Efficient Alternative Wiring Techniques and Applications

SZE, Chin Ngai
BEng

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

碩士研究畢業論文

高效率的基於可替換線邏輯變換及其應用

作者：施展毅

摘要

可替換線邏輯綜合方法研究的基本問題是怎樣針對電路中的一條目標線，尋找一條可替換線。由於把可替換線加入電路中的時候，該可替換線是冗餘的，因此電路的邏輯功能不變。可替換線邏輯綜合方法的用途十分廣泛，包括電路優化 (circuit optimization)，時延優化 (timing optimization)，電路分割規劃 (circuit partitioning) 以及現場可編程門陣列綜合方法 (FPGA synthesis)。

可替換邏輯綜合方法的廣泛用途引起了不少研究小組的興趣及參與。其中大多數的研究採用自動測試生成 (ATPG) 的算法來尋找可替換線，因爲這種算法都比其他算法快捷有效。但這類算法的運行速度仍然緩慢，不足以滿足速度上的要求，原因是它們涉及了大量的邏輯蘊涵 (logic implication) 和冗餘測試。爲了解決以上有關尋找可替換線的問題，我們的研究提出了兩種不同的算法。這兩種算法都比現有的方法更爲有效，而且執行速度更爲快捷。

首先我們提出「基於邏輯蘊涵的可替換線檢驗算法」 (Implication Based Alternative Wiring Logic Transformation)。這個算法是根據不同邏輯蘊涵之間的關係，把傳統基於自動測試生成的算法加以改善。從實驗結果得知，我們的算法有效地減少冗餘測試的次數，從而加快可替換線檢驗的運算速度。

第二種算法是「基於圖像的可替換線檢驗算法」 (Graph Based Alternative Wiring)，它將可替換線辨驗的過程轉化爲一個圖像配對的問題。我們把電路中的線與門和一組預先定義含有可替換線的圖像配置作配對，來尋找目標線及可替換線。因爲這種算法不涉及任何邏輯蘊涵，所以比傳統基於自動測試生成的算法更快捷。但是以往這種算法檢驗出的可替換線數目不及傳統基於自動測試生成的算法，原因是預先定義的圖像配置不足。我們的研究不但大大增加了圖像配置的數目，還採用了一種圖像配置組織分類的方法，從而增加檢驗出的可替換線數目和算法的運行速度。

「基於圖像的可替換線檢驗算法」的概念與傳統基於自動測試生成的算法有很大分別，圖像配對的過程之中，我們完全不知道有關邏輯蘊涵的資料。因此我們的研究提出了一種新的方法，應用圖像配對算法於邏輯優化的問題之中。實驗結果證明了我們的邏輯優化算法不但保持了邏輯優化的質素，而且更大大加快了傳統邏輯優化算法的運行速度。

Abstract

Alternative wiring refers to the process of adding a redundant connection to a circuit so as to make a target connection redundant and removable from the circuit without altering the functionality of the circuit. The technique of alternative wiring has a wide range of applications including logic synthesis such as circuit optimization, timing optimization; and physical design such as partitioning, post-layout logic restructuring and FPGA synthesis.

Due to wide applications of alternative wiring, the study of locating alternative wires effectively and efficiently has drawn the attention of many research groups. They used to adopt ATPG-based approaches to identify alternative wires but their methods involved time-consuming logic implications and redundancy checks. Thus, we explore two different approaches to provide efficient solutions to the alternative wiring problem in this thesis.

Our first approach is to improve the traditional ATPG-based alternative wiring algorithms by implication analysis. Our algorithm, the implication-based alternative wiring logic transformation (IBAW), aims at reducing the number of alternative wire checks by skipping unnecessary identification steps. By exploring the implication relationship among alternative wires, IBAW successfully improves the efficiency of traditional ATPG-based approaches.

By modeling the alternative wire identification as a graph matching process, the Graph Based Alternative Wiring algorithm (GBAW) suggests a different approach to the problem. In GBAW, alternative wires are found by matching the elements in the network with a set of pre-defined graph configurations, which is termed the pattern family. Since GBAW does not involve any logic implication, it runs much faster than traditional ATPG-based alternative wiring algorithms. However, the solution-quality is slightly worse than ATPG-based algorithms since only a small subset of patterns are included in GBAW. Our work is to group the patterns in an organized manner and hence to simplify the pattern implementation as well as the logic transformation. Besides, we propose a number of new patterns which help to increase the solution-quality of GBAW effectively.

Since the concept of GBAW is completely different from the traditional ATPG-based alternative wiring algorithms, the implication information is unknown throughout the GBAW process. Therefore, we develop a new algorithm to apply GBAW in logic optimization. Experimental results in area minimization show that our GBAW optimization algorithm is more efficient than other logic minimization tools with competent optimization ability.

Acknowledgments

First, I would like to express my deepest gratitude to my advisor, Professor David Yu-Liang Wu for his guidance and kindness. He aroused my interest in the research of alternative wiring, piloted me when I was confused and encouraged me when I felt depressed. Besides, I would thank Professor Chak-Kuen Wong, Professor Kam-Wing Ng, Professor Kin-Hong Lee and Professor Fung-Yu Young for their advice.

Special thanks are given to Rachel Wai-Yiu Wong, Ray Chak-Chung Cheung, Jessica Gut-Yee Hui, Ada Kwan, Martinian Chan, Nancy Chan, Wangning Long, Professor Shih-Chieh Chang, Professor Wayne Wolf, Professor Eby G. Friedma and Robbin Dodson.

Last but not least, the greatest gratefulness are owed to my family. I am just nothing without them.

Curriculum Vitae

Born - Hong Kong - 1st September, 1977.

Education

1996 - 1999	BEng in Computer Engineering,
	The Chinese University of Hong Kong
	Major: Computer Engineering, Minor: Mathematics
	Final Year Project: Low Earth Orbit Satellite Systems

Publications

- **C.N. Sze**, Y. L. Wu, "Improved Alternative Wiring Scheme Applying Dominator Relationship", Proceedings of Asia and South Pacific Design Automation Conference (ASPDAC) 2001, Pages: 473-478.

- Y. L. Wu, **C.N. Sze**, C.C. Cheung, "On Improved Graph-Based Alternative Wiring Scheme for Multi-Level Logic Optimization", Proceedings of IEEE International Conference on Electronics Circuits and Systems (ICECS),2000.

- P. T. S. Tam, J. C. S. Lui, H. W. Chan, **C. C. N. Sze**, C. N. Sze, " An optimized routing scheme and a channel reservation strategy for a low Earth orbit satellite system ",Proceedings of Vehicular Technology Conference, 1999. VTC 1999-Fall. IEEE VTS 50th Volume:5, 1999 ,Pages: 2870-2874 vol.5

- Y. L. Wu, **C.N. Sze**, W. N. Long, J. N. Bian, "Accelerated Alternative Wiring Logic Transformation by Implication Analysis", in revision of IEEE Transactions on Very Large Scale Integration Systems.

Contents

List of Figures

List of Tables

Efficient Alternative Wiring Techniques and Applications

History teaches us that men and nations behave wisely once they have exhausted all other *alternatives.*

– Abba Eban

Chapter 1

Introduction

1.1 Motivation and Aims

Traditionally, the Computer-Aided Design (CAD) for Very Large Scale Integration (VLSI) can simply be separated into three steps: high level synthesis (involving Behavioral Synthesis and Sequential Synthesis), logic synthesis (including technology mapping) and physical design synthesis[HS96, She99]. The detailed design process is shown in Figure A.1 in Appendix A. High level synthesis consists of the construction of behavioral and functional specification and the conversion from specification into hardware descriptions such as Finite State Machine (FSM). Logic Synthesis refers to the translation of high-level language descriptions into logic designs (a set of technology specific gates and interconnects, or netlist) and the optimization of chip area, speed (delay) and testability. Physical design synthesis transforms the circuit representation into a geometric representation, the physical layout. It involves the process of circuit partitioning, floorplanning, placement, and routing and the objective is to minimize the chip area and to maintain chip performance. Since the design processes are very complicated, these steps often operate separately and, in other words, there is no interaction between them. However, the design processes are changing due to the development of sub-micron and deep sub-micron VLSI technology in the last decade.

Year of 1st Product shipment	1998	2001	2004	2007	2010
Minimum feature size (nm)	250	180	130	100	70

Table 1.1: Projections of VLSI chips feature size

Due to the development of fabrication technology, the VLSI synthesis processes enter the deep sub-micron range (feature size $< 1\mu m$). According to several reports [FS97, Ass97, Ass00] , the VLSI chips feature size will be 100nm and 70nm in 2007 and 2010 respectively. The information is shown in Table 1.1. The decrease in chip feature size brings the following effects:

- The interconnect delay dominates the total path delay, which includes interconnect and gate delays. (Refer to Figure 1.1 [Ass97])
- noise effect is exacerbated.
- power dissipation becomes unmanageable.

As a result, a verified design in logic synthesis steps may violate the rules in physical design steps. The unmanageable violations are related to the timing constraints and power dissipation requirements.

With the development of sub-micron VLSI technology, there is an appeal for the mergence of logic synthesis and physical design. In a panel session in 34th DAC entitled "Physical Design and Synthesis: Merge or Die"[Ped97], all panel members agreed that Physical Design and Synthesis must merge when "deep sub-micron effects", which refer to the increasing dominance of interconnect delay versus gate delay in the total path delay, grow stronger.

In the last decade, there are several research directions relating to Logic synthesis and Physical design. Some of the directions are the buffer insertion, gate resizing, or logic reconstruction after the floorplanning and placement steps. Another direction is to find a new timing model during the logic synthesis steps.

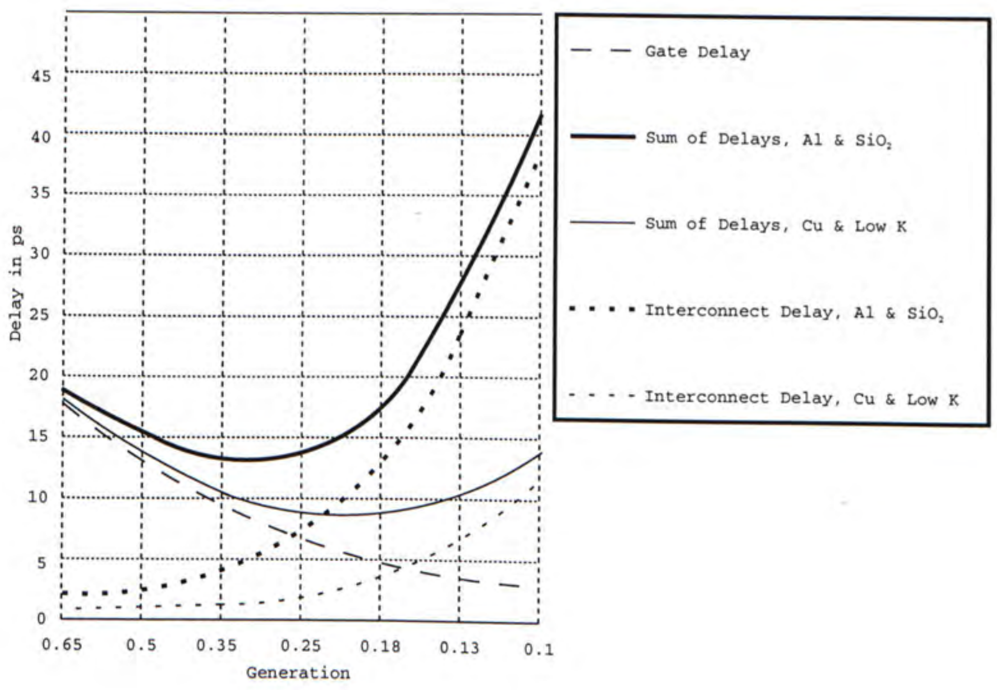


Figure 1.1: Interconnection projections [Ass97]

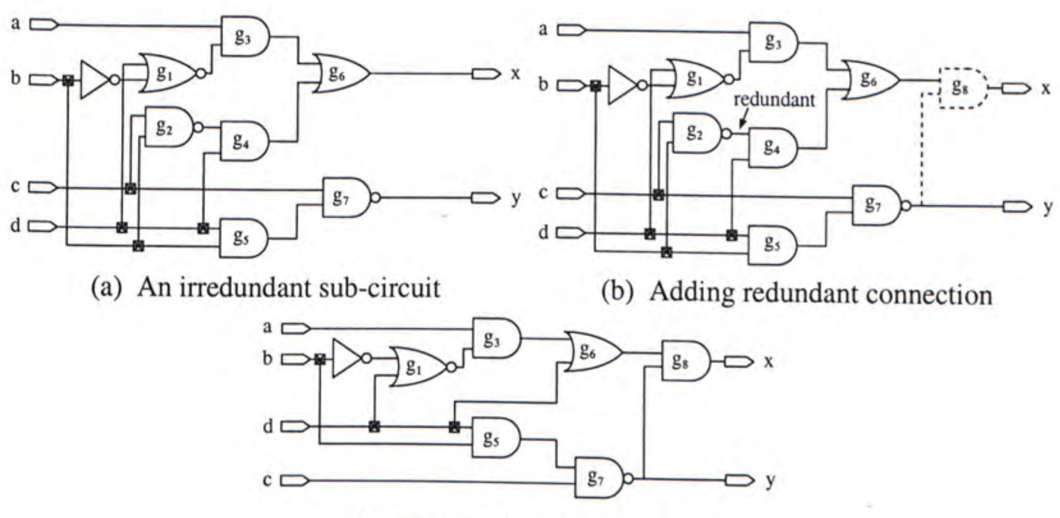
Buffer insertion can decrease the number of fanouts, and in turn minimize the connection delay. Gate resizing refers to the increase in gate size in critical path and the decrease in gate size in non-critical path. The arrangement can decrease the overall chip delay. However, most of the researches cannot provide a satisfactory solution to the problem, which is the divergence between logic synthesis and physical design. A better method to merge the logic synthesis into physical design steps would be performing logic reconstruction, or logic transformation, during physical design processes. For example. a circuit may not fit into a specified chip because of the inflexibility of routing resources. In such case, to route an unroutable circuit successfully, we may need to change the placement by swapping or duplicating some logic functional blocks, and to alter the circuit topology.

For logic transformations, traditional multi-level logic synthesis systems such as SIS [SsLea92] usually adopt algebraic and Boolean methods. These methods are based on different sets of don't cares in circuit simplification. They perform transformation in each logic expression which can be viewed as a sub-circuit transformation. Thus, they are usually unable to perform engineering change [LCMS99], that is the minimization of change in the logic and physical specification for VLSI chips.

In recent years, some Automatic Test Pattern Generation (ATPG) based alternative wiring logic transformation algorithms were proposed [CE93, CLMS95, KM94, CPK98, CMSC96, CGLMS96, IK98]. Unlike SIS, this kind of methods utilize the ATPG techniques, such as logic implication [FS93] and recursive learning [KP92], to substitute the target connection by adding a new wire without changing the network's logic behavior. One of these ATPG-based alternative wiring algorithms is the Redundancy Addition-and-removal for Multilevel Boolean Optimization (RAMBO) [CE93, CLMS95] which has been well developed for different applications.

The concept of alternative wiring logic transformation is to add a redundant connection to the circuit which in turn makes another wire redundant. A wire in a circuit is redundant if its addition and removal does not alter the functionality of the circuit. An example of alternative wiring logic transformation is shown in Figure 1.2. Figure 1.2(a) shows an irredundant circuit. The additional connection $g_7 \rightarrow x$ with an AND gate in Figure 1.2(b) is redundant, but the new connection makes the originally irredundant wire $g_2 \rightarrow g_4$ redundant. After removing the connection $g_2 \rightarrow g_4$, a new circuit with the same functionality but much simplified is obtained (shown in Figure 1.2(c)). The connection $g_7 \rightarrow g_8$ is known as the alternative wire of connection $g_2 \rightarrow g_4$. Alternative wiring refers to the replacement of a target wire with its alternative wires.

Alternative wiring algorithms succeed in merging logic synthesis with physical design tools because they are able to replace a circuit element(gate or connection) with another element elsewhere. This kind of logic transformation



(a) An irredundant sub-circuit

(b) Adding redundant connection

(c) Circuit after redundancy adding and removing

Figure 1.2: An example of alternative wiring

fulfills the requirement of engineering change researches since the change can be minimized and it does not affect any other circuit element. For example, an unroutable wire can be replaced by a routable wire, or a "long" wire which violates timing constraints can be replaced by a "short" wire. In the examples, the violations are prevented, while the replacement can be made so that no other connection is affected.

Alternative wiring technique also has a wide range of applications such as circuit optimization [CE93, EC93, EC95, CMSC96, CGLMS99, EEOU96, CGLMS96],timing optimization [EEOU96], partitioning [CLMS95], post-layout logic restructuring [CCWMS97a, CCWMS97b] and FPGA synthesis [CCWMS94]. The latter three applications are for physical design automation of the VLSI circuits.

The first example is depicted in Figure 1.3. Assuming that the thick line is an unroutable or a long wire, it is then possible to replace the connection with its alternative wire. The flexibility in selecting wires will improve the routability and reduce overall delay of the circuit.

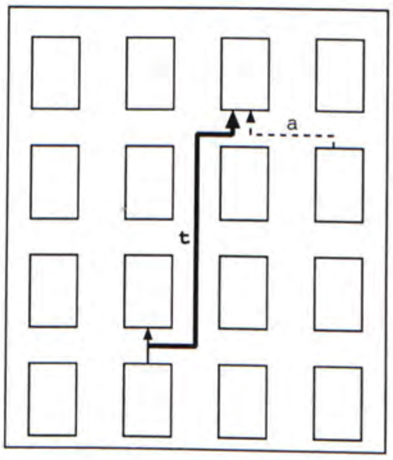


Figure 1.3: Application of alternative wiring - placement and routing

Figure 1.4 shows another example. If connection t is on the critical path and it connects two functional blocks, it will bring long propagation delay to the system. In this case, we can replace the wire with its alternative wires to reduce the critical path delay.

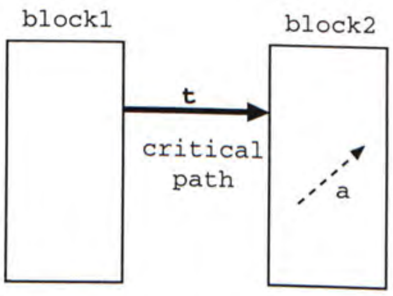


Figure 1.4: Application of alternative wiring - critical path removal

Another application of alternative wiring in partitioning is shown in Figure 1.5. If wire t violates the pin constraints of chips 1 and 2, it is possible to replace t with the alternative wire a, which is inside the chips. The transformation can reduce the number of off-chip connections in the system and hence increase the flexibility of partitioning.

Alternative wiring can also be applied in the area of logic optimization. If

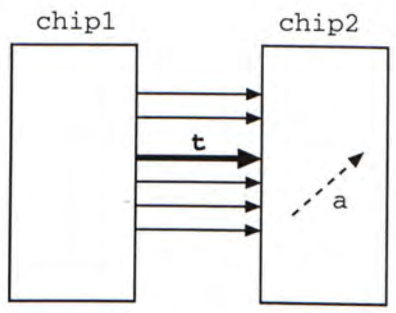


Figure 1.5: Application of alternative wiring - partitioning

the number of circuit elements newly added to the network is less than the number of elements removed, the circuit will be simplified. In Figure 1.2, after replacing $g_2 \rightarrow g_4$ by its alternative wire $g_7 \rightarrow g_8$, the number of gates is reduced. All the above applications demonstrate that alternative wiring is not only a competent solution to fill the gap between logic synthesis and physical design, but it also can be a logic optimization tool.

However, the main problem of the current ATPG-based alternative wiring scheme is that it runs slowly, mainly due to the time-consuming nature of the ATPG procedures. The RAMBO algorithm first selects a target wire, locates a new wire that makes the target wire redundant, and then substitutes the new wire for the target wire. The new connection is a feasible alternative wire only if it is redundant when adding to the network. Otherwise, the addition of the new wire will change the network logic behavior. The most time consuming procedure in RAMBO is the redundancy check of the candidate alternative wires. Our research aims at the acceleration of ATPG-based alternative wiring schemes. To improve the efficiency, the number of redundancy identification must be reduced.

In [WLF00], a new graph-based technique of identifying alternative wire, named Graph-Based Alternative Wiring (GBAW), was first proposed. It first models a circuit network as a directed acyclic graph (DAG).Then it identifies alternative wires by performing graph pattern matching between local sub-graph of the network and the pre-specified minimal sub-graph configurations, which contains alternative wires within a given range limit. Experiments show that the number of all such local minimal sub-graphs is limited and most of the alternative wires are located topologically "near" to their target wires. It is found that about 96% of the closest alternative wires are of only 2-edge-distance from their target wires.

GBAW produces a competitive result in finding alternative wire when comparing to RAMBO. Not only does GBAW perform well in searching alternative wires but it also runs very fast. Experiments show that on average, the CPU running time of GBAW is just 1.4% of that of RAMBO. This significantly short running time makes GBAW a better technique for identifying alternative wires. The efficiency of GBAW is gained mainly because of its avoidance of running the CPU-expensive Boolean implications. Since common alternative wire patterns repeatedly occur in the same circuit, the one-time analysis effort of pattern based rewiring scheme is made practical. However, we observed that GBAW finds less 2-local alternative wires, which are of 2-edge-distance from the target wire, than RAMBO due to the small pre-defined set of minimal configurations shown in the early version [WLF00]. In this problem, we intend to improve the GBAW 's solution quality, especially in locating alternative wires.

1.2 Contribution

In this paper, we first propose an algorithm to accelerate the speed of alternative wiring using the implication relationship between nodes of candidate alternative wires. We have noticed that the implication relationship among source nodes and destination nodes of alternative connections can help us to accelerate the ATPG-based alternative wiring algorithms. In fact, the identification of an alternative wire consists of the selection of a source node and a destination node. Hence, the alternative wiring algorithm becomes more efficient by dividing the procedure of redundancy checking into two parts: selection of suitable source and destination nodes. For example, it is unnecessary to investigate the destination node of a candidate alternative wire if merely the implication relationship of the corresponding source node is sufficient in identifying redundancy.

We propose an algorithm in which redundancy identification is carried out by analyzing the relationship among destination nodes. In [CGLMS99], it is shown that the destination nodes of an alternative wire are always with forced mandatory assignments. In large circuits, the number of feasible destination nodes of an arbitrary target wire can be very large. So, the acceleration in destination node identification would highly expedite the whole alternative wiring scheme. Then, by exploring the implication relationship among destination nodes, we have obtained encouraging results in improving the efficiency of our alternative wiring scheme when compared with RAMBO.

Our algorithm, Implication Based Alternative Wiring Logic Transformation (IBAW), is a general-purposed alternative-wiring algorithm. Since it is capable of finding alternative wires for target wires efficiently, it is also powerful in other applications such as logic optimization, delay optimization and circuit partitioning. Experimental results show that, to find alternative wires for target wires, our scheme runs 6.7 times faster than the complete RAMBO algorithm.

Secondly, a much extended Graph-Based Alternative Wiring (GBAW) scheme to identify alternative wires in multi-level logic with promising results is presented. By modelling subsets of circuits as minimal graphs and applying purely graph-based local pattern search technique, we have found more than 40 graph patterns which contain alternative wires within 2-edge-distance from the target wire. Applying proper grouping technique for the patterns with similar configurations, the complexity of our rewiring technique as well as the CPU run time can be reduced.

Experimental results on MCNC benchmark circuits show that our technique is much faster than the ATPG-based technique RAMBO with competitive number of found alternative wires. With this much augmented pattern family of alternative wires, we are able to find 30% more alternative wires compared to RAMBO, with 75 times speedup on average. To demonstrate GBAW technique in logic minimization, we applied it as a perturbation engine and simplify the target circuit by *SIS* algebraic operations. Results show a further reduction of 11.1% in literal count compared to applying algebraic operations alone.

Besides, we propose the GBAW optimization algorithm, which applies GBAW for logic optimization in circuit area. The idea of our algorithm is based on the simplification and perturbation optimization scheme in [CMSC96]. There are two different parts in GBAW optimization algorithm, which are the logic simplification algorithm and the incremental perturbation heuristic. Both of them focus on the characteristics of all patterns in our pattern family. We classify the patterns into different groups with the same function for logic optimization. Our experimental results show that our algorithm produces competent optimization quality and better efficiency when comparing with other optimization schemes.

1.3 Organization of Dissertation

The rest of the paper is organized as follows. Definitions and notations are introduced in Chapter 2. Discussions of some previous works are included in Chapter 3. Chapter 4 presents the redundancy identification by destination node relationship. Chapter 5 explains the graph based approach to the alternative wiring problem. Chapter 6 shows the GBAW logic optimization algorithm. Conclusion is presented afterwards.

Chapter 2

Definitions and Notations

A Boolean network can be formulated as a directed acyclic graph (DAG). The nodes can be primary inputs (PI), primary outputs (PO), or internal nodes. PI have only out-going edges and PO have only an in-coming edge while internal nodes have at least two in-coming edges and one out-going edge, and they are associated with Boolean functions. Inverter is not treated as an internal node but is considered as the polarity of an edge. In this paper, all gates are assumed to be simple gates including *AND*, *OR*, *NAND*, and *NOR*.

A wire w is an edge which connects two nodes. It can be represented as a triplet, $< S, D, P >$, where S denotes the source node , D denotes the destination node, and $P \in \{0, 1\}$ denotes the polarity of the wire. The input value of the wire w to node D, $\beta(S, D, P)$, is $(\overline{S}P + S\overline{P})$. An absolute dominator, or namely dominator, D of a target wire w is defined as a node such that all paths from w to any PO should go through D.

The controlling value of a node G, $cv(G)$, is defined as the input value of G, which can determine the output of G. Conversely, the non-controlling value of a node G is $ncv(G)$, where $ncv(G) = [cv(G)]'$. The controlled output value of a node G, is defined as the output value when one of the inputs of G is $cv(G)$. Similarly, the non-controlled value is simply the negation of the controlled output value.

The stuck-at-x fault of a wire w, $(w|s{-}a{-}x)$ fault, $x \in \{0, 1\}$, is defined as a fault model of a circuit such that the fault circuit can be modeled as one with w setting to a constant logical value of x.

The test generation of $(w|s\text{−}a\text{−}x)$ fault is defined as the generation of input vectors, known as test vectors, such that with the input vector, the output values of the faulty circuit is different from those of the fault-free circuit.

The set of mandatory assignments (SMA) for the test of $(w|s\text{−}a\text{−}x)$ fault is defined as the value assignment to nodes which must be satisfied by all test vectors of the fault. The node with an assigned logic value is defined as a determined node. The set of mandatory assignments can be classified into three different subsets. The driving mandatory assignment (DMA) is defined as the value assignments in order to drive the target wire to a fault-free value. The observability mandatory assignment (OMA) is defined as the non-controlling value assignments to nodes such that it sensitizes a fault-propagating path to one of the primary output during the $(w|s\text{−}a\text{−}x)$ fault test. The fault propagation mandatory assignment (FPMA) is defined as the value assigned to all dominators of the target wire with the value driven and only controlled by the target wire. The FPMA are denoted as $\{1/0\}$ or $\{0/1\}$ indicating the values in the fault-free circuit and the faulty circuit respectively. In this paper, the symbols D and $\overline{D}$ denote $\{1/0\}$ and $\{0/1\}$ respectively. Forced mandatory assignments, first introduced in [CMSC96], is a subset of SMA, which consists of OMA, DMA and the mandatory assignments obtained by backward propagation. Similarly, forced node is defined as the node with forced mandatory assignment.

Direct implicationis defined as the determination of input (output) values of gates by the corresponding input or/and output value. Take an *AND* gate as an example, one of the inputs with value 0 would imply the output to be 0 while the output value of 1 would imply both input values to be 1. The former example is said to be a forward implication and the latter is a backward implication. In an implication process, implying value always implies the implied value. Take the backward implication of an AND gate as an example, the output value 1 of AND is the implying value and the input values 1 of AND are the implied values.

A wire is said to be redundant in a circuit if and only if the removal of the wire will not change the functionality of the circuit. Based on the concept, we define a wire $w < S, D, P >$ to be ATPG-based redundant if the SMA generated by direct implication for the $(w|s{-}a{-}x)$ fault test is inconsistent, where x is the non-controlling value of D. For convenience, the SMA of $(w < S, D, P > |s{-}a{-}ncv(D))$ fault test is abbreviated as MA_w. Throughout the paper, the word redundant stands for ATPG-based redundant in short. By the abbreviation, MA_{w_a} is defined to be stricter than MA_{w_b} if and only if

$$MA_{w_b} \subseteq MA_{w_a}$$

A wire $w_a < S_a, D_a, P_a >$, which does not exist in the circuit, is defined as an alternative wire of an existing target wire $w_t < S_t, D_t, P_t >$ if and only if it satisfies the following conditions:

*A*1. The addition of w_a to the circuit will change the mandatory assignments for $(w_t < S_t, D_t, P_t > |s{-}a{-}ncv(D_t))$ fault test to inconsistent.

*A*2. After the addition of w_a, the mandatory assignments for $(w_a|s{-}a{-}ncv(D_d))$ fault test is inconsistent.

We demonstrate the above definitions by examples. In Figure 1.2(a), the circuit is not redundant since all connections are irredundant. In Figure 1.2(b), after adding the dotted connection and an *AND* gate, the generation of SMA of $(< g_2, g_4, 0 > |s{-}a{-}1)$ fault is described as follow. FPMA is $\{g_2 = g_6 = g_4 = g_8 = \overline{D}\}$, DMA is $\{c = 1,\ b = 1\}$ and OMA is $\{d = 1,\ g_3 = 0,\ g_7 = 1\}$. By direct implication, we get $g_1 = 0,\ g_5 = 0,\ d = 0$. It is obvious that the SMA of $(< g_2, g_4, 0 > |s{-}a{-}1)$ fault is inconsistent since it contains $d = 1$ and $d = 0$. As a result, $< g_2, g_4, 0 >$ is a redundant wire.

In Figure 1.2(b), it can be proved that the additional connection $< g_7, g_8, 0 >$ is also redundant. The addition of $< g_7, g_8, 0 >$ makes the originally irredundant wire $< g_2, g_4, 0 >$ redundant. So, $< g_7, g_8, 0 >$ is the alternative wire of $< g_2, g_4, 0 >$.

Chapter 3

Literature Review

3.1 Logic Reconstruction

Traditionally, there are two kinds of logic reconstructions, which are either technology-dependent or technology-independent. For technology-dependent logic reconstruction, local optimizations are performed in order to reduce the area delay and power consumption while leaving the structure of the circuit substantially unchanged. The local optimization refers to, for example, the change of gates impedance or transformation between serial and parallel connection structures. They are carried out based on the specific knowledge of the implementation technology.

For technology-independent logic reconstruction, the change of gate-function and local logic structure is concerned. The reconstruction, multi-level minimization, may lead to a simpler, smaller circuit, with shorter delay and higher testability. In most cases, multi-level minimization relies on the externaland internal don't cares. External don't cares exist when the circuit has incompletely specified function while the internal don't cares are related to the local circuit structure.

An important step for multi-level minimization is substitution, or resubstitution. It refers to the addition of new input to a gate. The additional input should be an existing function. In this case, some of the functions can be reused and shared. When part of the function of the gates is replaced by the additional input, some circuit elements can be eliminated. This is actually based on the concept - division of function. For two Boolean functions f and d, it is possible to express their relationship by $f = Qd + R$. With such division relationship, we can perform substitution on the functions.

There are two kinds of substitution: algebraic substitution and boolean substitution, which use algebraic division and boolean division respectively. Both of them use the technique of cube extraction, kernel extraction and a set of don't cares to find the product of the division. Boolean substitution produces better logic minimization results since the don't cares set of boolean division is larger. However, boolean substitutions are more time-consuming.

The following is an introduction of a logic synthesis system which provides us a platform with well-developed functions for performing logic resubstitutions.

3.1.1 SIS: A System for Sequential and Combinational Logic Synthesis

SIS is an open-source logic synthesis system which is developed by the Department of Electrical Engineering and Computer Science in the University of California, Berkeley. It is built on the basis of MISII [BRSVW87], a multi-level logic optimization system, and much enhancement has been made to the logic optimization techniques. In fact, SIS is not only an interactive tool for synthesis and optimization of sequential circuits, but also a comprehensive logic synthesis platform on which developers can implement their systems. The system of RAMBO [CE93] and REWIRE [CGLMS99] are built on the SIS synthesis platform. All experiments in this paper are also built on the same platform.

There are several scripts in SIS for logic optimization by reconstructions such as *script.algebraic*, *script.boolean* and *script.rugged*. *script.algebraic* and *script.boolean* can simulate the algebraic and boolean substitutions. For *script.rugged*, it gives better logic minimization results but it takes more CPU-time since it uses the binary decision diagram (BDD [Lee59, Bry86])-based techniques to find a larger don't cares set. However, if the BDD of the circuits cannot be built, the script will not terminate.

3.2 ATPG-based Alternative Wiring

The basic idea of alternative rewiring is the concept of redundant - adding a redundant wire to make another wire redundant. It has been found that an untestable stuck-at fault indicates a redundant connection. For example, if the stuck-at-1 fault at a connection is untestable, we can replace the connection with a constant "1" signal without changing the circuit's functionality. Under this consideration, the ATPG algorithms can then be applied to identify redundancy (untestable fault).

There are two types of ATPG-based alternative wiring algorithms: the add-first schemes and the target-first schemes. The add-first schemes first add a redundant wire to the network and try to locate the wires which become redundant after the addition of the new wire. On the contrary, the target-first schemes try to locate all alternative wires for a specified target wire. When logic optimization is concerned, some algorithms adopt an add-first approach, instead of the target-first approach used by the original RAMBO [CMSC96]. These schemes are good for logic optimization purpose because adding a new wire (with or without an additional gate) may lead to the removal of more than one wire. Several methods are proposed to quickly identify the redundant wires for the add-first scheme [CGLMS96, IK98]. By excluding some wires that are vital for keeping the new connection redundant, reference [CGLMS96] diminishes the search space of possible redundant wires, and accelerates the process.

In [IK98], a new direct RID method was proposed to expedite the redundancy identification process. However, we notice that the add-first scheme is not suitable for post-layout logic transformation. The reason is that post-layout logic synthesis tools prefer selecting a target wire first and then searching for alternative wires for substitution. As a result, most alternative wiring algorithms for post-layout logic transformation utilize the target-first approach.

3.2.1 Redundancy Addition and Removal for Logic Optimization

In [CE93], the idea of alternative wiring based on ATPG techniques was first published. The algorithm is named Redundancy Addition and Removal for Logic Optimization (RAMBO). The authors first proposed the connection fault model. In the model, an extra connection is generated by the fault. Then, by identifying the testability of the fault, a feasible redundant connection can be found. Having the redundant connection added into the circuit, the feasible target wire can be found by ATPG-based redundancy removal algorithms. The paper also explained the implication of mandatory assignments in test generation. The outline of their algorithm of logic optimization by alternative wiring is shown in Figure 3.1. The paper stated that the destination nodes of a feasible alternative wire should be the dominators of the target wire.

3.2.2 Perturb and Simplify Logic Optimization

Based on RAMBO, the paper [CMSC96] proposed new techniques to improve both the alternative wiring scheme and the logic optimization algorithm. First, the authors had extended the alternative wiring algorithm and showed that the alternative wiring logic transformation can be divided into several different types. The types include single alternative wire, multiple wire addition, gate function substitution and simultaneous alternative wires. After that, a

```
logic_optimization()
{
    while(Under a specified time limit) {
        for each gate $G$ {
            based on the MA of $G$, find all redundancy;
            add all found redundancy to the circuit;
        }
        if (redundancy is added){
            remove redundant wires according to a cost value;
            /* the cost value is calculated based on
               its effect on the circuit area */
        }
}
```

Figure 3.1: Outline of the logic optimization algorithm in [CE93]

logic optimization algorithm, termed Perturb and Simplify, was proposed and the experiment shows that their algorithm is better than RAMBO in both efficiency and the quality of optimization.

Single alternative wire is the removal of a wire by the addition of an alternative wire. Using the forward MA (generated by forward implication) together with the backward MA (by backward implication), it is possible to identify more single alternative wires. Moreover, by exploring the fact that the destination nodes of a feasible alternative wire can be nodes other than the dominators, their algorithms are able to consider more connections. In order to further improve the efficiency, the authors had concluded a complete set of all possible transformations (connection fault models).

For the multiple alternative wires, the paper explored the cases where adding one wire may cause the removal of multiple wires. The number of circuit elements(connections and gates) decrease in such cases so the research is very important for circuit area minimization. Besides, the authors also found that for some cases, the change of a gate's function will not alter the functionality of the whole circuit. After the explanation, they proposed procedures to locate these two special logic transformations by observing special SMA patterns.

Simultaneous alternative wires are a pair of wires w_1, w_2 in the circuit such that each of them is not redundant, but simultaneously adding (removing) w_1 and removing (adding) w_2 will not alter the circuit functionality. In this case, we can add w_1 (w_2) and then remove w_2 (w_1) but the alternative wire cannot be identified by redundancy check. The paper presented the conditions for simultaneous alternative wires to be identified. It stated that if the fanin of a XOR/XNOR gate have a specific don't care term, the simultaneous alternative wire will exist in the fanin connection.

After all the alternative wiring types are presented, a circuit minimization algorithm, Perturb and Simplify, was proposed in the paper. The algorithm consists of two steps: greedy optimization and circuit perturbation.

In the greedy optimization step, it tries to remove as many connections as possible when adding one wire. For each node D, all the connections with D as their dominator are selected. Then, for each selected wires, all its alternative wires are recorded. After that, a list of alternative wires is obtained. From the list, it can be concluded that some wires, e.g. w_1, w_2, w_3, may share the same alternative wire, e.g. w_a. That means adding the wire w_a makes those wires redundant. In other words, the addition of w_a and the removal of w_1, w_2, w_3 do not change the functionality of the circuit. The removal of at least one connection by adding one connection is named Incremental Transformations. This step is recursively performed until no more minimization can be done.

The greedy step may stuck at a local minimum, so the perturbation step is proposed to bring the circuit out of the local minimization. Circuit perturbation is single alternative wire transformations in order to increase the internal don't cares in the circuit and in turn facilitate the greedy optimization step. There are three rules in choosing alternative wires to increase the internal don't cares. It is advisable to have:

- nodes with fewer fanout
- nodes far from primary output
- wires with more parallel fanin

The authors stated that the rules are intuitively correct and do not guarantee the increase of internal don't cares. Therefore, it is necessary to keep track of the quantity change of the internal don't cares. Their heuristic uses the number of dominators as the cost function for a specific wire.

The overall optimization algorithm iteratively performs greedy optimization and circuit perturbation for a limited number of times. Experimental results show that their algorithm performs very well in optimizing combinational circuits and sequential circuits. Besides, their memory requirement is quite low when compared with RAMBO.

3.2.3 REWIRE

In [CGLMS99], the ATPG-based alternative wiring algorithm is further improved in its efficiency. First, the paper refined the concept of MA and introduced a new notation, forced mandatory assignment. From the concept of forced MA, the backward alternative wires (identified by backward MA) is better-defined.

Then, the authors proposed five conditions for a wire to be an infeasible alternative wire. If a wire possesses some properties, we know that it must be an infeasible alternative wire without any identification procedure. A wire is not feasible (irredundant) if:

- a wire is not visited during the implication process.
- its destination node has a forced observability MA.
- it holds an MA of destination node's ncv and the destination node has no MA

- its parallel fanin holds a MA of destination nodes of value cv
- its removal does not change the observability MA.

Utilizing these five filtering conditions, the omission of a number of unnecessary redundancy checks is possible. As a result, the alternative wiring algorithm is greatly accelerated.

The authors then deduced two necessary and sufficient conditions for a redundant wire to be an alternative wire of a target wire. For a candidate alternative wire $w < S, D, P >$, the conditions are:

- S should have an MA with value $cv(D) \oplus P$ for the stuck-at fault test of target wire
- D should have a forced MA with value $ncv(D)$ for the stuck-at fault test of target wire

According to the conditions as well as some previous works, the authors proposed a procedure for logic optimization. The algorithm is shown in Figure 3.2.

The algorithm is much more efficient than [CMSC96] since it first uses a destination node to select "good" target wire. Then it uses the selected target wires to skip "bad" source nodes. The two filters trim out a large number of unnecessary redundancy identification.

3.2.4 Implication-tree Based Alternative Wiring Logic Transformation

In [LWB00], the redundancy identification by the implication relationship among source nodes of the alternative wires is proposed. For two candidate alternative wires that share a common destination node, we assume that the two source nodes are g_1 and g_2, which have determined logic value a and b

```
logic_optimization()
{
    for each $D$ in the circuit {
        find the OMA of $D$
        for each wire $w\_t$ in the fanin and fanout cone of $D$ {
            use previous five conditions to skip $w\_t$ if it
            cannot be redundant;
            find MA of $w\_t$ stuck-at fault;
            use two necessary and sufficient conditions to
            skip all wires $S \rightarrow D$ which is not a feasible
            alternative wire;
            store all possible $S$ into source_node_array;
        }
        optimize the circuit by 'Greedy Optimization' in [CMSC96];
    }

}
```

Figure 3.2: Outline of the logic optimization algorithm in [CGLMS99]

respectively. If $g_1 = a$ implies $g_2 = b$ and the candidate wire from g_1 is irredundant, then the candidate wire from g_2 is also irredundant. Utilizing this relationship, the approach can accelerate the ATPG-based scheme significantly, because many infeasible alternative wires are discarded without invoking the time-consuming redundancy identification procedure.

In order to apply the implication relationship of the source nodes for efficiency improvement, the authors proposed a data structure, the implication-tree, to store the implication relationship. The source node selection of an alternative wire from the implication-tree avoids many needless redundancy checking.

The details of the Implication-tree Based Alternative Wiring [LWB00] will be discussed in section 4.1.

3.3 Graph-based Alternative Wiring

Graph Based Alternative Wiring (GBAW) was first proposed in [WLF00]. First, it models a circuit network as a directed acyclic graph (DAG). Then, it identifies alternative wires by performing graph pattern matching between local sub-graphs of the network and the pre-specified minimal sub-graph configurations which contain alternative wires within a given range limit. The scheme works because it has been shown that most alternative wires are of only 2-edge-distance from their target wires.

The paper also demonstrates that GBAW produces a competitive result in finding alternative wire when comparing to RAMBO. GBAW not only performs well in searching alternative wires but also runs very fast as no ATPG-based implication is involved in alternative wire identification. The significantly short running time makes GBAW a potentially and considerably different technique for identifying alternative wires.

The details of the Graph-Based Alternative Wiring [WLF00] will be discussed in section 5.

Chapter 4

Implication Based Alternative Wiring Logic Transformation

4.1 Source Node Implication

4.1.1 Introduction

In this section, the technique in [LWB00] is reviewed. By exploring the implication relationship between alternative wires with the same destination node but different source nodes, we develop a theorem as well as an implication-tree data structure. Selecting source nodes from the implication-tree, the technique is able to trim out a significant number of unnecessary redundancy identifications.

4.1.2 Implication Relationship and Implication-tree

In a network after mandatory assignment and logic implication process, many nodes are assigned with a determined logic value. We define an implication relationship between two determined nodes as the following.

Definition 1 (Implication relationship between two determined nodes)

Suppose g_0 and g_1 are two determined nodes in the network under consideration, $g_0 = b_0$, and $g_1 = b_1$. If $g_0 = b_0$ results in $g_1 = b_1$, we say $g_0 = b_0$

implies $g_1 = b_1$, denoted $g_0 = b_0 \Rightarrow g_1 = b_1$; or for brevity, g_0 implies g_1, denoted $g_0 \Rightarrow g_1$.

The implication relationship between determined nodes possesses transitivity property, i.e., if $g_0 \Rightarrow g_1$ and $g_1 \Rightarrow g_2$, then $g_0 \Rightarrow g_2$. And, g_0 directly implies g_1 if g_0 implies g_1 through no transition,. In Figure 4.1, for example, g_5 directly implies g_2, d and g_6, while implies g_1 and g_4 through transition.

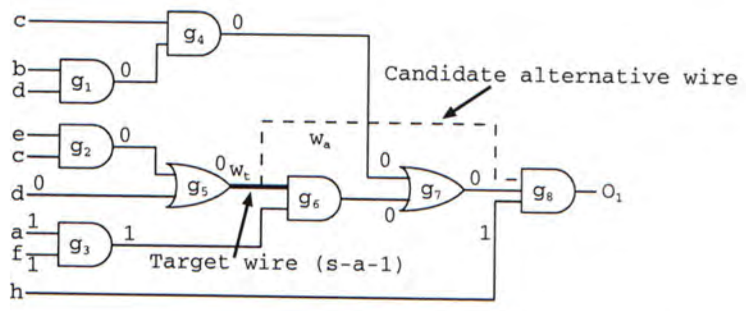


Figure 4.1: An example of implication relationships

Theorem 1 Considering two determined nodes g_0 and g_1, where $g_0 = b_0$ and $g_1 = b_1$, such that $g_0 \Rightarrow g_1$, let $< g_i, D, p_i >$, where $(i = 0, 1)$, be two candidate wires while D is an internal node. p_i is the correspondent polarity that makes $\beta(g_i, D, p_i) = cv(D)$. If $< g_0, D, p_0 >$ is irredundant, $< g_1, D, p_1 >$ is also irredundant.

Proof As the two candidates are connected to the same destination node, they share the same fault propagation path. Hence, the OMA are the same for the two candidates. Besides, considering the fault driving assignment when the redundancy checking is performed for wires $< g_i, D, p_i >$ $(i = 0, 1)$, the fault that may cause $< g_i, D, p_i >$ redundant is $(s{-}a{-}ncv(D))$. Therefore, the DMA on $< g_i, D, p_i >$ requires $((g_i, D, p_i) = cv(D)$. Recall that when $g_i = b_i$ is given, the value of p_i is defined to make $\beta(g_i, D, p_i) = cv(D)$. Hence, the above DMA equals to those of $g_i = b_i$. As g_0 implies g_1, after the mandatory assignment and logic implication for $(< g_0, D, p_0 > |s{-}a{-}ncv(D))$, both g_0 and g_1 become determined, i.e. $g_0 = b_0$ and $g_1 = b_1$. So if $< g_1, D, p_1 >$ is redundant , $< g_0, D, p_0 >$ must be also redundant. By contrapositive, if $< g_0, D, p_0 >$ is irredundant, $< g_1, D, p_1 >$ is also irredundant. □

It is observed that most candidate alternative wires are not redundant. Hence, applying Theorem 1, many irredundant candidate wires can be easily eliminated without calling the time-consuming ATPG procedure. For example in Figure 4.1, $g_3 = 1 \Rightarrow a = 1$, so if $< g_3, g_8, 1 >$ is irredundant, $< a, g_8, 1 >$ is also irredundant. Thus the whole logic transformation process can be greatly accelerated. To store the implication relationship and apply Theorem 1, we propose a data structure, implication-tree. We simplified the relationship to a tree, since the number of checks in node selection procedure are similar to that of a graph while the construction of the tree is more efficient.

Definition 2 (Implication-tree) Given a determined-node set $V_1 = v_1, v_2, ..., v_k$, an implication-tree is a tree whose vertex set is $V = v_R, v_1, v_2, ..., v_k$, where v_R is the root which does not correspond to any nodes in the network. The children of v_R are the determined nodes that are not implied by any other nodes. A leaf vertex corresponds to a node in the network that does not imply any other nodes. Apart from the root and leaves, others are called internal vertices. An internal vertex directly implies all its children. There are two kinds of edges in an implication-tree, parent-child edge and sibling edge. The parent-child edge, denoted by a vertical solid line, links a parent to its first child. The sibling edge, denoted by a horizontal dotted line, links two close siblings. Of a parent, all children form an array linked by sibling edges. All the vertices except v_R have a logic value marked next to it. Besides, every vertex in the tree must possess one and only one parent. If two or more nodes imply a vertex, we only assign the vertex as a child of any one of them.

For example, Figure 4.2 gives an implication-tree corresponding to the SMA shown in Figure 4.1. The target wire is $w_t = g_5 \rightarrow g_6$. Let the determined-node set under consideration be $V_1 = a, f, d, g_1, g_2, g_3, g_4, g_5$. Then, the vertex set of the implication-tree is $V = v_R, a, f, d, g_1, g_2, g_3, g_4, g_5$, where v_R is the root. g_3 and g_5 are the children of v_R , since they are not implied by any other nodes. The edge from g_3 to a is a parent-child edge, with a being the first child of g_3. The edge from a to f is a sibling edge, which means that a and f are two close siblings in the child array. Both a and f are leaf vertices. The rest of the implication-tree is built similarly according to definition 2.

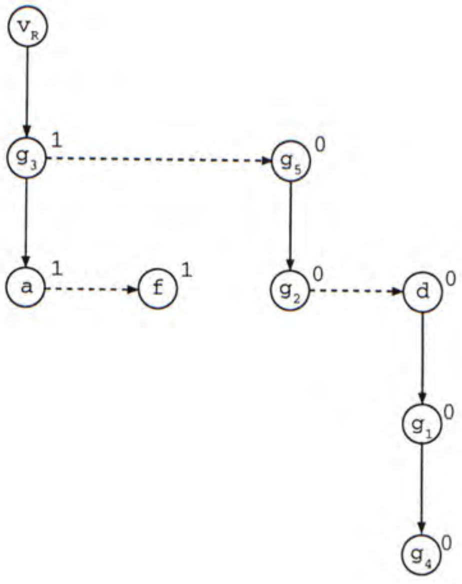


Figure 4.2: An example of an implication-tree

In the above example, g_6, g_7 and g_8 are excluded from V_1, the reason is that according to [CMSC96], the connections from the nodes in the fanout-cone of the target wire cannot be a feasible alternative wire. As a result, only the determined nodes outside the output-cone of the target wire are selected to establish an implication-tree.

To conclude, after the mandatory assignment and logic implication for a target wire, the corresponding implication-tree can be established in the following two steps: (Assume that the nodes under consideration are outside the output-cone of the target wire)

Step 1 For each determined node, store the implication relationship between its inputs and the output. For example, let i_1 be an input of node n. If n implies i_1 which has no parent until now, i_1 is put into the child-array of node n.

Step 2 Assign all the determined nodes which have no parent to the child-array of v_R.

4.1.3 Selection of Alternative Wire Based on Implication-tree

From $A1$ and $A2$, a candidate alternative wire is required to make the target wire redundant while it should be redundant after its addition into the network. Besides, as mentioned previously, the selection of a candidate alternative wire can be divided into two parts: selection of source and destination nodes.

In [CMSC96], it is proved that the destination node of an alternative wire should have the following properties.

R1. It should have forced mandatory assignment which is consists of the OMA, DMA and the MA obtained by backward propagation.

R2. The logic value of the on-input of the dominator should be D for OR and NOR gates, and $\overline{D}$ for AND and $NAND$ gates.

We adopt the above criteria to select destination nodes while our algorithm merely focuses on the efficient selection of a source node. In RAMBO, the process is quite time-consuming. However, by applying implication-tree, we can speed the process up significantly.

Given destination node D, a node S is called a feasible source node if the candidate wire $< S, D, P >$ is a feasible alternative wire. In Figure 4.3, the algorithm describes the selection of a source node from the implication-tree.

The main idea of the source node selection algorithm is explained as follows. Before the procedure is called (except the first time), it always checks whether the node, which is selected last time, is a feasible source node. If the node is feasible, the node-pointer is modified to its first child if it exists, else it is pointed to its close sibling in the child array. Otherwise, if it is an infeasible node, the node-pointer is modified to its close sibling or its parent's close sibling. At the end of the procedure, if the node-pointer is pointing to a non-root vertex of the implication-tree, the vertex is returned. Otherwise, NULL is returned.

Throughout the procedure, the parameter R_1 is a flag that marks whether the node selected last time is a feasible source node. For the first time when the procedure is called, P_1 is Null. So P_1 is pointed to the first child of the root v_R and returned. For the next times, R_1 is first checked. If R_1 is TRUE, it is necessary to visit its children, and P_1 is modified to P_1's first child if there exists any, or else to its next sibling or parent's next sibling and so on if it has no children. If R_1 is FALSE, (the node selected last time is an infeasible source node, and its children are also infeasible), P_1 is pointed to P_1's sibling or its parent's sibling and so on. Moreover, the vertex is returned if P_1 points to a non-root vertex of the tree finally. Otherwise, NULL is returned, showing that no more nodes can be selected.

Consider the implication-tree in Figure 4.2 as an example. Supposing g_8 in Figure 4.1 is selected as the destination node of the candidate wire, the above procedure is invoked for several times as following.

(1) At the first time when the procedure is called, P_1 = NULL. So P_1 is

```
Select-a-node-from-implication-tree($R_1$)
$R_1$ is TRUE or FALSE; /* ($R_1$=TRUE) means the node
visited last time is a feasible source node. */
Begin
/* $P_1$ = global variable pointing to the last visited node.
   $P_1$'s initial value is Null. After the tree has
   been traversed, $P_1$ is set as $v_R$, the root of
   the implication-tree. */
    if ($P_1$ == Null) {
        $P_1$ = first_child_of ($v_R$);
        return $P_1$;
    } else if ($R_1$==TRUE) {
        if ($P_1$ has at least one child) {
            $P_1$ = first_child_of ($P_1$);
            return $P_1$;
        } else if ($P_1$ has unconsidered siblings) {
            $P_1$ = next_sibling_of ($P_1$);
            /* close sibling that has not been examined */
            return $P_1$;
        } else {
            /* $P_1$ has no child and
               no unconsidered siblings. */
            while ($P_1$ has no unconsidered siblings
                   && $P_1 \neq v_R$)
                $P_1$ = parent_of ($P_1$);
            if ($P_1$ == $v_R$)
                return NULL; /* No more nodes */
            else {
                $P_1$ = next_sibling_of ($P_1$);
                return $P_1$;
            }
        }
    } else /* if $R_1$ == FALSE */
        if ($P_1$ has unconsidered sibling) {
            $P_1$ = next_sibling_of ($P_1$);
            /* close sibling that has not been examined */
            return $P_1$;
        } else {  /* $P_1$ has no other siblings */
            while ($P_1$ has no unconsidered siblings
                   && $P_1 \neq v_R$)
                $P_1$ = parent_of ($P_1$);
            if ($P_1$ == $v_R$)
                return NULL; /* No more nodes */
            else {
                $P_1$ = next_sibling_of (parent_of ($P_1$);
                return $P_1$;
            }
        }
    }
End
```

Figure 4.3: Implementation of selecting source node from implication-tree

modified to v_R's first child, g_3, which is then returned.

(2) As g_3 is an infeasible source node, R_1 = FALSE, and P_1 is modified to g_3's next sibling g_5 and returned.

(3) As g_5 is a feasible node, R_1 = TRUE. So P_1 is modified to g_5's first child, g_2 and returned.

(4) As R_1 = FALSE, P_1 is modified to g_2's next sibling, d and returned.

(5) As R_1 = FALSE, P_1 is modified to v_R. Hence, Null is returned, which means that no more nodes can be selected. Finally, g_5 becomes the only feasible node in the network.

After the procedure, we know that the corresponding feasible alternative wire is $< g_5, g_8, 0 >$. Our algorithm needs only 4 trials, while the original RAMBO needs 8.

4.1.4 Implication-tree Based Logic Transformation

The pseudo-code in Figure 4.4 presents the framework of our implication-tree based logic transformation.

```
SrcId(net)
net is the network under consideration;
Begin
    for_each_node (net, $n_1$)
        for_each_fanout ($n_1$, $o_1$) {
            $w_t$ = $n_1$->$o_1$;
            SrcId-transform (net, $w_t$);
        }
End
```

Figure 4.4: Framework Implementation of Implication-tree Based Logic Transformation

The scheme tries to find alternative connections for every wire in the network. In AW, SrcId-transform() is a key function, which tries to find an alternative wire to substitute the target-wire, w_t. In Figure 4.5, the pseudo-code describes the details of SrcId-transform().

At the beginning of the process, the redundancy identification for w_t is performed. If w_t is redundant, it is removed; otherwise, an implication-tree is built from the results of the logic implication process. For each candidate destination node X, a temporary buffer node D is added right after X, where D is assumed to be the destination node of the candidate alternative wire. The buffer node can be converted into an AND gate or OR gate when the candidate wire is introduced. Then the implication-tree is traversed in order to identify a feasible source node S such that wire $S \rightarrow D$ is a feasible alternative wire. The alternative wire is stored in an array named *alt_array* with 4 parameters, S, X, P and T, where P and T are the polarity and the gate type required for the alternative connection respectively.

After the implication-tree has been traversed, the buffer node is removed to keep the original circuit unchanged. If *alt_array* is not empty, which means there exists at least one alternative wire, the other destination nodes will be ignored. However, if we want to find as many alternative wires as possible, line 24 and 25 can simply be removed.

If *alt_array* is not empty at the end of the procedure, a candidate wire is chosen from the array to substitute the target wire. The procedure Add(net, w_a) is intended to add the alternative wire w_a into the network net. As mentioned previously, we use S, X, P and T to store a candidate alternative wire. If X has only one fanout node whose gate type is T, the alternative wire is directly connected from S to X's fanout node with polarity P. Otherwise, a new gate D with type T is added right after X to be the destination node of the alternative wire. Actually, the possibility for the alternative wires to exist is higher if we add a new gate behind the destination node.

```
SrcId-transform(net, $w_t$)
net is the network under consideration;
$w_t$ is the target-wire;
Begin
1     if ($w_t$ is redundant) {
         /* Redundancy checking is called for $w_t$. */
2         Remove (net, $w_t$);      /* remove $w_t$ from net. */
3         return TRUE;
4     }
5     $v_R$ = Generate-implication-tree(net, $w_t$);
      /* $v_R$ is the root of implication-tree. */
6     Put $w_t$s candidate destination nodes into $S_d$;
      /* $S_d$ is an array. */
7     Sort-destination-node-array ($S_d$);
8     for ($i_1$=0; $i_1$ < length of $S_d$; $i_1$++) {
9          X = $S_d$ [$i_1$];
10         D = Insert-buffer-node-after (X);
11         $P_1$ = NULL; /* $P_1$ is a global node pointer. */
12         S = Select-a-node-from-implication-tree (TRUE);
13         while (S != $v_R$) {
14              Determine P;
                /* the polarity of the candidate alt. wire */
15              $w_a$ = <S, D, P>; /* candidate alt. wire */
16              if ($w_a$ is redundant) {
                     /* $w_a$ is a feasible alt. wire. */
17                   Put $w_a$ into alt_array;
18                   $R_1$ = TRUE;
19              }
20              else $R_1$ = FALSE;
21              S = Select-a-node-from-implication-tree ($R_1$);
22         }
23         Delete-buffer-node (D);
24         if (alt_array is not empty)
25              break;          /* break from the for loop. */
26    }
27    if (alt_array is not empty) {
28         Choose a wire $w_a$ from alt_array;
29         Add (net, $w_a$);     /* Add $w_a$ into net. */
30         Remove (net, $w_t$); /* Remove $w_t$ from net. */
31         return TRUE;
32    }
33    else return FALSE;
End
```

Figure 4.5: Implementation of Implication-tree Based Logic Transformation

4.2 Destination Node Implication

4.2.1 Introduction

In this section, we focus on the add-first alternative wiring procedures. The procedures are able to further improve the efficiency of the technique in Section 4.1[LWB00]. Based on the definitions (A1,A2) in previous sections, the procedure of locating alternative wires of a target wire is simply the addition of a candidate alternative wire between any two nodes in a circuit, SMA generation by direct implication, and inconsistence checking. It is necessary for the SMA generation and inconsistency checking, being very time-consuming, to be carried out selectively for efficiency reason. The trivial approach to accelerate alternative wire identification is based on the properties of alternative wire. The following lemma presents one of the properties as an example.

Lemma 2 If a wire $w_a < S_a, D_a, P_a >$ is an alternative wire of target wire $w_t < S_t, D_t, P_t >$, for the test of $(w_t|s{-}a{-}x)$ fault, the MA of S_a should be assigned such that $\beta(< S_a, D_a, P_a >) = cv(D_a)$.

> *Proof* From $A1.$, the addition of $w_a < S_a, D_a, P_a >$ should change the mandatory assignments for $w_t|s{-}a{-}x$. The only way to change a node's value, if it can be changed, is the addition of an input to the node with a controlling value. □

Since acceleration can be made merely by source nodes implication relations, it is expected that further improvement can be made by developing the identification trimming by relationship among destination nodes. In this section, the second technique is presented. We first concentrate on the properties of the destination nodes of alternative wires for a target connection. Based on the properties, we develop theorems on the relationship between alternative connections with the same source node but different destination nodes, in order to accelerate the alternative wiring process.

4.2.2 Destination Node Relationship

It is observed that the selection of destination node based on $R1$ and $R2$ in the previous section is not efficient for large circuits, since the number of forced nodes of a target wire is still large. An efficient procedure to select destination node further improves the alternative wiring algorithm. Our solution to this problem is to introduce a new method to select destination nodes of a target wire. Before describing our algorithm in detail, some theorems are derived.

Theorem 3 All FPMA of the target wire are not involved in the inconsistency of the SMA of the target wire.

> *Proof* The inconsistent SMA of a wire is that a logic conflict occurs between the implication from its inputs and output. For FPMA, the assigned value follows the fault propagation path and all FPMA is driven by their input. So there is no inconsistency involving FPMA. □

By Theorem 3, the derivation of the following corollary is trivial.

Corollary 4 For ATPG-based redundancy check, it is not necessary to include FPMA in the checking of inconsistent SMA.

We continue to discuss the relationship between different candidate destination nodes with the same source node in constructing an alternative wire. For the candidate alternative wires with the same source node but different destination nodes, their DMA are the same. As a result, we merely concentrate on the OMA, which is implied by the destination node.

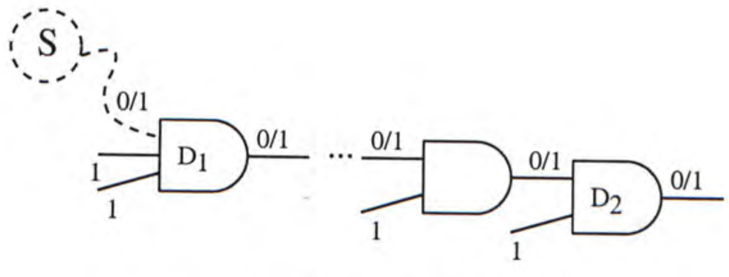
Theorem 5 For a target wire, the OMA of a candidate destination D_1 node is the same or stricter than that of another candidate destination node D_2 if D_2 is the transitive fanout of D_1, excluding nodes on the path from D_1 to D_2.

Proof As mentioned in $R1$, all candidate destination nodes are forced node of the target wire. For any two forced node D_1 and D_2 which are candidate destination nodes with D_1 is the transitive fanin of D_2, the OMA of all fanout gates of D_2 driven by both D_1 and D_2 should be the same since all side-inputs are set to non-controlling value in the fault propagation path. However, for the backward implication of D_2, if the inputs of D_2 can be implied, they should be set to non-controlling value. Since for backward implication, the "implying value" should be non-controlled value and the "implied value" should be the non-controlling value. Similarly, if backward implication can be done in all gates between D_1 and D_2, all side-inputs should be set to non-controlling values. As a result, excluding the path from D_1 to D_2, the OMA implied by the destination node D_1 should be stricter than that of D_2 in most cases. The OMA implied by destination node of D_1 and D_2 are the same only when all gates between D_1 and D_2 are involved in backward implication. □

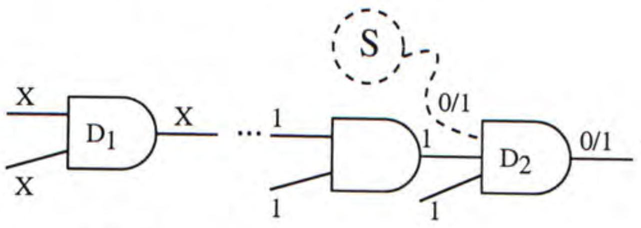
Example of Theorem 5 is shown in Figure 4.6. Figure 4.6(a) demonstrates a subset of OMA $< S, D_1, 0 >$ and (b) for a subset of OMA $< S, D_2, 0 >$. It can be observed in the figures that the OMA $< S, D_1, 0 >$ is stricter than OMA $< S, D_2, 0 >$.

Based on Theorem 5, we conclude the relationship of the destination nodes selection for alternative wires.

Theorem 6 For two new connections $< S, D_1, P_1 >$ and $< S, D_2, P_2 >$ with D_1 being in the transitive fanin of D_2, if $< S, D_1, P_1 >$ is irredundant, $< S, D_2, P_2 >$ is also irredundant, where S is an internal node, D_1 and D_2 are internal nodes satisfying the condition in $R1$ and $R2$, P_1 and P_2 are the polarities that make the corresponding connection the controlling values of D_1 and



(a) Subset of OMA of <S,D$_1$,0>



(b) Subset of OMA of <S,D$_2$,0>

Figure 4.6: An example of Theorem 5

D_2 respectively.

> *Proof* For the two connections, $< S, D_1, P_1 >$ and $< S, D_2, P_2 >$, having the same source node, their DMA should be the same. However, from Theorem 5, the OMA of the former connection is stricter than or equal to the latter one excluding nodes on the path from D_1 to D_2. So if the former connection is irredundant, there is no conflict in corresponding SMA for both connections. Thus, the latter connection should be irredundant. □

Corollary 7 For the same condition stated in Theorem 6, if $< S, D_2, P_2 >$ is redundant , $< S, D_1, P_1 >$ should be also redundant.

> *Proof* The contrapositive of Theorem 6. □

An example illustrating Corollary 7 is shown in Figure 4.7. The circuit demonstrates the addition of a new connection to the circuit in Figure 1.2(a).

According to Figure 1.2(b), the new connection can be added after g_6. However, when applying Corollary 7, the new connection can also be added before g_6 and the addition is concluded in Figure 4.7.

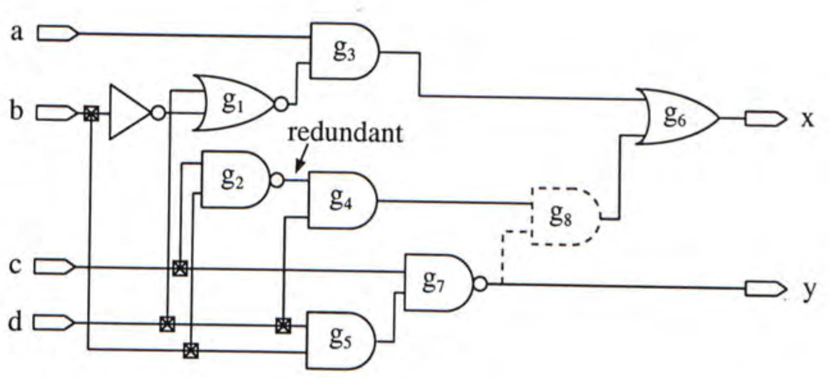


Figure 4.7: An example of Corollary 7

Based on Theorem 6, a large number of alternative wire checking can be skipped. When a candidate wire with a destination node D is identified to be irredundant, all connections with forced nodes which is in the transitive fanout of D and the same source node can be skipped from checking. The complete procedure to identify alternative wires with the technique is presented in the next sections.

4.2.3 Destination Node Implication-tree

Based on Theorem 6, we can build a destination node implication-tree to store all relationships among candidate destination nodes, with forced mandatory assignments. For nodes with FPMA, we first set the dominator which is closest to PO to be the root of the tree, while for other dominators, if dominator D_i is in the fanin of dominator D_j, D_i is set to be the child of D_j. After that, we assign all other nodes with MA obtained by backward implication to be the children of their fanin nodes.

An example is shown in Figure 4.8 and Figure 4.9. Assuming that all nodes in the Figure 4.8, a subset of a circuit, are forced nodes, the corresponding destination node implication-tree is shown in Figure 4.9.

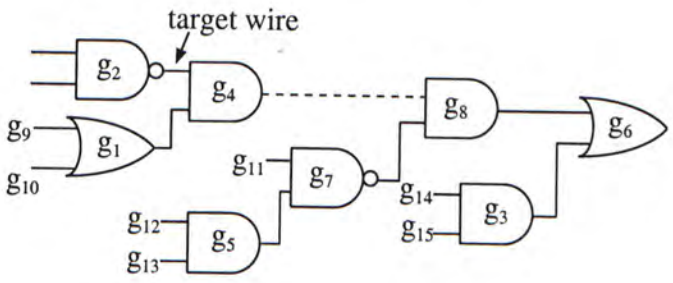


Figure 4.8: A sub-circuit

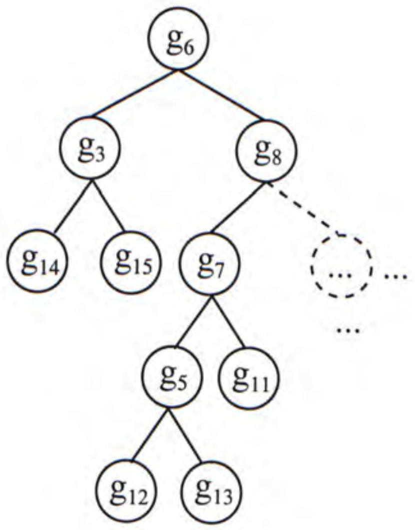


Figure 4.9: A destination node implication-tree

Destination node selection can be made from the implication-tree. According to Theorem 6, we conclude that if a node in the implication-tree is identified to be infeasible, all its ancestors should also be infeasible. In other words, we can skip all the redundancy checks of alternative wires associating with the ancestor nodes.

4.2.4 Selection of Alternative Wire

The selection of appropriate candidate alternative wires is crucial to the acceleration of the whole process of alternative wiring. Since the basic implementation of alternative wiring transformation is described in Figure 4.3, 4.4 and 4.5, only the alternative wire selection procedure of a target wire w_t, which is based on destination node relationship, is briefly described as the pseudo-code in Figure 4.10.

In the alternative wiring identification procedure of a target wire w_t, a candidate source node is selected according to the SMA of w_t. When all potential nodes are checked, the procedure would restart with the next target wire. For the source node selection, we can choose any algorithm to accelerate the process on condition that it is independent of the destination node selection.

For a selected candidate source node, candidate alternative wires can be formed with candidate destination node of the target wires. So in our implementation, the destination nodes are first stored in the destination node implication-tree. Then, destination nodes are obtained and a redundancy check is performed for the candidate alternative connection in the inner while loop. If the connection is identified to be redundant, it is an alternative wire and is stored for further use, for example, when selecting suitable alternative wires for optimization.

As indicated by Theorem 6, a destination node is skipped if any of its descendants is infeasible. In the function *select_dest_node*, we select destination node according to the implication-tree, last traversed node and the result of last redundancy check. At the first time, the last traversed node is $NULL$, the function returns a leaf node of the implication-tree. If the last node is not $NULL$, the function will switch the node's flag based on the result of last redundancy check. For example, if the last identification returns FALSE, the flags of all ancestors of the last traversed node would be set such that all of

```
find_all_candidate_AW ($w_t$)
$w_t$ is the target wire;
Begin
    dest_impl_tree = get_dest_node ($w_t$);
    /* the source node selection loop
     * can be different depends on which
     * algorithm is adopted
     */
    S = select_source_node ($w_t$);
    D = NULL;
    while (S != NULL){
        while((D = select_dest_node (dest_impl_tree,R,D)) != NULL ){
            /* dest. node of candidate wire */
            insert_buffer_gate (D);
            P = find_polarity ($w_t$,S,D);
            if (is_redundant (S,D,P)){
                store_candidate_AW (S,D,P);
                R = TRUE;
            }
            else{
                R = FALSE;
                remove_buffer_gate (S);
            }
        }
        S = select_source_node ($w_t$);
    }
End

/* select destination node from the destination node
 * implication-tree, according to the last redundancy check result
 */
select_dest_node (dest_impl_tree,R,D)
R = TRUE,FALSE, result of last redundancy check
D = last traversed node
Begin
if (D == NULL){
    D = get_a_leaf_node($v_R$); /*$v_R$ is the root node */
}
else{
    if (R == TRUE){
        node_set_flag(D,TRUE);
    }
    else{
        all_ancestors_set_flag(D,FALSE);
    }
    D = get_next_sibling(D);
    if (D == NULL){
        D = get_parent_sibling(D);
    }
    D = get_a_leaf_node(D);
}
return D;
}
End
```

Figure 4.10: Implementation based on destination node identification

them will be skipped. Then it will try to get its next sibling or, if no sibling exists, try to get its parent or its parent's next sibling. At this step, if a node is obtained, its descendant leaf node will then be found and returned. When no sibling or parent exists, the function will return a NULL value and the inner while loop will break and another source node would be selected.

4.3 The Algorithm

Our algorithm, the Implication Based Alternative Wiring Logic Transformation (IBAW), is an integration of the source node and destination node implication relationship techniques. They work independently and are complementary to each other as they both focus on different parts of alternative wires when trimming unnecessary redundancy identifications. As a result, IBAW speed up the alternative wiring process significantly.

4.3.1 IBAW Implementation

The core implementation of IBAW is presented in Figure 4.11. Only the logic transformation part is shown for the framework of IBAW is similar to the pseudo-code shown in Figure 4.4. The implementation is simple and clear since it is an integration of Figure 4.5 and Figure 4.10.

4.3.2 Experimental Results

In order to illustrate the performance of our algorithm, we compared IBAW with the original RAMBO[CE93]. Both algorithms are implemented to locate as many alternative connections as possible for all wires in the benchmark circuits. All experiments are performed on Sun Enterprise 4500 machines. Results are listed in Table 4.1. In the table, the columns display the CPU-time usage for running RAMBO and IBAW. From the results, we know that the CPU-time usage of our implementation is only 14.7% of that of RAMBO. The most encouraging finding is that the acceleration of IBAW increases as the size of the circuit increases, which can be observed from the ratio between RAMBO and IBAW in larger circuits such as $C3540$ and *too_large*.

Circuits	RAMBO	IBAW	Ratio
5xp1	24.89	6.75	3.69
C1355	82.3	14.36	5.73
C1908	247.27	32.33	7.65
C2670	328.08	97.48	3.37
C3540	6886.63	736.9	9.35
C432	65.05	18.97	3.43
C499	82.61	14.41	5.73
C5315	477.69	110.22	4.33
C6288	1016.82	311.01	3.27
C7552	992.71	261.03	3.80
C880	58.8	21.55	2.73
alu2	945.96	142.91	6.62
alu4	4878.6	591.61	8.25
apex6	286.85	64.03	4.48
b9_n2	5.84	1.64	3.56
comp	15.98	4.94	3.23
duke2	893.92	109.21	8.19
f51m	34.1	8.3	4.11
misex3	1042.42	108.39	9.62
pcler8	7.79	1.49	5.23
rot	100.43	27.95	3.59
sao2	82.06	21.82	3.76
term1	53.27	14.03	3.80
too_large	602.78	79.26	7.61
ttt2	36.45	9.93	3.67
x3	150.71	34.35	4.39
Total	19400.01	2844.87	6.82
Relative	1	0.147	

Table 4.1: Comparison between RAMBO, IBAW in CPU time usage

4.4 Conclusion

Based on implication relationship among nodes in alternative wiring, we propose a novel algorithm, Implication Based Alternative Wiring Logic Transformation (IBAW), to accelerate the ATPG based alternative wiring logic transformation algorithms. IBAW demonstrates its high efficiency in our experimental results by running 6.7 times faster than the original RAMBO in exhaustively finding alternative wires in a circuit. To conclude, IBAW is a fast and general-purpose alternative wiring logic transformation tool which is applicable to many other EDA problems.

```
IBAW-transform(net, $w_t$)
net is the network under consideration;
$w_t$ is the target-wire;
Begin
    if ($w_t$ is redundant) {
        /* Redundancy checking is called for $w_t$. */
        Remove (net, $w_t$);      /* remove $w_t$ from net. */
        return TRUE;
    }
    $v_R$ = Generate-implication-tree(net, $w_t$);
    /* $v_R$ is the root of implication-tree. */
    dest_impl_tree = get_dest_node ($w_t$);
    X = NULL;
    while((X = select_dest_node (dest_impl_tree,R,X)) != NULL ){
        D = Insert-buffer-node-after (X);
        $P_1$ = NULL; /* $P_1$ is a global node pointer. */
        S = Select-a-node-from-implication-tree (TRUE);
        while (S != $v_R$) {
            Determine P;
            /* the polarity of the candidate alt. wire */
            $w_a$ = <S, D, P>; /* candidate alt. wire */
            if ($w_a$ is redundant) {
                /* $w_a$ is a feasible alt. wire. */
                Put $w_a$ into alt_array;
                $R_1$ = TRUE;
            }
            else $R_1$ = FALSE;
            S = Select-a-node-from-implication-tree ($R_1$);
        }
        Delete-buffer-node (D);
        if (alt_array is not empty)
            R = TRUE;
        else
            R = FALSE;
    }
    if (alt_array is not empty) {
        Choose a wire $w_a$ from alt_array;
        Add (net, $w_a$);     /* Add $w_a$ into net. */
        Remove (net, $w_t$); /* Remove $w_t$ from net. */
        return TRUE;
    }
    else return FALSE;
End
```

Figure 4.11: Implementation of IBAW

Chapter 5

Graph Based Alternative Wiring Logic Transformation

5.1 Introduction

In [WLF00], Graph-Based Alternative Wiring (GBAW), a new graph-based technique to identify alternative wires, was first proposed. It first models a circuit network as a directed acyclic graph (DAG). Then, alternative wires are identified by performing graph pattern matching between local sub-graphs of the network and the pre-specified minimal sub-graph configurations which contain alternative wires within a given range limit. Experiments show that the number of all such local minimal sub-graphs is limited and most of the alternative wires are located topologically "near" to their target wires. The paper states that about 96% of the alternative wires are only of 2-edge-distance from their target wires.

GBAW is proved to produce a competitive results in finding alternative wires when compared with RAMBO. Not only does GBAW perform well in searching alternative wires, but it also runs very fast. The paper [WLF00] shows that the CPU running time of GBAW is, on average, just 1.4% of that of RAMBO. Due to its significantly short running time, GBAW is potentially considerable to be a different and better technique for identifying alternative wires. The efficiency of GBAW is gained mainly due to its avoidance of running the CPU-expensive Boolean implications. In addition to the existence of common alternative wire patterns which repeatedly occur in the same circuit, the one-time analysis effort of pattern based rewiring scheme is made practical.

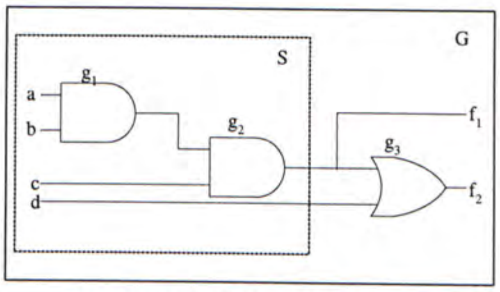
It is also observed that GBAW is unable to find all 2-edge-distant patterns which can be found by ATPG-based alternative wiring, if GBAW is limited to search merely the small pre-defined set of minimal configurations like the early version shown in [WLF00]. In this section, we present a much extended GBAW scheme, mainly with more 2-local patterns, to improve the effectiveness of GBAW significantly. We also define the concept of pattern cluster in order to keep GBAW simple while expanding the pattern library. With the refined GBAW, we achieve an encouraging result in identifying 2-local alternative wires.

5.2 Notations and Definitions

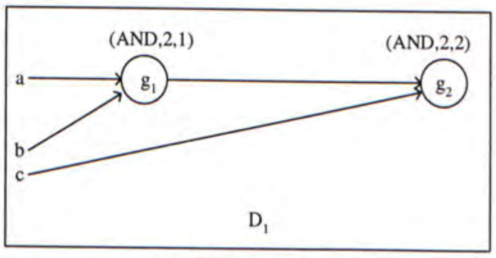
A Boolean network can be modeled as a directed acyclic graph (DAG). In a Boolean network G, the in-degree of node y, denoted by $d^-(y)$, is defined as the number of edges entering y. The out-degree of node y, denoted by $d^+(y)$, is defined as the number of edges leaving y. We define a node y by a triplet $(op, d^-(y), d^+(y))$, where op is the Boolean operator of y which can be AND, OR, NAND,or NOR.

A wire is replaceable if and only if it has at least one alternative wire. We use a graph configuration D to map the logic function from a Boolean Network G. For each node n_i in sub-network S in network G, n_i is mapped to a triplet (op, i_1, i_2) in D where op denotes the operator representing the boolean function of n_i and i_1, i_2 are non-negative integers. All edges in S are preserved, while the edges outside S are omitted in D. In most cases, i_1 equals $d^-(n_i)$ and i_2 equals $d^+(n_i)$. The element of a triplet $(op, d^-(y), d^+(y))$

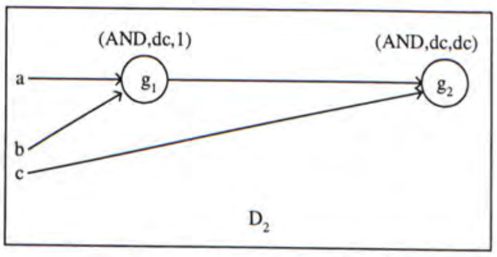
can also be don't care, *dc*. For the first element, *dc* means any operator. For the other elements, *dc* can be any positive integer. We use a configuration to denote a minimal pattern containing both the target wire and its alternative wires.



(a) Boolean network *G* and its sub-networks



(b) A configuration of *S*, D_1



(c) Another configuration of *S*, D_2

Figure 5.1: Configuration of a sub-network

The mapping is illustrated in Figure 5.1. S is a sub-network of G. D_1 and D_2 are two mappable configurations of S. The main difference between D_1 and D_2 is that the nodes in D_1 have definite number of fanins and fanouts while those nodes in D_2 can have any number of fanins and fanouts.

The k-local pattern is defined as the alternative wire pattern with the distance between the alternative wire and the target wire being k. The distance between two wires is the difference of maximum path length from any primary input to the destination nodes of the wires.

A pattern P covers an alternative wire pair (w_t, w_a) if the wire pair matches with P in the graph matching process of GBAW. In same senses, the cover set of pattern P is defined as all alternative pairs which are covered by P.

Figure 5.2 shows a sample of alternative wire patterns. The figure suggests that adding connection $a \rightarrow g_3$ and removing connection $a \rightarrow g_1$ is a feasible alternative wire transformation. However, for each pattern, it is always possible to match two types of alternative wire pairs: the forward alternative wire pairs and the backward alternative wire pairs. In Figure 5.2, the replacement of $a \rightarrow g_1$ with $a \rightarrow g_3$ is an example of forward alternative wires while the replacement of $a \rightarrow g_3$ with $a \rightarrow g_1$ refers to a backward alternative wire pair.

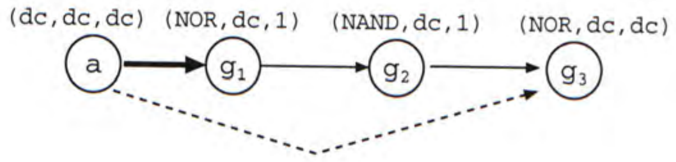


Figure 5.2: A sample alternative wire pattern

For pattern matching alternative wire transformation, if connection w_a replaces w_b, we define w_a as the addition wire and w_b as the removal wire. So for the pattern shown in Figure 5.2, connection $a \rightarrow g_3$ can be the addition wire or the removal wire, depending on the actual pattern matching.

5.3 Alternative Wire Patterns

In this section, the core of GBAW, alternative wiring patterns, is presented. In practice, the alternative wires are not too far away from the target wire. The paper [WLF00] states that about 96% of alternative wires can be found within 2-edge-distance. The original 0-local, 1-local and 2-local patterns [WLF00] can be found in Appendix B. The new patterns in this section are intended to help GBAW to identify 2-local alternative wires that the original GBAW is unable to locate. Proved by our experiment, the set of new patterns is capable of greatly enhancing GBAW in locating alternative wire.

Before introducing the new patterns, we first define the complete set of alternative wire patterns as the pattern family F and each of the member in the pattern family set as pattern member P. Besides, in order to analyze the patterns systematically, we introduce another terminology: the pattern cluster. Pattern cluster C is defined as a subset of F which contains more than one pattern member P. These members in the same pattern cluster should have the same topological order, but the operator op in each of the nodes can be different. In the following, we present some examples of the pattern clusters which we have implemented. The explanation and verification of partial implemented patterns are included in Appendix C for illustration.

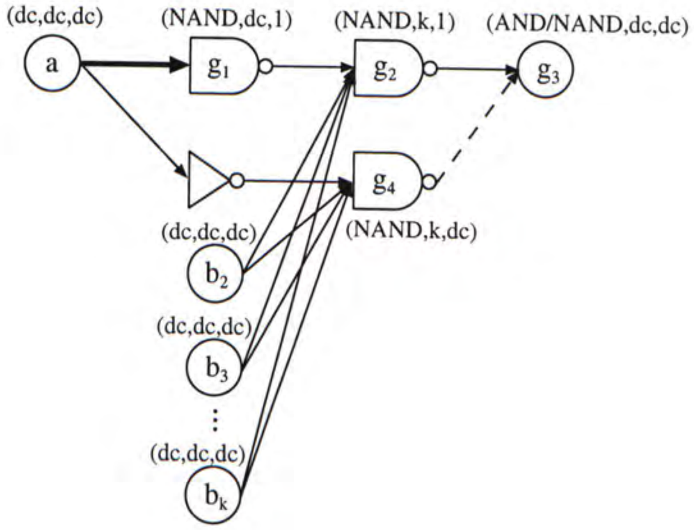


Figure 5.3: One member of Pattern Cluster C_1

One member in the first pattern cluster C_1 is shown in Figure 5.3. In the pattern member, except one fanin node, g_2 and g_4 should share the same set of fanin nodes b. Thus, after the alternative wire transformation, the signals of nodes b can be operated with a before they reach g_3. A detailed proof is shown in Appendix C.1. In fact, the pattern cluster is related to the case 2-2 of local-2 patterns shown in [WLF00]. The pattern members can have different operators. For example, the operator op_1 can be AND, OR, $NAND$, or NOR.

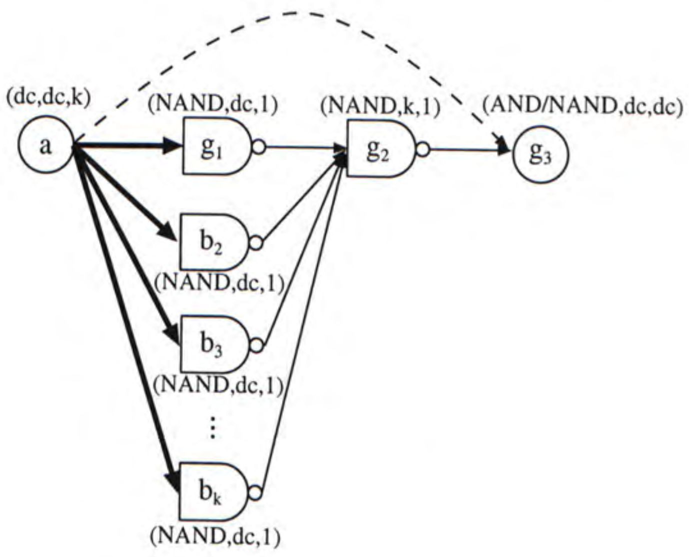


Figure 5.4: One member of Pattern Cluster C_2

For the second pattern cluster C_2, one of the pattern members is shown in Figure 5.4. In the pattern, it is obvious that the signal from a converges to g_2 and g_3. Therefore, we can replace all the corresponding fanouts of a with a direct connection to g_3. Actually, the pattern cluster is a generalization of fanout reconvergence.

Figure 5.5 demonstrates the third pattern cluster C_3. This pattern is different from the others since it is derived from the consensus property of Boolean Logic, which is shown in the following identities.

$$ab + a'c + bc \equiv ab + a'c$$

One member of the fourth pattern cluster C_4 is shown in Figure 5.6. This

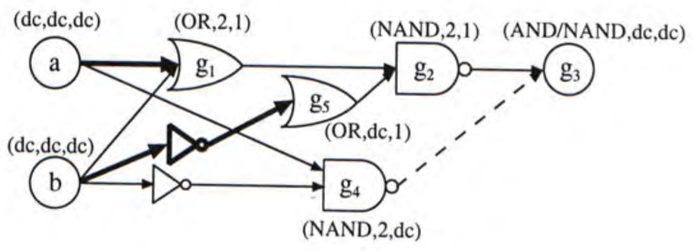


Figure 5.5: One member of Pattern Cluster C_3

pattern cluster is obtained by analyzing the result from an ATPG-based alternative wiring package, RAMBO. The pattern topology is more complicated than other that of pattern clusters.

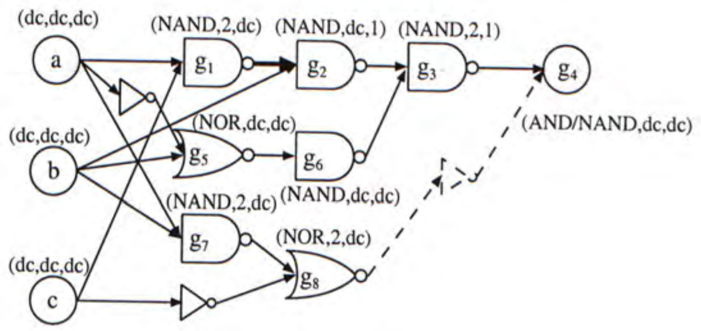


Figure 5.6: One member of Pattern Cluster C_4

For the pattern cluster C_5, an example is shown in Figure 5.7. This pattern cluster is constructed by the pattern extraction method from pattern cluster C_4. C_5 looks like pattern cluster C_2 but the topologies are different.

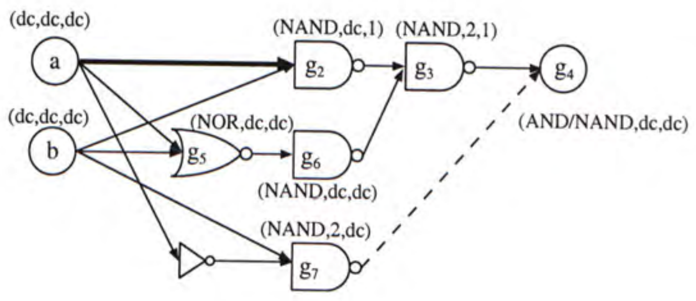


Figure 5.7: One member of Pattern Cluster C_5

5.4 Construction of Minimal Patterns

In this section, some approaches for constructing minimal patterns are discussed with several examples. Before the discussion, the minimality of patterns is explained in detail. The concept of minimality is extremely important because GBAW would be ineffective if an alternative wire pair could be matched with two different patterns. To avoid duplication of matching, all patterns should have the property of minimality.

5.4.1 Minimality of Patterns

In the process of pattern construction, the most significant concept is the "minimality" because it is always possible to encounter problems such as:

- Is the newly constructed pattern the same as any known pattern?
- Can two different patterns cover the same alternative wire pair?

To deal with these problems, it is necessary to explore the minimality, which is one of the pattern properties. Actually, the concept of minimal patterns involves the following requirements:

- For any two minimal patterns, the set of alternative wires they covered should be mutually exclusive.

- All elements (nodes and connections) in a minimal graph configuration should be vital for the pattern to maintain its correctness, i.e., no redundant elements exist in the pattern.

- For a minimal pattern with fixed topology, the cover set should be maximized.

We take the minimal pattern shown in Figure 5.2 as an example. Figure 5.8 shows a pattern by adding the node *b* in Figure 5.2. It is obviously not a minimal pattern since it consists of redundant element *b*. In fact, all alternative wires covered by pattern in Figure 5.8 are covered by the pattern in Figure 5.2.

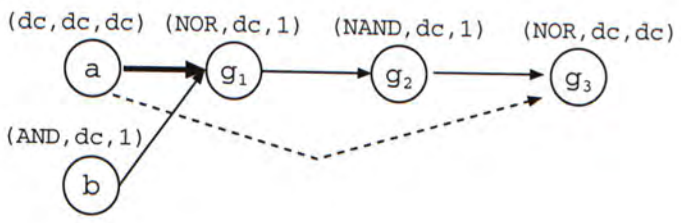


Figure 5.8: An example of minimality requirement

Another example is shown in Figure 5.9. Obviously, it is not a minimal pattern since its cover set is not maximum. For the corresponding minimal pattern, the in-degree should be of value *dc* so that it not only covers the cover set of pattern in Figure 5.9, but also covers more alternative wires in the same topology.

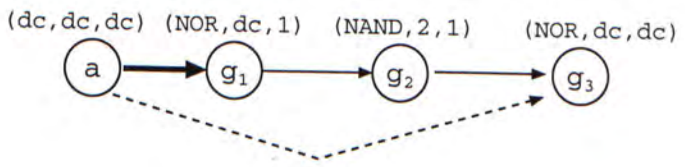


Figure 5.9: Another example of minimality requirement

Under these considerations, we have the following definition of minimal pattern.

Definition 3 (Minimal Pattern)

An alternative wire pattern P is minimal if and only if, with the constraint that the cover set of P cannot be reduced,

- the in-degree x_i, out-degree y_i and the operator set op_i of each nodes (op_i, x_i, y_i) in P is maximum,
- the number of nodes in P is minimum, and
- the number of connection in P is minimum,

where dc can be treated as the "largest" value during comparison.

5.4.2 Minimal Pattern Formation

Since the old version of GBAW locates limited number of 2-local patterns, in our work, we concentrate on concluding 2-local patterns. One effective approach is to extract minimal configurations from experimental results of some ATPG-based alternative wiring logic transformation algorithms. In this section, the steps of minimal pattern construction are explained.

For the ATPG-based alternative wiring logic transformation algorithm, our implementation is based on the RAMBO algorithm in [CE93]. In order to locate alternative pattern configurations, we compare the network transformed by RAMBO and GBAW [WLF00]. After the comparison, we have extracted a sub-circuit in which the transformed circuit of RAMBO differs from that of GBAW.

One of the extracted sub-circuit is depicted in Figure 5.10 and Figure 5.11 as an example. Figure 5.10 is a sub-network of the pre-transformed network and Figure 5.11 is the corresponding sub-network after the transformation by our implementation of RAMBO.

Direct conversion from the sub-networks to the graph configuration is shown in Figure 5.12. All "primary inputs" in the sub-network are converted

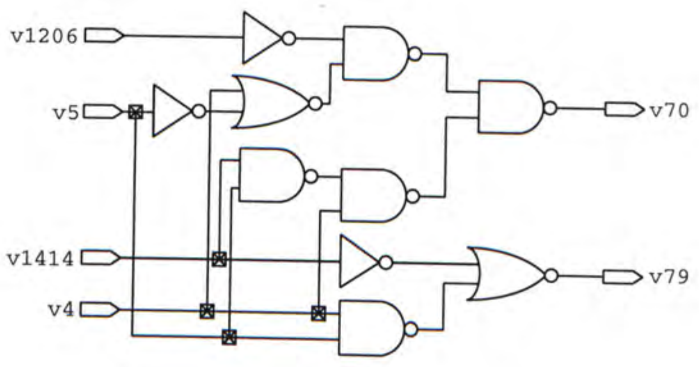


Figure 5.10: A sub-network S

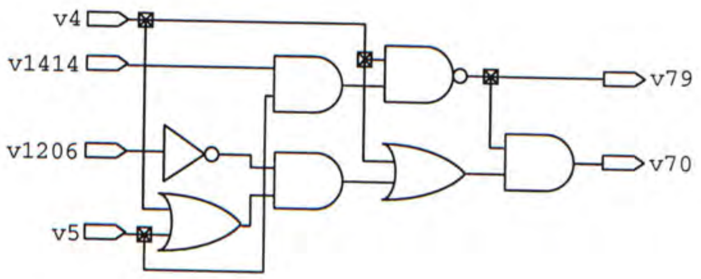


Figure 5.11: The sub-network S transformed by an ATPG-based alternative wiring algorithm

to nodes with triplets (dc, dc, dc), since there is no restrictions to the nodes. Similarly, the out-degree of "primary outputs" should be dc. At the same time, since gate $t1206$ has only one fanout to an AND gate, they are combined together to be the node g_6.

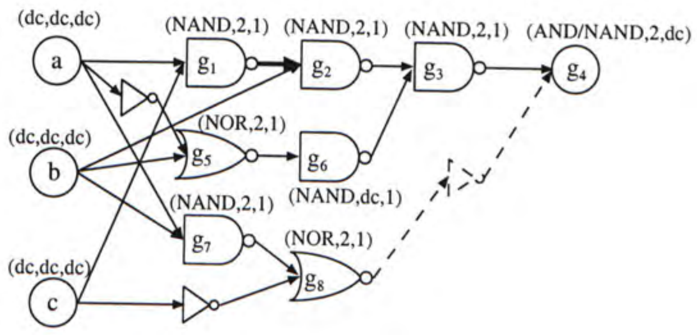


Figure 5.12: The resultant pattern configuration - Step 1

It is obvious that the graph configuration in Figure 5.12 is not a minimal pattern. Although both the number of nodes and the number of connections are minimum in this pattern (it can be easily proved by exhaustive search), the number of alternative wire pairs which it covers would increase when the in-degree or out-degree of some nodes increase. In other words, the values of in-degrees and out-degrees are not maximized in the example.

Hence, the next step is the attempt of adding fanin nodes (dc, dc, dc) to all nodes except those with dc value of in-degree. This is to ensure that the in-degrees of all nodes are minimum under the condition that the cover set does not shrink. For the addition of (dc, dc, dc) nodes, an example is shown in Figure 5.13.

After the trial addition of (dc, dc, dc) nodes, we should verify the correctness of the pattern and eliminate some improper nodes addition. However, it will be more effective if we eliminate some improper nodes before verification. From the observations, the dc node X cannot be added to the fanin of a node N when

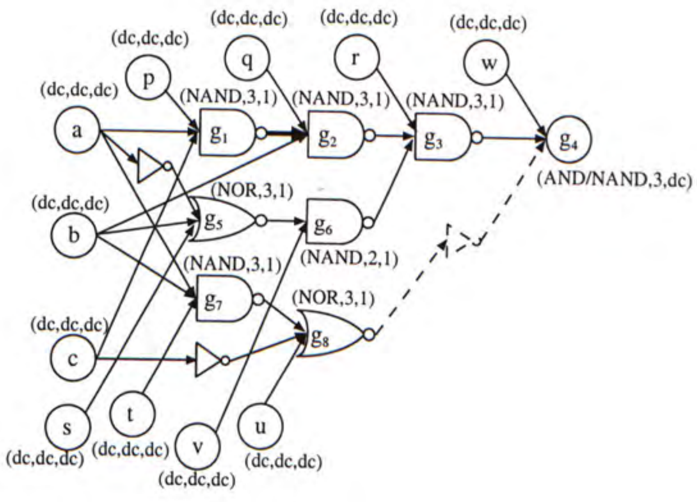


Figure 5.13: Example - Step 2

- N is a transitive fanin of the target wire but at the same time X is not a transitive fanin of the alternative wire.

- N is a transitive fanin of the alternative wire but at the same time X is not a transitive fanin of the target wire.

The two rules are obvious since the signal from X will be blocked after the logic transformation. The rules suggest that the addition of (dc, dc, dc) node is improper when the signal only passes through target wire or alternative wire.

According to the rules, we know that the (dc, dc, dc) nodes p, t, u in Figure 5.13 can be eliminated without performing verification. The resultant graph configuration is shown in Figure 5.14. And the verification details are demonstrated in Table 5.1.

Since the graph configuration is shown to be incorrect for g_4 during the verification, we need to eliminate some nodes for the correctness. As shown in the verification, it is obvious that the node r should be removed. After the removal of r, the pattern is correct for matching with alternative wire pairs. The resultant pattern is shown in Figure 5.15.

The final step includes the combination of (dc, dc, dc) nodes with their

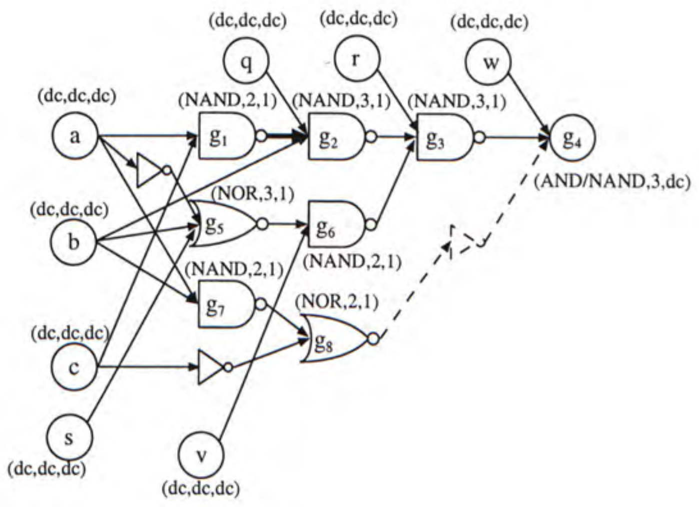


Figure 5.14: Example - Step 3

Verification:

Node	Before transformation	After transformation
g_1	$(ac)'$	$(ac)'$
g_5	$(a'+b+s)'$	$(a'+b+s)'$
g_7	$(abt)'$	$(abt)'$
g_2	$[(ac)'bq]'$	$(bq)'$
g_6	$[(a'+b+s)'v]'$	$[(a'+b+s)'v]'$
g_8	$[(ab)'+c']'$	$[(ab)'+c']'$
g_3	$\{[(a'+b+s)'v]'r[(ac)'bq]'\}'$	$\{[(a'+b+s)'v]'r(bq)'\}'$
g_4	$\{[(a'+b+s)'v]'r[(ac)'bq]'\}'w$ $= [ab's'v'+r'+bq(a'+c')]w$	$\{[(a'+b+s)'v]'r(bq)'\}'w[(ab)'+c']$ $= [ab's'v'+r'(a'+b'+c')+bq(a'+c')]w$

Table 5.1: Verification of the pattern in Figure 5.14

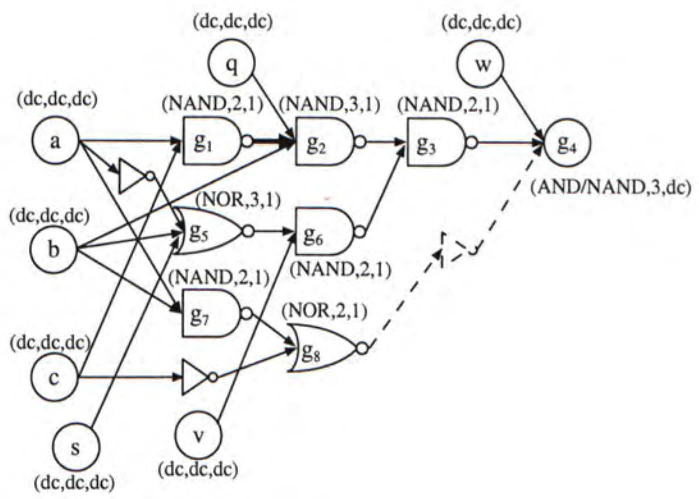


Figure 5.15: Example - Step 4

fanouts and the maximization of out-degree for all nodes. To maximize the fanout number, it is necessary to follow a simple rule.

- The out-degree of nodes cannot be increased when the node is a transitive fanout of target wire or alternative wire.

The reason behind this rule is simple: Only those signals to the transitive fanouts of the addition and removal wires will be changed throughout the alternative wiring transformation process.

After the combination and maximization of out-degree of nodes, a minimal graph configuration is obtained and shown in Figure 5.16.

5.4.3 Pattern Extraction

Practically, we can obtain more patterns by simplifying some known patterns. For a minimal pattern, if we eliminate any "primary input" from the graph configuration and maintain the correctness of the pattern, a different pattern will be obtained. The method is named Pattern Extraction.

Pattern Extraction method is explained by an example. Considering the pattern shown in Figure 5.16, if we eliminate the node c by putting its value

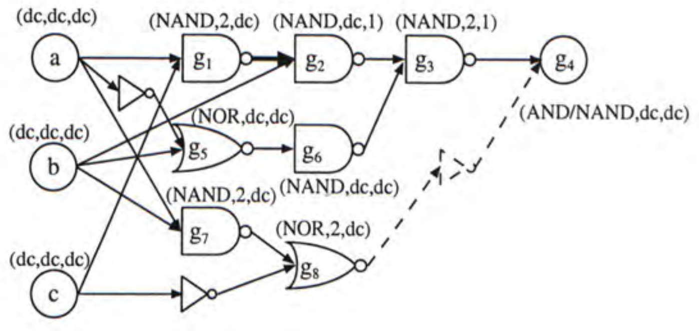


Figure 5.16: Example - minimal pattern

to 1, the graph configuration is still correct but it represents a different alternative wire pattern (the verification is shown in Appendix C.5). The newly extracted pattern is shown in Figure 5.17. The pattern is different from Figure 5.16 and their cover sets are mutually exclusive. If it is not the same as any known pattern in our pattern family, it can form a new pattern cluster with its variations.

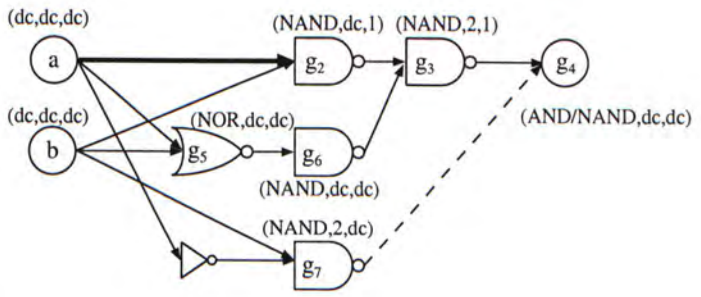


Figure 5.17: Minimal pattern extracted from Figure 5.16

5.5 Experimental Results

Table 5.2 shows the number of target wires which have 2-local alternative wires. The results are competitive with RAMBO (98%). For each target wire, it may have more than one alternative wire and some of them may be 2-local patterns. So the number of alternative wires is usually greater than the number of corresponding target wires. Besides, since RAMBO uses ATPG techniques to locate alternative wires, its search space is much larger than that of GBAW. Thus, RAMBO is able to find a larger number of alternative wires than our technique does. However, we are able to obtain a promising result since it is possible for GBAW to find backward alternative wire while the current RAMBO cannot.

We implemented the improved GBAW on Sun UltraSparc 5 workstation for MCNC benchmark circuits and results are shown in Table 5.3. The speed and capability of locating alternative wires between RAMBO and GBAW are also compared. In the table, it is shown that our improved GBAW is able to find 30% more alternative wires than RAMBO with only 1.38% CPU time on average.

5.6 Conclusion

In this chapter, an augmented Graph-Based Alternative Wiring (GBAW) scheme is presented. Although there are more than 40 patterns are included, an attractive efficiency is still maintained by using the concept of Pattern Cluster. GBAW has forward and backward search capability and can identify alternative wire efficiently. Experimental results showed that it is capable to find 30% more alternative wires comparing with the forward search RAMBO version. GBAW has a good coverage of alternative wires with 75 times speedup on average. When using GBAW as the perturbation engine and combining with

Name	RAMBO (target/alter.)	GBAW (target/alter.)	Searched (%) (target/.alter.)
5xp1	10/21	10/18	100/86
9sym-hdl	5/6	5/8	100/133
C1908	42/57	44/44	105/77
C2670	85/99	83/94	98/95
C3540	208/297	238/250	114/84
C432	33/44	40/40	121/91
C5315	76/113	69/73	91/65
C6288	3/17	3/18	100/106
C7552	132/219	76/82	58/37
C880	27/62	27/27	100/44
alu2	64/100	54/56	84/56
alu4	120/198	84/86	70/43
apex6	72/121	68/70	94/58
b9_n2	8/10	3/3	38/30
comp	28/44	17/17	61/39
des	671/907	795/795	118/88
duke2	35/64	29/29	83/45
misex3	50/167	41/41	82/25
rot	46/75	31/32	67/43
sao2-hdl	25/38	9/9	36/24
term1	41/77	27/28	66/36
ttt2	28/68	10/13	36/19
x3	71/82	76/76	107/93
Total	1880/2886	1839/1909	98/66

Table 5.2: 2-local pattern comparison between RAMBO and refined GBAW

SIS algebraic operations, there is a further reduction of 11.1% comparing with the result by algebraic operations alone.

Name	RAMBO alt. wires	RAMBO CPU	GBAW alt. wires	GBAW CPU
5xp1	36	10.17	62	0.24
9sym-hdl	27	1.56	40	0.16
C1355	185	12.82	250	0.89
C1908	127	33.52	240	0.68
C2670	267	83.57	344	1.33
C3540	569	273.80	816	2.15
C432	129	10.26	188	0.37
C499	16	6.05	34	0.6
C5315	511	155.91	713	2.88
C6288	1352	361.18	2191	4.18
C7552	1709	143.95	617	4.2
C880	151	9.86	239	0.66
alu2	169	214.71	263	0.84
alu4	333	270.50	493	1.61
apex6	239	34.32	377	1.23
b9_n2	48	1.65	71	0.17
comp	57	9.18	58	0.21
des	1468	729.92	2204	8.7
duke2	157	46.55	281	0.63
f51m	49	6.19	65	0.25
misex	216	124.48	439	0.97
my_adder	46	1.16	0	0.23
pcler8	29	1.3	30	0.12
rot	243	48.04	406	1.1
sao2-hdl	104	16.86	153	0.39
term1	106	16.81	169	0.37
ttt2	68	9.68	133	0.34
x3	228	23.13	348	1.2
Total	8639	2657.13	11224	36.7
Normalized	1	1	1.2992	0.0138

Table 5.3: Comparison between RAMBO and GBAW

Chapter 6

Logic Optimization by GBAW

6.1 Introduction

The Graph-Based Alternative Wiring (GBAW) algorithm, which has been applied in logic optimization, was first published in [WLF00]. The authors simply used GBAW to perform some random network perturbation and invoked SIS [SsLea92] *script.algebraic* script for logic minimization. The perturbation and simplification processes were iterated for several times and the best results (smallest in circuit area) were chosen.

However, the logic optimization approach in [WLF00] is not as good as the REWIRE algorithm [CGLMS99]. The reasons are listed below.

- The standard SIS minimization script *script, algebraic* is comparatively ineffective in logic minimization when comparing with other tools and scripts since it only adopt simple algebraic substitutions.

- Applying GBAW to perform random perturbation is undesirable since the circuit may become bigger or the internal don't care terms will be reduced after the perturbation.

In fact, GBAW can be applied effectively in logic optimization and incremental perturbation. Incremental perturbation refers to the logic transformation which aims at increasing internal don't cares and in turn increasing the number of possible alternative wires. In this chapter, a GBAW logic minimization algorithm and a GBAW incremental perturbation heuristic are presented. Afterwards, our algorithm in logic optimization is explained.

6.2 Logic Simplification

Logic minimization by alternative wiring logic transformation is possible since adding one logic element (usually a connection) to the Boolean Network may make more than one circuit element (connections or gates) become removable. In such situation, the circuit is simplified. In [CMSC96], by gathering a list of alternative wires information, a greedy circuit optimization algorithm is proposed based on this idea.

For GBAW, alternative wires are identified by pattern matching and each pattern possesses different features, so a different approach is needed for logic simplification.

In our approach, alternative wiring logic transformation by GBAW is basically divided into two types: the single-addition-multiple-removal and the single-addition-single-removal. The former refers to the alternative wiring process, which is able to remove more than one connection by adding only one connection. The latter transformation type refers to the removal of only one connection by adding one wire. In this section, these two types of transformations will be explained separately.

6.2.1 Single-Addition-Multiple-Removal by Pattern Feature

In our implemented alternative patterns, some of them belong to the type single-addition-multiple-removal, including both pattern cluster 2 and pattern cluster 3. All of them are able to remove more than one connection while one connection is merely added. In pattern cluster 2 (refer to Appendix C.2), all re-convergent fanout of a can be removed. In pattern cluster 3 (refer to Appendix C.3), two connections, $a \rightarrow g_1, b \rightarrow g_5$, can be removed.

6.2.2 Single-Addition-Multiple-Removal by Combination of Patterns

However, the transformations of most implemented patterns belong to the type single-addition-single-removal. For most cases, they are not able to decrease the total network area directly. In such situations, total circuit area can still be reduced when more than one minimal pattern share the same alternative wire.

Example is shown in Figure 6.1. In the sub-network, it is obvious that no single-addition-multiple-removal patterns matches with the circuit.

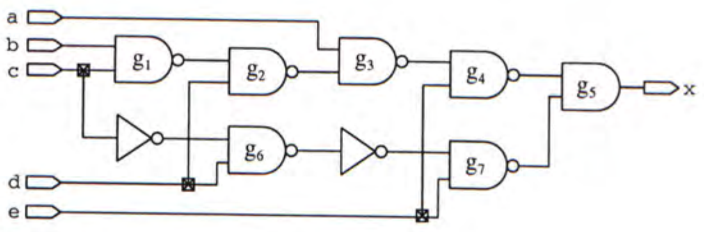


Figure 6.1: A sub-network

However, in Figure 6.1, the sub-network matches with two single-addition-single-removal patterns. The gates (g_1, g_2, g_3, g_6) match with the pattern member 1 in cluster C_1. At the same time, the gates $(g_3, g_4, g_5, g_6, g_7)$ match with the backward alternative wire pattern of member 1 in cluster C_1. It is depicted in Figure 6.2.

In this situation, if we perform the alternative wiring logic transformation for two matched sub-network simultaneously, we can remove the two wires by adding only one wire. The corresponding sub-network is simplified and shown

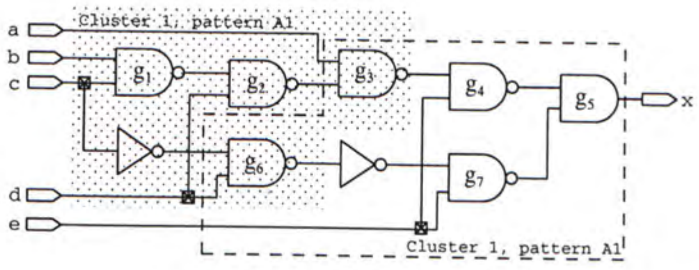


Figure 6.2: Simultaneous pattern matching

in Figure 6.3.

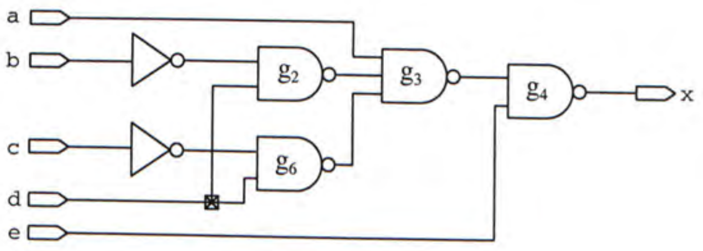


Figure 6.3: The corresponding sub-network after logic transformation

In the last example, the simultaneous pattern matching shows its ability in logic optimization. However, it is not always applicable for simultaneous alternative wiring transformation. The reason is that the transformation of pattern P_1 may change the logic structure of another originally matched pattern P_2. In the concept of ATPG-based alternative wiring, the transformation of P_1 may affect the implication of mandatory assignments for alternative wire pairs in pattern P_2. In order to ensure a simultaneous alternative wire transformation, we develop the following theorem.

Theorem 8 (Simultaneous alternative wire transformation)

The alternative wire transformation for two matched minimal patterns, P_1, P_2, where P_1, P_2 share the same addition wire, can be performed if and only if both of the following conditions are satisfied.

- P_1 does not contain the removal wire of P_2, and

- P_2 does not contain the removal wire of P_1.

  Proof From definition 3, all elements in a minimal pattern cannot be eliminated while the cover set is maintained not to be reduced. It implies that all elements are essential for the alternative wire pair to exist in the pattern. In the sense of ATPG-based alternative wiring, all gates and connections are vital to the implication of mandatory assignments of the corresponding connection faults. As a result, if the pattern does not contain the removal wire of another pattern, the mandatory assignments of both alternative wire pairs will be kept intact.

Based on Theorem 8, we can match the feasible simultaneous alternative wire patterns for logic optimization. First, the information of all matched patterns, the addition wires and removal wires are gathered. Then, we list and sort all feasible simultaneous alternative wire patterns in descending order of the total number of removable wires. After that, we are able to perform logic transformation according to the sorted list.

6.2.3 Single-Addition-Single-Removal

For the single-addition-single-removal alternative wire patterns which the simultaneous transformation technique cannot be applied, area can be reduced in a different way. We observed that the removal of one connection may cause the removal of one gate, and hopefully, some of its transitive fanins, when the gate has only one fanout.

Under this consideration, a matched alternative wire transformation ($S_a \to D_a$ replaces $S_t \to D_t$) will be performed when

- D_t has only two fanins.

- S_t has only one fanout.

In these two cases, at least one gate can in turn be removed after the removal of connection, $S_t \to D_t$. Hence, these conditions contribute to the logic simplification.

6.3 Incremental Perturbation Heuristic

Although some alternative wiring logic transformations would decrease the total network area, there are still a subset of transformations which do not alter the total area of the network. Such logic transformation refers to the circuit perturbation. However, logic perturbation can change the circuit structure and it is possible to increase the total number of feasible alternative wire pairs. We define this kind of logic perturbation to be the incremental perturbation.

Intuitively, the number of possible alternative wires in a network is related to the number of internal don't cares. Since in the view of ATPG-based alternative wiring, alternative wire pairs exist when the addition wire is initially a redundant wire, and afterwards the removal wire becomes a redundant wire.

In this section, we aim at proposing several heuristic considerations for applying alternative wiring transformation (incremental perturbation) to increase the potential number of alternative wire pairs.

In [CMSC96], it stated three conditions for a logic transformation ($S_a \to D_a$ replaces $S_t \to D_t$) to increase the internal don't cares in a network. The conditions are listed here.

- D_a should have fewer fanouts than D_t.
- D_a should be farther from PO than D_t.
- D_a should have more side inputs than D_t.

The underlying reason bases on the relation between internal don't cares and an informal term - the "observability". In [CMSC96], the term "observable" for a connection is intended to describe the difficulty of propagating the connection fault. So, the less observable a connection is, the more its corresponding internal don't cares are. For the three conditions above, all of them refer to the increase of difficulties in propagating the fault which makes the connections less observable.

The conditions do not absolutely guarantee the increase in internal don't cares since different logic structures such as fanout reconvergence would cancel the observability. However, they still serve as a good indicator to judge whether an alternative wire pattern should be transformed for incremental perturbation.

According to the conditions, we suggest an incremental perturbation heuristic to increase internal don't cares by alternative wiring transformation while the perturbation does not increase the network area. The following cost function quantifies the conditions and it is to represent the possibility of an alternative wiring transformation to increase the "observability".

For the addition wire $S_a \to D_a$ and removal wire $S_a \to D_a$, the cost function of the transformation is:

$$\text{Cost} = \alpha f + \beta l + \gamma s$$

where

$$\begin{aligned}
f &= \text{number of fanouts of } D_a - \text{number of fanouts of } D_t \\
l &= -(\text{minimum edge-distance from any PO node to } D_a \\
&\qquad - \text{minimum edge-distance from any PO node to } D_t) \\
s &= -(\text{number of side inputs of } D_a - \text{number of side inputs of } D_t)
\end{aligned}$$

and α, β, γ are constants to control the importance of each condition.

With the cost function, we propose a systematic procedure to perform incremental perturbation. Firstly, we list and sort all matched alternative wire patterns in ascending order of the cost function. Then, we perform all transformation according to the sorted order until all transformations with positive cost function are made.

6.4 GBAW Optimization Algorithm

The overall algorithm for logic optimization is depicted in Figure 6.4. First, the target network is simplified according to the approaches discussed in section 6.2. Then, the simplified network is perturbed in order to increase its internal don't cares according to section 6.3. These steps are repeated for a specific number of iterations. Finally, the network is further simplified before the end of the procedure.

6.5 Experimental Results

We implemented GBAW optimization algorithm on SIS platform. For the cost function of the incremental perturbation heuristic, we took ($\alpha = 1, \beta = 1, \gamma = 1$) in our experiment as it represents all three factors are equally weighted. For efficiency, our implementation run for only one iteration. All the experiments were performed on the MCNC combinational circuits. In the experiment, the circuit area and CPU-time of our optimization algorithm, which were compared to standard SIS scripts including *script.algebraic*, *script.boolean* and *script.rugged*, were recorded. Before running GBAW circuit optimization program, we pre-processed the circuits by mapping them with a library *mcnc*1*.genlib* which is a subset of *mcnc.genlib*. The mapping limits all gates in the circuit to be simple gates (NOT, OR, AND, NOR and NAND) with at most 2 fanins. Since the circuits consist of only simple gates after optimization, we post-processed all circuits with SIS commands $el; sweep; el; simplify$ and compared the factored literal counts among all four optimization schemes.

All experiments were performed in a Sun Enterprise 4500 machine. The

```
GBAW_logic_optimization($n$)
{
    for $n$ iterations {
        /* logic simplification */
        match and store all alternative wire patterns;
        perform transformations for
        single-addition-multiple-removal patterns;
        perform simultaneous pattern transformations;
        perform transformations for single-addition-single-removal
        patterns according to section 6.2.3;

        /* Incremental perturbation heuristic*/
        match and store all alternative wire patterns;
        perform transformations for single-addition-single-removal
        patterns according to section 6.3;
    }

        /* logic simplification before the end of the procedure*/
        match and store all alternative wire patterns;
        perform transformations for
        single-addition-multiple-removal patterns;
        perform simultaneous pattern transformations;
        perform transformations for single-addition-single-removal
        patterns according to section 6.2.3;

}
```

Figure 6.4: Outline of GBAW logic optimization algorithm

experimental results are shown in Table 6.1. The table shows the optimized circuit area (literal count) and CPU-time taken for five optimization approaches: *script.algebraic*, *script.boolean*, *script.rugged*, GBAW optimization algorithm and REWIRE [CGLMS99]. Columns 2 and 3 show the results and run-time of *script.algebraic*; Columns 4 and 5 show the results and run-time of *script.boolean*; Columns 6 and 7 show the results and run-time of *script.rugged*. For our GBAW optimization algorithm, the CPU-time for circuit simplification and incremental perturbation are recorded separately in columns 9 and 10 while columns 8 and 11 show the results and the total CPU-time. The last column shows the optimized area of REWIRE which is published in [CGLMS99]. In their paper, REWIRE was shown to be a competent optimization scheme with short CPU run-time.

The total area and CPU-time of all benchmark circuits were calculated in the second last row. In the last row, we normalized all total optimized circuit area and CPU-time with respect to the corresponding figures of GBAW optimization algorithm. Table 6.1 shows that the literal counts are about the same for GBAW and REWIRE. At the same time, GBAW optimizes benchmark circuits with 29%, 15% and 3% smaller area than *script.algebraic*, *script.boolean* and *script.rugged* do respectively. Moreover, GBAW optimization is, on average, 3.64, 4.34 and 215.02 times faster than the above three standard SIS scripts.

Since our experiments and the experiments in [CGLMS99] were done on different workstations, it is not appropriate to compare the CPU run-time directly. Thus, we compared GBAW optimization and REWIRE with standard SIS scripts and the comparisons are shown in Figure 6.2. In the table, column 2 shows the CPU time ratio of SIS scripts to GBAW optimization while column 3 shows the ratio of the same scripts to REWIRE. The ratios of REWIRE are calculated from the figures reported in [CGLMS99]. The last column shows that GBAW optimization runs 6% faster than REWIRE when both comparing

Circuit	algebraic		boolean		rugged		GBAW				REWIRE
	lits	time/s	lits	time/s	lits	time/s	lits	sim-t	perb-t	tot-t	lits
C1355	670	1.66	554	1.99	552	290.74	546	0.07	5.88	6.16	552
C1908	564	2.35	552	3.37	538	361.73	519	0.09	4.71	4.9	512
C2670	840	3.39	759	5.75	746	547.44	739	0.21	12.43	12.9	697
C3540	1486	7.69	1297	20.01	1200	143.57	1181	0.36	19.26	20	1127
C432	252	0.82	240	1.1	196	123.1	167	0.05	0.55	0.66	171
C499	558	1.07	554	1.71	552	23.4	546	0.07	5.89	6.17	550
C880	473	1.52	427	1.75	411	14.12	410	0.1	3.11	3.33	415
alu2	478	2.74	422	21.5	325	31.88	317	0.11	2.28	2.52	324
f51m	159	0.6	131	0.64	110	1.13	107	0.04	0.39	0.46	105
frg2	1118	8.4	934	10.87	758	59.88	745	0.17	6.8	7.14	761
term1	271	1.48	231	3.26	154	4.58	143	0.04	0.34	0.42	145
too_large	491	210.43	437	216.53	302	12748.6	273	0.08	1.23	1.4	301
ttt2	242	0.77	230	1.25	207	1.86	183	0.05	0.53	0.63	179
z4ml	42	0.14	39	0.21	36	0.36	36	0.01	0.04	0.06	36
total	7644	243.06	6807	289.94	6087	14352.4	5912	1.45	63.44	66.75	5875
relative	1.29	3.64	1.15	4.34	1.03	215.02	1.00			1.00	0.99

Table 6.1: Comparison between GBAW optimization algorithm, SIS scripts and REWIRE

with *script.boolean*, and at the same time, our algorithm runs 65% faster than REWIRE when both comparing with *script.rugged*.

CPU time ratio	GBAW	REWIRE	GBAW/REWIRE
script.boolean	4.34	4.02	1.08
script.rugged	215.02	130.00	1.65

Table 6.2: CPU time improvement of GBAW optimization and REWIRE against SIS scripts

Table 6.2 provides a indirect comparison between our GBAW optimization and REWIRE, which is the latest ATPG-based alternative wiring logic optimization algorithm. It is shown that GBAW optimization not only provides excellent circuit optimization capability which is comparable to REWIRE but also offers better efficiency.

6.6 Conclusion

In this chapter, the GBAW optimization algorithm is proposed. The algorithm consists of two parts: the logic simplification algorithm and incremental perturbation heuristic. The simplification and perturbation are based on different pattern matching of GBAW. The experimental results shows that our algorithm runs faster than [CGLMS99] with the same optimization ability.

Chapter 7

Conclusion

We have proposed two different approaches to the alternative wiring problem including IBAW and improved GBAW. At the same time, a comprehensive logic optimization algorithm by applying GBAW is clearly explained.

For IBAW, aiming at improving the efficiency of traditional ATPG-based alternative wiring algorithms, we explored the implication relationships among the source nodes and destination nodes of the alternative wire pairs. Experiment shows that our technique is able to quicken the original RAMBO by using only $\frac{1}{3.8}$ of RAMBO's CPU-time.

For GBAW, by adding more than 40 patterns into the pattern family, we successfully improved the solution-quality (number of alternative wires found) of GBAW, especially for 2-local alternative wire pairs. In order to decrease the complexity of pattern implementation, we group the patterns together as pattern clusters by concentrating on the similarity of patterns. Experimental results show that our improved GBAW is able to produce solution-quality comparable to ATPG-based alternative wiring algorithms.

For the GBAW optimization algorithm, we divided the logic optimization into two steps: the logic simplification and incremental perturbation. In the logic simplification step, patterns are divided into several groups depending on their functions. We observed that the logic transformations of the matched single-addition-multiple-removal patterns may lead to the decrease in network area. Besides, the logic transformation of some matched single-addition-single-removal patterns can be performed simultaneously such that the network area can be decreased. For some of other patterns, the removal of connection can in turn eliminate a number of gates. All these pattern transformations are competent in logic simplification.

Incremental perturbation is useful when the logic simplification is trapped into local minimum since it can be applied to increase the internal don't cares of the network. In such situations, the number of possible alternative wires would increase and this may bring the network out of the local minimum in the logic simplification process. Experimental results show that our logic optimization approach is able to obtain competent optimization results and better efficiency when comparing with other logic optimization tools.

Bibliography

[Ass97] Semiconductor Industry Associates. National technology roadmap for semiconductores, 1997.

[Ass00] Semiconductor Industry Association. International technology roadmap for semiconductores, 2000.

[BRSVW87] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1062–1081, November 1987.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computer*, 35(8):677–691, Aug. 1986.

[CCWMS94] Shih Chieh Chang, Kwang Ting Cheng, Nam Sung Woo, and M Marek-Sadowska. Layout driven logic synthesis for FPGAs. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 308–313, June 1994.

[CCWMS97a] Shih Chieh Chang, Kwang Ting Cheng, Nam Sung Woo, and M. Marek-Sadowska. Postlayout logic restructuring using alternative wires. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(6):587–596, June 1997.

[CCWMS97b] Shih Chieh Chang, Kwang Ting Cheng, Nam Sung Woo, and M. Marek-Sadowska. Postlayout logic restructuring for performance optimization. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 662–665, 1997.

[CE93] Kwang Ting Cheng and Luis A. Entrena. Multi-level logic optimization by redundancy addition and removal. In *Proceedings of European Conference on Design Automation, with the European Event in ASIC Design*, pages 373–377, Feb. 1993.

[CGLMS96] Shih Chieh Chang, Van Ginneken, L.P.P.P., and M. Marek-Sadowska. Fast boolean optimization by rewiring. *Digest of Technocal Papers of IEEE/ACM International Conference on Computer-Aided Design*, pages 262–269, 1996.

[CGLMS99] Shih Chieh Chang, Van Ginneken, L.P.P.P., and M. Marek-Sadowska. Circuit optimization by rewiring. *IEEE Transactions on Computer*, 48 9:962–969, Sept. 1999.

[CLMS95] D. I. Cheng, C C Lin, and M. Marek-Sadowska. Circuit partitioning with logic perturbation. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 650–655, 1995.

[CMSC96] Shih Chieh Chang, M. Marek-Sadowska, and Kwang Ting Cheng. Perturb and simplify: Multilevel boolean network optimizer. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15 12:1494–1504, Dec. 1996.

[CPK98] M. Chatterjee, D. K. Pradhan, and W. Kunz. Lot: Logic optimization with testability - new transformations for logic synthesis. *IEEE Transcations on Computer-Aided Design of Integrated Circuits and Systems*, 17(5):386–399, May 1998.

[EC93] Luis A. Entrena and Kwang Ting Cheng. Sequential logic optimization by redundancy addition and removal. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 310–315, Nov. 1993.

[EC95] Luis A. Entrena and Kwang Ting Cheng. Combinational and sequential logic optimization by redundancy addition and removal. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(7):909–916, July 1995.

[EEOU96] L. A. Entrena, J. A. Espejo, E. Olias, and J. Uceda. Timing optimization by an improved redundancy addition and removal technique. In *Proceedings of European Design Automation Conference, with EURO-VHDL '96 and Exhibition*, pages 342–347, 1996.

[FS93] H. Fujiwara and T. Shimono. On the acceleration of test generation algorithms. In *Proceedings of International Symposium on Fault Tolerant Computing*, pages 98–105, 1993.

[FS97] James F. Freedman and Scott Sibbett. Report of the ad hoc working group on interconnect. Focus Center Research Program, 1997.

[HS96] Gary D. Hachtel and Fabio Somenzi. *Logic Synthesis and Verification Algorithms.* Kluwer Academic Publishers, 1996.

[IK98] H. Ichihara and K. Kinoshita. Logic optimization: Redundancy addition and removal using implication relationships. *IEICE Transactions on Information & Systems (Special Issue on Test and Diagnosis of VLSI)*, E81-D(7):724–730, 1998.

[KM94] W. Kunz and P. R. Menon. Multilevel logic optimization by implication analysis. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 6–13, 1994.

[KP92] W. Kunz and D. K. Pradhan. Recursive learning: an attractive alternative to the decision tree for test generation in digital circuits. In *Proceedings of International Test Conference*, pages 816–825, Sept. 1992.

[LCMS99] Chih-Chang Lin, Kuang-Chien Chen, and M. Marek-Sadowska. Logic synthesis for engineering change. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(3):282–292, March 1999.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technology Journal*, 38:985–999, July 1959.

[LWB00] Wangning Long, Yu Liang Wu, and Jinian Bian. IBAW: An implication-tree based alternative-wiring logic transformation algorithm. In *Proceedings of Asia and South Pacific Design Automation Conference*, pages 415–421, 2000.

[Ped97] Massoud Pedram. Panel: Physical design and synthesis: Merge or die! In *Proceedings of ACM/IEEE Design Automation Conference*, pages 238–239, 1997.

[She99] Naveed Sherwani. *Algorithms for VLSI Physical Design Automation.* Kluwer Academic Publishers, 1999.

[SsLea92] E. M. Sentovich, K. J. singh, L. Lavagno, and et. al. SIS: A system for sequential circuit synthesis. *ERL Memorandum No. UCB/ERL*, M92/41, 1992.

[WLF00] Yu Liang Wu, Wangning Long, and Hongbing Fan. A fast graph-based alternative wiring scheme for boolean networks. In *Proceedings of International Conference on VLSI Design*, pages 268–273, 2000.

Appendix A

VLSI Design Cycle

The overview of the VLSI systems design cycle is shown in Figure A.1 which is extracted from [She99]. The cycle can be generally divided into three parts: high-level synthesis, logic synthesis and physical design synthesis.

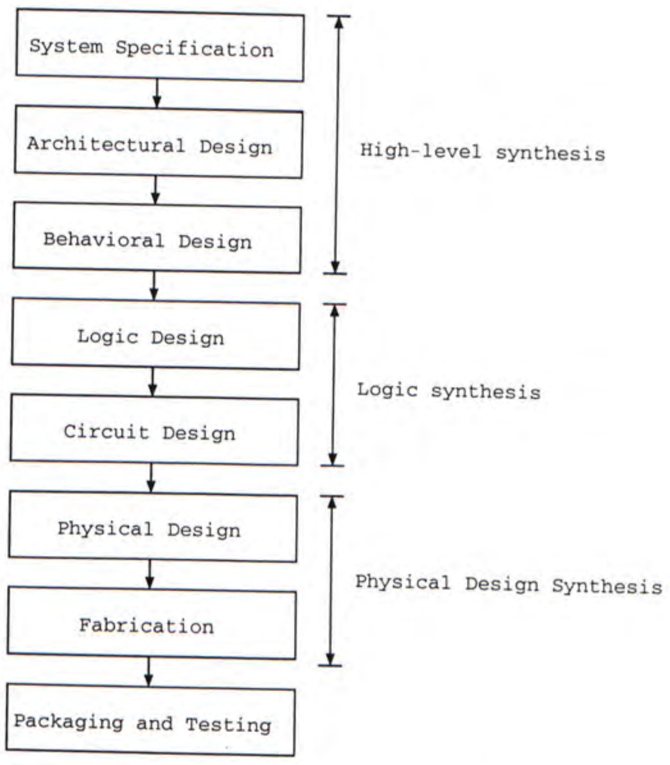


Figure A.1: A simplified VLSI design cycle

In high-level synthesis, the first steps are system specification and architectural design. The specification is a high-level representation of the system and it initiates the whole design process. As a result, it should consider all factors inside the design process such as, functionality, performance, technology, market value, etc. For the Architectural design, instruction sets and system elements such as ALUs, caches are specified. The Micro-Architectural Specification is the output of architectural design step.

The next step is the functional design and behavioral synthesis. In this step, all functional unit and the connection between the units are defined. Besides, for each unit, the system requirements are specified while the limitations are estimated.

For logic design, the output is the Register Transfer Level (RTL) description such as Verilog, Hardware Description Language (HDL) and VHDL. The description specifies logic expressions of each functional unit. In this step, logic and timing simulation and testing are performed.

The circuit design step is intended to convert logic specification into circuit representation. The representation is always called a netlist. It represents all circuit elements including gates and connections. The conversion is always guided by timing and power limitation.

For physical design processes, circuit level representation is converted into geometric representation. The process includes partitioning, floorplanning, placement and routing. The output of physical design is a layout. Throughout the processes, the conversion should strictly satisfy some design rules, such as metal width, size, layers and chip area specification. Verification is very important for layout quality assurance. If the limitation cannot be fulfilled, engineering changes must be performed.

Fabrication, packaging and testing are the last steps of the design cycle. In the process, wafers are fabricated and diced into chips. The chips should be packaged and tested before delivery. The final product should satisfy all system specification and performance requirement.

Appendix B

Alternative Wire Patterns in [WLF00]

In this chapter, the basic minimal patterns (0-local, 1-local, 2-local) in for rewiring [WLF00] are introduced.

B.1 0-local Pattern

Figure B.1 shows a 0-local pattern. As g_1 and g_3 have the same fanins and the same operators, they possess duplicated signals. Let $g_1 \rightarrow g_2$ be the target wire, adding $g_3 \rightarrow g_2$ makes $g_1 \rightarrow g_2$ redundant and thus can be removed. Furthermore, the node g_1 (and its fanins: $a_1 \rightarrow g_1, a_2 \rightarrow g_1, ..., a_k \rightarrow g_1$) can also be removed if g_1 has only one fanout.

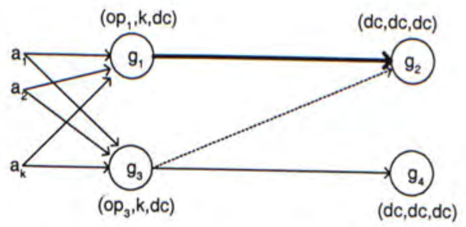
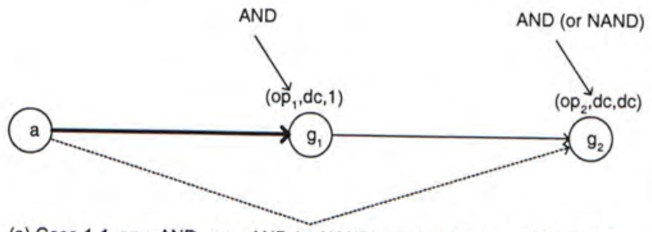


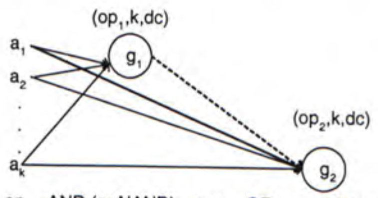
Figure B.1: 0-local pattern

B.2 1-local Pattern

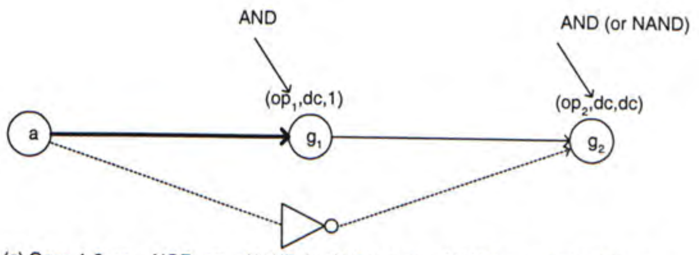
There are three 1-local patterns, which are case 1-1, case 1-2 and case 1-3. All of them are shown in Figure B.2. If g_1 has only one fanout g_2, $op_1 = AND$ and $op_2 = AND$ (or $NAND$), then $a \rightarrow g_1$ can be replaced by $a \rightarrow g_2$. Similarly it applies for op_1 = OR and op_2 = OR (NOR). Case 1-2 is similar to the 0-local pattern, except the target wire is $a \rightarrow g_2$ and the alternative wire is $g_1 \rightarrow g_2$. In case 1-3, if g_1 has only one fanout, op_1 = NOR and op_2 = NAND (or AND), then $a \rightarrow g_1$ can be replaced by $a \rightarrow g_2$ with an inverter. Similarly it applies to op_1 = NAND and op_2 = NOR(OR).



(a) Case 1-1, op$_1$=AND, op$_2$ =AND (or NAND); or op$_1$=OR, op$_2$=OR(or NOR)



(b) Case 1-2, op$_1$=AND, op$_2$ =AND (or NAND); or op$_1$=OR, op$_2$=OR(or NOR)



(c) Case 1-3, op$_1$=NOR, op$_2$ =NAND (or AND); or op$_1$=NAND, op$_2$=OR(or NOR)

Figure B.2: Three cases of 1-local patterns

B.3 2-local Pattern

For 2-local patterns, the alternative wire is 2-edge far away from the target wire. There are three cases of 2-local patterns. All of them are shown in Figure B.3. For case 2-1, let $a \rightarrow g_1$ be the target wire. The pattern requires that g_1=(NOR,dc,1), g_2 = (NAND, dc, 1), g_3 = (NOR,dc,dc). For case 2-2, let $< a_1, g_1 >$ be the target wire. The pattern requires that g_1 = (OR,dc,1), g_2 = (AND,k,1), g_3 = (NOR,dc,dc) and g_4 = (AND,k,dc). Case 2-3 is also shown in Figure B.3.

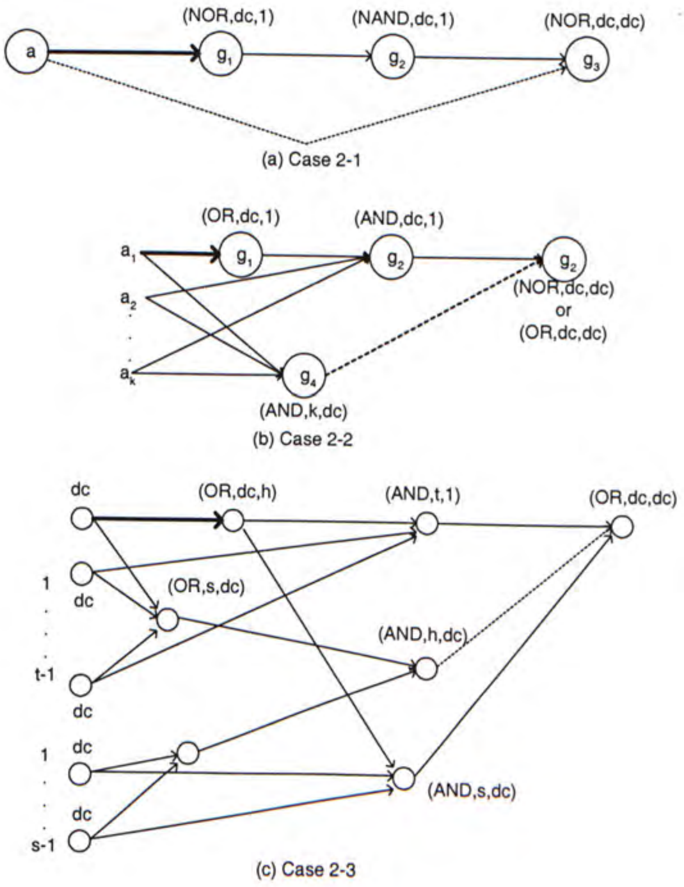


Figure B.3: Three cases of 2-local patterns

B.4 Fanout-reconvergent Pattern

The 2-local patterns involving re-convergent fanouts of this kind can also be easily located by graph-based algorithm for example, GBAW. The fanout-reconvergent pattern is adopted by GBAW and it is shown in Figure B.4.

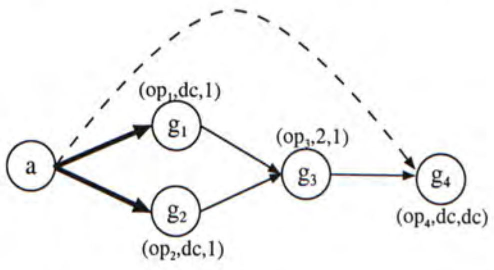


Figure B.4: Example of fanout-reconvergent patterns

In the figure, g_1 and g_2 should have the same logic operator, $op_1 = op_2$. They can be AND, OR, NAND and NOR. Besides, g_1 and g_2 should have at least 2 fanins. For node g_4, the operator should be AND or OR. In this pattern, the alternative wire may have inverter when $op_1 = op_2 =$ NAND or NOR.

Appendix C

New Alternative Wire Patterns

The construction of new minimal patterns is a joint project with another M. Phil. student, Chak-Chung Cheung, under the supervision of Professor Yu-Liang Wu, in the Department of Computer Science and Engineering. In this thesis, we only show partial patterns for illustration.

In the following figures, dark thick lines represent target wires and thin dotted lines are alternative wires.

C.1 Pattern Cluster C_1

The first pattern cluster is in the form of two gates in series with only one fanout. And there are two parallel gates with same fanins. This pattern cluster is of type single-addition-single-removal.

C.1.1 NAND-NAND-AND/NAND;AND/NAND

Figure C.1 shows a pattern in pattern cluster C_1. The proof is shown as follows. Without loss of generosity, we assume g_3 is an AND gate. In the proof, let x be the AND-product of all other inputs of g_1 except a and z be the OR-summation of all other inputs of g_3 except g_2. b is the AND-product of all other inputs of g_2 except g_1, i.e. $b = b_2b_3b_4...b_k$, where $k = 2, 3, 4....$

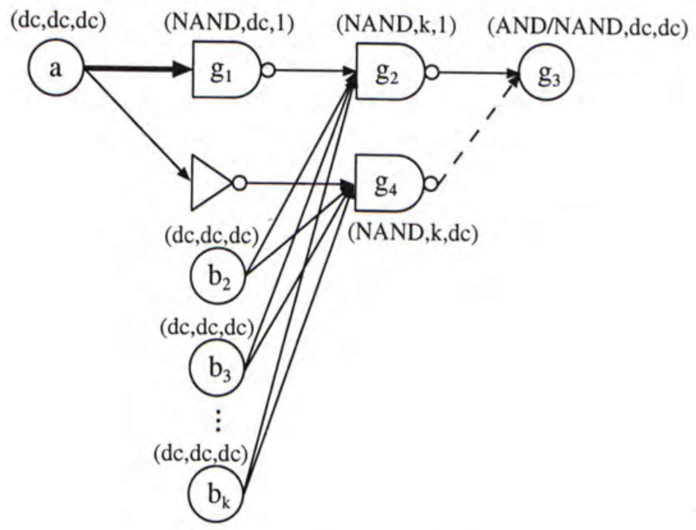


Figure C.1: Cluster C_1, Pattern A1

Before transformation	$g_3 = [(ax)'b]'z = z(ax + b')$
After transformation	$g_3 = (x'b)'z(a'b)' = z(ax + b')$

The pattern shown in Figure C.2 is nearly the same as Figure C.1. The difference is that the NAND gate g_4 is separated into a AND and a NOT. We treat them as different patterns in pattern matching since g_4 would pass different values to its fanouts and hence it represents different logic structure.

C.1.2 NOR-NOR-OR/NOR;AND/NAND

Figure C.3 shows another pattern. The proof is shown as follows. Without loss of generosity, we assume g_3 is an OR gate. In the proof, let x be the OR-summation of all other inputs of g_1 except a and z be the OR-summation of all other inputs of g_3 except g_2. b is the AND-product of all other inputs of g_2 except g_1, i.e. $b = b_2 + b_3 + b_4 + ... + b_k$, where $k = 2, 3, 4....$

Before transformation	$g_3 = [(a + x)' + b]' + z = z + ab' + xb'$
After transformation	$g_3 = (x' + b)' + z + (a' + b)' = z + ab' + xb'$

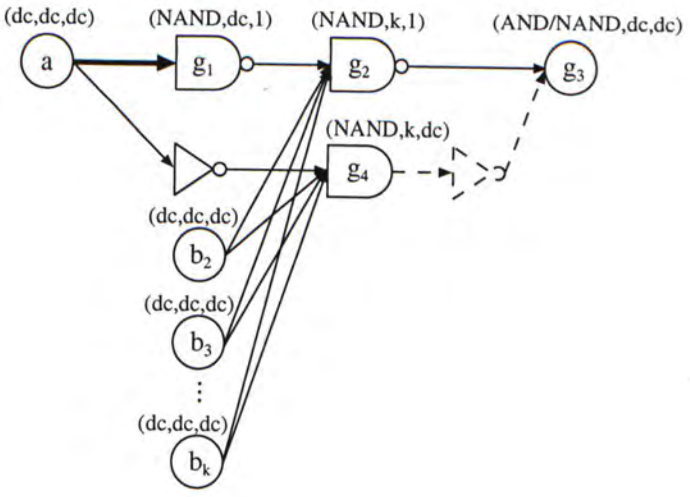


Figure C.2: Cluster C_1, Pattern A2

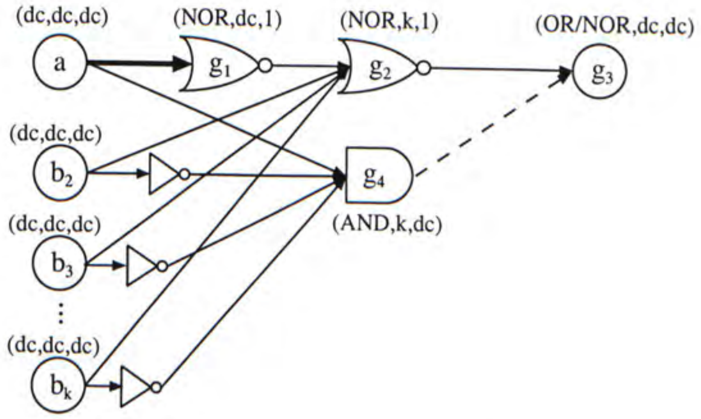


Figure C.3: Cluster C_1, Pattern C1

The pattern shown in Figure C.4 is nearly the same as Figure C.3. The difference is that the NOR gate g_4 is separated into a OR and a NOT. We treat them as different patterns in pattern matching since g_4 would pass different values to its fanouts and it represents different logic structure.

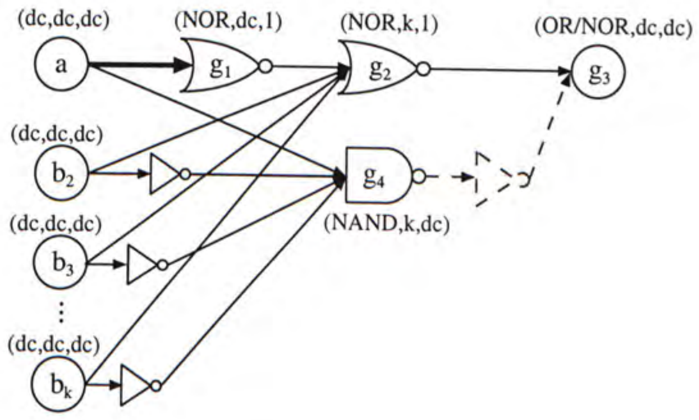


Figure C.4: Cluster C_1, Pattern C2

C.1.3 AND-NOR-OR/NOR;OR/NOR

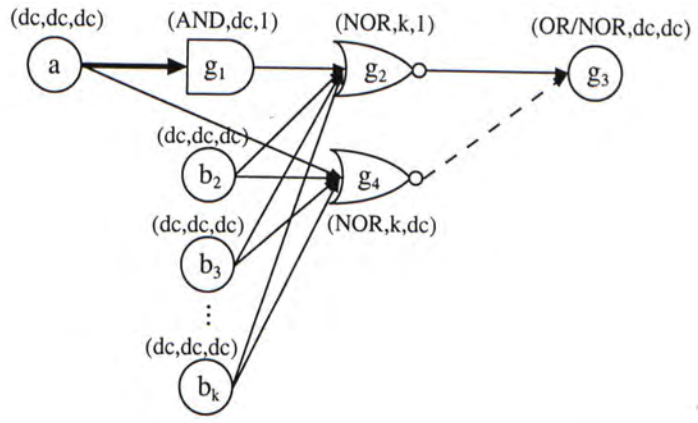


Figure C.5: Cluster C_1, Pattern E1

Figure C.5 shows another pattern. Actually, this pattern is logically the same as the second pattern of 2-local patterns in [WLF00]. The proof is shown as follows. Without loss of generosity, we assume g_3 is an OR gate. In the proof, let x be the AND-product of all other inputs of g_1 except a and z be the OR-summation of all other inputs of g_3 except g_2. b is the OR-summation of all other inputs of g_2 except g_1, i.e. $b = b_2+b_3+b_4+...+b_k$, where $k = 2, 3, 4....$

Before transformation	$g_3 = (ax+b)' + z = z + a'b' + x'b'$
After transformation	$g_3 = (x+b)' + z + (a+b)' = z + a'b' + x'b'$

The pattern shown in Figure C.6 is nearly the same as Figure C.5. The difference is that the NOR gate g_4 is separated into a OR and a NOT. We treat them as different patterns in pattern matching since g_4 would pass different values to its fanouts and it represents different logic structure.

C.1.4 OR-NAND-AND/NAND;AND/NAND

Figure C.7 shows another pattern. Actually, this pattern is logically the same as the second pattern of 2-local patterns in [WLF00]. The proof is shown

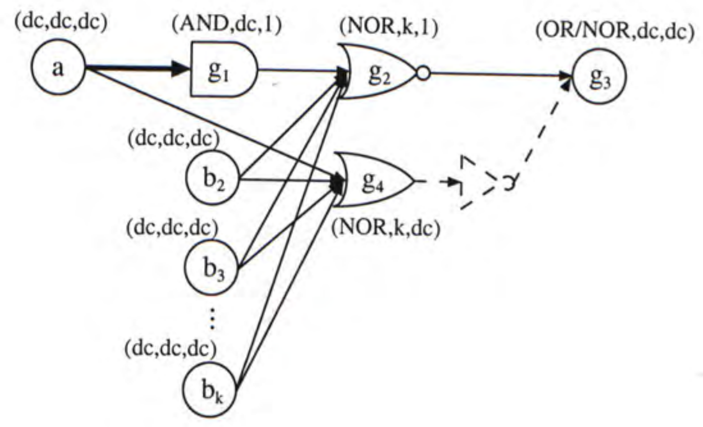


Figure C.6: Cluster C_1, Pattern E2

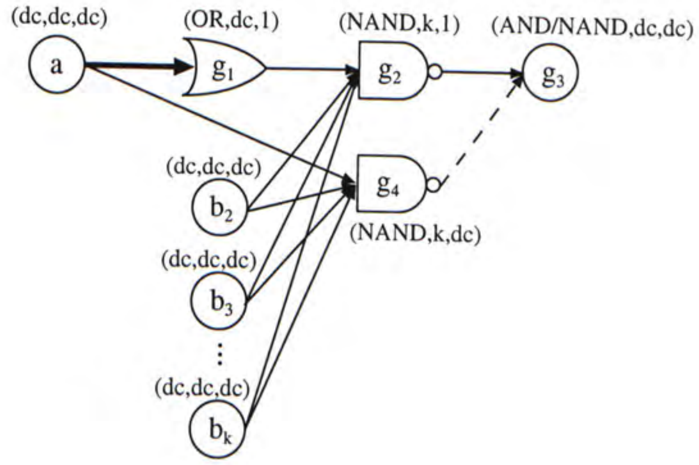


Figure C.7: Cluster C_1, Pattern F1

as follows. Without loss of generosity, we assume g_3 is an AND gate. In the proof, let x be the OR-summation of all other inputs of g_1 except a and z be the AND-product of all other inputs of g_3 except g_2. b is the AND-product of all other inputs of g_2 except g_1, i.e. $b = b_2b_3b_4...b_k$, where $k = 2, 3, 4....$

Before transformation	$g_3 = [(a+x)b]'z = (a'x' + b')z$
After transformation	$g_3 = (xb)'z(ab)' = z(a'+b')(x'+b')$

The pattern shown in Figure C.8 is nearly the same as Figure C.7. The difference is that the NOR gate g_4 is separated into a OR and a NOT. We treat them as different patterns in pattern matching since g_4 would pass different values to its fanouts and it represents different logic structure.

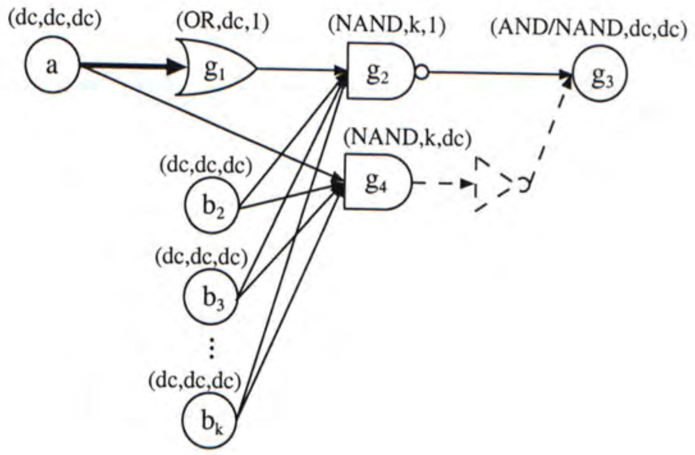


Figure C.8: Cluster C_1, Pattern F2

C.2 Pattern Cluster C_2

All patterns in Cluster C_2 are related to fanout reconvergence. Since they are of type single-addition-multiple-removal, they are very useful in logic simplification.

Figure C.9 shows the first pattern in the cluster. The proof is as follows. Without loss of generosity, we assume g_3 is an AND gate. In the proof, let x_1 be the AND-product of all other inputs of g_1 except a and x_2 be the AND-product of all other inputs of b_2 except a. Similarly, $x_3 ... x_k$ are the AND-product of all other inputs of $b_3 ... b_k$ except a.

Before transformation	$g_3 = [(ax_1)'(ax_2)'...(ax_k)']'z = az(x_1 + x_2 + ... + x_k)$
After transformation	$g_3 = az(x_1 + x_2 + ... + x_k)$

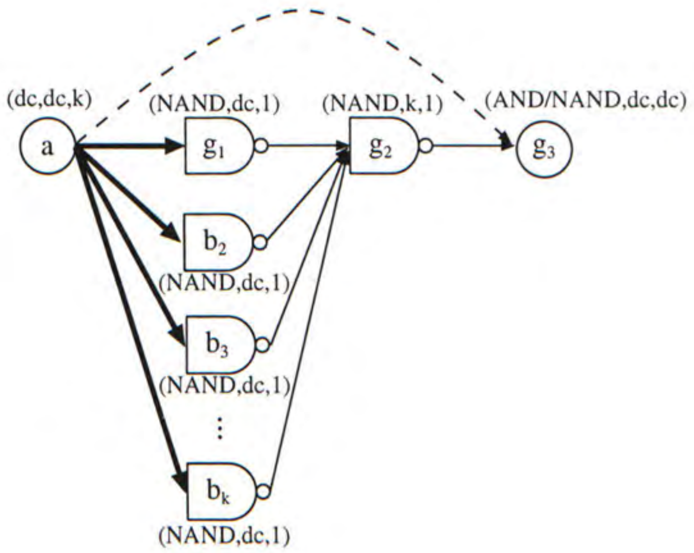


Figure C.9: Cluster C_2, Pattern A1

The logic structure of Figure C.10 is the same as Figure C.9 while NAND-NAND is replaced with AND-OR.

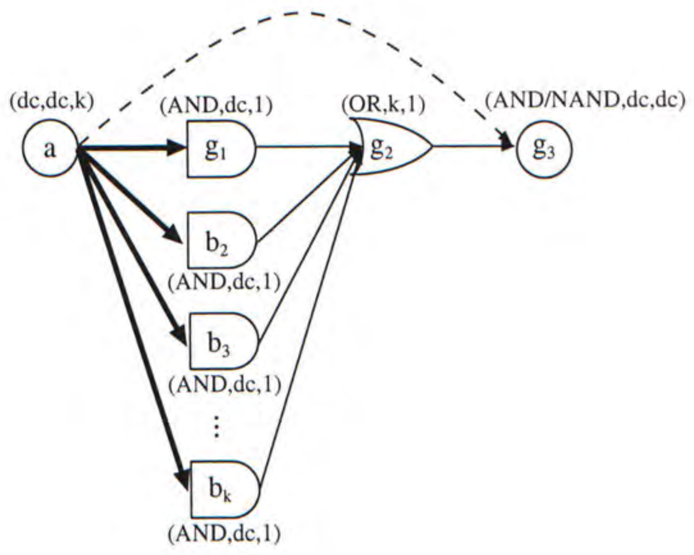


Figure C.10: Cluster C_2, Pattern A2

C.3 Pattern Cluster C_3

This pattern is related to the consensus property of logic. Belonging to the type single-addition-multiple-removal, they are also very useful in logic simplification. For the patterns with the same alphabetical name but different numerical name, they are of the same logic structure. For example, patterns A1, A2, A3 and A4 are the same. In fact, our implementation includes similar variation of patterns (e.g. D1,D2,D3,D4) but all variations of B1, C1, D1, E1, F1 and G1 are omitted in this chapter.

The pattern in Figure C.11 is different from all specific member in pattern cluster C_1 and C_2. The proof is shown as follows. Without loss of generosity, we assume g_3 is an AND gate. In the proof, let z be the AND-product of all other inputs of g_3 except g_2, and y is the OR-summation of all inputs of g_5 except b'.

Before transformation	$g_3 = [(a+b)(y+b')]'z = z(a'b' + y'b)$
After transformation	$g_3 = (yb)'z(ab')' = z(a'b' + y'b + a'y')$

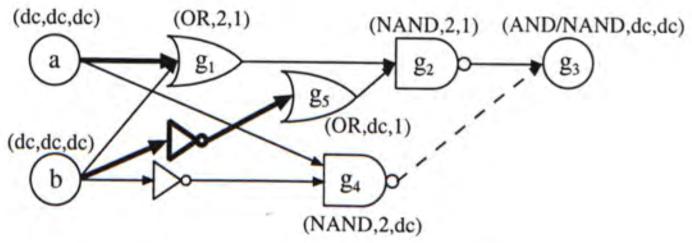


Figure C.11: Cluster C_3, Pattern A1

This pattern member has another form and it is shown in Figure C.12,

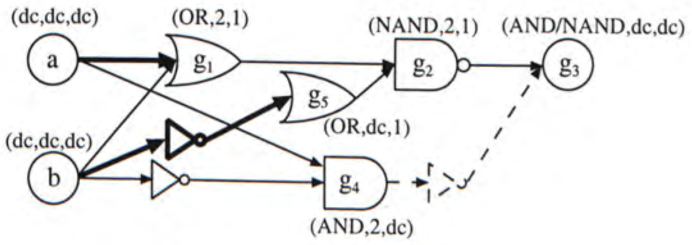


Figure C.12: Cluster C_3, Pattern A2

Figure C.13 shows another pattern member. The proof is shown as follows. Without loss of generosity, we assume g_3 is an AND gate. In the proof, let z be the AND-product of all other inputs of g_3 except g_2, and y is the OR-summation of all inputs of g_5 except b'.

Before transformation	$g_3 = [(a+b)(y+b')]'z = z(a'b' + y'b)$
After transformation	$g_3 = (yb)'z(a'+b) = z(a'b' + y'b + a'y')$

Figure C.14 shows another pattern member. The proof is shown as follows. Without loss of generosity, we assume g_3 is an AND gate. In the proof, let z be the AND-product of all other inputs of g_3 except g_2, and y is the AND-product of all inputs of g_5 except b'.

Before transformation	$g_3 = [(a+b)(yb)']'z = z(a'b' + yb)$
After transformation	$g_3 = (y'b)'z(ab')' = z(a'b' + yb + a'y)$

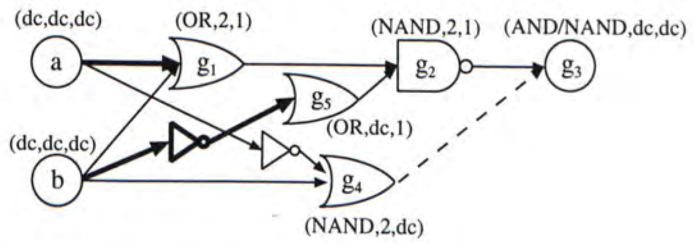


Figure C.13: Cluster C_3, Pattern B1

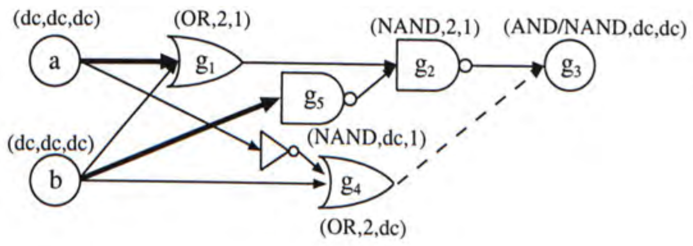


Figure C.14: Cluster C_3, Pattern C1

Figure C.15 shows another pattern member. The proof is shown as follows. Without loss of generosity, we assume g_3 is an AND gate. In the proof, let z be the AND-product of all other inputs of g_3 except g_2, and y is the OR-summation of all inputs of g_5 except b'.

Before transformation	$g_3 = [(a+b)' + (yb)]z = z(a'b' + yb)$
After transformation	$g_3 = (y + b')z(a' + b) = z(a'b' + yb + a'y)$

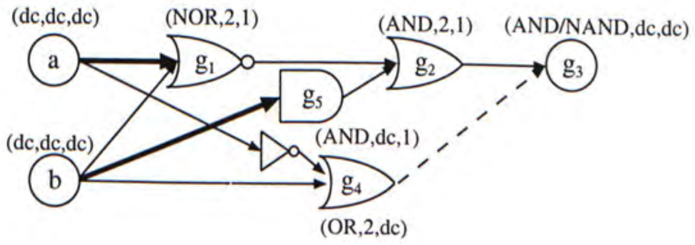


Figure C.15: Cluster C_3, Pattern D1

Figure C.16 shows another pattern member. The proof is shown as follows.

Without loss of generosity, we assume g_3 is an AND gate. In the proof, let z be the AND-product of all other inputs of g_3 except g_2, and y is the OR-summation of all inputs of g_5 except b'.

Before transformation	$g_3 = [(ab') + (yb)]z = z(ab' + yb)$
After transformation	$g_3 = (y + b')z(a + b) = z(ab' + yb + ay)$

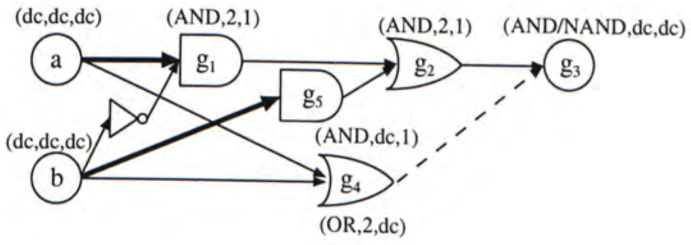


Figure C.16: Cluster C_3, Pattern E1

Figure C.17 shows another pattern member. The proof is shown as follows. Without loss of generosity, we assume g_3 is an AND gate. In the proof, let z be the AND-product of all other inputs of g_3 except g_2, and y is the OR-summation of all inputs of g_5 except b'.

Before transformation	$g_3 = [(a + b')(y + b)]'z = z(a'b + y'b')$
After transformation	$g_3 = (yb)'z(ab)' = z(a'b + y'b' + a'y')$

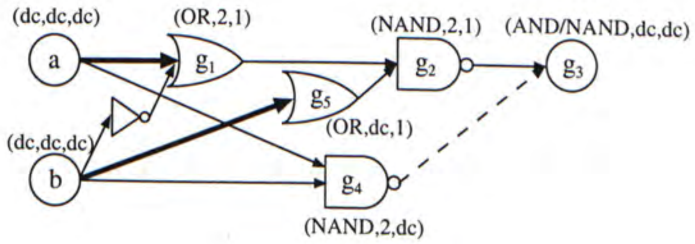


Figure C.17: Cluster C_3, Pattern F1

Figure C.18 shows another pattern member. The proof is shown as follows. Without loss of generosity, we assume g_3 is an AND gate. In the proof, let

z be the AND-product of all other inputs of g_3 except g_2, and y is the OR-summation of all inputs of g_5 except b'.

Before transformation	$g_3 = [(a'b) + (y + b)']z = z(a'b + y'b')$
After transformation	$g_3 = (y' + b)z(ab)' = z(a'b + y'b' + a'y')$

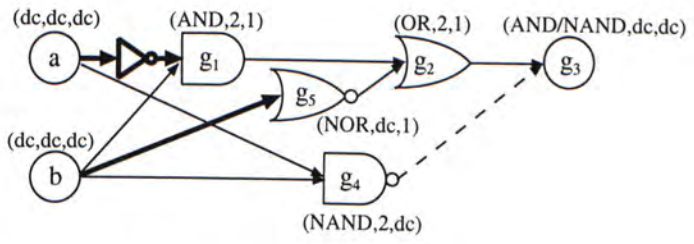


Figure C.18: Cluster C_3, Pattern G1

C.4 Pattern Cluster C_4

The pattern cluster is extracted from experimental results. We compare the result of original GBAW and other ATPG-based alternative wiring programs. Figure C.19 demonstrates one pattern in the cluster. The pattern is bigger in size as it consists of 11 nodes.

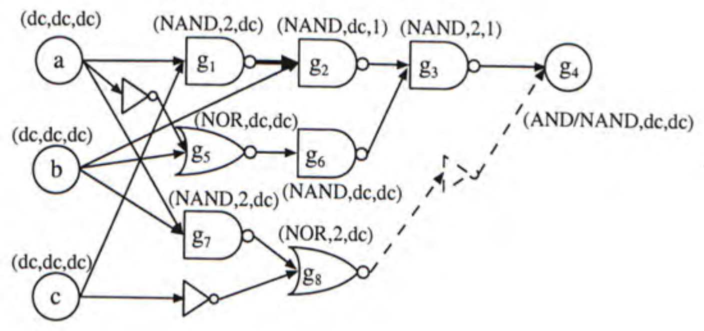


Figure C.19: Cluster C_4, Pattern A1

The proof is shown as follows. Without loss of generosity, we assume g_4 is an AND gate. In the proof, let q be the AND-product of all other inputs of g_2 except g_1 and b, w be the AND-product of all other inputs of g_4 except g_3 and g_8', s be the OR-summation of all inputs of g_5 except a' and b, and v be the AND-product of all inputs of g_6 except g_5.

Before transformation	$g_4 = \{[(a'+b+s)'v]'[(ac)'bq]'\}'w$ $= [ab's'v' + bq(a'+c')]w$
After transformation	$g_4 = \{[(a'+b+s)'v]'(bq)'\}'w[(ab)'+c']$ $= [ab's'v' + bq(a'+c')]w$

C.5 Pattern Cluster C_5

The pattern cluster is obtained from experimental results and is extracted from pattern cluster C_4. Figure C.20 demonstrates one pattern in the cluster.

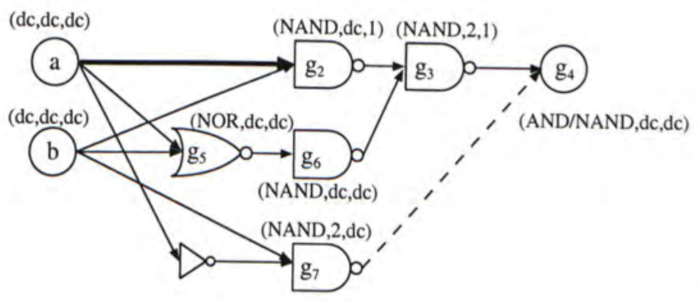


Figure C.20: Cluster C_5, Pattern A1

The proof is shown as follows. Without loss of generosity, we assume g_4 is an AND gate. In the proof, let w be the AND-product of all other inputs of g_2 except a and b, x be the AND-product of all other inputs of g_4 except g_3 and g_7, y be the OR-summation of all inputs of g_5 except a and b, and z be the AND-product of all inputs of g_6 except g_5.

Before transformation	$g_4 = \{(abw)'[(a+b+x)'y]'\}'z$ $= (abw + a'b'x'y)z$
After transformation	$g_4 = \{(bw)'[(a+b+x)'y]'\}'z(a'b)'$ $= [bw + a'b'x'y)z(a+b')$

Glossary

absolute dominator a node D on the transitive fanout of a target wire such that all paths from the target wire to any primary output should go through D.

alternative wire pattern a graph configuration containing an alternative wire pair.

ATPG abbreviation for Automatic Test Pattern Generation.

cv abbreviation for controlling value.

dc abbreviation of any operator or any positive integer, appeared in the triplets of nodes in a GBAW graph configuration.

DMA abbreviation for Driving mandatory assignments.

dominator short form of absolute dominator.

FPMA abbreviation for fault propagation mandatory assignments.

GBAW abbreviation for Graph Based Alternative Wiring Logic Transformation.

IBAW abbreviation for Implication Based Alternative Wiring Logic Transformation.

OMA abbreviation for observability mandatory assignments.

pattern short form of alternative wire pattern.

perturbation short form of logic perturbation.

PI abbreviation for primary inputs.

PO abbreviation for primary outputs.

RAMBO abbreviation for Redundancy Addition-and-removal for Multilevel Boolean Optimization, refer to [CE93].

REWIRE one of the latest logic optimization algorithms based on ATPG-based alternative wiring logic transformation, refer to [CGLMS99].

SIS a open-source system for sequential and combinational logic synthesis.

SMA abbreviation for set of mandatory assignments.

Index

Strategies to Enhance Performance of Pre-sorting based Algorithms in Building Decision Trees for Large Data Sets

Jian Tang* Ada Wai-chee Fu+ Yin Ling Cheung+

* Department of Computer Science
Memorial University of Newfoundland,
St. John's, NF, A1B 3X5 Canada
jian@cs.mun.ca

+ Department of Computer Science and Engineering
Chinese University of Hong Kong
Shatin, Hong Kong
{adafu,ylcheung}@cse.cuhk.edu.hk

Abstract

Classification is a function that identifies a new object as belonging to one of several predefined classes. Since late 70's, it has been used to assist in decision making process in a variety of applications such as medical diagnosis, credit approval, weather prediction, etc. It has emerged now as an important branch of data mining in databases. Among the techniques for classification, decision tree has caught most attention recently due to its conceptual simplicity and accuracy. One class of methods for building a decision tree pre-sort the attribute values for each attribute. As the decision tree grows, the attribute values will be distributed (recursively) to each node in such a way that their relative orders are preserved. The advantages of this pre-sorting method are twofold. First, it maintains the pre-sorted order of the attribute values without incurring sorting-related overhead at each node. Second, compared with other methods, the input data set it creates at each node is less sensitive in size to the inter-dependencies between different attributes. Thus its performance is relatively stable under different data distributions. In this paper, we study several strategies for pre-sorting based methods in building decision trees under database oriented constraint: the main memory space is limited, and is smaller than the dataset. We pay particular attention to the problem of how to minimize the I/O operations under the limited memory space. Our study shows that by emphasizing on different aspects, we can obtain schemes with different performance characteristics. Thus they can be used to meet different requirements for applications.

Keywords: Knowledge Discovery in Databases, Classification, Decision Trees, Sorting.

1 Introduction

Classification has emerged as one of the main branches in data mining since the 90's. Early applications were restricted only to small datasets, which were assumed to fit into main memories and accuracy was the only concern. With the rapid growth of the computer's capability to collect and store large amount of data, assuming the entire dataset to fit into the main memory is no longer realistic. This results in new issues such as fast classification, scalability, etc. Several kinds of classifiers were proposed in the past [5, 14, 8, 4, 18]. Among them *decision tree* has caught most attention recently due to its simplicity, conceptual cleanness, and accuracy [4, 1, 11, 12, 13].

SPRINT [15] employs a *pre-sorting* method, which sorts attirbute values only once. The advantages of pre-sorting attributes are twofold. First, it maintains the pre-sorted order of the attribute values without incurring sorting related overhead at each node. Second, compared with other methods, the input data it creates at each node is less sensitive in size to the inter-dependencies between different attributes. Compared with other work, such as [10, 3, 17, 7, 6, 9], SPRINT eliminates the introduction of inaccuracy in the result, sorting at every node, memory resident data structure, and the dependence of sample selection.

In this paper, we study strategies to speed up the pre-sorting based approaches to building decision trees under database oriented constraint: the main memory space is limited, in comparison to the dataset size. We make no assumption that the main memory can hold any dataset dependent on the structures of the original dataset for the major evaluation steps. Also, we do not introduce any restriction that may compromise the accuracy. Therefore we provide absolute improvements on previous methods with no trade-offs. Since in general the growing phase dominates the performance, we consider the growing phase only.

In our schemes, we pay special attention to the problem of how to minimize the I/O operations under limited main memory space. We use a technique, called 'pre-evaluation', whereby split points can be evaluated for an attribute list while the attribute list is being generated, instead of after it has been generated and written to disk. This virtually reduces the amount of I/O operations required for evaluating split points to zero. Our study also shows that by emphasizing on different aspects, one can obtain schemes with different performance characteristics. Thus they can be used to meet different requirements for the applications.

The rest of this paper is organized as follows. In Section 2, we describe the problem and explain the framework introduced in SPRINT. In Section 3, we introduce pre-evaluation technique. In Section 4, we present several strategies for building decision trees. In Section 5, we analyze the performance for those strategies. We conclude the paper by summarizing the main results.

2 A framework using sorted attribute lists in Decision Trees

The construction of decision tree for large dataset in SPRINT [15], is based on the use of attribute lists, one for each predictor attribute. An **attribute list** for attribute X at node N is a projection of the associated dataset $S(N)$ on X, C and *Rid*. If X is a continuous attribute, then the attribute

```
Preprocessing: construct a set $A$ of sorted attribute lists.
BuildTree(Node $N$, AttriSet $A$)
0. if all data in $N$ are of the same class then return
1. further splitting is necessary: find splitting attribute list $L$ and splitting point $v$
2. Generate $N_L$ and $N_R$ as the left child and right child of $N$;
3. Split($A, L, v, A_L, A_R$);
4. BuildTree($N_L$,$A_L$);
5. BuildTree($N_R$,$A_R$);
```

Figure 1: Scheme 0 (SPRINT)

list is ordered by the values of X. A framework for this approach is shown in Figure 1. In the preprocessing, attribute lists are created for both the splitting attribute and non-splitting attributes. Then BuildTree(N,A) is called, where parameter A is the set of constructed attribute lists. Further splitting is necessary if all records in $S(N)$ do not have a unique class label. In this case we search for the best splitting point in any attribute list using Gini Index.

Let L be an attribute list for attribute X. Assume X is a continuous attribute and $x \in X$. Let $S_x = \{r : r.X \leq x \ \&\ r \in L\}$ and $R_x = \{r : r.X > x \ \&\ r \in L\}$. Then the Gini Index at x is

$$Gini_x = \frac{|S_x|}{|L|}Gini(S_x) + \frac{|R_x|}{|L|}Gini(R_x) \qquad \textbf{I}$$

To evaluate all the splitting points for X, we scan L top down. Suppose r and e are two entries, we say that $r \preceq e$ if and only if the value of the splitting attribute in r is less than or equal to that of e. For each entry e scanned, for any class label i, the count $y_i = | \{r : r \in S(N) \ \&\ r \preceq e \ \&\ r.C = i\} |$ is accumulated. These counts can be used to calculate the Gini Index for each splitting point on the way[1].

Now assume X is a categorical attribute and $x \subseteq X$. Let $U_x = \{r : r.X \in x \ \&\ r \in L\}$ and $V_x = \{r : r.X \notin x \ \&\ r \in L\}$. Then the Gini Index at x is

$$Gini_x = \frac{|U_x|}{|L|}Gini(U_x) + \frac{|V_x|}{|L|}Gini(V_x) \qquad \textbf{II}$$

In SPRINT, a *count matrix* is constructed for each attribute list to calculate the Gini Index for each possible partition of the attribute domain. In either case, we can determine the best splitting point for each attribute list. Among all these splitting points, we choose the best one as the final splitting point and the associated attribute list as the splitting attribute list. Then, we can split every attribute list according to the splitting point, with one portion going to N_L and the other to N_R. This task is carried out by the subroutine Split.

In the subroutine Split, we first scan, in top down manner, the splitting attribute L and split it into L_L belonging to N_L and L_R belonging to N_R. For each entry scanned, we compare the

[1]In SPRINT, these counts are stored and updated incrementally in a data structure called 'histogram'.

attribute value it contains with the splitting point, and determine its destination (i.e., either L_L or L_R) immediately. Then, we split non-splitting attribute lists. Between L_L and L_R, we bring in the shorter list, say L_L, to the main memory. Then we can bring in non-splitting attribute lists L_N one by one to do the splitting for them. For each record in L_N, we see if it exists in L_L by matching the record id (*rid*). If so it belongs to N_L else it belongs to N_R. To reduce the search time, we build a hash table for L_L using the *rid* as the key. Each record in L_N will "probe" the hash table using the *rid* value.

If an attribute list cannot fit into main memory, the attribute list is divided into **buckets.** The buckets are brought in one at a time to the main memory. For the splitting attribute, each time a bucket X is brought in from L_L, a hash table is built for the bucket only. With the hash table in main memory, we bring in every bucket from each non-splitting attribute list one by one, and do the probing of the hash table to determine whether each record must go to N_L or N_R. Then we repeat the entire process with the next bucket in L_L. When the last bucket of L_L is brought into the memory, all the entries in each bucket of each non-splitting attribute list can be determined for their destinations. This essentially splits each non-splitting attribute list into two sub-lists, one for N_L and the other for N_R.

3 Post-evaluation vs Pre-evaluation of Splitting Points

In the SPRINT framework, evaluation of splitting points occurs after the attribute lists have been constructed, as indicated by the order of the related statements (i.e, preprocessing precedes step 1, and step 2 precedes step 3.). We call this way of evaluating splitting points **post-evaluation.** The post-evaluation scheme requires multiple write-back and fetching for splitting point evaluation when the attribute lists cannot fit into the main memory.

The SPRINT framework can be improved in such a way that once the attribute lists are generated at a node we can determine for further splitting the splitting attribute list and the final splitting point without incurring any extra I/O. We call this scheme of evaluating splitting points **pre-evaluation**. (The prefixes 'post' and 'pre' are with respect to writing the attribute lists to the disk.)

4 Schemes to Construct Decision Trees

In this section, we describe several schemes to construct decision trees (only for the growing phase) based on pre-sorting. All the schemes use pre-evaluations.

4.1 One-to-many hashing

This is the SPRINT approach except that the pre-evaluation of splitting points is used. As described before, in this scheme for each bucket of records in the splitting attribute list, we create a hash table and then read every bucket from each non-splitting attribute list to probe the hash table.

The reason for the phrase **one-to-many** is because the buckets of the splitting attribute are loaded into memory *one* time while the buckets of the non-splitting attributes are loaded *many* times.

4.2 Many-to-one and Horizontal hashing

A merit for one-to-many hashing is that the hash table is created only once in the memory. However, for each bucket from splitting attribute list, each bucket of a non-splitting attribute list can possibly be brought into the memory multiple times. For each load, it must be subsequently written back to the corresponding disk file. To reduce the I/O cost, we can use an alternative to do the hashing, which we call **many-to-one hashing**. Like one-to-many hashing, we divide each attribute list into buckets. However we fetch the buckets from non-splitting attribute lists first. For each of these fetched buckets, we bring in all the buckets from the splitting attribute list and create hash tables one by one. These hash tables are probed by the bucket of the non-splitting attribute list. When all the records in that bucket are resolved, we write them back to the corresponding files, and then repeat this process for the next bucket from the non-splitting attribute list. Thus each bucket from a non-splitting attribute list will be read and written only once, while each bucket from the splitting attribute list can possibly be fetched multiple times, but without involving write operations. We call this scheme many-to-one because the splitting attribute list has to be brought into main memory *many* times while the non-splitting attribute lists are brought in only *one* time.

One way of constructing a bucket from the non-splitting attribute lists is to let each bucket contain the records entirely from one list. We call the resulting scheme the **many-to-one simple hashing scheme.** The other way is to divide the bucket into k slots, where k is the number of the non-splitting attribute lists, and let each slot contain the records from one list. We call this second method **horizontal hashing.**

4.3 A scheme using Paired Attribute Lists

This scheme differs from the previous schemes in the structure of the attribute lists and the way they are split. At each internal node N we maintain a set of **paired attribute lists**. Let X and Y be two attributes. The paired attribute list for **X paired with Y**, denoted as $\langle X, Y, C, Rid\rangle$, is a list of tuples of the values for these four attributes. Furthermore, if X is a continuous attribute the tuples are listed in the ascending order of the values of X. We say that X is the **host** and Y is the **guest** in the list.

The idea behind the attribute pairing is the following. Suppose there exists another attribute list $\langle Y, Z, C, Rid\rangle$ at the same node. Then the values of Y in $\langle X, Y, C, Rid\rangle$ can reference those of Y in $\langle Y, Z, C, Rid\rangle$. If $\langle Y, Z, C, Rid\rangle$ is the splitting attribute list at the left child, this reference can facilitate the process to distribute the entries in $\langle X, Y, C, Rid\rangle$, as described below. To make it possible for any attribute to reference any other attribute at a node, we maintain a cycle of m paired attribute lists: $\langle X_1, X_2, C, Rid\rangle$, $\cdots$, $\langle X_{m-1}, X_m, C, Rid\rangle$, $\langle X_m, X_1, C, Rid\rangle$.

Now, consider how to construct the attribute lists at the two children N_L and N_R of P. Suppose the attribute lists for P have been created, and attribute X_1 has been chosen as the splitting at-

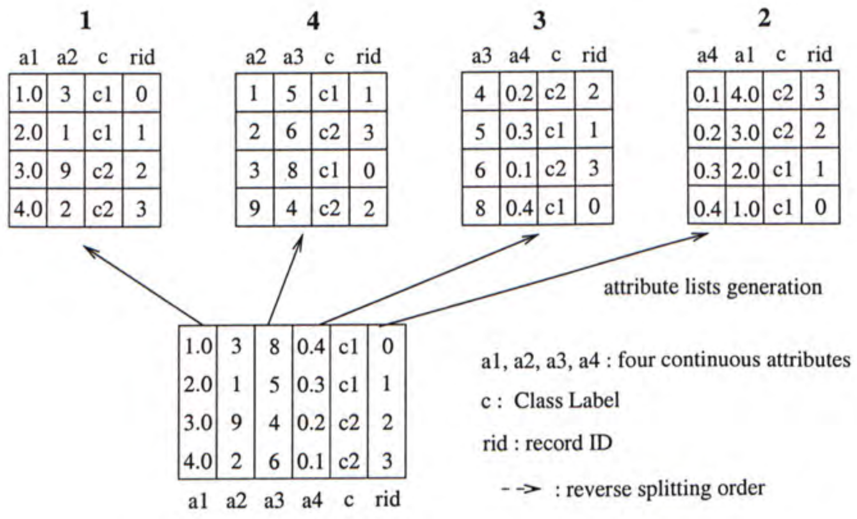


Figure 2: Splitting attribute by attribute pairing

tribute and x as the splitting point of X_1. There is no difficulty to split attribute list $\langle X_1, X_2, C, Rid\rangle$ for node N_L and N_R. Let the attribute list at N_L be $\langle X_1, X_2, C, Rid\rangle^L$ as a result of that splitting. To determine the entries that must go to node N_L (and respectively N_R) for the non-splitting attribute lists at P, we start from $\langle X_m, X_1, C, Rid\rangle$.

- First, assume X_1 is continuous. Since x is the splitting value for X_1 at node P, any entry in $\langle X_m, X_1, C, Rid\rangle$ with a value for X_1 being less than or equal to x goes to N_L and otherwise goes to N_R. Thus we scan the entries in $\langle X_m, X_1, C, Rid\rangle$ top down. For each entry scanned, we can determine where it goes by examining its value for X_1.

- Second, assume X_1 is categorical. We scan each entry in $\langle X_m, X_1, C, Rid\rangle$, and check if its X_1 value is in set x. If it is, the entry goes to N_L, otherwise it goes to N_R.

Let $\langle X_m, X_1, C, Rid\rangle^L$ be the attribute list at node N_L as a result of splitting $\langle X_m, X_1, C, Rid\rangle$. Note that this list is still sorted on the value for X_m in case it is a continuous attribute. Now consider $\langle X_{m-1}, X_m, C, Rid\rangle$. Assume X_m is a continuous attribute. Let x'_m be the largest value for X_m in $\langle X_m, X_1, C, Rid\rangle^L$. We scan the entries in $\langle X_{m-1}, X_m, C, Rid\rangle$ top down. If an entry contains a value for X_m larger than x'_m, it will surely go to N_R, otherwise we use hashing similar to the previous schemes to determine its destination.

If X_m is a categorical attribute, we use hashing for each entry of $\langle X_{m-1}, X_m, C, Rid\rangle$ to determine which child it goes to. Once the list $\langle X_{m-1}, X_m, C, Rid\rangle$ splits, we then split $\langle X_{m-2}, X_{m-1}, C, Rid\rangle$, then $\langle X_{m-3}, X_{m-2}, C, Rid\rangle$, etc., using the same procedure. Eventually, the list $\langle X_2, X_3, C, Rid\rangle$ will be split. We observe that these lists are split following the reverse order of their subscripts. We therefore refer to this order *reverse splitting order*. Figure 2 gives an example of this scheme. (The boldface numbers indicate the order the attribute lists are split, assuming the top left most list is the splitting attribute list.)

4.4 A scheme using Database Replication

We notice that when we do the reverse splitting in the attribute pairing scheme, splitting the first non-splitting attribute list is much easier than splitting other non-splitting attributes. This is because the value used for splitting that list is the splitting value of the splitting attribute list. Is it possible to use this splitting value for the splitting of all other non-splitting attribute lists? The answer is yes. We can make u copies of the entire dataset, where u is the number of continuous attributes. These copies are sorted based on the values of different continuous attributes. We use notation $\langle X_1, \cdots, X_i^*, \cdots \rangle$ to indicate that this dataset copy is sorted based on the value of X_i. (If there is no continuous attribute, a single copy must be maintained.)

At the root, all the copies are created in a preprocessing phase. As usual for any continuous attribute the best splitting point can be determined when the copy is sorted. For categorical attributes we can choose any copy to do the evaluation when that copy is in memory. Thus the splitting attribute and the splitting point are determined in a pre-evaluation of the splitting points. Now consider the children, N_L and N_R of an internal node N, assuming the existence of u copies of the datasets and the splitting attribute X and splitting point x at node N. Since every copy contains X, splitting is easy. We can simply compare the value for X in each entry with x. If X is continuous, a value smaller than or equal to x implies the entry must go to N_L, otherwise to N_R. If X is categorical, a value belonging to x implies the entry must go to N_L, otherwise to N_R.

5 Performance Analysis

We have described the following schemes:

(1) **Scheme 1**: one-to-many hashing,

(2) **Scheme 2**: many-to-one simple hashing,

(3) **Scheme 3**: many-to-one horizontal hashing,

(4) **Scheme 4**: paired attributes (one-to-many),

(5) **Scheme 5**: paired attributes (many-to-one),

(6) **Scheme 6**: dataset replication.

We also call the scheme of SPRINT **Scheme 0**. In this section, we compare the performance among these schemes. We first formulate the amount of I/O operations involved in each scheme (in the worst case behavior). Then we apply these schemes to a specific database to gain some concrete idea of their performance.

We assume the size of main memory is fixed. All the data initially resides on disk, grouped as blocks. A block is the minimum unit of transfer between the disk and the main memory. The original dataset is a table with $k+1$ fields. Listed in Table 1 are the symbols used in our derivation. Note that to simplify the derivation, we assume all the fields (including the *rid* field) are of equal sizes. Since all schemes need preprocessing, we first concentrate on the number of disk block accesses (read or write)i when splitting takes place at any particular node for each scheme in the worst case. We call this number the **cost**. After that we will derive the cost for the preprocessing.

Table 1: Notations for the parameters

notations	unit	meaning
Z	bytes	available main memory space
f	bytes	field size of a record
b	bytes	block size
n	records	size of dataset at a node
k	attri	number of predictor attributes
u	attri	number of continuous attributes
B_s	bytes	bucket size for the splitting attribute
B_n	bytes	bucket size for non-splitting attributes

- First consider Scheme 0. Let N be an internal node of the decision tree.

  (1) The attribute list for the splitting attribute contains $\frac{3nf}{b}$ blocks. To split the attribute list, all its blocks are read and then written back. This amounts to $\frac{6nf}{b}$ block accesses.

  (2) After the splitting attribute list has been split into two lists, the smaller of the two lists, letting it be L_s, will be fetched into memory to form hash tables. Therefore the greatest possible size of this list is $\lfloor n/2 \rfloor$ entries, or bounded by $\lceil \frac{3nf}{2b} \rceil$ blocks. This list will form at most $\lceil \frac{3nf}{2B_s} \rceil$ buckets.[2]

  To split non-splitting attribute lists, for each bucket of L_s brought into the main memory, all the entries of non-splitting attribute lists must be fetched. The total size of them is $\frac{3knf}{b}$ blocks. Each of these blocks is read and subsequently written $\frac{3nf}{2B_s}$ times. Hence the total cost for splitting non-splitting attribute lists is $\frac{3nf}{2b} + \frac{3knf}{b}\frac{3nf}{B_s}$. Note that the hash table created for L_s and the non-splitting bucket B_n must fit into the main memory. If H is the hash table size, then $H + B_n \leq Z$. The number of entries in the hash table should be greater than that of L_s. However, each entry in the hash table has a small size, since it only need to record the record id. In the implementation of the hash table, we can vary the utilization factor in the hash table, and typically $H = fB_s$, where f is a factor close to 1. We have

$$B_s + B_n \approx Z \tag{1}$$

  (3) After both the splitting and non-splitting attributes have been split, they must be brought to memory again to evaluate the split points. This requires another $\frac{3knf}{b}$ block accesses.

  Adding the above three values together, the total I/O cost for Scheme 0 at node N is

$$\frac{6nf}{b} + \frac{3nf}{2b} + \frac{k(3nf)^2}{bB_s} + \frac{3knf}{b} = (7.5 + 3k)\frac{nf}{b} + \frac{k(3nf)^2}{bB_s} \tag{2}$$

- For Scheme 1, we save on the third step because of the pre-evaluation. Hence, the total I/O cost is

$$\frac{6nf}{b} + \frac{3nf}{2b} + \frac{k(3nf)^2}{bB_s} = (7.5)\frac{nf}{b} + \frac{k(3nf)^2}{bB_s} \tag{3}$$

[2]To simplify our discussion, we shall ignore the floors and ceilings in similar terms in the following

- For Scheme 2, we have many-to-one hashing. First, to split the splitting attribute list, we use $\frac{6nf}{b}$ block accesses. After the splitting attribute list has been split into two lists, let the the smaller of the two lists be L_s. Let the size of L_s be $n/2$ entries, or $\frac{3nf}{2b}$ blocks. The non-splitting attribute lists amounts to $\frac{3knf}{B_n}$ buckets. Then to split the non-splitting attribute lists, L_s is brought into main memory $\frac{3knf}{B_n}$ times. Hence the cost involved in this step is $\frac{3knf}{b} + \frac{3nf}{2b}\frac{3knf}{B_n} = \frac{3knf}{B_n} + \frac{k(3nf)^2}{2bB_n}$. The total I/O cost is

$$\frac{6nf}{b} + \frac{3knf}{B_n} + \frac{k(3nf)^2}{2bB_n} = \left(6 + \frac{3kb}{B_n}\right)\frac{nf}{b} + \frac{k(3nf)^2}{2bB_n} \tag{4}$$

- The analysis for Scheme 3 is the same as that for Scheme 2.

- For Schemes 4 and 5, the cost to split each of the first two attribute lists in the reverse splitting order is $\frac{8nf}{b}$, which includes costs for both reading and writing. The derivation for the block accesses for each of the remaining $k-2$ attribute is similar to that for splitting a non-splitting attribute list in Scheme 1 and Scheme 2, except that now each record in the attribute list contains four fields, instead of three. Thus we omit the detail. The cost of Scheme 4 is:

$$\frac{8nf}{b} + \frac{8nf}{b} + \frac{4nf}{2b} + \frac{(k-2)(4nf)^2}{bB_s} = 18\frac{nf}{b} + \frac{(k-2)(4nf)^2}{bB_s} \tag{5}$$

The cost of Scheme 5 is:

$$\frac{8nf}{b} + \frac{8nf}{b} + \frac{4knf}{b} + \frac{(k-2)(4nf)^2}{2bB_n} = 20\frac{nf}{b} + \frac{(k-2)(4nf)^2}{2bB_n} \tag{6}$$

- Lastly, we consider Scheme 6. Now an attribute list for an attribute (i.e, the entire dataset sorted on the value of that attribute) contains $k+2$ fields. This amounts to $(k+2)nf$ bytes, or $\frac{(k+2)nf}{b}$ blocks. Twice this value is the cost to split any replicated copy. Assuming there are u continuous attribute lists, the total cost for splitting all the attribute lists is

$$\frac{2u(k+2)nf}{b} \tag{7}$$

Equation (7) indicates that Scheme 6 would have better I/O performance for large dataset since it is linear in n, it also requires less CPU time since no hashing is necessary, but this scheme requires much more disk space to accommodate the replicated datasets.

From Equations (2), (3), and (5), we see that for the one-to-many schemes, it is better to use a larger B_s to minimize the I/O cost. From Equation (4) and (6), for the many-to-one schemes, it is better to use a larger B_n. Since the buckets from the splitting and non-splitting attribute lists must share the limited memory space, a larger B_s (B_n) would lead to a smaller B_n (B_s). Thus if we set B_n too large, we will have a lot of overhead in bringing in small portions of records from disk and in creating a large number of small hashing tables. The CPU cost will go up. Similarly, if we set B_s too large, we will have a lot of overhead in bringing in small portions of records for the non-splitting attributes. Therefore, we expect that there is a value of B_s or B_n neither too large nor too small, where the overall performance is optimized.

The equations also indicate that the many-to-one schemes have better I/O cost than one-to-many schemes if B_s and B_n are comparable. However, the many-to-one approach requires each bucket of the splitting attribute list to be brought into memory and the corresponding hash table to be created multiple times. This will incur more CPU costs. Therefore the choice would depend on the I/O performance and the CPU performance of the system.

Schemes 4 and 5 have a probabilistic advantage. In addition to hashing, attribute values are used to resolve (i.e, find the destination of) each record of the attribute list being split. With such a double resolution method, each record in a bucket of the attribute list being split may be resolved earlier, and never later than it would be in the single resolution scheme where only hashing were used. In other words, the fraction of the buckets being brought into the memory from the splitting attribute list in the double resolution scheme is likely to be smaller than it is in the single resolution scheme.

Now consider the preprocessing. For the first three schemes, each attribute list contains $\frac{3nf}{b}$ blocks. Thus it requires $\frac{3nf}{b}log\frac{3nf}{b}$ block accesses to be sorted. Thus the cost to sort all the attribute lists is $\frac{3knf}{b}log\frac{3nf}{b}$. Scheme 4 requires $\frac{4knf}{b}log\frac{4nf}{b}$ block accesses to sort all the attribute lists. Finally, Scheme 5 requires $\frac{(k+2)unf}{b}log\frac{(k+2)nf}{b}$ block accesses to sort all copies. These results show that, except for Scheme 5, the pre-sorting costs for all the schemes are dominated by the costs for splitting attribute lists.

6 Experimental results

We use the synthetic database proposed in [2] for all of our experiments. We employ two classification functions proposed in [2].

6.1 Performance

We compare the total response time and the number of logical page access for all the schemes presented in this paper. Experiments are conducted under UNIX on a Sun Ultra 5 with 128MB main memory and 270MHz clock rate. We use a 9GB hard disk, for which the average I/O rate is 8000 blocks per second. We force each page access to go to the hard disk and hence the logical page accesses corresponds to physical page accesses. Figure 3 shows various parameters setting for our experiments. We have employed both linear probing and coalesced chaining [16] for hash operations and find that the latter incurs less number of collision.

As illustrated before, when we split a non-splitting attribute list, we bring into memory the buckets from the splitting attribute list to create hash tables. In the actual implementation, however, to make efficient use of memory space we do not need to put an entire bucket of the splitting attribute list in the memory. Instead we use a working space the size of a single block and fetch into it the blocks from the splitting attribute list one after another. For each block fetched, we insert entries into the hash table. Thus we need to put only a hash table and a bucket from non-splitting attribute list into the memory. As a result we have $H + B_n \approx Z$. To increase the hit ratio when probing the hash table, we let B_n be smaller than H. Normally the entries in the hash table are only partially used. We use *load factor* to denote the percentage of the entries that are actually used in the hash table. In the implementation, we adjust the load factor to a high value (around 80%) to make $H \approx B_s$. Thus we have

$$B_s + B_n \approx Z \tag{8}$$

Although we have 128MB main memory, we restrict the usage of the main memory to be 5MB for the hash table and the buckets. This is to simulate the case where the main memory is substantially smaller in size than the data set. Under this circumstance the scalability for each scheme can be best observed.

6.2 Test 1 : Smaller Decision Tree

The first set of experiments is conducted with Function 2, which has 4 predictor attributes, and which produces a very small decision tree. The range of total number of nodes is from 25 to 31 with different dataset sizes. We call this set of experiments **Test 1**. Our experiments confirm our expectations in the earlier analysis. Scheme 6 stands out as the best in response time and in page accesses for large datasets, since both I/O cost and CPU cost are low. It also has the linear scalability. However, the disk space requirement is much bigger than the other methods.

For the other schemes, given the fixed main memory allocation, we vary the bucket sizes B_s and B_n. We discover that the overall response time follows a U-shaped curve with an optimal minimal point. This is shown in Figure 4 (a). If we measure only the page accesses (I/O time), the performance of the many-to-one schemes would improve with the value of B_n, while that of the one-to-many schemes deteriorates. This is shown in Figure 4 (b).

Next we choose the optimal bucket allocation for each scheme and measure the response time and page accesses with varying database size. For page accesses, the many-to-one single and horizontal schemes are the second best methods, which we can predict from Equation (4). The paired attribute schemes are not as good since the attribute lists becomes bigger in size and requires more I/O costs. The proposed methods are better than SPRINT as expected. The total response time include both the CPU time and the I/O time. Since the system we use has a very high I/O performance, the effect of the CPU time is quite dominant. The one-to-many schemes are better than many-to-one since much less hashing are needed. In particular, the one-to-many paired attribute scheme is the second best. This is because there is no hashing for the first two attribute lists at each node. On the other hand, SPRINT and the horizontal scheme show very similar behavior, because in this case the SPRINT's good CPU performance is off-set by its poor I/O performance while the other way is true for horizontal scheme. We also observe that second to the database replication scheme, one-to-many scheme and one-to-many paired attribute scheme have excellent scalability in terms of the total response time. This is because the decision tree is relatively small and therefore the inferior I/O performance for one-to-many schemes is not significant. It is largely compensated by its good CPU performance.

We find that the disk usage is 1542MB and 623MB for the database replication scheme and SPRINT respectively for a 10M dataset size using function 2. If the disk usage is acceptable, the database replication scheme is superior to the other schemes.

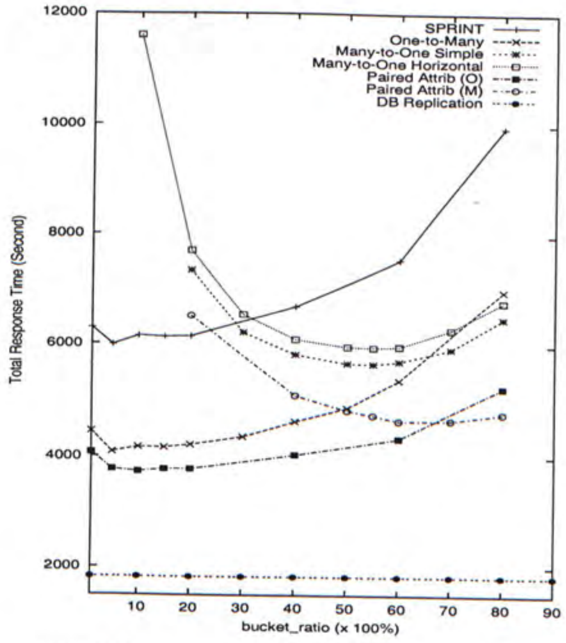
Parameters	Descriptions
page	size of a page (4K bytes)
$data\_size$	training dataset size (number of records) = 1000K, 2000K, 3000K, ..., 10000K
$record\_size$	number of bytes for a record entry in an attribute list $= 3 \times 4$ for SPRINT, One-to-Many, Many-to-One Simple and Horizontal $= 4 \times 4$ for Paired Attrib (O), Paired Attrib (M) $= n \times 4$ for DB Replication where n is the total number of continuous attributes and field size = 4 bytes
$buffer\_size$	total main memory allocated = 5MB $\geq hashSize + otherBucketSize$
$bucket\_ratio$	ratio of $otherBucketSize$ to $hashSize$ = 0.01, 0.05, 0.1, 0.15, ..., 0.9
$hashSize$	memory size required for hash table (in bytes)
$hashRow$	number of entries in hash table $= \lfloor \frac{hashSize}{8} \rfloor$ for linear probing $= \lfloor \frac{hashSize}{12} \rfloor$ for coalesced chaining
$load\_factor, \alpha$	load factor (utilization) of hash table = 50%, 60%, 70% or 80%
$splitBucketSize$	main memory size used for the splitting attribute bucket ($= B_s$ in Table 2) $= record\_size \times load\_factor \times hashRow$
$splitBucketRow$	number of entries in splitting attribute bucket $= \lfloor \frac{splitBufferSize}{record\_size} \rfloor$
$otherBucketSize$	main memory size used for the non-splitting attribute bucket ($= B_n$ in Table 2) $= buffer\_size$ - $hashSize$ (bytes)
$otherBucketRow$	number of entries in non-splitting attribute bucket $= \lfloor \frac{otherBucketSize}{record\_size+1} \rfloor$

Figure 3: Parameters setting for the experiment

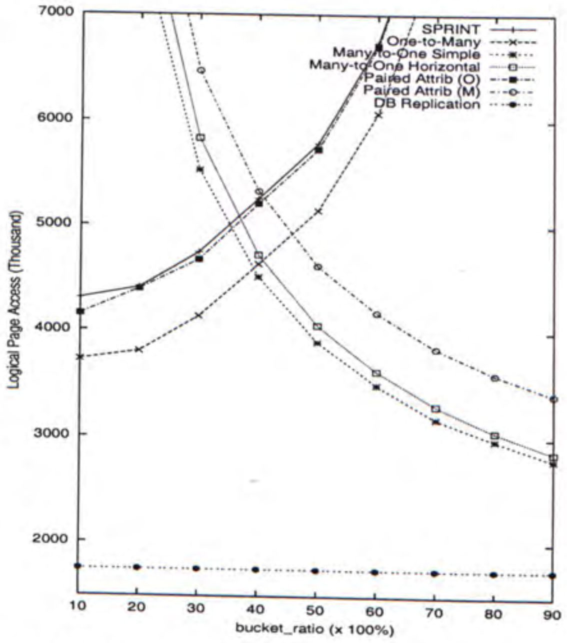
6.3 Test 2: Bigger Decision Tree

Next we experiment with Function 6 (Figure 9), which has 6 predictor attributes, and which produces a much larger decision tree. The range of total number of nodes is from 2500 to 6000 with different dataset size. We call this **Test 2.** We again carry out experiments by varying the dataset size and choosing the optimal bucket sizes in each scheme. The result is shown in Figure 6.

From Figure 6, the other schemes again outperform SPRINT. The I/O cost of the database replication scheme has linear scalability in the dataset size. The many-to-one schemes also have good scalability. (The many-to-one simple and many-to-one horizontal are nearly linear.) For the total response time, the results are somewhat different from Test 1. In Test 1, the paired attribute schemes have pretty good performance. However, in Test 2, they are the worst among the proposed schemes. This is because the number of attributes have increased from 4 to 6. The paired attributes provides advantages for the first two attribute lists at each node, but demands slightly bigger attribute list sizes. In Test 1, we provide savings to 2 out of 4 attributes, while here the ratio becomes 2 out of 6. The benefit becomes less significant while the overhead becomes more significant. The other difference is that the many-to-one schemes now have a better performance

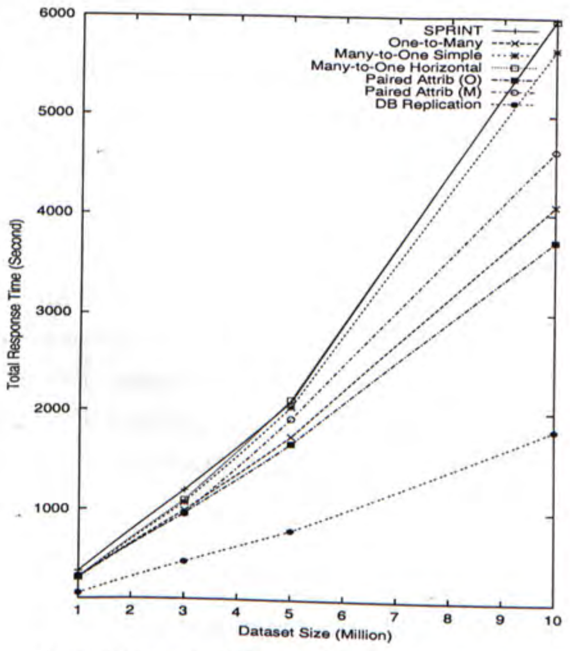


(a) Total response time by varying the bucket size

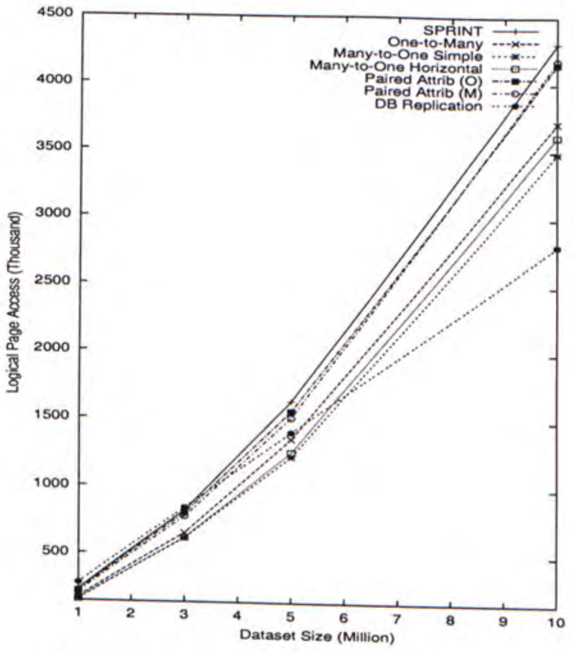


(b) Logical page access by varying the bucket size

Figure 4: dataset size=10M, total buffer size=5MB, load factor=80%, using function 2

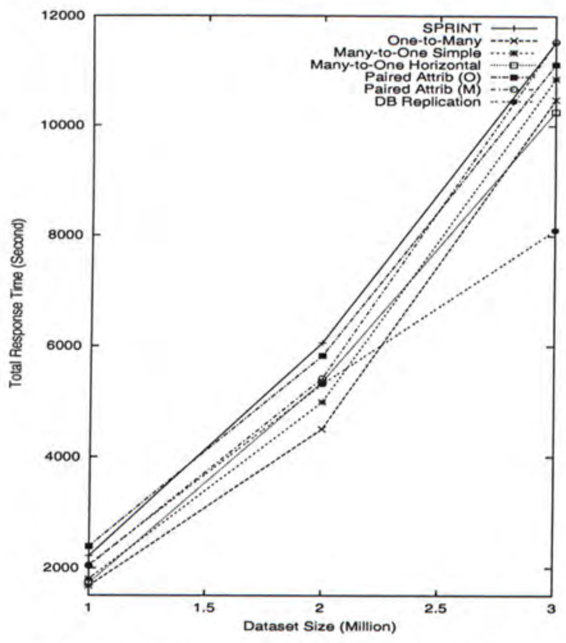


(a) Total response time with optimal bucket size

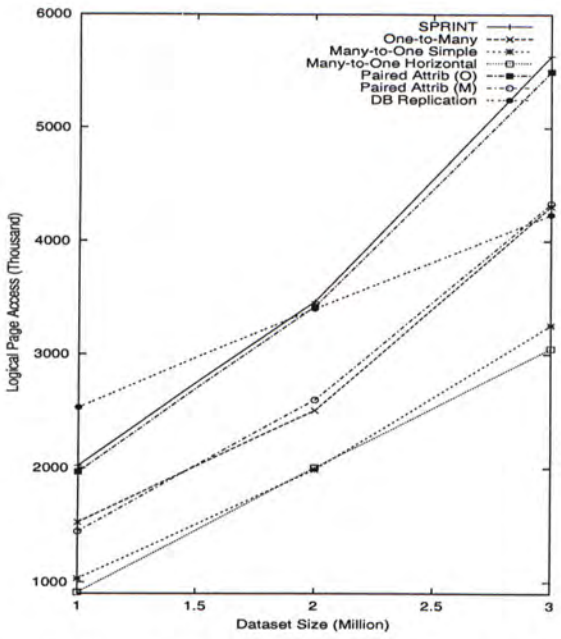


(b) Logical page access with optimal bucket size

Figure 5: total buffer size=5MB, load factor=80%, using function 2



(a) Total response time with optimal bucket size



(b) Logical page access with optimal bucket size

Figure 6: total buffer size=5MB, load factor=80%, using function 6

than they did in test 1. We explain this as follows: since our decision trees are binary, when the decision tree becomes bigger, the number of levels increases and many nodes are found at the deeper levels of the tree. Such nodes typically correspond to smaller data sets. If data sets are large, the disk access are more contiguous, and the average disk access time per page is smaller. With smaller data sets, the disk access are fragmented, and the average disk access time per page is increased. Therefore the I/O time becomes more significant when the tree is bigger. Since the many-to-one schemes win the one-to-many schemes in I/O, this results in better overall performance for the many-to-one schemes.

From the above analytical and experimental results, we conclude that the different proposed schemes are all better compared to SPRINT. The database replication scheme can be chosen given sufficient disk space. The other proposed schemes can be used in different scenarios. The many-to-one scheme has better I/O performance, while the one-to-many scheme has better CPU performance. The paired attribute scheme provides more savings if the number of attributes is small.

7 Conclusion

In this paper, we present a family of schemes that grow decision trees based on pre-sorting. We start from the framework proposed in SPRINT. Then we show how the performance can be improved by using careful design and implementation of the procedures for splitting the attribute lists. We introduce techniques for the pre-evaluation of split points. We study several methods to split the dataset, including one-to-many and many-to-one hashing, horizontal hashing, attribute pairing and database replication, and derive results relating to their performance. We also report on experimental results, providing evidence to our expectations of the new schemes.

References

[1] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proceedings of VLDB*, pages 560–573, 1992.

[2] R. Agrawal, T. Imielinski, and A. Swami. Database mining: a performance perspective. *IEEE Transactions of Knowledge and Data Engineering*, 5(6):914–925, 1993.

[3] K. Alsabti, S. Ranka, and V. Singh. Clouds: a decision tree classifier for large datasets. In *Proceedings of KDD*, pages 2–8, 1998.

[4] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and regression trees.* Wadsworth, Belmont, 1984.

[5] P. Cheeseman, J. Kelly, and M. Self. Autoclass: a bayesian classification system. In *Proceedings of 5th Int. Conf. on Machine Learning.* Morgan Kaufman, June 1988.

[6] J. Gehrke, V. Ganti, R. Ramakrishnan, and Wei-Yin Loh. Boat–optimistic decision tree construction. In *Proceedings of ACM SIGMOD*, pages 169 – 180, 1999.

[7] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest - a framework for fast decision tree construction. In *Proceedings of VLDB*, pages 416 –427, 1998.

[8] D. Godberg. *Genetic algorithms in search, optimization and machine learning.* Morgan Kaufmann, 1989.

[9] M. Mehta, R. Agrawal, and J. Rissanen. Sliq: a fast scalable classifier for data mining. In *Proceedings of fifth Int. Conf. on EDBT*, March 1996.

[10] Y. Morimoto, T. Fukuda, Matsuzawa H., Tokuyama T., and Yoda K. Algorithms for mining association rules for binary segmentations of huge categorical databases. In *Proceedings of the VLDB*, 1998.

[11] J. Quinlan. Discovering rules by induction from large collections of examples. In *Expert Systems in the micro Electronic Age*, 1979.

[12] J. Quinlan. Induction of decision trees. In *Machine Learning*, pages 81 – 106, 1986.

[13] J. Quinlan. *C4.5: programs for machine learning.* Morgan Kaufmann, 1993.

[14] B. Ripley. *Pattern recognition and neural networks.* Cambridge university Press, Cambridge, 1996.

[15] J. Shafer, R. Agrawal, and M. Mehta. Sprint: a scalable parallel classifier for data mining. In *Proceedings of the 22nd VLDB*, pages 544 – 555, 1996.

[16] F. Daniel Stubbs and W. Neil Webre. Data structures with abstract data types and modula-2. pages 340–363. Brooks/Cole Publishing Company, 1987.

[17] H. Wang and C. Zaniolo. Cmp: a fast decision tree classifier using multivariate predictions. In *Proceedings of the 16th ICDE*, pages 449 – 460, 2000.

[18] S. Weiss and C. Kulikowski. *Computer systems that learn: Classification and prediction methods from statistics, neural nets, machine learning and expert systems.* Morgan Kaufmann, 1991.