 2005. ACM.
- [LAC⁺09] David Lewis, Elias Ahmed, David Cashman, Tim Vanderhoek, Chris Lane, Andy Lee, and Philip Pan. Architectural enhancements in Stratix III TM and Stratix IV TM. In *International Symposium on Field-Programmable Gate Arrays*, pages 33–42, 2009.
- [Lam03] David Lammers. Spending on masks can pay off, Sematech finds. *EEtimes*, 2003.
- [Lam05] David Lammers. Shift to 65 nm has its costs. *EEtimes*, November 2005.
- [Lar92] Tracy Larrabee. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(1):4–15, January 1992.
- [LBJ⁺03] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Y Marquardt, Cameron McClintock, Bruce Pedersen, Giles Powell, Srinivas Reddy, Chris Wysocki, Richard Cliff, and Jonathan Rose. The Stratix TM routing and logic architecture. In *International Symposium on Field-Programmable Gate Arrays*, pages 12–20. ACM Press, 2003.
- [LBP08] Adrian Ludwin, Vaughn Betz, and Ketan Padalia. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In *International Symposium on Field-Programmable Gate Arrays*, pages 14–23, 2008.
- [LJHM07] Chih-Chun Lee, Jie-Hong R. Jiang, Chung-Yang (Ric) Huang, and Alan Mishchenko. Scalable exploration of functional dependency by interpolation and incremental SAT solving. In *International Conference on Computer-Aided Design*, pages 227–233, 2007.
- [LKJ⁺09] Jason Luu, Ian Kuon, Peter Jamieson, Ted Campbell, Andy Ye, Mark Fang, and Jonathan Rose. VPR 5.0: FPGA CAD and architecture exploration tools with single-driver routing, heterogeneity and process scaling. In *International Symposium on Field-Programmable Gate Arrays*, 2009.
- [LMS04] Qinghua Liu and Malgorzata Marek-Sadowska. Pre-layout wire length and congestion estimation. In *Proceedings of Design Automation Conference*, pages 582–587, 2004.
- [LS91] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

-
- [LSB05a] Andrew C. Ling, Deshanand P. Singh, and Stephen D. Brown. FPGA PLB evaluation using quantified Boolean satisfiability. In *International Conference on Field-Programmable Logic and Applications*, pages 19–24, August 2005.
- [LSB05b] Andrew C. Ling, Deshanand P. Singh, and Stephen D. Brown. FPGA technology mapping: a study of optimality. In *Proceedings of Design Automation Conference*, pages 427–432, New York, NY, USA, 2005. ACM Press.
- [LXZ07] Chuan Lin, Aiguo Xie, and Hai Zhou. Design closure driven delay relaxation based on convex cost network flow. In *International Conference on Design and Test in Europe*, pages 63–68, San Jose, CA, USA, 2007. EDA Consortium.
- [MB08] Alan Mishchenko and Robert K. Brayton. Recording synthesis history for sequential verification. In *International Conference on Formal Methods in Computer-Aided Design*, pages 27–34, 2008.
- [MBJJ09] Alan Mishchenko, Robert Brayton, Jie-Hong Roland Jiang, and Stephen Jang. Scalable don’t-care-based logic optimization and resynthesis. In *International Symposium on Field-Programmable Gate Arrays*, pages 151–160, New York, NY, USA, 2009. ACM.
- [MBR99] Alexander (Sandy) Marquardt, Vaughn Betz, and Jonathan Rose. Using cluster-based logic blocks and timing-driven packing to improve FPGA speed and density. In *International Symposium on Field-Programmable Gate Arrays*, pages 37–46, New York, NY, USA, 1999. ACM Press.
- [MBR00] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-driven placement for FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, pages 203–213, New York, NY, USA, 2000. ACM.
- [MBV06] Valavan Manohararajah, Stephen D. Brown, and Zvonko G. Vranesic. Heuristics for area minimization in LUT-based FPGA technology mapping. *IEEE Journal on Technology in Computer Aided Design*, 25:2331–2340, November 2006.
- [MCB89] J. C. Madre, O. Coudert, and J. P. Billon. Automating the diagnosis and the rectification of digital errors with PRIAM. In *International Conference on Computer-Aided Design*, pages 30–33, 1989.
- [MCB06a] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG Rewriting: A fresh look at combinational logic synthesis. In *Proceedings of Design Automation Conference*, pages 532–536, 2006.

-
- [MCB06b] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Improvements to technology mapping for LUT-based FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, 2006.
- [McM03] Ken L McMillan. Interpolation and SAT-based model checking. In *International Conference on Computer Aided Verification*, volume 2725, pages 1–13, 2003.
- [McM08] Sile McMahan. Gartner revised semiconductor market growth in 2008. *FABTECH*, 2008.
- [MCSB06] Valavan Manohararajah, Gordon R. Chiu, Deshanand P. Singh, and Stephen D. Brown. Difficulty of predicting interconnect delay in a timing driven FPGA CAD flow. In *Workshop on System Level Interconnect Prediction*, pages 3–8, March 2006.
- [ME95] Larry McMurchie and Carl Ebeling. PathFinder: A negotiation-based performance-driven router for FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, pages 111–117, 1995.
- [Men05] Mentor Graphics. *Precision Synthesis: product overview*, 2005.
- [Min93] Shinichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of Design Automation Conference*, pages 272–277, New York, NY, USA, 1993. ACM Press.
- [Mis01] Alan Mishchenko. An introduction to zero-suppressed binary decision diagrams. Technical report, 2001.
- [MMZ⁺01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of Design Automation Conference*, 2001.
- [Mor06a] Kevin Morris. Time for a change: Mentor modernizes the ECO. *FPGA and Structured ASIC*, May 2006.
- [Mor06b] Kevin Morris. Virtex 5 is alive. *FPGA and Structured ASIC*, May 2006.
- [MOS08] MOSEK. *The MOSEK optimization tools manual*, 2008.
- [MSB05] Valavan Manohararajah, Deshanand P. Singh, and Stephen D. Brown. Post-placement BDD-based decomposition for FPGAs. In *International Conference on Field-Programmable Logic and Applications*, pages 31–38, August 2005.
- [MSS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

-
- [Pla05] Daniel Platzker. FPGA design meets the heisenberg uncertainty principle. SOCcentral, November 2005.
- [RBM99] Jonathan Rose, Vaughn Betz, and Alexander Marquardt. *Architecture and CAD for Deep-Submicron FPGAs*. February 1999.
- [RK62] J. P. Roth and R. M. Karp. Minimization over Boolean graphs. *IBM Journal of Research and Development*, pages 227–238, 1962.
- [RMM⁺03] Michael L. Rieger, Jeffrey P. Mayhew, Lawrence Melvin, Robert Lugg, and Daniel Beale. Anticipating and controlling mask costs within EDA physical design. In *SPIE*, volume 5130, pages 617–627, 2003.
- [Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *International Conference on Computer-Aided Design*, pages 42–47, 1993.
- [SB98] Subarnarekha Sinha and Robert K. Brayton. Implementation and use of SPFDs in optimizing Boolean networks. In *International Conference on Computer-Aided Design*, pages 103–110, 1998.
- [Sha38] Claude E Shannon. A Symbolic Analysis of Relay and Switching Circuits. *Transactions of the American Institute of Electrical Engineers AIEE*, 57:713–723, 1938.
- [SKB01] Subarnarekha Sinha, Andreas Kuehlmann, and Robert K. Brayton. Sequential SPFDs. In *International Conference on Computer-Aided Design*, pages 84–90, Piscataway, NJ, USA, 2001. IEEE Press.
- [SMB05a] Deshanand Singh, Valavan Manohararajah, and Stephen D. Brown. Two-stage physical synthesis for FPGAs. In *IEEE Custom Integrated Circuits Conference*, pages 171–178, September 2005.
- [SMB05b] Deshanand P. Singh, Valavan Manohararajah, and Stephen D. Brown. Incremental retiming for FPGA physical synthesis. In *Proceedings of Design Automation Conference*, pages 433–438, 2005.
- [Smi04] Alexander D.S. Smith. Diagnosis of combinational logic circuits using boolean satisfiability. Master’s thesis, University of Toronto, 2004.
- [SMS02] Amit Singh and Malgorzata Marek-Sadowska. Efficient circuit clustering for area and power reduction in FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, pages 59–66, 2002.
- [SMV⁺07] Sean Safarpour, Hratch Mangassarian, Andreas G. Veneris, Mark H. Liffiton, and Kareem A. Sakallah. Improved design debugging using maximum satisfiability. In *International Conference on Formal Methods in Computer-Aided Design*, pages 13–19, 2007.

-
- [Som98] F. Somenzi. CUDD: CU decision diagram package release 2.3.0. University of Colorado at Boulder, 1998.
- [Som99] F. Somenzi. Binary decision diagrams. *NATO Science Series F: Computer and Systems Sciences*, 173:303–366, 1999.
- [SR99] Y. Sankar and J. Rose. Trading quality for compile time: Ultra-fast placement for FPGAs. In *International Symposium on Field-Programmable Gate Arrays*, 1999.
- [SVV04] Alexander Smith, Andreas Veneris, and Anastasios Viglas. Design diagnosis using Boolean satisfiability. In *Proceedings of Asia South Pacific Design Automation Conference*, pages 218–223, 2004.
- [ter09] terasIC. *DE3 User Manual*, 2009.
- [VKT02] Navin Vemuri, Priyank Kalla, and Russell Tessier. BDD-based logic synthesis for LUT-based FPGAs. *ACM Trans. Des. Autom. Electron. Syst.*, 7(4):501–525, 2002.
- [Wil97] Steven J.E. Wilton. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories*. PhD thesis, 1997.
- [Wor06] Jerry Worchel. FPGA market will reach \$2.75 billion by decade’s end. *In-Stat*, (IN0603187SI), 2006.
- [WRV96] Steven J. E. Wilton, Jonathan Rose, and Zvonko G. Vranesic. Memory/logic interconnect flexibility in FPGAs with large embedded memory arrays. In *International Conference on Custom Integrated Circuits Conference*, pages 144–147, 1996.
- [WZ05] Dennis Wu and Jianwen Zhu. FBDD: A folded logic synthesis system. In *International Conference on ASIC*, pages 746–751, Shanghai, China, October 2005.
- [Xil00] Xilinx Corporation. *Xilinx 2000 Databook*, 2000.
- [Xil08] Xilinx Corporation. *SmartCompile Technology: SmartGuide*, 2008.
- [Yan01] Chiang Yang. Challenges of mask cost & cycle time. *Intel*, January 2001.
- [YCS00] Congguang Yang, Maciej J. Ciesielski, and Vigyan Singhal. BDS: a BDD-based logic optimization system. In *Proceedings of Design Automation Conference*, pages 92–97, 2000.
- [YSC99] Congguang Yang, V. Singhal, and M. Ciesielski. BDD decomposition for efficient logic synthesis. In *International Conference on Computer Design*, pages 626–631, 1999.

- [YSVB07] Yu-Shen Yang, S. Sinha, A. Veneris, and R. K. Brayton. Automating logic rectification by approximate SPFDs. In *Proceedings of Asia South Pacific Design Automation Conference*, pages 402–407, 2007.
- [Zha97] Hantao Zhang. SATO: an efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, volume 1249, pages 272–275, 1997.
- [ZLM06] Yue Zhuo, Hao Li, and Saraju P. Mohanty. A congestion driven placement algorithm for FPGA synthesis. In *International Conference on Field-Programmable Logic and Applications*, pages 1–4, August 2006.