DESIGN AND IMPLEMENTATION VERIFICATION OF FINITE STATE SYSTEMS

by

Rajeev Kumar Ranjan

Memorandum No. UCB/ERL M97/99

18 December 1997

DESIGN AND IMPLEMENTATION VERIFICATION OF FINITE STATE SYSTEMS

by

Rajeev Kumar Ranjan

Memorandum No. UCB/ERL M97/99

18 December 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Design and Implementation Verification of Finite State Systems

Rajeev Kumar Ranjan

Please cite as

Rajeev Kumar Ranjan.
Design and Implementation Verification of Finite State Systems.
PhD thesis, University of California, Berkeley, 1997.
Available as UCB/ERL M97/99.
`http://www.eecs.berkeley.edu/~rajeev/publications/publications.html`

Abstract

Design and Implementation Verification of Finite State Systems

by

Rajeev Kumar Ranjan

Doctor of Philosophy in Engineering
University of California, Berkeley
Professor Robert K. Brayton, Chair

WITH the increasing complexity of VLSI circuits, design and implementation verification have become important components in current day design flows and can have major impact on the timely delivery of a functionally correct product. This work investigates a spectrum of techniques targeted towards making the verification process practical for real designs. The key contributions can be divided into three parts.

The first part exploits the architecture of a computer system for efficiently manipulating Binary Decision Diagrams (BDDs), the core technology in the symbolic techniques. Various methods are presented to localize the memory accesses and thereby leverage off the highly different access times for different levels of memory hierarchy in a workstation. These are further extended to exploit the main memory of several workstations connected together in a network. It is also shown that the locality of breadth-first manipulation can be merged with the parallel computing power of a shared-memory multiprocessor to efficiently leverage the parallel architecture.

The second part presents efficient BDD-based schemes for the representation and traversal of the state-space of large designs. Key contributions in this area include efficient image and pre-image computations, the core tasks in symbolic synthesis and verification algorithms.

The last part of this work targets the sequential equivalence problem which in its most general form is much harder than the combinational equivalence problem. An algorithm is presented to reduce the sequential equivalence problem to an extended

form of combinational equivalence problem for designs which have undergone iterative retiming and resynthesis transformations. This allows the sequential equivalence checking process to leverage off the advancements in the combinational equivalence domain, making it viable for large designs. Further, the optimization potential and the verification complexity of optimization transformations consisting of retiming and combinational synthesis are formally characterized.

R. K. Brayton

... to Renu

Table of Contents

List of Figures

List of Tables

List of Definitions and Theorems

Preface

THIS thesis has evolved with an aim towards providing efficient techniques targeting the functional verification (i.e., whether the design is specified correctly) and implementation verification (i.e., whether the design is implemented correctly) of finite-state systems.

The emphasis of the solutions proposed in this work has been more on the practical aspects, in particular their usability and capacity. Keeping usability in mind we have mainly focused on automatic techniques for verifying systems. The underlying motivation behind our work on efficient manipulation of BDDs comes from the fact that most of the current automated design verification algorithms use BDDs as basic data structure.

To establish the capacity of the proposed techniques, we have performed comprehensive experiments with reasonably big instances of designs. In particular, we have been able to build very large BDDs in a order of two less time compared to the state-of-the-art techniques. Our technique for sequential circuit verification can verify sequential equivalence of two large ISCAS benchmarks in few minutes.

A significant amount of effort was put in the software implementation aspect of various techniques presented in this thesis. Most of them have been successfully implemented and the software is available in public domain. In particular, our BDD package, named CAL, has also been adopted inside a commercial EDA tool. The package is general enough to be integrated inside any BDD-based system.

I was also involved in the development of the verification tool, VIS, as one of the primary architects. The work on state-space traversal described in this thesis constitutes a fundamental part of model-checking engine of this tool. Over the last 2 years, VIS has been adopted by over 500 people all over the world and has been incorporated as a core engine inside some commercial model checking tools.

—Rajeev Kumar Ranjan
Berkeley, California.
December 1997.

Acknowledgements

WHEN I was making plans to write my thesis, I thought that the acknowledgment section would be the easiest one to write. After all, there is a standard format – first of all acknowledge your advisor, other members of your qualifying examination committee, some other faculty members / colleagues you collaborated with, the funding sources – followed by the acknowledgment of your friends for their good company during the long stay for Ph.D. – and finally some sentimental lines for spouse / partner / parents. Mine too would pretty much follow the similar outline. However, what made this process difficult was my intention of succinctly capturing the true essence of my interactions with friends, colleagues, people, and the environment at Berkeley.

In any case, here I go ...*

First and foremost, I would like to thank my advisor, Prof. Robert K. Brayton, for his guidance, inspiration, encouragement, and support during my graduate years at Berkeley. I consider myself very fortunate to have worked with Bob. His ability to patiently listen to "uncooked ideas" and suggest useful directions is remarkable and so are his perseverance, commitment to work, intellectual curiosity, and technical depth. His constant enthusiasm for everything that the graduate students were involved in is what kept us motivated to achieve higher and higher goals.

I would like to thank Prof. Alberto Sangiovanni-Vincentelli for being the chair of my qualifying examination committee and for his valuable comments on the draft of my thesis. I would like to thank the third reader of this dissertation, Prof. Ilan Adler from IEOR department, for timely review of the thesis. Prof. Paul Hilfinger from Computer Science department and Prof. Adler agreed to serve in my qualifying at a very short notice. I am thankful to them for their consideration.

The continuous sources of funding during the last 5 years have been essential for my survival as a graduate student. First of all, I would like to acknowledge the fellowship

*As a true researcher and to pre-empt any criticism of plagiarism, I would like to mention that similar sentiments have been expressed in the past. In particular [Mur93] and [Swa96] (I got paid to refer to these theses) are worth mentioning.

from Motorola under the URP program. I would also like to acknowledge the funding from California Micro, Fujitsu, and Cadence.

I would like to mention Prof. M. A. Pai from University of Illinios at Urbana-Champaign. He was my advisor at U of I and it was due to his undivided attention to my program that I could finish my M.S. thesis and publish a couple of conference and journal papers, all in less than a year. He always supported me in all my decisions including my transfer to Berkeley. Even after I came to Berkeley, he was always concerned about me and would keep track of my progress throughout my Ph.D.

I take this opportunity to thank my alma-mater Indian Institute of Technology, Kanpur. Getting a Bachelors degree from IIT Kanpur was a turning point in my career, since it opened up a multitude of exciting opportunities.

During my stay at Berkeley, I got a chance to collaborate with a lot of smart people and had the privilege to learn from them. I would like to take this opportunity to express my thanks to my research collaborators.

The work on breadth-first BDD manipulation was done with Jagesh Sanghavi. From him I learned the technique of developing complex software packages using "unit-level debugging". It was with this philosophy that we were able to develop and release an industrial-strength BDD package. Wilsin and I collaborated on the dynamic reordering work inside the CAL package. Often he and I would get together to discuss other research issues. The work on state-space traversal started at Motorola, Austin, where I was an intern in the summer of 1994. I collaborated with Bernard Plessier and Carl Pixley and later with Adnan Aziz. The last part of this dissertation (sequential circuit verification), is the result of my internship at Cadence Berkeley Labs. There I collaborated with Vigyan Singhal and Prof. Fabio Somenzi from University of Colorado at Boulder. I thank them for putting in long hours in brain-storming sessions and discussions.

It was a pleasure being part of the VIS team and I would like to acknowledge the members: Adnan Aziz, Szu-Tsung Cheng, Stephen Edwards, Sunil Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Sriram Rajamani, Tom Shiple, Gitanjali Swamy, and Tiziano Villa. It was with the group chemistry and team spirit that we managed to put together a high quality model checking tool in just about six months. The success of this project is evidenced by over 500 downloads from various parts of the world in

the last 2 years. I would also like to thank them for their support and collaboration in organizing the VIS technology-transfer course. It was a great learning experience.

During my stay at Berkeley, going through the myriad of academic requirements would have been very difficult, had it not been for an excellent set of administrative staff to help us. I would like to express my heartful appreciation to the staff at the graduate office. Heather Brown (now Levien), Ruth Gjerde, Mary Byrnes, and Pat Hernan have done an excellent job in patiently dealing with the graduate students' problems and explaining the various intricacies of departmental requirements on a day-to-day basis. I would also like to acknowledge the CAD Group staff for assisting us in various reimbursements, registrations to conferences, business travel, etc. In particular, I would like to mention Flora Oviedo, Kia Cooper, Elise Mills, Gwyn Horn, and Pearl. The last I heard, I was Baby # 3 in the Flora's favorite babies' list. I guess I was bumped down since I moved to the South Bay. During my work at Cadence Berkeley Labs, I solicited the assistance of our administrator Kris Lamanno. She is by far the best sys-admin I have come across. I would remember her for her superb efficiency in managing day-to-day activities at the labs and for her sensitive gesture in throwing an impromptu birthday party for me. Thanks to our system administrators Brad and Judd who have done an excellent job in maintaining our huge network of workstations.

I would like to acknowledge all my friends / colleagues / seniors who made work environment a fun place (at 550 Cory, Cadence Berkeley Labs, Synopsys).

Tom Shiple was one of the first few senior graduate students I met in the CAD group. Starting from the days of the logic-synthesis course project for which he was my mentor, we became very good friends. I have always been amazed at his methodical ways in life. I have learned a lot from him. He always found time to listen to my problems – academic or personal – and provide advice as a friend, colleague, or mentor.

I have known Fabio for over 3 years, but it was only during his sabbatical at Cadence Berkeley Labs that I got a chance to closely collaborate with him. Besides him being a true BDD guru, I was quite impressed with his finesse and expertise in Perl, Emacs, deep understanding of ISCAS/MCNC benchmarks, to name a few. Just recently, during ICCAD he asked a question after one of the paper presentation - "How come s1269 is not in the table of results?" I think most of the people in the audience would have been amazed, I was.

"Pre-doctoral qualifying examination" – also known as "prelims" – was the first stressful academic experience I had to go through at Berkeley. I would like to thank my preparation team members – Yuji Kukimoto, Serdar Taşıran, and Harry Hsieh. It would have been difficult going through the whole bunch of conference and journal papers (some of them very boring and irrelevant) without sharing the workload.

Yuji, Serdar, Harry, and I have been good friends since the beginning of graduate years. I admire Yuji's sense of responsibility and ability to deliver on time. So far I have not met anyone else who would consistently turn in the DAC/ICCAD submissions on the advertised date. Serdar was kind enough to bring "Turkish Delights" after every visit to his native country, Turkey. At times, he provided us accompaniment on guitar in our informal singing sessions. Harry enjoyed eating Indian food, and we enjoyed watching him eat. He would love eating spicy food while sweating all over his body.

I would like to acknowledge my cubicle mates. Sriram Krishnan always had passionate ideas on all issues – religion, politics, research – to name a few. Often his outbursts at the tennis court and politically incorrect statements used to be our source of entertainment. Adrian Isles has been a good cubicle mate to me. I was truly impressed by his ability to work long hours in Cory Hall and to combine community service with that. I have known Gitanjali Swamy since IITK days. When she was in the cubicle, she provided us a lot of entertainment in various controversial discussions on feminism, Indian politics, etc., especially those with Sriram Krishnan. After moving to Boston she would call us West coast folks at 7:00am Pacific time. She has done this with me many times and she would start with – "Hope I did not wake you guys up...". Regardless, it is always a pleasure chatting with her.

I also had the privilege of sharing my office space with Dominique Borrione from Grenoble, France. It was great fun celebrating "Bastille Day" at her place. The best part was when people from different countries sang their national anthems. Earlier this year, I had an opportunity to go to Grenoble to teach a VIS tutorial. I will remember Dominique for her hospitality, inviting me over for French cuisine, taking time off to take me around Grenoble and vicinity areas, taking me to the music festival, and not to forget, helping me out in shopping for Renu (my wife).

I had many useful discussions with Desmond Kirkpatrick on various computer related issues – emacs, tcsh, Intel's performance – to name a few. It was refreshing to

talk with him about my research, since he always brought in a new perspective. I will remember Chris Lennard for his Australian ways (come to think of it, he IS Australian !!). I remember the time when he threw a party and he and his fellow band members played some rock numbers for the guests. Stephen Edwards was a good source of knowledge about various computer-related things. Arrogant as his attitude may be, we did not hesitate calling him over for dinner and getting some fundamentals on issues like java vs. javascript vs. hotjava. It would always be time effective to know about these things from him (taking into account cooking time, etc.).

Rajeev Murgai, an "All Time Berkeley Resident", has been a very good friend. His pet peeve with me has been my login name – "rajeev". I would often receive emails from his friends inviting me for some delicious dinner and such and sometimes unpleasant notes from his bosses (hmm... am I lying here?). Anyways, I will remember him most as a wonderful host in parties and for his passion for ballroom dancing. Marco Sgroi was our "Maestro Italiano". I thank him for taking the initiative to teach Italian to some of the enthusiasts in the CAD group. Shaz Qadeer and Amit Mehrotra provided company during bridge sessions. I thank Drs. Alpa and Jagesh Sanghavi for their wonderful Diwali parties which typically involved innumerable delicious dishes. Thanks also to Premal Buch, Luca Carloni, Wilsin Gosti, Harry Hsieh, Sunil Khatri, Gurmeet Singh Manku, Amit Narayan, Roberto Passarone, Mukul Ranjan Prasad, Subarna Rekha, for making 550-Cory a lively place.

Thanks to Vijay (wife) and Kamal (husband) for being wonderful hosts at Berkeley, Santa Cruz, and now in Milpitas. Thanks to Zeina and Enrico for all the wonderful times we have had at various parties (surprise or otherwise) and not to mention for throwing the annual barbecue parties. Satyajit Ranganathan (aka Satya) has been a good friend of mine for over a decade now (boy, do I sound old ...). He and I share many common interests – movies, Hindi songs, football, philosophical discussions about life – to name a few. At times, he has given me advice on various issues which helped me see things more clearly. Charutosh was always a good host when we would go to his big house in Mountain View (that used to be our weekend getaway). I was always impressed with his dedication to Hindustani Classical music.

It was one of the rare instances in Cory Hall (may be it was first), that Renu and I were doing Ph.D. together and were concurrently occupying desks in 550 Cory for

about 5 years. As a result, I had a chance to interact with many of Renu's close friends and sometimes take their sides in troubling her.

Varghese George (hmm... just now I realized that I always called him by his last name) was my tennis Guru at Berkeley. Whenever we would play doubles, he would pick me as partner even though that would mean losing all the sets. It was fun chatting with him about latest movies. Marlene Wan was always a cheerful victim of my continuous small-time pranks. After moving to South Bay, whenever I would go up to Berkeley and meet her in Cory Hall, she would greet me in her unique way – a scream, followed by a long "hellooooooo", and then "Rajeeeev". This picture would not be complete without Jeff Gilbert. I would refer to the "Jeff" and "George" duo as "Jeff George" (for the football deficient, the only quarterback to use expletives for his coach on national TV during a game, not that they ever did the same to their advisors). I will remember Jeff for his wittiness and for his ability to act innocent after playing pranks on Marlene and Renu. Kathy Lu has been a good friend since the time I was a T.A. for one of her courses.

During the last phases of this dissertation, I was working at Synopsys. The folks at Synopsys were quite supportive of me in this effort. In particular, J.C. Madre prepared coffee on a regular basis and helped me out in locating documentation-related softwares. Others who wished me luck at one point or other include – Dinesh Ramnathan, Jim Kukula, Randy Allen, Tony Ma, Kurt Keutzer, Tom Shiple, Robert Damiano, Stephanie Aitken, Theresa Botha, Ramsey Haddad, Joe Hutt, and Narendra Shenoy.

Like many Indian students, I learned cooking only after coming to US (rather after coming to Berkeley). After couple of years of trial and error, I learned how to cook what I think is reasonably well. More than cooking and enjoying the meal afterwards, I took pleasure in feeding other people. It always gave me satisfaction when people ate with good appetite and with gratifying pleasure. The list of people who I would like to acknowledge includes (not quite in order) - Satya, Stephen, Gitanjali, Gurmeet, Rajeev (the other one). Vice versa, I would be quite disappointed if the guest did not follow the above protocol. This list contains just one person who shall remain nameless.

Walking through the Sproul Plaza on a sunny afternoon would always enlighten my day. I remember that I would deliberately delay my departure to school on Friday mornings so as to be able to see and hear the unique "Berkeleyan Sounds of Music". In

particular, I was highly enchanted by the so called "Piano Man" playing piano endlessly under his pink umbrella. The UC Berkeley octet's performance on Wednesdays used to be a treat. I was always amazed by their vocal numbers combined with some innovative sounds serving as accompaniment in perfect harmony. The Sanponia (a variation of flute) band from Peru who played Quitus music from South America (I am not making this all up, I actually took notes one day) used to be a regular presence at the intersection of Bancroft and Telegraph. It was a pleasure listening to their numbers played on Sanponias of various sizes. And not to forget the lonesome drummer, Larry Hunt, drumming away (sometimes on a set of buckets) on the upper plaza, day in and day out. He got a notice from campus authorities against his "noise-making practice". I was quite happy to sign in his support book. And last but not the least was the the Irish Celtic Harp player, Aryeh Frankfurter. He would play just beside Sather gate towards Wheeler Hall, oblivious to all the commotion at Sproul Plaza. With all these distractions, it would often be the case that the twenty minutes walk from home to Cory Hall would turn into an hour or more.

Finally, it's time to get a tissue handy.

I thank my parents, brother Sandeep, and sister Rimjhim for having faith in me for all these years.

Above all, I would like to express my gratitude to my wife Renu Mehra. Words cannot express my feeling. We have been through a great deal together and I am indebted to her forever for being so supportive of me through thicks and thins of life. Without her help, encouragement, and criticism I would not have been what I am today. I thank her for her constant love and support and for being so patient with me. Renu spent innumerable hours helping me prepare for preliminary exam, qualifying exam, and various conference talks. Needless to say, she has played a crucial role in my being able to finish this dissertation on time. Without the millions of red marks made by her on various drafts, it could not have taken this shape today. I dedicate this thesis to her.

Chapter 1

Introduction

As we move towards 21st century, integrated circuits (ICs) are becoming an integral part of our day-to-day lives, being embodied in various forms – microprocessors in home computers, embedded controllers in automobile fuel-injection systems, graphics controllers in video games, micro-controllers in toasters, answering machines etc., automated data-acquisition and manipulation components in bio-medical systems, etc. These complex electronic systems have requirements on their functionality, speed, and reliability. Of these three, functional correctness is the most fundamental requirement because the speed and reliability of an incorrectly functioning electronic system is of no interest. Moreover, of these three requirements, functional correctness is becoming the largest bottleneck in design [Keu96].

It has been widely estimated that over 70% of the design time for integrated circuits is spent in performing various kinds of verification tasks and the effort devoted to this process eclipses all other aspects of the design process. In addition, this problem is likely to become increasingly worse due to the following two reasons:

1. The first one is the result of phenomenal growth in the design complexity. Transistor density increases by about 50% per year, quadrupling in just over three years. The increasing number of transistors leads to larger number of components to analyze in the verification problem making it more and more complex and harder to solve. Also, with the growing complexity of the digital systems, the complexity of protocols interacting between components and ensuring the security and reliability of data being passed amongst them is also increasing at a rapid rate. These factors combined together significantly affect the performance demands on the verification technology.

   Another way to look at the affect of increasing design complexity on verification is by examining the following equation that gives the number of bugs per

chip [Keu96]:

$$\frac{bugs}{chip} = \frac{logic\_transistors}{chip} \times \frac{lines\_in\_design}{logic\_transistors} \times \frac{bugs}{lines\_in\_design}$$

Assuming that number of logic transistors per line of the high-level code is constant, since the number of logic transistors doubles every 18 months, we must reduce the number of bugs per line of HDL by half every 18 months to ensure the correctness of the design.

2. The second factor is the shrinking time to market. It is common to turn around an application specific integrated circuit (ASIC) design with few hundred-thousand gates within a couple of months. Timely delivery of a product is the key to its success and imposes additional performance demands on the verification technology.

These factors combined together have stressed the capabilities/capacity of functional verification as never before.

1.1 Goals and Scope of the Thesis

The goal of this thesis is to investigate a spectrum of techniques targeted towards making the verification process practical for real designs. Later in this chapter we will precisely identify the nature of verification algorithms we are targeting in this work. Note that we are only concerned with the computational aspects of functional verification. We do not address the issue of the specification of the system or the correct behavior. Also, the scope of this work includes finite state systems only. There are similar (and more complex) verification problems in the area of infinite state systems (e.g., hybrid systems).

Chapter Organization: In Section 1.2, we will review some trends in the area of computer architecture and allude to areas of interest from the point of view of this work. In Section 1.3, we describe various notions of verification and also provide the context of our work. Sections 1.4 and 1.5 discuss various issues and solution approaches to BDD-based verification techniques. In Section 1.6, we give background on sequential circuit verification. Finally, we outline the organization of this thesis in Section 1.7.

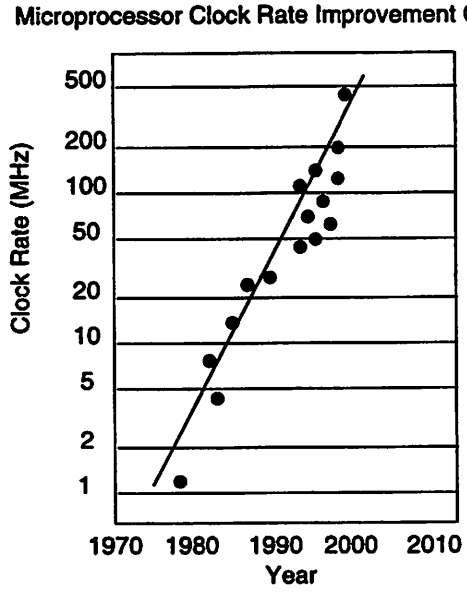


Figure 1.1 Clock rate improvement for microprocessors over last 27 years.

1.2 Computer Architecture Trends

We first analyze various components of computer architecture – microprocessor, memory, disks, etc. – to better understand how we can leverage from each of them to obtain efficient algorithms for functional verification.*

1.2.1 Microprocessors

Over the last decade, microprocessor performance has grown at a tremendous pace. The contributing factors to this growth are the following:

- The clock rates for the leading microprocessors increase by about 30% per year. In Figure 1.1, we show the improvement in the clock frequency for several important microprocessor families. In particular, the clock rate for x86 family from Intel has increased from 16MHz in 1986 for i386 to 300MHz in 1997 for Pentium-II.

*Some of the material in this section has been derived from [HP90, San96, CSG97].

- Transistor density: The rate of increase of number of transistors in a chip is about 40% per year. This leads to increased raw computing power per year.

- Architectural trends: To feed the increasing computing power of microprocessors a lot of architectural advancements have been made. Some of the major contributors are – instruction level pipelining, superscalar instruction issue, out-of-order execution, speculative execution, etc.

- Software technology: Compiler technology attempts to extract the total available parallelism for a specific computer architecture by loop optimization, software pipelining, and scheduling techniques.

These factors combined together lead to a significant rate of increase in performance of microprocessors on standard benchmarks. SPEC integer performance has been increasing at about 55% per year and SPEC floating-point performance at 75% per year.

1.2.2 Memories

One of the physical laws of computer architecture is that fast memories are small, large memories are slow. This occurs as a result of many factors, including the increased address decode time, delays on the increasingly long bit lines, small drive of increasingly dense storage cells, and selector delays. This is why memory systems are constructed as a hierarchy of increasingly larger, slower, and less expensive (per byte) memories further away from the processor. A simplified memory hierarchy consists of processor registers, several levels of on- and off-chip caches (SRAM), main memory (DRAM), and a hard disk. The important characteristics of each memory system component are given in Table 1.1. A large hierarchical memory system provides fast access on average as long as the references exhibit good locality.

Main Memory (DRAMs)

The main memory satisfies the demands of caches and serves as the I/O interface. Performance measures of main memory emphasize both latency and bandwidth. The DRAM capacity has quadrupled every three years due to finer line-widths, larger chip area, and advances in the design of basic DRAM cells. However, the performance of

Level	Registers	Cache	Main Memory	Disk Storage
Typical Size	< 1KB	< 4MB	< 4GB	> 1GB
Access time (in ns)	2-5	3-10	80-400	10,000,000
Bandwidth (in MB/sec)	4000-32000	800-5000	400-2000	4-32
Managed by	Compiler	Hardware	Operating System	OS/User
Backed by	Cache	Main memory	Disk	Tape

Table 1.1 Typical levels in memory hierarchy.

DRAMs is growing at a much slower rate. Figure 1.2 shows a performance improvement in row access time of about 7% per year.

To compensate for the low rate of access time and cycle time improvement for standard DRAMs, innovative operating modes, novel memory architectures, and application-specific DRAMs have emerged [KOKD96]. Despite these advances, for memory accesses that show little or no correlation, the access time remains the measure of DRAM performance that characterizes the main memory performance.

Secondary Memory (Disks)

Disk capacity has grown at an enormous rate in the last ten years as shown in Figure 1.3. In 1986 the largest commercial disk had a capacity of 20MB. These days we find disks with capacity up to 8GB. However, the access time has improved by only one-third in 10 years. The average price per megabyte of magnetic disks has reduced from $35 per megabyte in 1986 to $0.1 per megabyte in 1997.

1.2.3 Microprocessors vs DRAMs: Performance Gap

In Figure 1.5, we have shown the comparison between the performance trends of microprocessor and DRAMs. The trend shows a growing disparity between memory and processor performance. This is because memory performance has increased at less than 10% per year whereas processor performance has increased at about 50% per year since 1986. Several researchers predict that memory bandwidth will limit the performance of the future microprocessors.

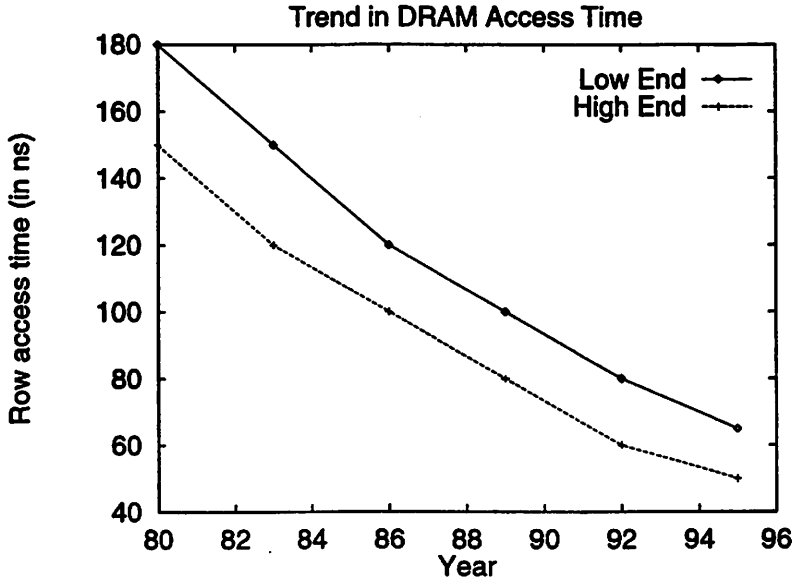


Figure 1.2 DRAM access time improvement over years.

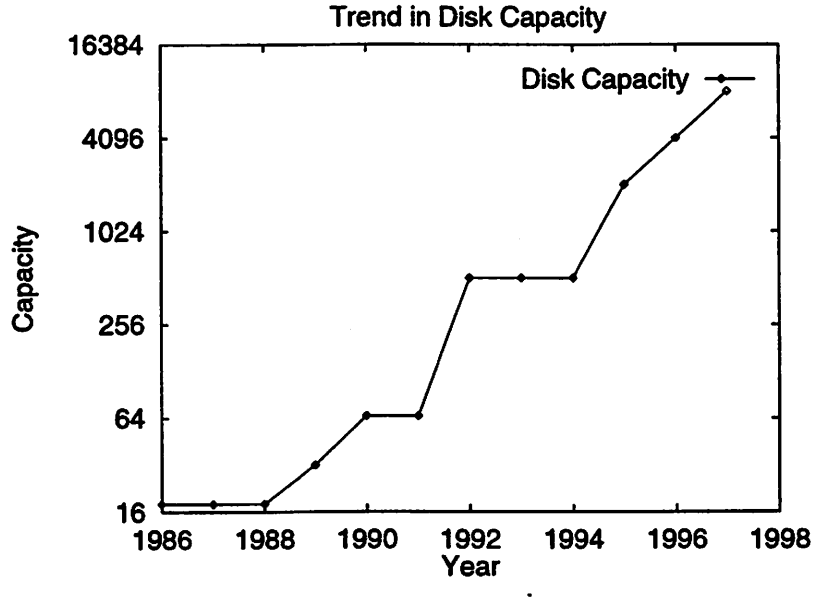


Figure 1.3 Trend in disk capacity over the last decade.

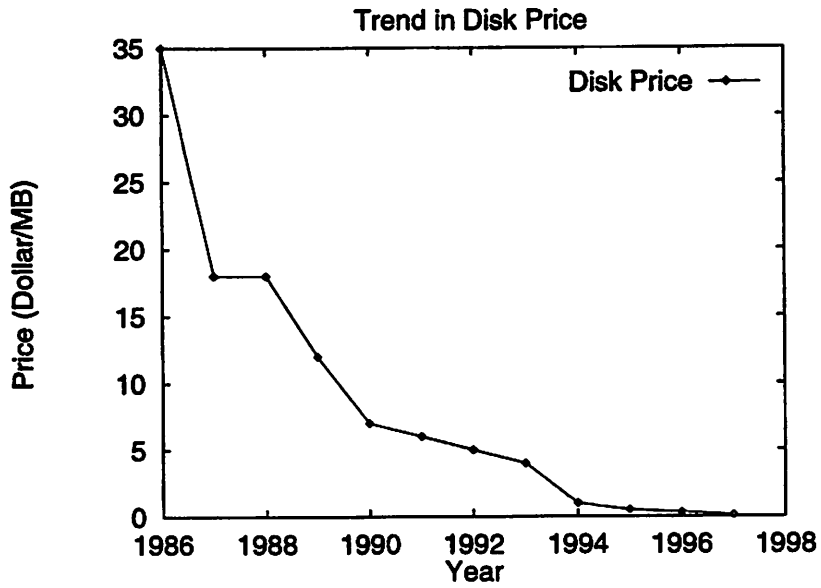


Figure 1.4 Trend in disk price over the last decade.

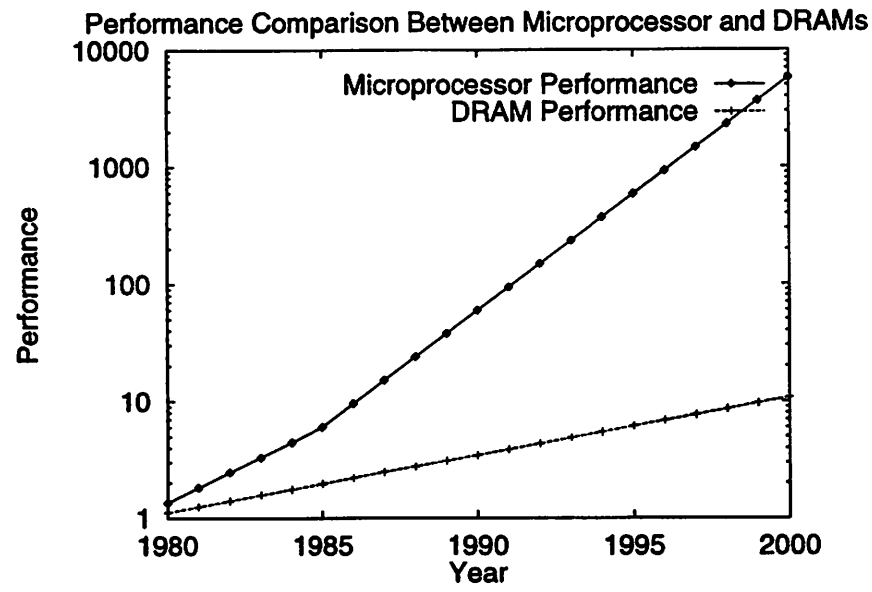


Figure 1.5 Performance trend comparison of microprocessors and DRAMs.

The disparity between DRAM and processor performance is also evidenced by extensive use of caches. In 1980, most of microprocessor designs did not have caches. In 1996, most of microprocessor designs have two levels of caches. However, for application that exhibit little or no correlation between addresses, the caches serve no useful purpose and the microprocessor may spend 75% of all CPU cycles waiting for memory accesses.

1.2.4 Disks vs DRAM: Price vs Performance

In Figure 1.6, we have plotted the capacity ratio of disks vs DRAM per constant dollar. It shows the trend in how much more disk capacity comapred ot the DRAM capacity can be purchased for the same amount of money. The ratio has increased from about 8 in 1986 to about 160 in 1997 indicating that on average the rate of decrease in the price of unit disk capacity has outpaced the rate of decrease in the price of unit DRAM capacity by a factor of 20.

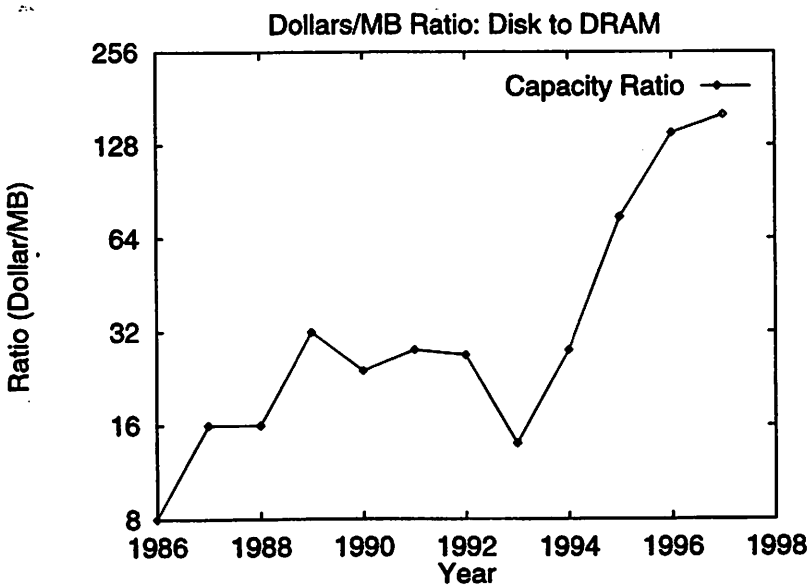


Figure 1.6 Disk to DRAM capacity ratio per constant dollar.

1.2.5 Parallel Computing

In response to increasing chip capacity various functional units have been replicated to increase the parallelism available in the processor. In order to utilize the parallelism in the hardware, superscalar execution, instruction-level pipelining, and out-of-order execution have emerged as mainstream technologies.

Given the expected increases in chip-density, the natural question to ask is how far will instruction-level parallelism go and at what point will the emphasis shift to thread-level parallelism. Studies have shown that a 2-way superscalar architecture is very profitable and 4-way offers substantial additional benefit, but wider issue widths, e.g., 8-way superscalar, provide little additional gain. The design complexity increases dramatically since on average control transfers occur once every five instructions. Recent works provide empirical evidence that to obtain significantly larger amounts of parallelism, multiple threads of control must be pursued simultaneously.

Also, the performance of the highly integrated, single-chip CMOS microprocessor is steadily increasing and is surpassing the larger, more expensive alternatives. The advantages of using small, inexpensive, low power, mass-produced processors as the building blocks for computer systems with many processors is now greater ever than before. In particular, the shared-memory multiprocessor appear to be a promising direction that could have a wide impact. We are beginning to see the emergence of parallel processing in the mainstream of computing as two to eight procesor shared-memory multiprocessors.

1.2.6 Computer Architecture: Conclusions

From the analysis of the technology and architectural trends, we derive following conclusions which will serve as guidelines for our effort to develop computer architecture based efficient functional verification techniques.

- A large hierarchical memory system provides fast access on average as long as the references exhibit good locality.
- The performance gap between a microprocessor and DRAM is increasing rapidly over the years. Hence maintaining locality of reference of memory accesses will become more and more critical.

- Although DRAM prices have reduced over the years, disk prices have fallen at a much faster rate. Therefore the efficient usage of disks will prove to be economically advantageous.

- The ever increasing growth in the transistor density on a chip will serve as the enabling technology for parallel computers. These days, 2- to 4-way shared-memory multiprocessors have become a common presence on desktops.

1.3 Design and Implementation Verification

To put our work in perspective, let us consider the first few levels in a typical top-down design methodology (illustrated in Figure 1.7). Consider the synthesis steps shown on the left in Figure 1.7. The design starts with some formal or informal specification that results from somebody's idea about what the design is supposed to do. This specification is manually translated to a register transfer level (RTL) description in high-level language like Verilog or VHDL. A hardware compiler is used to synthesize this to a gate-level design. The gate-level design can in turn be further optimized by manual or automated transformations. The optimized gate-level net-list is placed, routed, and converted to layout (not shown in the figure).

The verification steps corresponding to the various stages of the design flow are shown on the right side of Figure 1.7. First of all, we need to establish if the manual description in the Verilog or VHDL satisfies the specification we started with. We can think of the specification consisting of a "set of properties", and we would like to check if our described design implements these properties. This verification problem is known as design verification or property verification.

Next, we need to verify that the gate-level description has the same functionality as the RTL description. This verification problem is known as implementation verification, since we are checking if one description correctly implements the other. Implementation verification is needed to check if the gate-level optimizations maintain the functionality of the original design. Similar implementation verification steps are required at lower levels in the design methodology.

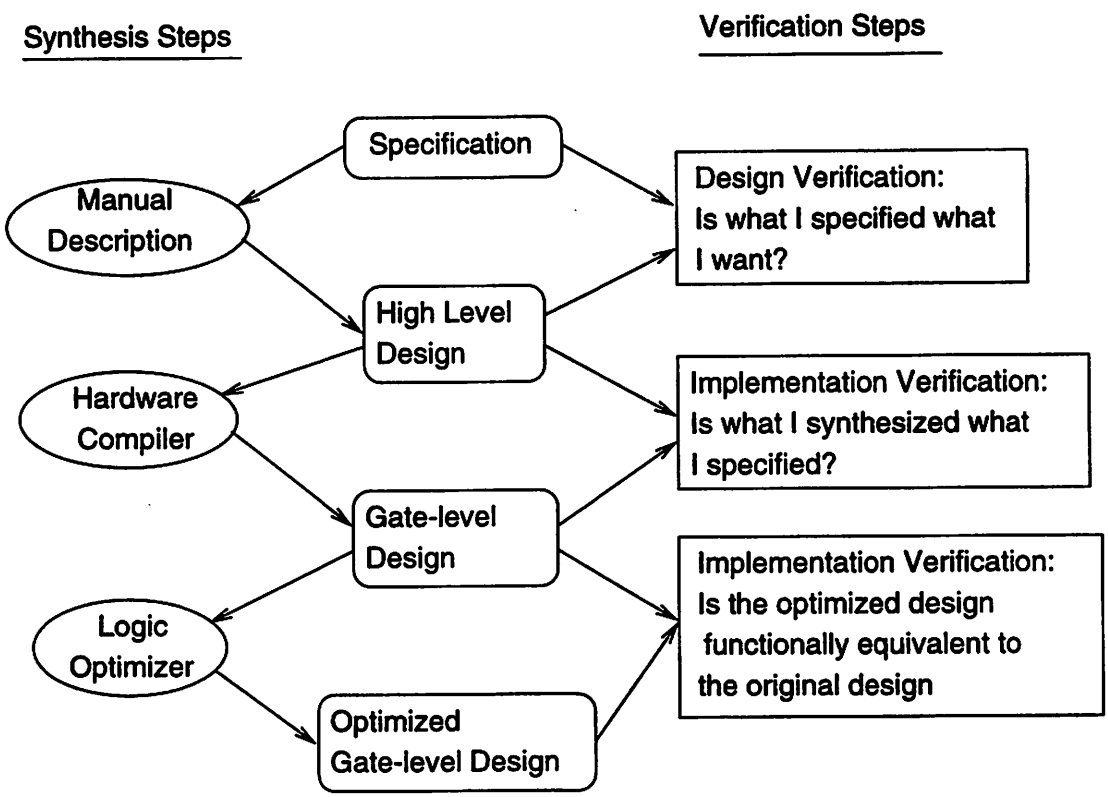


Figure 1.7 Typical top-down design methodology.

Verification Methodology

The methodology used to verify designs has evolved from an event-driven simulator to the use of a plethora of different techniques to reduce the simulation time and manage the design complexity. An overview is given in Figure 1.8. We briefly discuss few of these technologies.

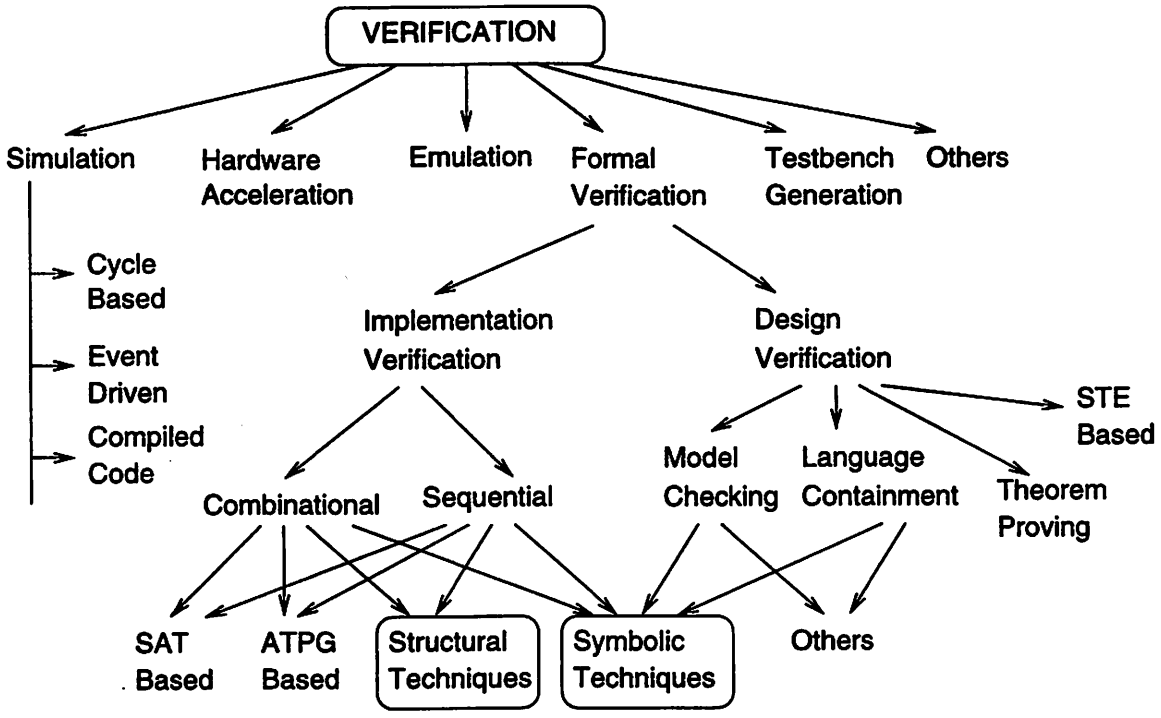


Figure 1.8 Overview of verification methodology.

In simulation, a set of inputs are applied to a model of the system and the outputs are checked for the correctness. Simulation still forms the backbone verification technology in the current day design methodology. For complex designs, however, exhaustive simulation exploring all behaviors is infeasible.

In formal verification, rigorous analytical techniques are used to establish the relationship between a mathematical model of a system and the system specification.

Formal verification can be applied for both implementation verification and design verification.

Of the various formal verification methods, *symbolic simulation* is closest in spirit to current engineering practice. In a symbolic simulation of a circuit, initial values and inputs are given not as Boolean values (0 or 1), but instead as symbolic variables. At each simulation step, the simulator computes the values of signals as Boolean functions of these variables, rather than as definite Boolean values. The Boolean functions obtained at the outputs of a circuit can be compared against the desired functions [Bry87]. This technique is primarily used to verify the combinational equivalence of two designs.

Model checking is another method of property verification that is somewhat more abstract than symbolic simulation. In this case, the desired properties to be verified are written in some form of mathematical logic which supports temporal relation between the signals (also known as temporal logic [Pnu86]). Specifications of temporal properties may also be described using finite automata instead of temporal logic. In either case, a *model checker* translates the implementation model (the design) into a finite state system, which is given by sets of states (defined by the values of memory elements in the design) and the transition amongst the states based on the input. The model checker then checks automatically that the specification is satisfied [CES86]. In the last couple of years few commercial model checking tools have emerged – Formal Check from Lucent Technologies, RuleBase from IBM, CheckOff from Abstract Hardware Ltd. to name a few.

The most general and powerful methods of verification are based on general purpose *theorem provers*. A theorem prover is based on a logic – a formal language for stating mathematical propositions. A logic is equipped with a proof system – a set of axioms and inference rules that make it possible to reason in a step-by-step manner from premises to conclusions. Most theorem provers are interactive, requiring guidance from the user in order to generate proofs [McM94]. Due to this reason, theorem-provers have not achieved the broad level of acceptance despite their impressive demonstration in some government pilot projects [Kur97].†

In this thesis we will focus on the design verification ("Is what I specified what I wanted?") and the gate-level implementation verification ("Is the optimized gate-

†For a survey on various formal techniques, refer to [Gup92].

level design functionally equivalent to the original gate-level design?"). We notice that a large number of automated verification methods make use of symbolic techniques as the core engine. Also, structural techniques are heavily used for implementation verification. Our work specifically targets these two techniques as indicated by the boxes in Figure 1.8.

1.4 BDD-based Verification Methodology

In most automated verification techniques, the designs are modeled as finite state machines where the states are defined by the values of the latches in the circuit and the transitions among the states are determined by the input values. The verification is performed by appropriately traversing the states of the circuit.

A Binary Decision Diagram (BDD) is a graph-based data structure used for representing Boolean functions (refer Section 2.1 for a detailed description). In a BDD-based verification set-up, entities (design behavior, sets of states, etc.) are represented as BDDs and appropriate BDD manipulation is done to perform verification of the design.

In Figure 1.9, we present various BDD-based verification techniques and identify the core operations and data structures.

Cycle-based simulation using decision diagrams is emerging as a new technology for fast simulations of large designs. An example is [MSSS95] which uses the Multi-valued Decision Diagram (MDD) [SKMB90], which is a wrapper around the BDD.

BDDs play an important role in the implementation verification of combinational circuits, since they provide a canonical representation of the functions. Many successful combinational verification techniques use BDDs as the basic data structure [RWK95, Mat96, KK97, JMF97]. For sequential circuits, in addition to representing the behavior of the circuit, we need to represent and traverse the state transition graph (STG). States are represented and manipulated using BDDs. State enumeration forms the core operation in this case.

For the two automated ways to perform design verification – model checking and language containment – the core operation is state enumeration, which is based on BDDs as mentioned above.

From the above analysis we observe that *BDDs* form the core data-structure for ver-

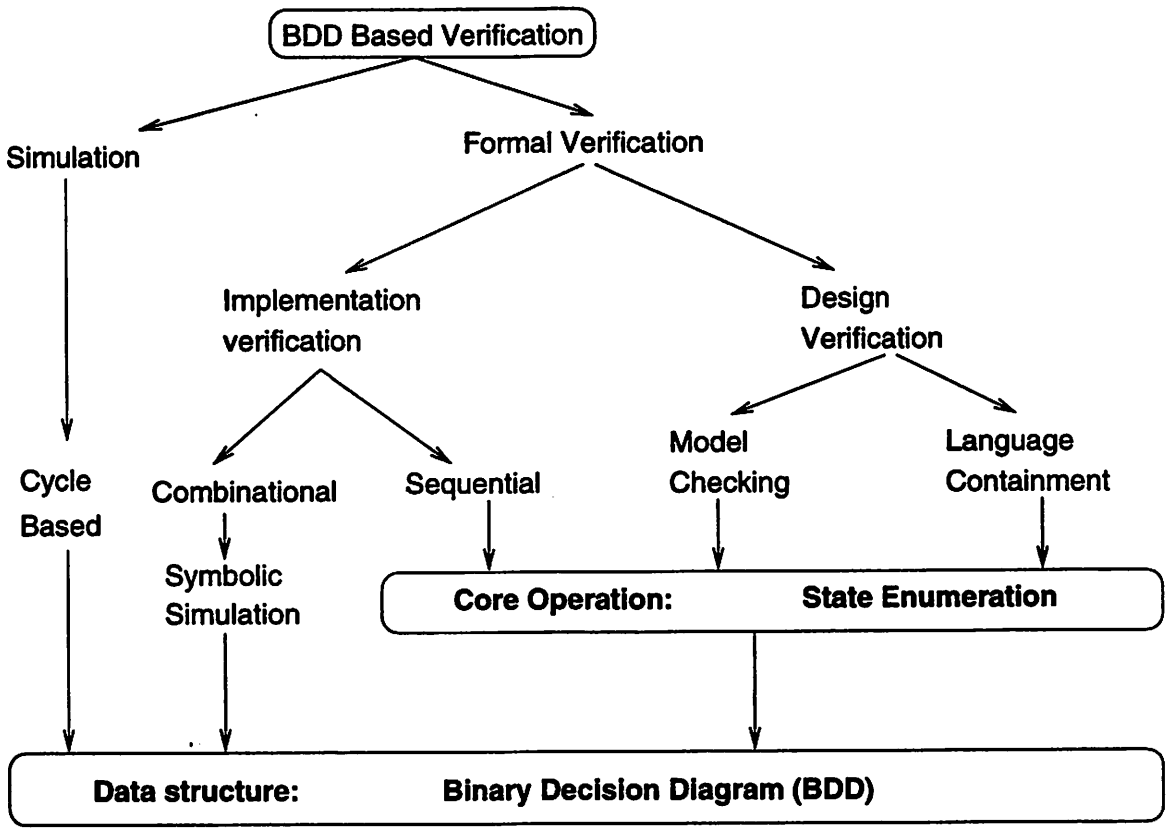


Figure 1.9 Overview of BDD-based verification methodology.

ification of all digital circuits – combinational or sequential, and *state enumeration* is the core operation for design/implementation verification of sequential circuits.

1.5 BDD-based Techniques – Problems and Solution Approaches

The two main characteristics of BDD-based algorithms are:

Memory: The algorithms are memory intensive and involve little computation. For a complex design the size of the BDD (the number of BDD nodes) gets too large to fit in the memory. Due to organization of the memory hierarchy in a typical computer, manipulating large BDDs results in large memory access time.

Computation: Even though computational complexity (in both space and time) of BDD operations is linear in the product of the BDD sizes of the functions being manipulated, for large BDDs, even polynomial complexity becomes excessive.

During the last decade or so, researchers have investigated various approaches to overcome these limitations. These can be classified under several categories described next.

1.5.1 Computer Architecture Based Solutions

Computer architecture has been used in two ways to improve BDD manipulation.

Exploiting memory hierarchy: When the size of a BDD exceeds the main memory, BDD nodes are swapped to the hard disk. The conventional depth-first BDD manipulation algorithm results in random accesses to the memory leading to a large number of page faults. Since a page access time is of the order of tens of milliseconds, a large number of page faults leads to excessive wall clock time, even though the time spent by the processor doing useful work is quite small. Ochi *et al.* [OYY93] proposed the breadth-first implementation approach to regularize memory accesses, which leads to fewer page faults. As a result, BDDs of very large size (up to 12 million nodes) can be handled. Ashar *et al.* [AC94] have presented an improved breadth-first algorithm, which enables manipulation of BDDs with up to 100 million nodes.

Parallel computation: Kimura *et al.* [KC90] have presented a parallel algorithm to construct BDDs that uses a shared-memory multiprocessor to divide the tasks that can be performed concurrently on several processors. Shared-memory machine allows the use of a single global hash table to maintain canonicity. Ochi *et al.* [OIY91] have proposed a breadth-first manipulation approach that uses a vector processor to exploit the high vectorization ratio and long vector lengths by performing a BDD operation on a level-by-level basis. Several other researchers have made use of distributed shared memory architecture [PSC94, SB96] and data-parallel architecture [CGRR94].

1.5.2 Application Specific Solutions

In this approach, particular domains are targeted and their characteristics are exploited to limit BDD sizes and intermediate BDD size blow up. Some examples are:

Combinational verification: The goal in BDD-based combinational verification is to create BDDs for the outputs of the designs to be compared and then check for equality. However, the size of the output BDDs grows large for a complex design. The BDD size can be controlled by creating a cutset in the circuit and treating the intermediate signals as the primary inputs [Mat96, KK97]. This requires appropriate handling of false negatives. The other technique is to introduce intermediate variables while building the BDDs and appropriately compose them at the end [JNC+96]. This approach prevents the blowup in the intermediate BDD sizes.

Sequential verification: In BDD-based sequential verification the goal is to represent the behavior of the finite state machine and to traverse the corresponding state-transition graph. Partitioned transition relations are used instead of monolithic transition relation to control the BDD sizes [TSL+90, BCL91b, RAP+95]. The state-space traversal, which requires image/pre-image computation as the core operations, can be made efficient by proper use of early variable quantification [TSL+90, BCL91b, RAP+95, GB94, KHB96]. Approximation techniques can also be used for reachability analysis [CHM+93, CHM+94, RS95, SRSB97] where exact analysis is not feasible.

Design verification: Partitioned transition relations and efficient image computation techniques used for sequential implementation verification are also applicable in design verification. In addition, appropriate abstraction of the original design can be used to prevent the blow-up in BDD size [Gra94, Kur93]. Also, compositional techniques can be used to break the original problem into tractable sub-problems to keep the BDD size small [GL91].

1.5.3 Algorithmic Solutions

In this approach, various algorithmic techniques are used to control BDD size of the functions obtained at the end of some manipulation. In addition, some techniques are used to control the BDD size of functions which represent partial results in a complex computation. In particular:

Variable ordering: The size of the BDD is critically dependent on the variable order. In some cases, there could be an exponential difference in the BDD sizes of a function for two different variable orders. Finding the optimal variable ordering is co-NP complete. A lot of research has been done on this issue [MWBS88, FFK88] and recently dynamic variable ordering [FMK91, Rud93] has emerged as enabling technology in this area.

BDD partitioning: This approach is similar to the partitioned transition relation method or to creating a cutset in the network. However, the techniques proposed in this category are application independent. In particular, techniques are proposed to represent a function as a set of BDDs [NJFS96, BW97].

Avoiding intermediate BDD computation: In many BDD applications (symbolic simulation, reachability, etc.), some intermediate BDDs are obtained on the way to computing the final result. In some cases, the sizes of the intermediate BDDs could be large even though the final BDD size is small. Some techniques are proposed to avoid the computation of intermediate results. In [HDB96], new variables are introduced (called operational variables) which capture the desired operation to be performed on the operands. Then these variables are successively moved down in order until they reach the bottom. The final result is obtained by appropriately manipulating the pointers at the bottom and performing a reduction

operation. Even though no intermediate results are computed in this approach, the size of the whole BDD when the operational nodes are dynamically shifted down can be large.

In other approaches [SBS93b, Hor96], a technique is proposed to compute the result of a Boolean expression by extending the traditional two-operand operations. By maintaining an array of operands and operations, one can perform depth-first traversal while computing the new operands and finding the terminal conditions along the way. In this approach the difficulty lies in the complexity of detecting the terminal conditions. Also in some cases, the overall memory consumption can increase due to the overhead of maintaining arrays and large caches to store the results.

1.5.4 Solutions Based on Modification of Decision Diagrams

In the last decade, research in the area of decision diagrams (DDs) has resulted in various kinds of DDs resulting in an alphabet soup [Bry95]. Different decision diagrams target reduction in DD size, simpler DD manipulation, etc. However, in general there is a trade-off between representation size and the manipulation complexity. Some methods of generating different decision diagrams are given below:

Changing decomposition method: By changing the interpretation of the decomposition method one can obtain a significantly smaller representation. Some of them are OFDD [KSR92] and OKFDD [DST+94].

Relaxing the ordering requirement: For some functions all variable orderings result in exponential BDD sizes in the number of variables [Bry91]. In [ADG91], the decision diagram is modified to allow variables to appear (possibly multiple times) in different orders for different paths. However, with this generalization, the canonicity of BDDs is lost and the algorithmic complexity of some common operations is exponential. A slightly less generalized version allows different variable orderings along different paths but requires the variables to appear at most once along any path. Even though efficient manipulation techniques were presented in [GM94, SW95], this representation has not yet gained popularity.

Representing numeric valued functions: For functions of Boolean variables with non-Boolean ranges, a series of representation schemes have been proposed. In particular, allowing arbitrary values on the terminal nodes (MTBDD [CMZ+93], ADD [BFG+93]), incorporating numeric weights on the edges (EVBDD [LS92]), changing the function decomposition with respect to its variables (BMD and *BMD [BC95]), or a combination of some of the above techniques (HDD [CFZ95]) have been proposed. These techniques differ in their representation sizes and the manipulation complexity.

In general, any approach which targets BDD size, thereby reducing the memory consumption will also reduce the computation time, since the operand sizes, which determine the computation complexity, will reduce. Better variable ordering falls under this category. Different decomposition schemes, and new decision diagrams however do not necessarily reduce the computation time since in some cases the corresponding manipulation complexity is much higher than for BDDs. Some approaches keep the BDD sizes under control while increasing the number of BDD computations (e.g., network partitioning [Mat96, KK97, JNC+96], partitioned transition relation [TSL+90, BCL91b, RAP+95], etc.). These approaches trade-off memory consumption with computation time.

BDD-based techniques are quite general and they are applicable to a broad range of problems. Since the underlying computation model for BDD-based techniques is a finite state machine, these techniques cannot leverage the structural information in the design. In some applications, it is sometimes advantageous to make use of techniques which can efficiently exploit such information. In the next section, we discuss one such application – sequential circuit verification.

1.6 Structural Technique Based Sequential Circuit Verification

Often it is required to make modification in the gate-level design description to achieve certain objective, such as, engineering change orders, iterative refinements, re-synthesis, retargeting to a different technology, optimization, test insertion, design reuse, etc. It is important that the functionality of the design is preserved across these changes. Se-

quential circuit verification refers to checking two sequential designs for equivalent functionality. More specifically, given two designs C_1 and C_2, we want to verify if for any arbitrary input sequence, the output sequence produced by the designs is equal. Verifying this input-output equivalence is PSPACE-complete citeaziz93d.

Traditional simulation based methods apply a set of random input vector sequences and check the equivalence of the designs for those sequences. However, for a complex design this approach fails to provide any meaningful coverage of possible input sequences.

A number of formal techniques based approaches have emerged in the last decade. A popular approach is to "compose" the given designs together by appropriately connecting the inputs and outputs. The composed design is modeled as a finite state machine and starting from some initial state (defined by the values of the latches), all the states are visited. At each state the equality of the outputs is checked. Explicit state enumeration technique visits one state at a time [DMN88, DDHY92]. However, due to the explicit nature of this technique, it is limited to only a small number of state elements. Symbolic techniques, which model the sets of states and the transitions between them as Boolean functions, have been widely used [CBM89, Kuk89, BCMD90, TSL+90]. A salient feature of these techniques is that the size of the underlying data structure (some form of decision diagram) does not depend on the number of states or the state elements in the circuit. However, the capability of the state-of-the-art symbolic methods falls below the smallest size designs being optimized in industry.

All of the above solutions attempt to solve the general sequential equivalence problem. However, due to complexity of the problem, they are either limited to relatively small size circuits or to circuits which have undergone relatively fewer optimization transformations.

The second approach is to trade the optimization capability with the verification complexity. In this approach, the sequential optimization is constrained in order to reduce the verification complexity. In particular, by making all the latches observable, the sequential synthesis reduces to combinational optimization leading to combinational verification problem. Solution proposed in [AGM96] falls in this category.

We propose a practical verification technique for transformations which include arbitrary combination of retiming and combinational optimization operations on a con-

strained form of the circuit. In particular, we require certain constraints to be met on the feedback paths of the latches involved in the retiming process. For a general circuit, we can satisfy these constraints by fixing the location of some latches, e.g., by making them observable. We show that implementation verification after performing repeated retiming and synthesis on this class of circuit reduces to a combinational verification problem. Our methodology can also be viewed as offering another point in the tradeoff curve between constraints-on-synthesis versus complexity of verification [AGM96].

1.7 Thesis Organization

This thesis is organized into three parts. In the first part of the thesis (Chapters 3, 4, 5, and 6), we present a set of computer architecture based techniques which target the efficient manipulation of BDDs. In second part of the thesis (Chapter 7), we present some application specific techniques which target state transition graph representation and state-space traversal of digital systems. Complexity issues in retiming and synthesis transformations and a practical algorithm for sequential circuit verification are described in the third part (Chapters 8 and 9).

In Chapter 2, we present the preliminary material and definitions for terminology used in this thesis. In particular, the background on BDDs will be necessary to understand the material in Part I. Familiarity with finite state machines and various state transition graph related manipulations will be helpful for the material in Chapter 7.

In Chapter 3, we discuss new algorithms for BDD manipulation that exploit the memory hierarchy by reorganizing the overall computation. The algorithms described in this chapter extend the ideas presented by Ochi *et al.* [OIY91, OYY93] and Ashar *et al.* [AC94]. The new techniques based on the iterative breadth-first algorithm enable manipulation of very large BDDs by localizing the memory accesses. The main contributions of this chapter are i) new data structures and memory management techniques, ii) techniques to exploit the memory hierarchy across several BDD operations, and iii) a comprehensive set of high performance algorithms for different BDD operations. In cases where the BDD size exceeds the main memory capacity, we have found performance improvement of a factor of more than 10 compared to state-of-the-art packages (Long's [Lon93] and CUDD [Som97]).

In Chapter 4, we present distributed algorithms on a network of workstations that

use a collection of main memories to improve the performance of the BDD algorithms and a collection of disks to manipulate BDDs that exceed the disk capacity on one workstation.

In Chapter 5, we propose techniques to improve the BDD manipulation performance by using parallel architectures. After discussing previous work on parallel manipulation of BDD nodes, we identify the key elements needed for a successful parallel implementation of a BDD package. We establish that by combining the locality of access of a breadth-first manipulation approach with the parallel computing power of a shared-memory multiprocessor, one can achieve a high degree of performance over a conventional BDD package.

In Chapter 6, we address the problem of dynamic reordering, which is an indispensable feature of a general-purpose BDD package. We propose techniques to preserve the locality of reference during reordering. After identifying the computational and memory overheads associated with implementing variable swapping (the core operation in dynamic reordering) in breadth-first based packages, we propose techniques to handle these problems. We show that the dynamic reordering performance inside a breadth-first manipulation based package can be competitive with a state-of-the-art conventional depth-first based packages.

Chapter 7 describes the second part of our work. We investigate application-specific solutions for BDD-based verification algorithms. In particular, we look into the state-transition graph representation and state-space traversal of finite-state systems. We establish that the core operation in BDD-based state-space traversal is that of forming the image and pre-image of a set of states under the transition relation characterizing the system. For efficient computation of these core operations, we present several techniques including clustering of transition relations, ordering of clustered relations for early variable quantification, network partitioning, usage of don't cares, and removal of redundant latches.

In Chapter 8, we present a theoretical analysis of synthesis optimization potential and the corresponding verification complexity of various retiming and combinational synthesis transformations. In particular we make an attempt to give formal notions for the optimization capability of retiming and resynthesis operations. Also, we formally establish the computational complexity of corresponding implementation verification

problems. Our goal is to benefit from these observations in establishing practical retiming and resynthesis logic optimization and verification methodologies.

In Chapter 9, we investigate the sequential verification problem for circuits which have undergone repeated retiming and combinational synthesis transformation. We present a practical algorithm for verifying equivalence of two sequential circuits one of which is obtained from the other using a constrained form of repeated retiming and combinational synthesis. We demonstrate that our methodology covers a large class of circuits by applying it to a set of benchmarks and industrial designs.

Finally, Chapter 10 summarizes the work and outlines directions for future work.

Almost all of the algorithms presented in this thesis have been implemented and experimental results are given in the corresponding chapters. In particular, the work on breadth-first BDD manipulation has resulted in a comprehensive public domain BDD package – CAL. In Appendix A, we describe some of the software engineering aspects of CAL and its integration with synthesis and verification tools. In addition, our work on efficient state-space traversal and has been integrated inside the verification tool VIS [BSA+96a] as one of the core engines. In Appendix B, we briefly discuss the VIS package and integration of our work inside it.

Chapter 2

Preliminaries

In this chapter we will preview some background material required to understand various chapters of this thesis. The knowledge of BDD (Section 2.1) will be essential for understanding Chapters 3, 4, and 6. The terms and definitions used in Chapter 7 are described in Sections 2.3 and 2.4. The notation and terminology used in the work on sequential verification (Chapters 8 and 9) is described in Section 2.2.

2.1 Binary decision diagrams

The origin of Binary Decision Diagram (BDD) goes back to the seminal paper by Akers [Ake78], in which Boolean functions were represented by decision graphs. However their widespread usage has started only after 1986, when a set of algorithms were proposed to construct and manipulate these data structures [Bry86]. For a complete description on various kinds of decision diagrams and the related terminology, please refer to [BRB90, Bry91, DB97]. Here we present a brief description of the data structure and the relevant terminology.

The BDD of a function is a rooted, directed, acyclic graph. To illustrate the creation of a BDD for a function, let us first look at the ordered binary decision tree of a function, $f = x_1x_3 + x_2x_3$. The ordered binary decision tree for this function is shown in Figure 2.1. The tree consists of two types of nodes: terminal and non-terminal. Each *non-terminal node* is labeled by a variable such that node labels on each path from root to leaf node satisfy a particular order given by the ordering of variables (the variable ordering for this example is $x_1 \prec x_2 \prec x_3$). Each non-terminal node has two out-going edges (pointing to left child-node and right child-node) which correspond to a truth assignment of 1 and 0 respectively to the associated variable. The child node corresponding to the "1" assignment to the associate variable is also known as THEN cofactor (also referred to as T). The child node corresponding to the "0" assignment to

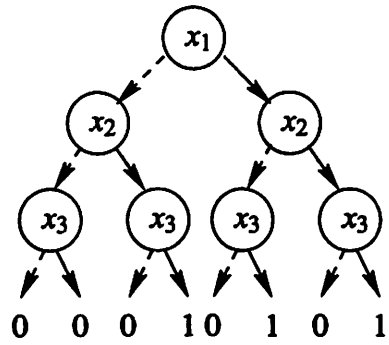


Figure 2.1 BDD tree for function $f = x_1x_3 + x_2x_3$. The dotted edges indicate the 0 assignment to the variable associated with the node.

the associate variable is also known as ELSE cofactor (also referred to as E).

A *terminal node* does not have any out-going edge and represents either a constant ZERO or constant ONE. Along every path from root to leaf, each variable appears exactly once in the order specified by the variable ordering. The binary decision tree has exponential size in the number of variables. Two reduction rules are applied to this tree representation to make it compact:

1. The *redundant nodes* (nodes with same left and right child nodes) are eliminated.

2. The *isomorphic nodes* (nodes with identical sub-tree and variable labeling) are merged.

By repeatedly applying these two reduction rules, we obtain a directed acyclic graph shown in Figure 2.2. This graph is known as Reduced Ordered Binary Decision Diagram (ROBDD). In the rest of this work, we will use the term "BDD" to refer to "ROBDD". Some important characteristics of this functional representation are given below:

1. Although the size of a BDD can be exponential in the number of variables, functions corresponding to real circuits do not typically exhibit that behavior (a well known exception being the multiplier function).

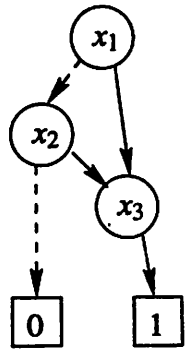


Figure 2.2 BDD graph for function $f = x_1x_3 + x_2x_3$.

2. The size of a BDD is a critical function of the ordering of variables. For some functions, the BDD size can vary from exponential to linear in number of variables for different variable orders. However, obtaining the optimal order of variables for a given function is a hard problem.

3. BDDs are canonical representation of Boolean functions. With proper implementation, the canonicity leads to constant time equality checks (satisfiability, tautology, or equivalence) making them highly suitable for formal verification of digital systems.

In Table 2.1 we compare the efficiency of representing and manipulating functions using BDDs with other data structures (prime-irredundant sum-of-products and multi-level network). In the last decade, extensive research has been done on variations of BDDs and the corresponding efficiency of representation and manipulation. A good survey can be found in [Bry95]. Below we briefly describe some of the terminology used in this work. For further details, refer to [BRB90].

Unique table: The unique table stores all the BDD nodes and facilitates the canonical representation of a function. For a given variable order, each Boolean function can be canonically represented by the top variable of the function and its two child nodes. The unique table facilitates this process by maintaining a hash table

Operation	Complexity		
	SOP	Multi-level	BDD
$f+g$	NP	$O(1)$	$O(\|f\| \cdot \|g\|)$
$f \equiv 0$	$O(1)$	NP	$O(1)$
$f \equiv 1$	$O(1)$	co-NP	$O(1)$
$f \equiv g$	NP	NP	$O(1)$
Size	Exponential	Exponential	Exponential

Table 2.1 Comparing manipulation complexity and expressiveness of BDD, sum-of-products and multi-level function representations.

of BDD nodes. Before creating a node for a function an associative look-up, based on the value of the child nodes, is performed in the unique table.

Computed table: The polynomial complexity in the product of operand BDD sizes for most BDD operations assumes that during a BDD operation, the result for each pair of operand BDD nodes is evaluated at most once during the computation. This requires the presence of a *computed table* which stores information about the operand BDDs and the result BDD. Theoretically, this would require implementation of a lossless cache. On a practical note, a computed table is implemented as a direct mapped or 2-way set associative. Without computed table, the complexity of even simple Boolean operations will be exponential in operand BDD sizes.

Index: Along each path in a BDD, variables (associated with the nodes along the path) appear in the same order. The *index* of a variable (also referred to as its "level") is equal to its distance from the root node. Variables in BDDs are ordered from 0 to $n-1$, starting from root to the constant nodes, n being total number of variables. A variable with "lower" value of index is "higher" in the order and vice versa.

Id: An id (identifier) is associated with each variable, and it also ranges from 0 to $n-1$. The identifier value for a variable remains constant throughout the life of the variable, however its index changes during reordering.

ITE: Given three Boolean functions F, G, and H, the **if-then-else** (ITE) operator is defined as follows:

$$ITE(F,G,H) = \overline{F} \cdot G + F \cdot H$$

2.2 Synchronous Sequential Circuits

A sequential circuit is an interconnection of combinational gates and memory elements along with input and output ports. Various notions of sequential circuits differ in the definition of memory elements. We focus on sequential circuits where all the memory elements are edge-triggered latches driven by the same clock. Formally, a sequential circuit is given as $C = (I, O, G, L, N)$, where I, O, G, L and N are sets of inputs, outputs, gates, latches,* and nets respectively. Inputs, outputs, combinational gates, and latches are collectively referred to as circuit elements. Each net $n \in N$ represents a directed connection between a primary input / gate / latch and a gate / latch / primary output. Often, for the purpose of analysis, it is convenient to represent a circuit using a directed graph also known as Boolean network. Formally, a Boolean network N is a directed graph $\vec{G} = (V, \vec{E})$, with a one to one correspondence between each node in the graph and the circuit elements (inputs, outputs, gates, and latches). Edges in the graph correspond to the nets in the circuit. An edge $e_{ij} \in E$ indicates a fanout from the circuit element corresponding to node i to that corresponding to node j. Nodes corresponding to combinational gates can have any number of fan-ins and fan-outs. Nodes corresponding to primary inputs do not have any fan-ins and that corresponding to primary outputs do have any fan-outs. Nodes representing latches have only one fan-in. Figure 2.3 shows a sequential circuit and the corresponding Boolean network.

2.3 Finite State Machines

A finite state machine (FSM) is used to model the behavior of systems with finite number of states. In general, an FSM is characterized by following elements:

- A finite set of *states*, S.

*In this thesis we will use the term "latch" as the short form for "edge-triggered latch" (also termed as flip-flop), unless otherwise noted.

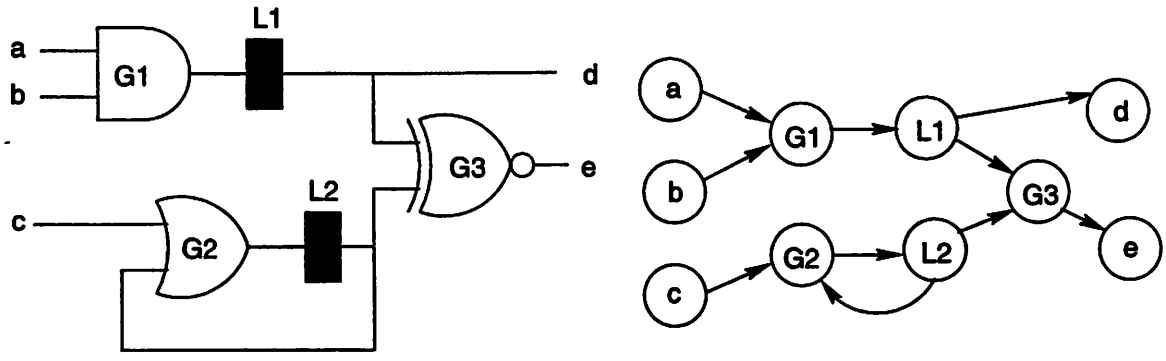


Figure 2.3 Example of a sequential circuit and corresponding Boolean network.

- A set of *initial states*, $S_I \subseteq S$.
- A finite *input alphabet*, Σ_I.
- A finite *output alphabet*, Σ_O.
- A *transition relation*, $T \subseteq S \times \Sigma_I \times S$. A tuple $(x,a,y) \in T$ implies that from state x, on applying the input a, the machine can move to state y. If T can be written as function, then the next state is deterministic, otherwise it is non-deterministic. An FSM is completely specified if for each $a \in \Sigma_I$ and $x \in S$, there exists a $y \in S$, such that $(x,a,y) \in T$.
- An *output relation*, O, given as $O \subseteq S \times \Sigma_O$ for *Moore* machines (output is solely defined by the state), and as $O \subseteq S \times \Sigma_I \times \Sigma_O$ for *Mealy* machines (output is defined by the state and the input). For deterministic machines, O can be expressed as a function ($O : S \mapsto \Sigma_O$ for Moore and $O : S \times \Sigma_I \mapsto \Sigma_O$ for Mealy).

A sequential circuit can be modeled by an FSM. The states are defined by the values of the latches, the circuit inputs and outputs define the input and output alphabets respectively, and the transition functions and output functions are defined by the logic of the circuit. In particular,

- All the symbols of the FSM – S, Σ_I, and Σ_O – are encoded in terms of binary variables (variables on the domain $\mathbb{B} = \{0,1\}$).

- A state is given as a minterm on the latch values.
- The input alphabet is determined by the minterms of the input variables.
- The output alphabet is determined by the minterms of the output variables.
- Since the circuit models a hardware, for any given state and the input value the next state and the output are uniquely defined.
- The non-deterministic behavior for the FSM can be simulated by adding few pseudo-inputs to the circuit.

Formally, for a sequential circuit with m inputs, k outputs, and n latches, the interpretation of FSM symbols is given as follows:

$$
\begin{aligned}
S = \mathbb{B}^n &: \text{Set of states} \\
S_I \subseteq \mathbb{B}^n &: \text{Set of initial states,} \\
\Sigma_I = \mathbb{B}^m &: \text{Set of input alphabets} \\
\Sigma_O = \mathbb{B}^k &: \text{Set of output alphabets} \\
\vec{\delta} &: \text{Vector of transition functions } \delta_i : \mathbb{B}^n \times \mathbb{B}^m \rightarrow \mathbb{B}, \quad i = 1,2,\ldots,n \\
\vec{\lambda} &: \text{Vector of output functions (Moore) } \lambda_i : \mathbb{B}^n \rightarrow \mathbb{B}, \quad i = 1,2,\ldots,k \\
&: \text{Vector of output functions (Mealy) } \lambda_i : \mathbb{B}^n \times \mathbb{B}^m \rightarrow \mathbb{B}, \quad i = 1,2,\ldots,k
\end{aligned}
$$

Some more notation used in this dissertation are given below:

- $\vec{x} = \{x_1, x_2, \ldots, x_n\}$: Vector of binary state variables.
- $\vec{u} = \{u_1, u_2, \ldots, u_m\}$: Vector of binary input variables.
- $\vec{z} = \{z_1, z_2, \ldots, z_k\}$: Vector of binary output variables.
- $\delta_i(\vec{x}, \vec{u}) : \mathbb{B}^n \times \mathbb{B}^m \mapsto \mathbb{B}$ Transition function. At places, we use f_i to indicate δ_i.
- $\vec{y} = \{y_1, y_2, \ldots, y_n\}$: Vector of binary variables used as a place holder for the next-state values of the latches.

- $T_i(\vec{x},\vec{u},y_i)$ is used to indicate the transition relation for i^{th} state bit and is given as $T_i = \overline{f_i \oplus y_i}$. The vector of transition relation is given as $\vec{T} = \{T_1, T_2, \ldots, T_n\}$.
- $T(\vec{x},\vec{u},\vec{y}) : \mathbb{B}^n \times \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}$ is the **transition relation**. It is given as $\prod_i T_i(\vec{x},\vec{u},y_i)$. $T(\vec{x},\vec{u},\vec{y}) = 1$ implies that in state $\vec{x}$ there exists a transition to state $\vec{y}$ on input $\vec{u}$.

2.3.1 State Transition Graph

An FSM can also be represented by a **state transition graph** (STG). An STG is a directed graph $G(V,\vec{E})$, where each vertex $v \in V$ corresponds to a state $s \in S$. An edge $e_{ij} \in E$ connects v_i to v_j if there exists a primary input $a \in \Sigma_I$ such that $(s_i, a, s_j) \in T$.

2.4 Implicit Boolean Set Manipulation

A Boolean function f from $\{0,1\}^n$ into $\{0,1\}$ denotes a unique subset S_f of $\{0,1\}^n$ that is defined by the equation

$$S_f = \{\vec{x} \in \{0,1\}^n / f(\vec{x}) = 1\}$$

Conversely a subset S of $\{0,1\}^n$ is denoted by a unique Boolean function from $\{0,1\}^n$ into $\{0,1\}$, χ_s, that is defined by the equation

$$\chi_s(\vec{x}) = 1 \Leftrightarrow \vec{x} \in S,$$

and is called its *characteristic function*. Since any Boolean function from $\{0,1\}^n$ into $\{0,1\}$ has a unique BDD representation for a given variable ordering, any subset of $\{0,1\}^n$ also has a unique BDD for this variable ordering.

Characteristics functions are a very interesting representation of Boolean sets because there is no relation at all between the number of elements in a set and the size of the BDD that denotes this set, so that huge Boolean sets can potentially be denoted by small BDDs and vice versa there exists subsets of $\{0,1\}^n$ whose BDDs have an exponential size with respect to n. Boolean operators correspond with set operators, e.g., disjunction corresponds with union, and negation with complementation. All elementary set operations can thus be evaluated with a quadratic complexity on BDDs [CM95].

Part I

Computer Architecture and BDD Manipulation

Chapter 3

Breadth-First BDD Manipulation

THE manipulation of very large BDDs is the key to success for BDD-based algorithms for simulation [Bry91], synthesis [CBM89, TSL+90], and verification [BCMD90, McM93, BSA+96a] of integrated circuits and systems. State-of-the-art BDD packages, based on the conventional depth-first technique, limit the size of the BDDs that can be manipulated due to disorderly memory access patterns that result in unacceptably high elapsed time when the BDD size exceeds the main memory capacity. In this chapter we present the design and implementation of a high performance BDD package that enables manipulation of very large BDDs by using an iterative breadth-first technique directed towards localizing memory accesses to exploit the memory organization in a computer system.

The basis of our work is the iterative breadth-first BDD manipulation algorithm proposed by Ochi *et al.* [OYY93] and later improved by Ashar *et al.* [AC94]. Our main contributions include i) an architecture-independent customized memory management scheme, ii) the ability to issue multiple independent BDD operations simultaneously (superscalarity), and iii) the ability to perform multiple BDD operations even when the operands of some BDD operations are the result of some other operations yet to be completed (pipelining). A comprehensive set of BDD manipulation algorithms are implemented using the above techniques. The new package is faster than the state-of-the-art depth-first BDD package by a factor of up to 100 for BDD sizes that do not fit the main memory. Even for the BDD sizes that fit within the main memory, our package outperforms depth-first manipulation based packages by a factor of 1.5. This is in contrast to the breadth-first algorithms presented in the literature which paid a performance penalty for smaller BDD sizes.

The rest of the chapter is organized as follows. We start with giving a background on the usage of BDD in synthesis and verification algorithms and the BDD manipulation

process in Section 3.1. In Section 3.2, we discuss the key features of the breadth-first manipulation algorithm and describe the related work based on this algorithm. In Section 3.3 we describe our approach and contrast it with the previous approaches. Section 3.4 describes the memory access pattern during breadth-first traversal. In Sections 3.5 and 3.6, we present the concepts of performing BDD operations in a superscalar and pipelined manner and discuss how it leads to better memory access patterns. Section 3.7 shows how superscalarity and pipelining are exploited to obtain efficient algorithms for common BDD operations, e.g., SUBSTITUTION and QUANTIFICATION. In Section 3.8, we present some implementation details. Experimental results demonstrating the effectiveness of our technique are presented in Section 3.9. Finally, we conclude in Section 3.10 and give directions for future work. Most of the work presented in this chapter was first reported in [SRBS96].

3.1 Introduction

For a preview on the basics of BDDs, please refer to the Section 2.1.

3.1.1 BDDs in Synthesis and Verification Algorithms

BDDs are used in two contexts in synthesis and verification algorithms: i) to represent Boolean functions which capture the functionality of the circuit under consideration, and ii) to implicitly represent and manipulate sets of elements, e.g., sets of states are represented using BDDs in verification algorithm.

In combinational verification, the BDDs representing the functionality of each primary output are built in terms of primary inputs. First, the BDD variables are created for primary inputs and the gates of the circuit are processed in topological order. For each gate, the BDD for its output is created by appropriately applying the logic function of the gate to the BDDs of its inputs. This process is repeated until the BDDs for the primary outputs are created.

In design and implementation verification of sequential circuits, the functionality of the circuit is represented by the BDDs for the outputs and the next-state functions of the latches. In addition, we need to represent and manipulate set of states (state traversal) which form the key operations in the sequential verification algorithm. The state traversal is performed by two basic operations: image and pre-image computation,

which in turn makes use of BDD operations: relational product, substitution, etc.

In synthesis, BDDs are used to obtain and manipulate sets of don't cares. These sets are implicitly represented using BDDs. The main operation again involves image and pre-image computation.

The application of BDD-based synthesis and verification algorithms to industrial designs requires performing Boolean operations on very large BDDs with millions of nodes.

3.1.2 BDD Manipulation

Let us now look at what is involved in manipulating functions using BDDs. Consider two functions f and g. For illustration purposes assume that we want to compute the "AND" of these two functions. Suppose x is the top variable in the BDDs for both functions. The cofactors of functions f and g are given as: $f_x, f_{\bar{x}}, g_x, g_{\bar{x}}$. Using Shannon decomposition of the functions, the "AND" can be computed as follows:

$$\begin{aligned} f &= x f_x + \bar{x} f_{\bar{x}} \\ g &= x g_x + \bar{x} g_{\bar{x}} \\ h &= f \cdot g \\ &= x(f_x g_x) + \bar{x}(f_{\bar{x}} g_{\bar{x}}) \\ &= x h_x + \bar{x} h_{\bar{x}} \end{aligned}$$

This is illustrated in Figure 3.1.

We notice that to compute the "AND" of the two functions, we need to compute the "AND" of the left cofactors (f_x and g_x) and the "AND" of the right cofactors ($f_{\bar{x}}$ and $g_{\bar{x}}$). The natural implementation of this computation is recursive, i.e., the results for the cofactors are recursively obtained.

3.1.3 Conventional BDD Manipulation and Limitations

Conventional BDD algorithms are based on a recursive formulation (shown in Figure 3.4) that leads to a depth-first traversal of the directed acyclic graphs representing the operand BDDs. The depth-first traversal visits the nodes of the operand BDDs on a path-by-path basis (Figure 3.2). In this traversal, a node is visited right after its parent node is visited. The access pattern is illustrated with an example in Figure 3.3.

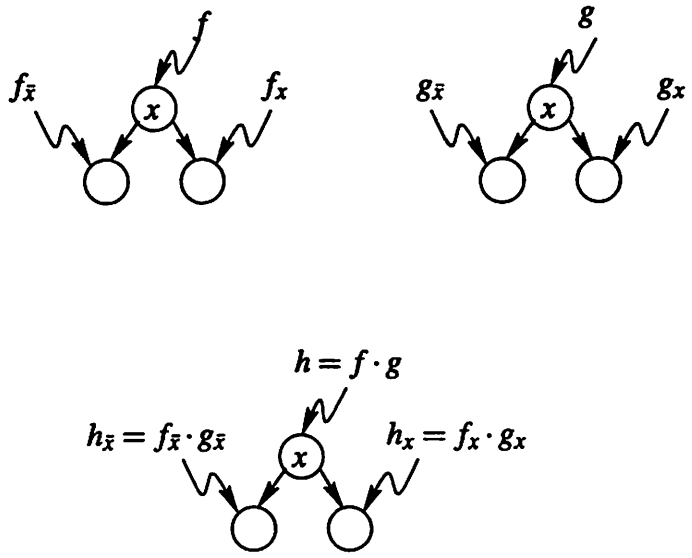


Figure 3.1 Computing "AND" of two functions.

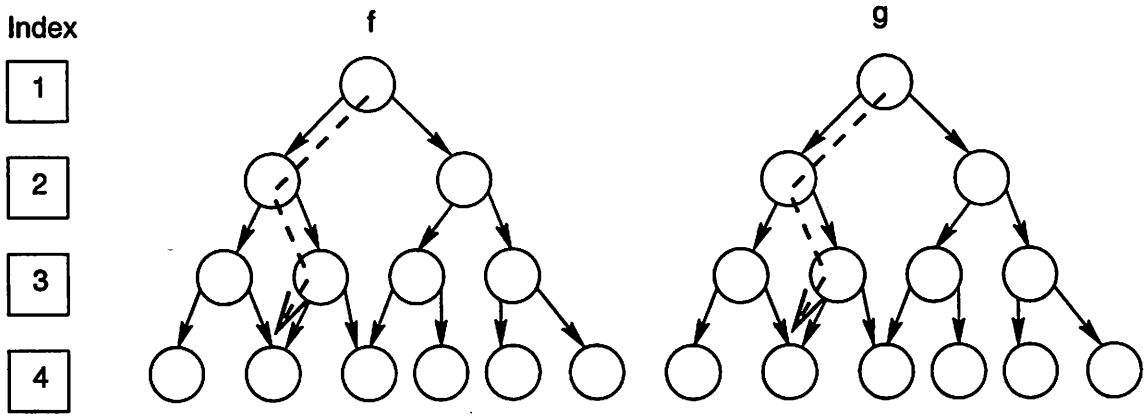


Figure 3.2 Operand access pattern during conventional manipulation.

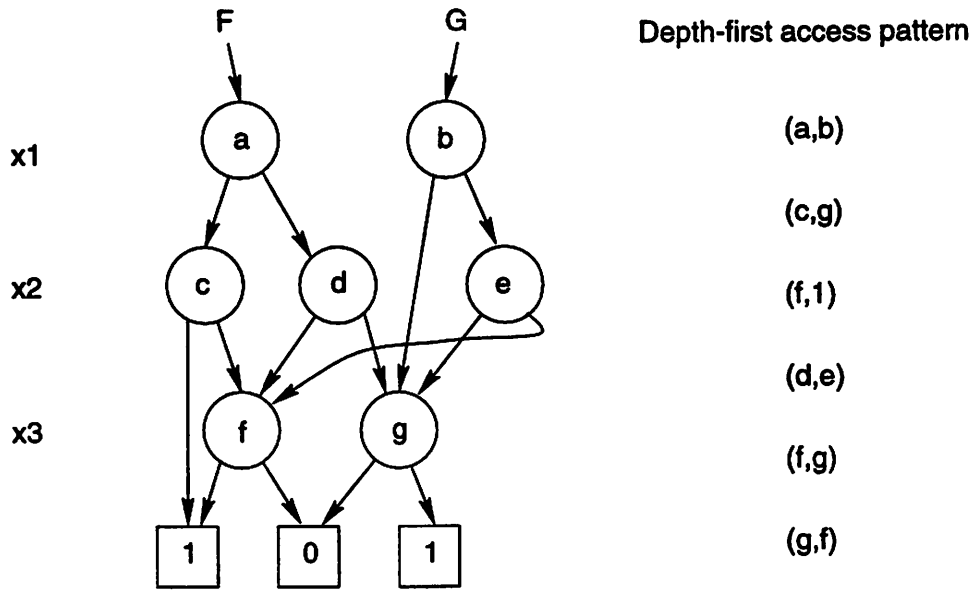


Figure 3.3 Depth-first traversal of operand BDDs in conventional manipulation.

```
df_op(op,F,G)
    if (terminal case(op, F, G)) return result;
    else if (computed table has entry(op, F, G)) return result;
    else{
        let x be the top variable of F, G;
        T = df_op (op, F_x, G_x);
        E = df_op (op, F_x', G_x');
        if (T equals E) return T;
        result = find or add in the unique table (x, T, E);
        insert in the computed table ((op, F, G) , result);
    }
    return result;
```

Figure 3.4 Depth-first BDD manipulation algorithm.

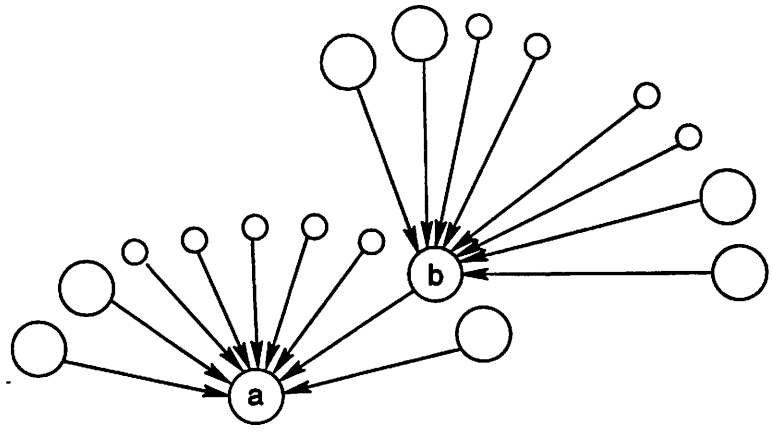


Figure 3.5 Problem in localizing memory accesses in depth-first traversal.

In order to keep the memory accesses local, we would need to put a node close to its parents node in the memory. This is difficult due to two reasons:

1. The large in-degree of a typical BDD node makes it impossible to assign contiguous memory locations for the BDD nodes along a path. This is illustrated in Figure 3.5. Since node *a* is visited right after any of its parent nodes are visited, it needs to be close in memory to all of them (including *b*). If *b* in turn has large in-degree as well, it would require that all the nodes shown in the figure should be located close in the memory, making it an infeasible task.

2. Even when the in-degree is not large for any node, the graph pattern corresponding to all the BDDs change over time as the manipulation progresses. Due to this, the number of parent nodes of a node cannot be predicted a priori and hence local memory assignment is not possible.

Therefore, the recursive depth-first traversal leads to an disorderly memory access pattern. Let us now look at the performance implications of the random memory access pattern.

We refer to the data on various levels in memory hierarchy as presented in Table 1.1 on page 5. When the BDD size exceeds the capacity of a given level in the memory

system, the disorderly pattern of the depth-first algorithms translates to a performance penalty. For example, when the BDD size exceeds the cache size, a slowdown by a factor of 2-10 may be observed due to a high cache miss rate. When the BDD size measured in the number of memory pages exceeds the number of translation look-aside buffer (TLB) entries, a further slowdown may be observed. However, the most dramatic degradation in performance is observed when the BDD size exceeds the main memory size; the depth-first algorithms thrash the virtual memory leading to unacceptably high elapsed time even though the amount of CPU time spent doing useful work is low. Therefore, the depth-first algorithms place a severe limit on the size of the BDD that can be effectively manipulated on a given computer system.

To a first approximation, the performance of BDD manipulation algorithms is dominated by the performance and capacity of each level in the memory system hierarchy. Hence, the design of high performance BDD algorithms require a careful consideration of memory related issues.

3.1.4 Breadth-first BDD Manipulation Technique

The iterative breadth-first technique for BDD manipulation attempts to fix the disorderly memory access behavior of the recursive depth-first technique. Unlike the depth-first algorithm that traverses the operand BDDs on a path-by-path basis, the iterative breadth-first algorithm traverses the operand BDDs on a level-by-level basis (Figure 3.6), where each level corresponds to the index of a BDD variable. The BDD nodes corresponding to a level are allocated from the same memory segment so that temporally local accesses to the BDD nodes for a specific level are also spatially local. The node access pattern is illustrated in Figure 3.7.

The basic iterative breadth-first technique consists of two phases: a top-down (from root node to leaves) APPLY phase followed by a bottom-up REDUCE phase. The APPLY and REDUCE phases are illustrated in Figures 3.8 and 3.9 respectively. In Figure 3.8, suppose we want to compute "AND" of functions given by BDD nodes a and d. As an initial step a temporary node is created which acts as a place holder for the final result. In our exposition we will use the term REQUEST to indicate such temporary nodes or place holders. The REQUEST is indicated by $R = (AND, a, d)$. The appropriate operand information is duplicated in this REQUEST. Next the top-down APPLY phase starts. In

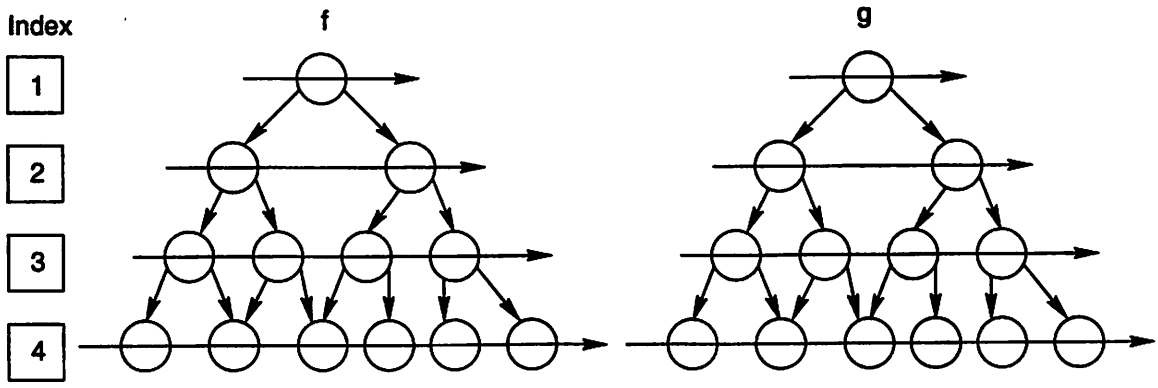


Figure 3.6 Operand access pattern during breadth-first manipulation.

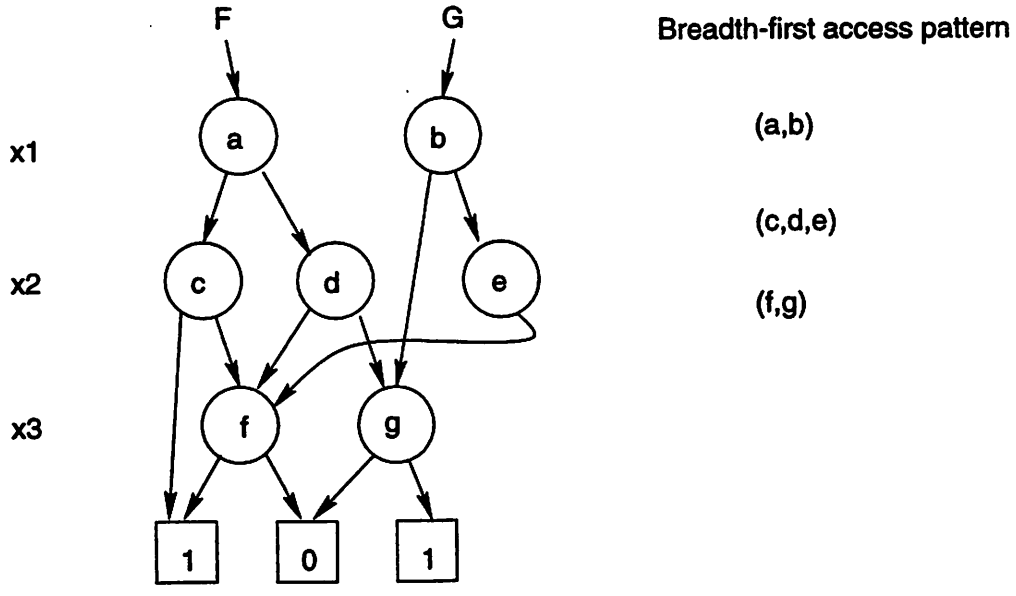


Figure 3.7 Operand nodes access pattern in breadth-first traversal.

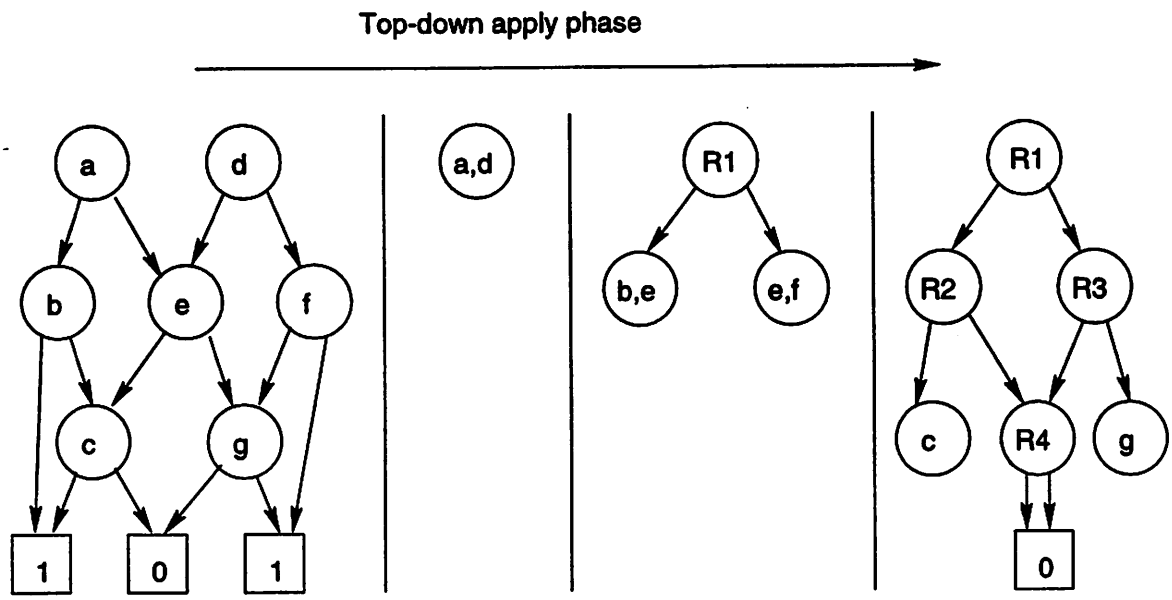


Figure 3.8 Illustration of APPLY phase in breadth-first manipulation.

this phase, REQUESTs are accessed and processed to obtain the final result of applying the Boolean function on the operands stored in the REQUESTs. In particular, to obtain results for the REQUEST $R1 = (AND, a, d)$, we need to obtain results for "AND"ing the left cofactors of the operands (b and e), and the result of "AND"ing the right cofactors of the operands (e and f). As opposed to computing these sub-results recursively, two new REQUESTs are generated. The locations of these new REQUESTs is determined by the minimum index of the operands' cofactors. In this example, the new REQUESTs, which act as place holders for the cofactor results, are created at the next index. The content of the original REQUEST R is overwritten and after the processing R contains pointers to the cofactor results place holders. The APPLY phase proceeds with the processing of REQUESTs at the next index and so on. We notice that in some cases no new REQUEST is generated if a terminal condition is found (node c) or a similar REQUEST is already created (request $R4 = (AND, c, g)$). The checking for duplicate REQUESTs is done by maintaining the collection of REQUEST at each index in a hash table which we refer to as REQUEST QUEUE. At the end of APPLY phase an unreduced BDD is obtained.

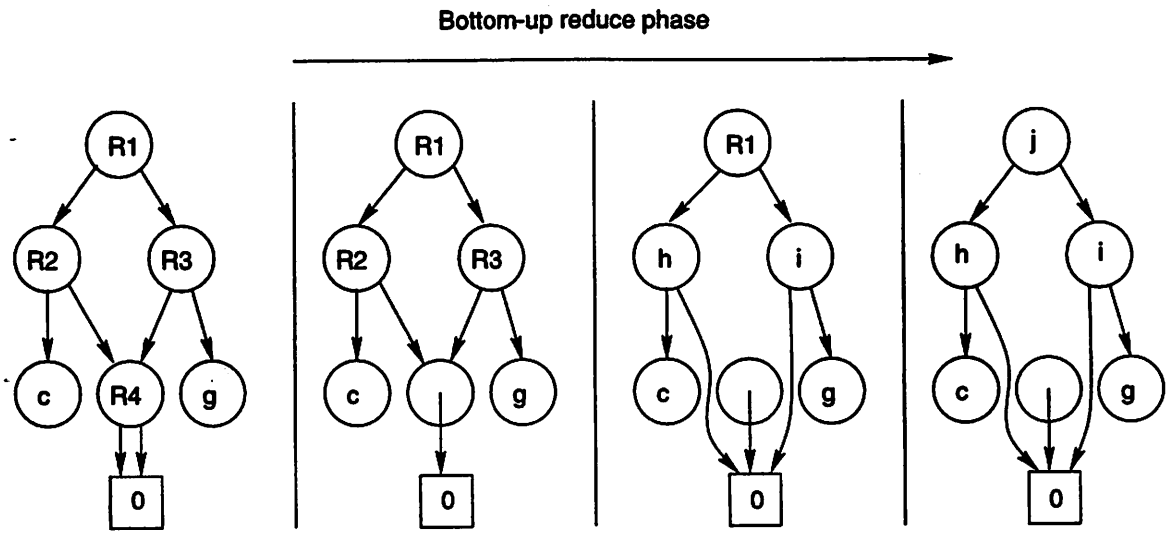


Figure 3.9 Illustration of REDUCE phase in breadth-first manipulation.

This is because, node R_4 is redundant (it has identical left and right child nodes). To eliminate such redundant nodes, a bottom up REDUCE phase is applied. In this phase, REQUESTs are again accessed one by one, and after updating of cofactor contents are either converted to a BDD node or are designated as redundant. In Figure 3.9, we start with identifying node R_4 as redundant. We put a forwarding pointer indicating the location of the result in R_4. Next REQUESTs R_2 and R_3 are processed. we first update their cofactor contents (by looking at the forwarding pointer of R_4). Then the corresponding UNIQUE TABLE is traversed to check if at that index a BDD node with identical left and right child nodes exists or not. In this particular case, both R_2 and R_3 get converted to BDD nodes *h* and *i* respectively.

The generic algorithm for a two-operand boolean operation is shown in Figures 3.10, 3.11, and 3.12.

During the APPLY phase, the outstanding REQUESTS are processed on a level-by-level basis. The processing of a REQUEST $R = (\text{op}, F, G)$, in general, results in issuing two new REQUESTs which represent the THEN and the ELSE cofactors of the result (F op G). Since certain isomorphism checks cannot be performed, the result BDD

```
bf_op(op, F, G)
    if terminal case (op, F, G) return result;
    min_index = minimum variable index of (F, G)
    create a REQUEST (F, G) and insert in REQUEST QUEUE[min_index];
    /* Top down APPLY phase */
    for (index = min_index; index ≤ num_vars; index++) bf_apply(op, index);
    /* Bottom up REDUCE phase */
    for (index = num_vars; index ≥ min_index; index--) bf_reduce(index);
    return REQUEST or the node to which it is forwarded;
```

Figure 3.10 Breadth-first BDD manipulation algorithm.

```
bf_apply(op, index)
    x is variable with index "index";
    /* process each request queue */
    while (REQUEST QUEUE[index] not empty)
        REQUEST (F, G) = unprocessed request from REQUEST QUEUE[index];
        /* process REQUEST by determining its THEN and ELSE */
        if (NOT terminal case ((op, F_x, G_x), result)){
            next_index = minimum variable index of (F_x, G_x);
            result = find or add (F_x, G_x) in REQUEST QUEUE[next_index];
        }     REQUEST → THEN = result;
        if(NOT terminal case ((op, F_x̄, G_x̄), result)){
            next_index = minimum variable index of (F_x̄, G_x̄);
            result = find or add (F_x̄, G_x̄) in REQUEST QUEUE[next_index];
        }     REQUEST → ELSE = result;
```

Figure 3.11 Breadth-first BDD manipulation algorithm – APPLY.

```
bf_reduce(index)
    x is variable with index "index";
    /* process each request queue */
    while (REQUEST QUEUE[index] not empty){
        /* process each request */
        REQUEST (F,G) = unprocessed REQUEST from REQUEST QUEUE[index];
        if (REQUEST→THEN is forwarded to T)
            REQUEST→ THEN = T;
        if (REQUEST→ELSE is forwarded to E)
            REQUEST→ ELSE = E;
        if (REQUEST→THEN equals REQUEST→ELSE)
            forward REQUEST to REQUEST → THEN ;
        else if (BDD node with (REQUEST→ THEN ,
            REQUEST → ELSE ) found in UNIQUE TABLE[index]){
            forward REQUEST to that BDD node;
        }
        else{
            insert REQUEST to the UNIQUE TABLE[index] with key
                (REQUEST → THEN , REQUEST → ELSE )
        }
    }
```

Figure 3.12 Breadth-first BDD manipulation algorithm – REDUCE.

obtained at the end of APPLY phase has redundant nodes. The REDUCE phase traverses the result BDD from the leaves to the root on a level-by-level basis eliminating the redundant nodes.

In the APPLY phase of the algorithm the following two things need to be determined for each REQUEST: i) indices of the operand BDDs to obtain the index of the variable with respect to which the cofactors should be taken, ii) indices of the cofactor nodes in order to place the new REQUESTs in the appropriate REQUEST QUEUE (see underlined in Figure 3.11). In order to preserve the locality of references, it is important to determine the variable index of the cofactor nodes without actually fetching them from the memory (i.e., without accessing them). In particular, the routine *bf_apply* called with index i should access nodes only at index i. Similar issues arise during the REDUCE phase. Also, the memory accesses to the REQUEST QUEUE during APPLY and REDUCE phases play a role in the locality of the references.

3.2 Previous Work

In [OYY93], Ochi, Yasuoka, and Yajima use a variant of BDDs, namely, Quasi Reduced BDD (QRBDD) to address various issues about locality of reference. A QRBDD is obtained from a binary decision tree by merging the isomorphic sub-graphs. The redundant nodes, however, are not eliminated. The main property of this graph is that along each path of the BDD, consecutive nodes differ in their indices by exactly one. Using this approach, they could localize memory accesses as follows: i) there is no need to determine the variable index of each operand BDD, since each has the same index equal to the current index for which REQUESTs are being processed, ii) since the indices of the cofactors is exactly one more than the current index, cofactor index determination can be done without any memory accesses, iii) since the new REQUESTs generated are always for the next higher index, checking for duplicate REQUESTs during the APPLY phase is done by searching the corresponding REQUEST QUEUE, and iv) similarly, during the REDUCE phase, isomorphic nodes are found by checking the REQUEST QUEUE of only one level.

However, their approach has two disadvantages: i) it is observed that the QRBDD is several times larger than the corresponding BDD [AC94], which makes this approach impractical for manipulating very large BDDs, and ii) because of larger number of

nodes in the operand BDDs, the total computation increases. Due to these problems, this approach performed poorly compared to the conventional depth-first approach for BDD sizes that fit the main memory.

Ashar and Cheong [AC94], use a BLOCK-INDEX table to determine the variable index from a BDD pointer by performing an associative lookup. Since this solution employs BDDs (as opposed to QRBDDs)*, an attempt is made to preserve the locality of reference during the check for duplicate requests during the APPLY phase and check for redundant nodes during the REDUCE phase. This is done by sorted accesses to co-factor nodes based on their variable indices. This approach overcomes the size problem of the previous approach by employing BDDs instead of QRBDDs. However, it suffers from significant overhead (about a factor of 2.65) as compared to a depth-first based algorithm for manipulating BDDs which fit within the main memory [AC94].

3.3 Our Approach

Our approach to handling the variable index determination problem differs from the works of Ashar *et al.* and Ochi *et al.* in the following aspects:

1. A new BDD node data structure is introduced to determine the variable index while preserving the locality of accesses. We represent a BDD using (variable index, BDD node pointer) pair (shown in Figure 3.20). Therefore a BDD node contains pointers to its THEN and ELSE cofactors as well as their variable indices. As a result, we do not need to fetch the cofactors to determine their indices. The BDD node data structure and related details are discussed in Section 3.8.1.

2. Optimized processing of REQUEST QUEUES for each level by eliminating the sorted processing of REQUESTs during APPLY and REDUCE phases as proposed by Ashar *et al.* Empirically, we have observed that this change does not affect the performance of our algorithm for manipulating very large BDDs.

3. Use of a customized memory manager to allocate BDD nodes which are quad-word aligned. The quad-word alignment improves the cache performance by

*Since the difference in the index of a node and its child node can be arbitrary in a BDD, strictly speaking, the traversal proposed in [AC94] is "level-by-level" and not breadth-first. However, in this dissertation, we overload the term "breadth-first manipulation" to mean "level-by-level" traversal [Shi97].

mapping a BDD node to a single cache line.

These three techniques eliminate the overheads associated with the previous breadth-first approaches. In addition we make use of multiple BDD operations (described next) resulting in new algorithms that are faster than the corresponding recursive algorithms even on those examples for which the BDDs fit in the main memory.

We propose two new concepts – superscalarity and pipelining – to optimize the memory performance of the iterative breadth-first BDD algorithms by exploiting locality of reference that exists among multiple BDD operations. The concepts of superscalarity and pipelining have their roots in the field of computer architecture in which superscalarity refers to the ability to issue multiple, independent instructions and pipelining refers to the ability to issue a new instruction even before completion of previously issued instructions. We shall see how these concepts can be applied in the context of the breadth-first BDD algorithms to exploit the memory system hierarchy.

3.4 Memory Access Pattern

Let us first take a closer look at the memory access pattern of the basic breadth-first algorithm. A custom memory manager assigns to each variable a memory segment that consists of a set of pages; the memory segment is expanded on demand in units of a page where each page holds PAGESIZE / NODESIZE number of nodes for a particular variable. The overall paging behavior resulting from the memory access pattern is determined by processing of a set of REQUESTs for each variable in the ascending order of their indices during the top-down APPLY phase and in the descending order of their indices during the bottom-up REDUCE phase.

Apply phase: The following accesses take place to process each REQUEST R, for an index i with associated variable x. Suppose REQUEST $R = (op, F, G)$. Without loss of generality, assume that $F_{index} = \min(F_{index}, G_{index}) = i$.

First of all R is accessed to determine the operand BDDs (F and G). The next step involves obtaining the cofactors of the operands, i.e., $F_x, F_{\bar{x}}, G_x$, and $G_{\bar{x}}$. Since $F_{index} == i$, we need to access the UNIQUE TABLE BDD node of F to obtain the THEN and ELSE BDDs that respectively represent the left cofactor (F_x) and right cofactor ($F_{\bar{x}}$). If $G_{index} > i$, $G_x = G_{\bar{x}} = G$ and we do not need to perform any access on G. In the next

step, the left and right cofactors of R is determined which are obtained as follows (we explain the steps involved in obtaining the left cofactor, the right cofactor is obtained along similar lines):

We first determine if the result of the cofactors can be trivially obtained, i.e., if the result of (op, F_x, G_x) is a terminal case. If it is indeed a terminal case, we make the left cofactor of R point to appropriate result. Otherwise, we need to create a new REQUEST $R_1 = (op, F_x, G_x)$ which is a place holder for the result of the operation on left cofactors. The new REQUEST is placed in the REQUEST QUEUE for index k, where $k = \min(F_{x_{index}}, G_{x_{index}})$. To avoid any duplicate computation, we perform an associate lookup in the REQUEST QUEUE for index k to check if a REQUEST equivalent to R_1 already exists.

Reduce phase: In this case, the THEN and ELSE fields of the REQUEST node are accessed. These fields are appropriately updated if they point to a redundant node. If the updated THEN and ELSE BDDs are not equal, the UNIQUE TABLE for the current index i is associatively searched to identify a duplicate BDD node and if no duplicate exists, a new BDD node is created. This leads to memory accesses to traverse the chain in the hash table that represents the set of UNIQUE TABLE BDD nodes for the current index i.

In summary, the following memory accesses take place for each index i in addition to memory accesses to each processed REQUEST. During the APPLY phase, to process each REQUEST for an index i, we have i) at least one and up to two pointer accesses to UNIQUE TABLE BDD nodes for i, ii) up to two associative lookups in appropriate REQUEST QUEUEs to check if the REQUESTs have been issued previously. During the REDUCE phase, to process each REQUEST for an index i, we have i) accesses to THEN and ELSE BDD nodes to check for redundancy, ii) associative lookups in the UNIQUE TABLE for the index i to determine if another node with the same attributes already exists.

The next step is to determine the memory accesses that contribute significantly to the total number of page faults when the BDD size exceed the main memory capacity. In general, we have empirically observed the following:

1. The size of the set of REQUEST QUEUEs is smaller, sometimes much smaller, than the size of the UNIQUE TABLE. Therefore, the UNIQUE TABLE for an

index has significantly more pages assigned to it as compared to the REQUEST QUEUE for that index.

2. Although for each cofactor, say (F_x, G_x), the cofactor index equal to the lower of indices for F_x and G_x can be arbitrarily large (up to maximum number of variables) than the current index, the difference between the cofactor index and the current index is small for most cofactors. Hence, we have the following: i) during the APPLY phase, most of the associative lookups in the REQUEST QUEUEs to identify or create cofactor REQUESTs are limited to REQUEST QUEUEs for next few variables, and ii) during the REDUCE phase, most of the redundancy checks for the THEN and the ELSE pointers are limited to nodes from next few variable indices. Therefore, the number of page faults that result from memory accesses to the REQUEST QUEUE constitute a small fraction of the total page faults when the BDD size exceeds the main memory capacity.

3. Accesses to the UNIQUE TABLE for variable x – accesses to operand BDDs during the APPLY phase and associative lookups that result in hash chain traversal during the REDUCE phase – are relatively random; therefore, a large number of pages assigned to the variable x is touched. The number of pages touched during accesses to the UNIQUE TABLE for the variable x is significantly less than the number in the absence of the memory management strategy. Nevertheless, even in the presence of the custom memory management, a large fraction of the total pages assigned to a variable determines the working memory set size. For very large BDDs that exceed the main memory capacity, the capacity page misses (page misses occurring because only a fraction of memory allocated to BDD nodes/REQUESTs can fit in the main memory) occur as computation progresses from one variable to another in the top-down APPLY phase and the bottom-up REDUCE phase.

The random accesses to the UNIQUE TABLE BDD nodes for each variable and accesses to each UNIQUE TABLE on a variable-by-variable basis result in the major fraction of the total page faults when the BDD size exceeds the main memory capacity. Therefore, for a very large BDD, the number of page faults is dominated by the memory accesses to the set of large UNIQUE TABLEs. *Superscalarity* and *pipelining* attempt to amortize the

cost of page faults for accessing UNIQUE TABLE pages for a specific variable among several BDD operations. These concepts are detailed in the next two sections.

3.5 Superscalarity

In the previous section we established that, for BDDs that exceed the main memory capacity, the performance is determined by the number of page faults, a major fraction of which is caused by accesses to the UNIQUE TABLE BDD nodes. Since the page faults to access the UNIQUE TABLE BDD nodes result from accesses to memory segments on variable-by-variable basis, the elapsed time for a BDD manipulation application depends on the number of passes of the APPLY and the REDUCE phases. To improve the performance of the breadth-first BDD algorithm, it is important to minimize the number of passes of the APPLY and REDUCE phases. The goal is realized by sharing a single pass of the APPLY and the REDUCE phases among several operations. The insight to perform several BDD operations simultaneously is obtained by viewing multiple outstanding REQUESTs during the APPLY phase as several BDD operations. The processing of several BDD operations simultaneously is an inherent part of the breadth-first paradigm; it does not have a natural counterpart in the depth-first technique.

In the context of breadth-first BDD algorithms, *superscalarity* refers to the ability to issue multiple, independent BDD operations simultaneously. Two BDD operations are said to be *independent* if their operands are reduced ordered BDDs, i.e., the nodes for the operand BDDs are in the UNIQUE TABLE. Intuitively speaking, a given set of operations are independent if none of the operations depend on the result of some other operation in the set.

This concept has been illustrated in Figure 3.13. In this example, we would like to compute the results of three "AND" operations given as, $h_i = f_i \cdot g_i$, $i = 1,2,3$. In the conventional method, the results can be individually computed as three separate operations. Using superscalarity, we issue REQUESTs for all three BDD operations simultaneously. Next, at each index during the APPLY and REDUCE phases, the REQUESTs generated from all three initial REQUESTs are processed. Hence we need only one pass of APPLY and REDUCE phase.

Performing multiple, independent BDD operations concurrently during the single

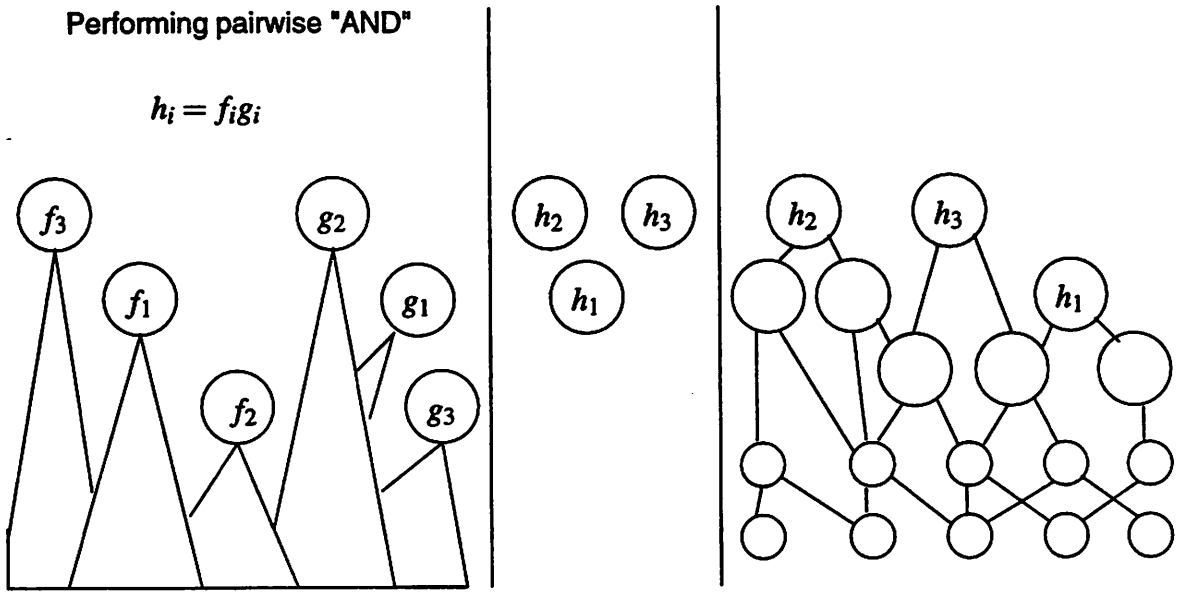


Figure 3.13 Multiple independent BDD operations using superscalarity.

APPLY and REDUCE phase amortizes the cost of page faults for accessing the UNIQUE TABLE entries. By issuing several independent operations simultaneously, the number of REQUEST nodes in the REQUEST QUEUE increases. However, the number of page faults for accessing UNIQUE TABLE nodes for a specific index does not increase proportionately; it increases at a lesser rate. Empirically, we observe a significant performance enhancement in the BDD algorithms by exploiting superscalarity.

Another major advantage of superscalarity is complete inter-operation caching of intermediate BDD results. A breadth-first algorithm for a single BDD operation provides complete caching of intermediate results during the operation by virtue of the REQUEST QUEUE. However, it is not possible to have inter-operation caching in the breadth-first algorithm without expending additional memory resources to store the cached results. It also requires additional computing resource to manage the complex caching scheme since the contents of a REQUEST node are destroyed after it is processed in the APPLY phase and the correct result is unavailable until the REQUEST is processed in the REDUCE phase. Superscalarity provides complete inter-operation caching for the set

of independent BDD operations that are issued simultaneously, thereby enhancing the performance of the breadth-first algorithm even further.

Superscalarity has few limitations. It is not guaranteed to reduce the number of page faults, in fact, it is possible that the number of page faults may actually increase. For example, consider performing n independent BDD operations simultaneously for which the number of memory pages required exceed the number of page frames in the main memory. If however, each of the n BDD operations were performed independently one after another, it may be possible to complete each of them such that maximum number of pages accessed during any one BDD operation does not exceed the number of page frames in the main memory. Assuming that same page is accessed during both the APPLY and the REDUCE phases, it is possible under some page replacement strategy to have twice as many page faults for n independent BDD operations performed simultaneously as n operations performed one by one.

We have assumed that most page faults during the APPLY and the REDUCE phase occur due to accesses to the BDD nodes in the set of UNIQUE TABLEs and only a small fraction of total page faults occur due to accesses to REQUESTs in the REQUEST QUEUEs. Due to superscalarity the size of the REQUEST QUEUE gets larger and the page fault balance may shift towards accesses to REQUESTs in the REQUEST QUEUEs. This shift may slow down or limit gains due to superscalarity as the number of operations that are performed simultaneously is increased.

How much gain can we expect from superscalarity? The answer depends on several factors. The most important is the number of independent operations available. However, several other interacting factors such as advantage due to inter-operation caching and disadvantage due to potential increase in working memory make it difficult to quantify or predict the potential gains. Except for some pathological cases, we have seen consistent performance improvement due to superscalarity.

3.6 Pipelining

For a very large BDD that exceeds the main memory capacity, the dominant cause of page faults, when using the breadth-first algorithm, is the sequential access to each set of UNIQUE TABLE BDD nodes. The heuristic to amortize the page faults, shares a single pass of the APPLY and the REDUCE phases among multiple BDD operations.

If multiple BDD operations are independent, the superscalarity enables us to perform them simultaneously in a single pass of the APPLY and the REDUCE phases. However, what if BDD operations are not independent? Is it still possible to extend the breadth-first technique suitably to complete several BDD operations in a single pass of the APPLY and the REDUCE phases? To answer these questions, let us first take a look at some of the features of the breadth-first algorithm.

1. Unreduced BDDs as operands: An unreduced BDD has some isomorphic and some redundant nodes. However, they can be used as an operand in a computation without any problem. The only outcome is that the resulting BDD is also unreduced. Therefore, the unreduced BDDs obtained at the end of the APPLY phase can be operand BDDs of another Boolean operation.

   Consider two unreduced BDDs R_1 and R_2 and an operation *op* which depends on the result of R_1 and R_2. The question is, do we need to wait until the end of APPLY phase (when R_1 and R_2 are created)? How soon can we start another Boolean operation *op* that uses the unreduced BDDs for R_1 and R_2, which are still under construction? This is answered by the next point.

2. Under what conditions can we use an unreduced node: Suppose R_1 and R_2 are the unreduced BDDs for functions F and G respectively. Let R be a REQUEST that implements the function (F op G) in the unreduced BDD H. Let x be the top variable of the REQUEST R. To process the node R in the unreduced BDD R, we only need $(F_{\bar{x}}, G_{\bar{x}})$ and (F_x, G_x) – cofactors of F and G with respect to x. Since the top variable for the node R is x, the index of each of the operands F and G is greater than or equal to the index of x. If the index of an operand, say F, is equal to the index of x, then the right and the left cofactors of F with respect to x is equal to the THEN and the ELSE BDDs of F respectively; since nodes for the variable x for the operand unreduced BDD R_1 is already constructed, the THEN and the ELSE BDDs of F are already known. If the index of F exceeds the index of x, then the cofactors of F with respect to x equal F itself. In either case, cofactors of F are known before processing a REQUEST R that uses F as one of the operands. Therefore, the APPLY phase for a variable x in constructing the unreduced BDD R can proceed as soon as the APPLY phases for the variable x in constructing the

unreduced BDDs R_1 and R_2 are complete. However, we need to process requests corresponding to R_1 and R_2 before processing those corresponding to R, i.e., we need to keep a partial order in the processing of the REQUESTs at each index during APPLY and REDUCE phase.

Based on these observations, we state the following theorem.

Theorem 1 *Let F_1, G_1, F_2, G_2 be regular BDDs. Given* REQUEST *s $R_1 = (op_1, F_1, G_1)$, $R_2 = (op_2, F_2, G_2)$ and $R = (op, R_1, R_2)$, the breadth-first algorithm with the modified* APPLY *and* REDUCE *phases, which processes the* REQUESTS *in level-by-level order while maintaining the partial order implied by the dependencies of* REQUESTS *for that index, correctly computes the reduced BDDs corresponding to the* REQUESTS *R_1, R_2, and R.*

Proof: The proof is based on the points discussed above. First, there is a one-to-one correspondence between processed requests and the nodes in the unreduced BDD created during the APPLY phase. Second, operand BDDs of a breadth-first BDD algorithm can be unreduced. Third, in constructing unreduced BDDs R_1 and R_2, at the end of the APPLY phase for variable x with index i, we have cofactors of each BDD node with index greater than or equal to i with respect to variable x. Fourth, in constructing the unreduced BDD R, processing a REQUEST $Q = (op, Q_1, Q_2)$ for index i during the APPLY phase, we only need cofactors of operands Q_1 and Q_2 with respect to x. Hence, nodes in the unreduced BDD R for index i can be constructed after constructing the nodes in the unreduced BDDs R_1 and R_2. This can be easily extended to the case with multiple dependencies. ■

In the context of breadth-first BDD algorithms, *pipelining* refers to the ability to issue multiple, dependent BDD operations simultaneously. A BDD operation op_1 is said to be *dependent* on another BDD operation op_2 if the result BDD of op_2 is an operand of op_1. The pipelining algorithm issues several dependent operations simultaneously using unprocessed requests to represent operands for the dependent operations. The result BDDs for these requests are obtained by a single APPLY and REDUCE phase that amortizes the cost of page faults for accessing the UNIQUE TABLE entries.

Consider the example shown in Figure 3.14. In this case, we would like to compute the result of the following Boolean equation: $y = e \cdot (f \cdot g)$. Conventionally, this is done

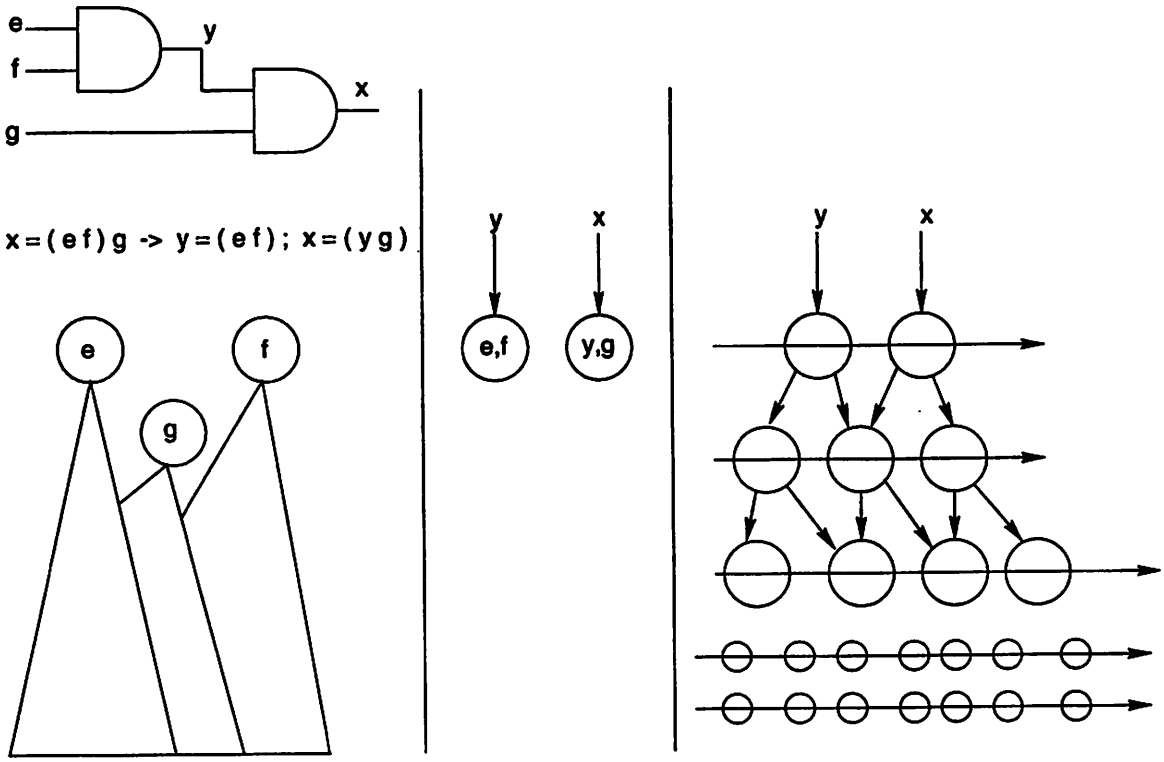


Figure 3.14 Multiple dependent BDD operations using pipelining.

in two steps: compute $x = f \cdot g$ in the first step and then compute $y = x \cdot e$. Using the pipelining approach, we issue REQUESTs for both the operations simultaneously. Then we perform a single pass of the APPLY and REDUCE phases. During these phases, at any given level, we process all the REQUESTs generated from the original REQUEST corresponding to the operation $x = f \cdot g$, before processing any REQUEST generated from the operation $y = x \cdot e$. In other words, for each index, we maintain a partial order while processing the outstanding REQUESTs. However, since partial order is on each index, we need to perform just one pass of APPLY and one pass of REDUCE.

The concept of pipelining improves the performance of the breadth-first algorithm by amortizing the cost of page faults across dependent BDD operations. However, pipelining results in operations on unreduced BDDs. Hence, there is an increase in the size of the working memory required and a corresponding increase in the amount of computation. If the increase in the working memory is large, the number of page faults will increase. The quantify the dependency amongst BDD operations by *pipedepth*. A pipedepth value of k indicates a dependency chain of length k amongst operations.

Pipelining will improve the amount of caching, since the intermediate BDDs in the dependent operation can use the cached result of all operations on which the current REQUEST depends directly or indirectly. However, it introduces the penalty of performing associative lookups in several REQUEST QUEUEs, which could offset the potential gain due to improved caching.

3.6.1 Application

Superscalarity and pipelining find their applications whenever a set of dependent and/or independent BDD operations needs to be performed. In this section we describe two such applications.

Creating Output BDDs of a Circuit: In many logic synthesis and verification applications we need to compute the BDDs for the outputs of a circuit. Given a network representing a circuit, we try to compute the function of the outputs in terms of the primary inputs. This requires computing the functions of the nodes of network starting from the primary inputs to the primary outputs. Pipelining and superscalarity can be employed to compute the output BDDs in several ways. Our algorithm is as follows:

1. Decompose the given network into two input NAND nodes.

2. Levelize the nodes of the new network.

3. Create the BDDs for nodes belonging to a particular level concurrently (using superscalarity), or

4. Create the BDDs for nodes belonging to two or more levels using pipelining.

The motivation behind decomposing the network into NAND nodes is to obtain as much superscalarity as possible. In Section 3.9 we provide experimental results indicating the effect of using superscalarity and pipelining while creating the output BDDs.

Multiway Operations: Applications of multiway operations arise when we want to perform some BDD operation on a set of functions. For example, suppose $\{f_i : \mathbb{B}^n \mapsto \mathbb{B}, i = 1, \cdots, m\}$ represent a set of m Boolean functions over n variables. Suppose we want to compute the BDD for function f given as $f = \Pi_{i=1}^{i=m} f_i$. We could compute this result by iteratively taking the product, at each step creating an intermediate result for the function $g_k = g_{k-1} \wedge f_k$, where $g_1 = f_1$. In this case we make $(m-1)$ passes of the APPLY and REDUCE phases each involving access to UNIQUE TABLEs and REQUEST QUEUEs. Using superscalarity and pipelining we can improve the performance significantly. Our algorithm is given below:

1. From the given set of arguments we make a binary tree where leaves represent the BDD arguments and the intermediate nodes represent the intermediate product BDDs.

2. Create the BDDs for all the nodes belonging to a particular level using superscalarity.

3. Using pipelining and superscalarity, we can process all the nodes belonging to two or more levels simultaneously.

This approach requires $\lceil \log k \rceil$ passes of APPLY and REDUCE phases. In Section 3.9.5 we present the performance comparison between computing a MULTIWAY AND iteratively and computing it employing superscalarity and pipelining.

```
df_substitute(F)
    if (terminal case(F)) return result;
    if (computed table has entry(F)) return result;
    let x be the top variable of F;
    if (x is to be substituted){
        g = function substituting x ;
    }
    else g = x ;
    T = df_substitute(F_x);
    E = df_substitute(F_x');
    return df_ite(g, T, E);
```

Figure 3.15 Depth-first algorithm for SUBSTITUTION.

3.7 Optimized BDD Algorithms

We have incorporated iterative breadth-first technique, superscalarity, and pipelining into a comprehensive set of high performance BDD algorithms for Boolean operations such as AND, OR, XOR, NAND, NOR, XNOR, ITE, COFACTOR, RESTRICTION, COMPOSITION, SUBSTITUTION, EXISTENTIAL QUANTIFICATION, UNIVERSAL QUANTIFICATION, RELATIONAL PRODUCT, and VARIABLE SWAPPING. In the following sections we describe a few of these algorithms. Each of these new algorithms raises specific issues that must be addressed to obtain a high performance BDD package. To the best of our knowledge, this is the first effort in this direction. In Section 3.9.3, we demonstrate the performance of our algorithms.

3.7.1 Substitute

In this operation a set of variables in the argument BDD is simultaneously substituted by a set of functions. Let $F(x_1,...,x_n)$ be a Boolean function. Without loss of generality, suppose variables $x_1,x_2,...,x_k$ in function F are to be substituted by functions $G_i(x_1,...,x_n), i = 1,..,k$. Then the result of the substitution operation is defined by the function $H(x_1,x_2,...,x_n) = F(G_1(x_1,...,x_n),G_2(x_1,...,x_n),...,G_k(x_1,...,x_n),x_{k+1},...,x_n)$.

The conventional depth-first algorithm for SUBSTITUTION is given in Figure 3.15. The algorithm recursively computes the BDD T and E that result from SUBSTITUTION operation on each of THEN and ELSE cofactors. However, unlike a simple BDD operation such as AND, each recursive call to SUBSTITUTION computes an ITE operation for BDDs. The need to perform an ITE operation on the results of the cofactors makes the SUBSTITUTION operation computationally more complex than other BDD operations. If the BDD size exceeds the main memory capacity, the SUBSTITUTION operation results in a large number of page faults.

The important step of the breadth-first algorithm for performing SUBSTITUTION is shown in Figure 3.16. At the end of the APPLY phase, the unreduced BDD constructed is structurally identical to the BDD for the function F. While processing a REQUEST during the REDUCE phase for an index i, we update its THEN and ELSE BDDs to T and E, where T and E BDDs are results of SUBSTITUTION on THEN and ELSE respectively. Next, we perform ITE operations. The result of the ITE operation is the BDD obtained by performing SUBSTITUTION for the REQUEST under consideration.

Note that it is not necessary to perform ITE operations one at a time for each of the REQUESTs at the given level. The concept of superscalarity in the breadth-first paradigm enables us to perform multiple ITE operations for all the REQUESTs for a specific index simultaneously during the REDUCE phase for that index. Since a set of ITE operations are computed for each index, the maximum number of the APPLY and the REDUCE passes are bounded by the number of variables. Employing superscalarity on the ITE operations significantly improves the overall performance.

3.7.2 Existential Quantification

EXISTENTIAL QUANTIFICATION of a function f with respect to a variable x is given by $\exists_x f = f_x + f_{x'}$. With respect to a set of variables $\mathcal{X} = \{x_1, x_2, \ldots x_n\}$, this is given as, $\exists_{\mathcal{X}} f = \exists_{x_n}(\exists_{x_{n-1}} \cdots (\exists_{x_1} f))$.

The conventional depth-first algorithm for EXISTENTIAL QUANTIFICATION (given in Figure 3.17) is quite similar to the depth-first recursive algorithm for other Boolean operations such as AND OR, and XOR. However, it is different from other basic depth-first algorithms in two aspects, both of which are related to the quantified variables. First, during the recursion, if the current variable x is to be quantified from the func-

```
bf_substitute_reduce(index)
    x is variable with index "index";
    if (x is to be substituted){
        g = function substituting x ;
    }
    else g = x ;
    /* process each request queue */
    while (request_queue[index] not empty){
        /* process each request */
        REQUEST (F, G) = unprocessed request from REQUEST QUEUE[index];
        Update THEN and ELSE cofactors;
        if (THEN ≡ ELSE ) forward REQUEST to THEN ;
        else {
            create an ITE REQUEST (g, T, E);
            forward REQUEST to the ITE REQUEST;
        }
    Perform superscalar ITE ;
    Update the forwarding information of REQUESTS;
```

Figure 3.16 Breadth-first substitute operation algorithm - REDUCE phase.

```
df_exist(F)
    if (terminal case(F)) return result;
    if (computed table has entry(F)) return result;
    let x be the top variable of F;
    T = df_exist(F_x);
    if (x is to be quantified and T == 1) return 1;
    E = df_exist(F_x');
    if (x is to be quantified) return df_or(T, E);
    else result = find or add in the unique table (x, T, E);
    return result;
```

Figure 3.17 Depth-first algorithm for existential quantification.

tion, we compute OR of the result of quantification on two cofactors. Since the OR is performed for each REQUEST for all the quantified variables, the QUANTIFICATION operation is usually more expensive than other basic Boolean operations. Second, if a variable is quantified, the recursion can possibly terminate early. For a quantified variable, if the result of one of the cofactors is 1, it is no longer necessary to recursively quantify the other cofactor, since in this case the BDD representing disjunction of quantified cofactors is available immediately. Therefore, it is possible to significantly prune the recursion for the QUANTIFICATION algorithm.

A naive breadth-first algorithm for QUANTIFICATION is as follows. During APPLY phase, both the cofactors of a REQUEST are processed. At the end of the APPLY phase, the unreduced BDD created is structurally identical to the BDD for the function f. While processing a REQUEST during the REDUCE phase for an index i, we update then and else cofactors (given by T and E). Next, if variable with index i is quantified, we compute the BDD for the OR of T and E. If the variable is not quantified, the REQUEST under consideration is the desired BDD. Superscalarity is employed in performing the OR operations simultaneously for the quantified variables. However, since we process both the cofactors during the APPLY phase, the pruning of recursion for the depth-first QUANTIFICATION algorithm is not available for this naive breadth-first algorithm.

The savings in computation due to pruning can be quite significant, and hence the naive breadth-first QUANTIFICATION algorithm is expected to perform poorly against the conventional depth-first algorithm, especially for BDDs that fit within the main memory.

To retain the advantage of the breadth-first algorithm while exploiting the pruning feature of the depth-first QUANTIFICATION algorithm, we propose a new mixed breadth- and depth-first approach. The essence of the new algorithm is to follow the depth-first technique for the quantified variables and breadth-first technique for the un-quantified variables. For REQUESTs belonging to un-quantified variables, we process both the cofactors, i.e., we process a REQUEST by issuing, in general, two new REQUESTs. However, for each quantified variable, we process each REQUEST as in the case of the depth-first algorithm, which means only one new REQUEST is issued – a REQUEST that corresponds to one of the quantified cofactors. The result BDD for each quantified cofactor REQUEST is obtained simultaneously (using superscalarity) by recursively using the new breadth-first QUANTIFICATION algorithm. If the resulting quantified cofactor is different from unity, a new REQUEST is issued to compute the other quantified cofactor. This avoids redundant computation if the result is a tautology. Once the result of QUANTIFICATION is available for both cofactors, we employ superscalarity to compute the set of multiple independent OR operations for the quantified variables. Note that processing of one cofactor at a time for quantified variable does not amount to path-by-path traversal of BDD as in the depth-first technique. Since we process the cofactors of all the REQUESTs at any index, we essentially are performing - multi-path traversal. The idea is to retain the locality of access of breadth-first manipulation while leveraging the quantification efficiency of depth-first scheme.

The RELATIONAL PRODUCT of functions f and g with respect to a set of variables $\vec{x}$ is the QUANTIFICATION of these variables from the product of f and g. Therefore the RELATIONAL PRODUCT algorithm works along the same lines.

3.7.3 Compose

The COMPOSE operation is a special case of SUBSTITUTION where in a function F, variable y is substituted by another function. Hence COMPOSE can also be optimized along the lines of the SUBSTITUTION algorithm. In a function F with top variable x,

to substitute a variable y by a function G, we have three possibilities depending on the values of y_{index} (index of variable y) and x_{index} (index of variable x).

1. $y_{index} = x_{index}$: In this case we compute ITE$(G, F_x, F_{\bar{x}})$.

2. $y_{index} < x_{index}$: In this case the function F is independent of the variable y and hence no computation need to be performed.

3. $y_{index} > x_{index}$: In this case we need to compute the results of COMPOSE(F_x, y, G_x) and COMPOSE$(F_{\bar{x}}, y, G_{\bar{x}})$. Unlike the SUBSTITUTION operation, we need to compute ITE operations only for one variable that is substituted, which makes COMPOSE computationally less complex than SUBSTITUTION. It exploits superscalarity by simultaneously computing multiple independent ITE operations for the substituted variable.

3.7.4 Swapping Variables

Given the BDD for a Boolean function $F(x_1, x_2, ..., x_i, ..., x_j, ..., x_n)$, SWAP VARS function obtains a new Boolean function $G(x_1, x_2, ..., x_j, ..., x_i, ..., x_n)$, i.e., the variables x_i and x_j are swapped in function F. An optimized depth-first algorithm exists for SWAP VARS. which is a special case of the SUBSTITUTION operation. However, the recursive depth-first algorithm for SWAP VARS is conceptually complex because it calls several different recursive functions. The depth-first algorithm is shown in Figures 3.18 and 3.19. Without loss of generality, the algorithm assumes that index of x is less than that of y. The recursion considers the following three cases: i) If the index of the top variable of F exceeds the index of x, then F does not contain x. Hence, compositions of F with $y = 0$ and $y = 1$, give the left and the right cofactors of the result BDD with the top variable x. ii) If the index of the top variable of F is less than the index of x, then we recursively call the SWAP VARS routine for THEN and ELSE cofactors. iii) If the index of the top variable of F equals the index of x, then we call another auxiliary recursive routine that computes the BDDs for the expressions $y(F_x|_{y=1}) + \bar{y}(F_{\bar{x}}|_{y=1})$ and $y(F_x|_{y=0}) + \bar{y}(F_{\bar{x}}|_{y=0})$ that give the left and the right cofactors of the result BDD with the top variable x.

The design of the breadth-first algorithm from this relatively complex depth-first algorithm proceeds by using the following principles: i) each different type of recursion

```
SwapVars(f, x, y){
    if (f == Constant) return f;
    if (CacheLookUp(f,x,y, result)) return result;
    if (f_index > x_index){
        t1 = Compose (f,y,0);
        t2 = Compose (f,y,1);
        result = FindOrAddInUniqueTable(x_index,t1,t2);
    }   else if (f_index < x_index){
        t1 = SwapVars(f_then,x,y);
        t2 = SwapVars(f_else,x,y);
        result = FindOrAddInUniqueTable(f_index,t1,t2);
    }   else {
        t1 = SwapVarsAux(f_then, f_else,y,0);
        t2 = SwapVarsAux(f_then, f_else,y,1);
        result = FindOrAddInUniqueTable(x_index,t1,t2);
    }
}
```

Figure 3.18 Depth-first algorithm for swapping variables.

– (COMPOSITION, ITE as part of COMPOSITION, SWAPVARS, SWAPVARSAUX with $y = 1$ and SWAPVARSAUX with $y = 0$) – is performed using a separate set of REQUEST QUEUEs; therefore, a total of five separate set of REQUEST QUEUEs are maintained, ii) a depth-first recursive call is replaced by the processing of a REQUEST in the breadth-first algorithm.

3.8 Implementation Details

3.8.1 Data Structure

The important issues in designing the BDD node data structure are the following:

Compact Representation The size of the BDD node should be as small as possible, because most of the memory is used by BDD nodes. Further we need to efficiently use the memory so as to fit as many nodes as possible in a given level of

```
SwapVarsAux(f1, f2, h, flag){
    if (f1_index == h_index){
        if (flag == 1) f1 = f1_then ;
        else f1 = f1_else
    }
    if (f2_index == h_index){
        if (flag == 1) f2 = f2_then
        else f2 = f2_else
    }
    if (f1 == f2){
        if (flag == 1) return Compose(f1, h, 1);
        else return Compose(f1, h, 0);
    }
    if (f1_index == h_index) AND (f2_index == h_index){
        result = FindOrAddInUniqueTable(h_index, f1, f2);
        return result;
    }
    if (CacheLookUp(swapVars, f1, f2, result)) return result;
    minId = TopVar(f1, f2);
    Cofactor(f1, minId, f11, f12);
    Cofactor(f2, minId, f21, f22);
    t1 = SwapVarsAux(f11, f21, h, flag);
    t2 = SwapVarsAux(f12, f22, h, flag);
    result = FindOrAddInUniqueTable(minId, t1, t2);
    CacheInsert(swapVars, f1, f2, result);
}
```

Figure 3.19 Auxiliary routine for SWAP VARS.

```
struct Bdd {
  int bddIndex;                 /* 2 Bytes */
  struct BddNode *bddNode;      /* 4 Bytes */
}
```

```
struct BddNode {
  struct BddNode *next;         /* 4 Bytes */
  struct Bdd thenBdd;           /* 6 Bytes */
  struct Bdd elseBdd;           /* 6 Bytes */
}
```

Figure 3.20 BDD and BDD node data structure.

the memory system hierarchy.

Variable Index Determination As mentioned in the section 3.2, the variable index determination is crucial to regular memory accesses. In particular, we should avoid fetching the BDD from memory to determine its index.

We propose the BDD data structure given in Figure 3.20. We represent a BDD using {variable index, BDD node pointer} pair. Unlike the conventional BDD data structure that stores its variable index in the BDD node, the new BDD data structure stores the variable indices of its THEN and ELSE BDD nodes. The new BDD node data structure is very compact: it requires 16 Bytes and 28 Bytes on 32-bit and 64-bit architectures respectively, which is the same as the memory required to represent the conventional BDD node structure. A customized memory allocator is used to align the BDD nodes to quad-word boundaries so that a total of 12 bits (last four bits of THEN, ELSE, and NEXT pointers) can be used to tag important data such as complement flags, marking flags, and the reference count. The tag bits are assigned so as to minimize the amount of computational overhead.

3.8.2 Memory Management

We use a customized memory manager to allocate and free BDD nodes in order to ensure locality of reference. We associate a *node manager* with each variable index. A BDD node is allocated by the node manager associated with the index of the node. The node manager maintains a free list of BDD nodes that belong to the same index. Memory blocks are allocated to a node manager in such a way that all the BDD nodes in the free list are aligned on a quad-word boundary. In addition to providing 12 free bits as explained above, the quad-word alignment helps improve the cache performance as it maps a BDD node to a single cache line.

3.8.3 Miscellaneous Details

Overloading of REQUEST data structure: A BDD node is obtained from a REQUEST node that is not redundant. To overload the use of the REQUEST data structure with the BDD node data structure [AC94], the REQUEST data structure is limited to 16 bytes. Before the APPLY phase, each REQUEST represents operand BDDs, each of which requires 6 bytes. Therefore, we allow only two operand operations. Three operand operations such as $ITE(f,g,h)$ are simulated by using indirect addressing and two request nodes.

Hashing function: Since hash tables are used extensively in the breadth-first BDD manipulation, it is important to optimize their performance. The analysis of source code revealed that about 15% of the total CPU time is spent in computing the unsigned remainder when using a prime number hashing function. A power of two hashing function reduces this time drastically without degrading the hash table access performance.

3.8.4 Repacking After Garbage Collection

Sometimes garbage collection results in a large number of free nodes. Since the memory pages containing the nodes are associated with a particular variable, it is possible that after garbage collection, a large number of pages collectively contain small number of nodes. This has the following disadvantages:

1. Because the idle pages cannot be used to allocate nodes for other variables, the memory usage is increased.

```
repack_after_gc{
    for (index = numVariables; index > 0; index--){
        if the current index does not need repacking{
            update the cofactors of the nodes, if necessary;
            continue;
        }
        N1 = number of pages allocated for this index;
        N2 = number of pages required for this index;
        for (page = 1 through N2){
            update the cofactors of the nodes, if necessary;
        }
        for (page = N2 + 1 through N1){
            update the cofactors of the nodes, if necessary;
            copy the contents of the nodes on pages ≤ N2;
        }
        free the pages from N2 + 1 through N1.
    }
}
```

Figure 3.21 Repacking after garbage collection.

2. Since the free nodes cannot be used by another variable, it could possibly lead to memory allocation problems for some other variables.

3. Since a small number of nodes are scattered on a large number of pages, it leads to non-local memory access during the UNIQUE TABLE traversal.

To overcome this problem, we adopt the strategy of repacking the nodes after every garbage collection which results in a large number of dead nodes. Essentially, we perform a bottom-up traversal of the BDD; at each index after deciding on the number of pages required, we copy the node contents from extra pages to make it as compact as possible. The idle pages can then be recycled to be used by other variables. The algorithm is shown in Figure 3.21.

Notice that we need to perform just one bottom-up pass of BDD nodes and for each page, the nodes are touched in consecutive memory locations. Hence computational

overhead of this algorithm is very low and due to excellent locality of access in the algorithm, it leads to very small run-time overhead.

3.8.5 Node Reallocation for Cache Locality

Before creating a new BDD node during the REDUCE phase of each operation, we need to look-up in the UNIQUE TABLE for its existence. This leads to a traversal of the collision chain in the table, if one exists. Often this chain contains nodes belonging to different pages and most likely different cache lines. Hence during the chain traversal, we incur a large number of cache misses and possibly page faults as well.

We propose a strategy where nodes are reallocated in the memory such that on an average, the collision chain traversal does not result in more than one cache miss. This strategy has no memory overhead (the reallocation is done in-place) and requires only two passes of the whole BDD. The first pass is done on a page-by-page basis, leading to local memory accesses. The second pass requires tracing the next pointers of the nodes which could be on multiple pages. However, the resulting improvement in performance due to cache locality far offsets this computational cost. The algorithm is shown in Figure 3.22 and the node layout before and after reallocation is shown in Figure 3.23. An important aspect of this technique is that it is useful for both depth-first and breadth-first traversal based manipulations. In the current work we have not implemented this algorithm.

3.9 Experimental Results

The algorithms described in this chapter were implemented in a comprehensive BDD package – CAL. The architecture and some implementation details of this package are briefly described in Appendix A. In this section we demonstrate the performance of our package. For experimental purposes, we integrated our package within SIS [SSL+92], and compared the performance against two depth-first manipulation based packages – Long's BDD package [Lon93] and Colorado decision diagram package [Som97].

```
reallocate_nodes{
    /* First pass, get the new address */
    for (index = numVariables; index > 0; index--){
        update the cofactors of the nodes;
        find the collision chain length for each bin;
        initialize the pointer for each bin;
        traverse the nodes on page-by-page basis {
            put the new address of the node in the NEXT pointer;
        }
    }
    /* Second pass, copy the nodes on to new addresses */
    for (index = numVariables; index > 0; index--){
        foreach page corresponding to the index{
            while there are still nodes to be updated {
                    tmp = content of next pointer of node;
                    store the content of the node in the next pointer;
                    node = next pointer of the new location (tmp);
            }
        }
    }
}
```

Figure 3.22 Reallocating nodes to achieve cache locality.

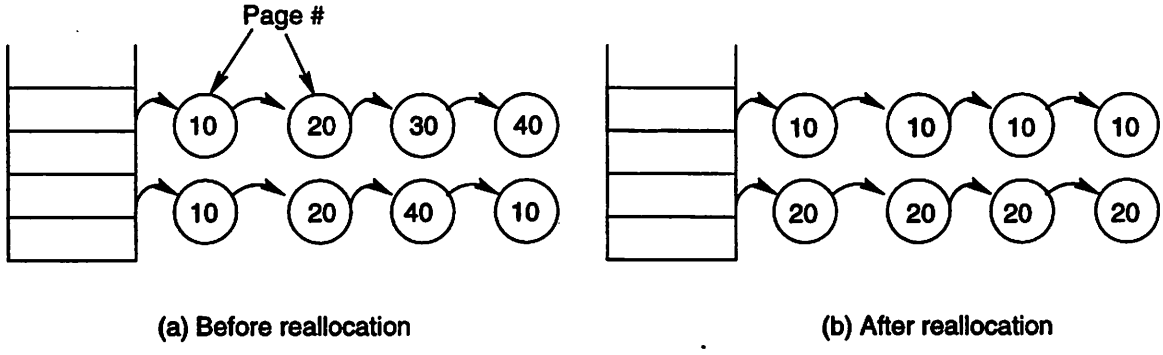


Figure 3.23 Node allocation before and after fixing collision chains.

3.9.1 Experimental Setup

All our experiments were performed on a DEC5400 with 128KB processor cache, 64 MB main memory and 1GB of disk storage. An alpha version (0.1) of the CAL package was used for the experiments presented here. The latest version for CAL (2.0), is much more efficient, both computation and memory consumption wise. However, due to lack of time all the experiments with the latest CAL package could not be reperformed. Results in Section 3.9.8 use version 2.0 of CAL.

In addition to using standard ISCAS and MCNC benchmark examples for the set of experiments, we use a series of sub-networks of the MCNC benchmark C6288 in order to systematically analyze the performance of our algorithms as BDD size increases. These artificially created examples have the property that the number of BDD nodes needed to represent the BDDs corresponding to the outputs are roughly multiples of one million. This enabled us to illustrate the gradual change in various performance metrics with the change in example size. These examples are denoted as "C6288_iM.blif", implying that the total number of BDD nodes in the manager after computing the BDDs for the outputs of C6288_iM.blif is i millions.

We use "dfs-ordering" in SIS to order the variables. The number of BDD nodes needed to represent a particular circuit may be significantly different from those reported in the literature (e.g. [AC94]) due to a different variable ordering. However, this issue is orthogonal to demonstrating the performance of our package for a given variable ordering.

The following set of experiments were carried out to demonstrate the performance of our BDD package:

1. For each of the benchmark examples, we create the BDDs for the outputs of the circuit. The number of nodes in the BDDs range from a few thousands to tens of millions.

2. We compare the performance of various BDD operations in our package with those in Long's package. We use the output BDDs as argument BDDs in our experiments. For instance, to compare the performance of the AND operation, we iteratively select random pairs from the output BDDs and compute the AND of the pair. Similarly, to compare the performance of QUANTIFICATION operation,

we randomly select one of the output BDDs and a set of variables to be quantified. The same functions and variables are selected for both packages.

3. To estimate the performance improvements from superscalarity, we compare the time taken in performing independent BDD operations with and without employing superscalarity.

4. Similarly, we compare the time taken in performing dependent BDD operations with and without employing pipelining to assess the gains from pipelined operations.

5. We also estimate the memory overhead in breadth-first manipulations with increasing value of pipedepth.

6. The improvement in memory usage due to repacking after garbage collection is shown.

7. Finally, we present few results from the latest CAL package (version 2.0) by comparing it against Long's package and Colorado decision diagram package [Som97].

3.9.2 Creating Output BDDs for Circuits

Small and Medium Size Examples

From Table 3.1, We observe that our package has competitive performance for most of the examples Long's BDD package, and we achieve a performance improvement upto a factor of 1.2 on some examples. However, for examples s15850, s35932, and s35854 our package performs significantly worse than Long's package. Upon analysis we found that these examples share one property, which is that even though BDD sizes were small, all three of them have a large number of primary inputs. This results in small number of BDD nodes for each variable. In this case the penalty of traversing the REQUEST QUEUE on a level by level basis becomes dominant and results in increased computation time.

Large Size Examples

In Table 3.2 we present the performance for output BDD creation for large examples.

Example	# Nodes	CPU Time (in secs)		Ratio
		Long's	CAL	
C1355	139998	15.30	13.57	1.13
C1908	56842	5.57	4.27	1.30
C432	193671	21.66	17.29	1.25
C499	109685	12.78	10.75	1.19
C5315	111798	10.39	11.13	0.93
C880	38721	3.38	3.02	1.12
s1423	56178	4.94	3.99	1.24
s15850	188035	19.36	33.09	0.59
s35932	25557	3.13	27.56	0.11
s38584	93055	10.10	30.99	0.33

Table 3.1 Performance comparison for creating output BDDs with Long's BDD package.

A: Long's BDD package

B: Our package.

Example	# Nodes	CPU Time		Elapsed Time		# Page Faults	
		Long's	CAL	Long's	CAL	Long's	CAL
C6288_1M	1,001,855	112	98	127	110	0	0
C6288_2M	2,066,878	273	215	403	306	0	0
C6288_3M	3,123,327	491	347	21281	1218	502059	17800
C6288_4M	4,273,510	820	490	106110	2433	2661738	42509
C6288_5M	5,337,005	t.o.	631	–	4140	–	80621
C6288_6M	6,381,496	–	804	–	6295	–	126977
C6288_7M	7,489,064	–	981	–	8454	–	168794
C6288_9M	9,193,222	–	1147	–	10864	–	213976

Table 3.2 Performance comparison for creating output BDDs: Long's BDD package (A) vs. our package (B).

t.o:Process killed after 21.5 hours of elapsed time.

–: Not tried since.

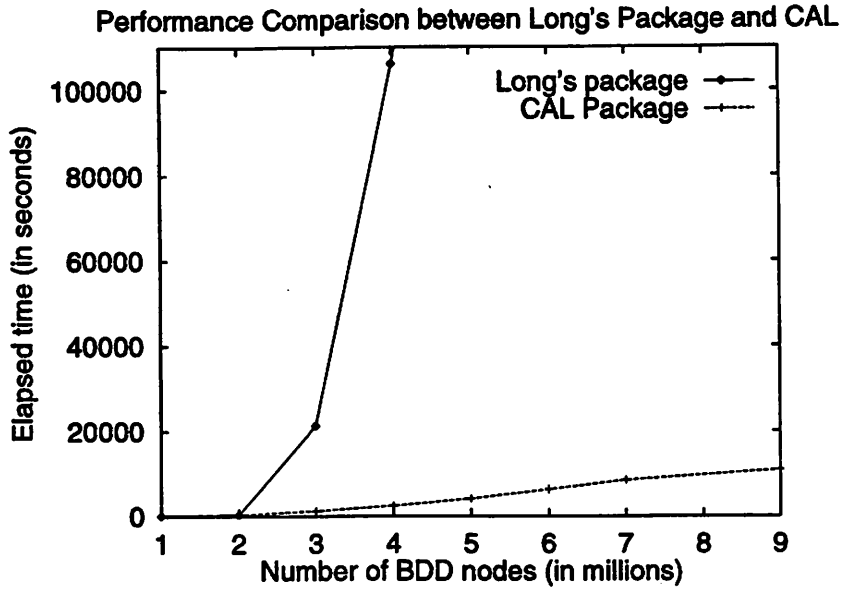


Figure 3.24 Variation of elapsed time with example size.

When the number of BDD nodes becomes too large to fit in the main memory, the number of page faults and the elapsed time increase drastically for Long's package. In Figures 3.24 and 3.25, we show the number of page faults and the elapsed time as a function of example size. For Long's package, an increase in the BDD size beyond the main memory size results in a sharp increase in the number of page faults and explains excessive elapsed time. This is in contrast to the page fault behavior of our package which increases linearly with an increase in the example size.

Very Large Size Examples

Table 3.3 gives the performance of our package on building very large BDDs for some benchmark examples. It is seen that BDDs with more than 23 million nodes are built in less than nine hours.

We also made a "black box" comparison with the BFS algorithm (developed at NEC) [AC94] on SUN Sparc2 workstation with 40MB main memory. The results for creating output BDDs for C6288 sub-circuits are shown in Table 3.4. On average our

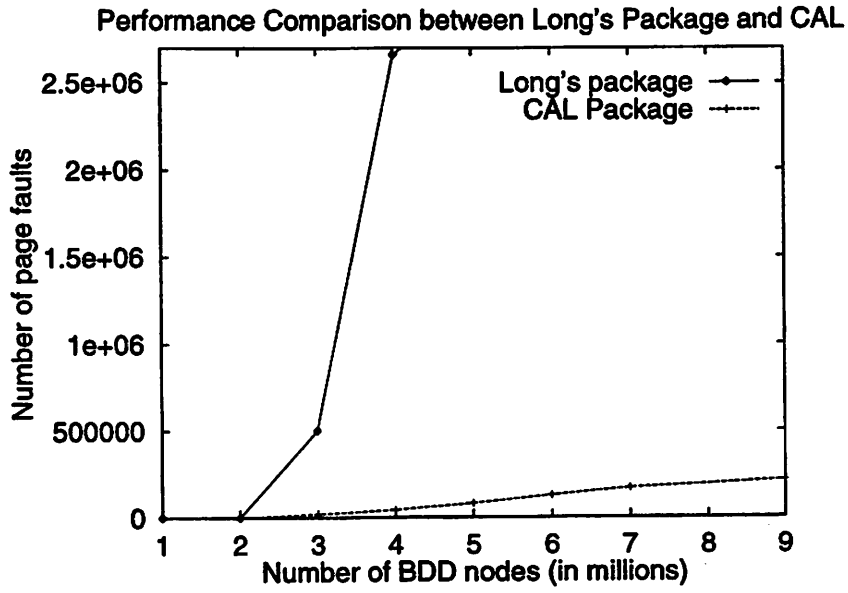


Figure 3.25 Variation of number of page faults with example size.

Example	# Nodes	Elapsed Time	# Page Faults
C2670	1.04×10^6	4 hrs 4 mins 58 secs	357005
C3540	2.76×10^6	25 mins	26603
C6288_12M	12.80×10^6	6 hrs 39 mins 54 secs	719697
s38417	23.15×10^6	8 hrs 49 mins 26 secs	868442

Table 3.3 Performance comparison for creating output BDDs for some very large examples.

Example	Elapsed Time		
	CAL	NEC	Ratio
C6288_1M	85	298	3.5
C6288_2M	180	631	3.5
C6288_3M	506	2558	5.1
C6288_4M	931	4603	4.9
C6288_5M	1643	6284	3.8
C6288_6M	2476	13361	5.4
C6288_7M	3386	16576	4.9

Table 3.4 Performance comparison with breadth-first approach by Ashar *et al.*.

approach is faster by a factor of 4.4 for creating output BDDs for C6288 sub-circuits with the number of BDD nodes ranging from one million to seven million. The performance improvement is mainly due to the new implementation technique, and the use of superscalarity, and pipelining. Note, however, due to slightly different orderings, the number of BDD nodes were not exactly identical in the two cases which could also have an impact on the performance.

3.9.3 Performance Comparison For Various BDD Operations

One of our objectives was to provide a comprehensive set of algorithms for all BDD operations. In the following subsections we compare the performance of some of our algorithms with those of Long's package. We provide the comparison with respect to small and medium sized examples only, since for large examples Long's package will have the obvious disadvantage of excessive page faults.

Performance Comparison for Small Size Examples

All examples considered in this category have less than 7000 BDD nodes. This implies that with a processor cache size of 128KB, it is possible that all the nodes can reside in the cache if node addresses are properly aligned. Since our node data structure is quad word aligned, the node address does not overlap across cache lines. Hence we can expect a significant cache hit rate during BDD manipulations. Long's package,

however, does not provide the word alignment and hence it is likely that the BDD node addresses could overlap across cache lines. We ran experiments to compare the performances of various BDD operations. We observed a performance ratio of 1.92 over Long's BDD package across all small sized examples and four BDD operations.

Performance Comparison for Medium Size Examples

In Table 3.5 we provide the performance comparison between packages for medium size examples. In this case the number of nodes are of the order of tens of thousands to hundreds of thousands, and the cache effect seen for the small size examples is not dominant. However, in most of the cases, we observe a performance improvement over Long's package. Overall performance ratio over all medium sized examples and across four BDD operations given in the tables, is about 1.5. The most significant is the relative performance on the SUBSTITUTE operation. We observe that on many examples, Long's package could not finish the SUBSTITUTION in 10,000 CPU seconds whereas our package took just about 1000 CPU seconds to complete. This substantiates the significant performance enhancement using superscalarity as mentioned in Section 3.7.1.

We notice that for the QUANTIFICATION operation (bdd_exist) our package consistently performs worse than Long's package by up to a factor of 0.6. Upon investigation we found that this was caused by book-keeping overhead of the multi-path traversal approach. It should be mentioned however that for large examples (ones which do not fit the main memory), all our operations consistently perform better than Long's package. Hence, for smaller sized examples we revert back to the depth-first strategy for QUANTIFICATION.

Example	BDD Operation											
	CPU			Elapsed			CPU			Elapsed		
	A	B	A/B	A	B	A/B	A	B	A/B	A	B	A/B
	bdd_and						bdd_substitute					
C1355	16.68	14.45	1.15	17	15	1.13	t.o.	1005.32	–	–	14710	–
C1908	7.80	7.05	1.11	8	7	1.14	t.o.	62.65	–	–	67	–
C432	18.23	17.77	1.03	19	18	1.06	3144.85	258.89	12.15	4746	476	9.97
C499	15.53	14.64	1.06	16	15	1.07	t.o.	1002.21	–	–	14749	–
C5315	12.01	6.52	1.84	16	7	2.29	0.91	0.16	5.69	1	1	1
C880	9.03	7.29	1.24	10	8	1.25	1.05	1.18	0.89	1	1.0	1
s1423	2.25	2.48	0.91	2	3	0.67	292.68	88.73	3.3	350	97	3.6
Example	**bdd_exist**						**bdd_swapvars**					
C1355	45.99	63.05	0.72	47	65	0.72	30.42	23.78	1.28	31	25	1.24
C1908	13.62	21.30	0.64	14	22	0.64	9.34	8.22	1.14	9	8	1.12
C432	59.08	99.80	0.59	60	102	0.59	33.94	29.67	1.14	34	31	1.10
C499	40.70	59.28	0.69	41	60	0.68	29.15	23.81	1.22	30	24	1.25
C5315	32.97	35.65	0.92	34	37	0.92	4.12	3.52	1.17	4	4	1
C880	17.28	21.98	0.79	18	22	0.82	10.65	8.62	1.24	11	9	1.22
s1423	16.57	28.31	0.59	17	29	0.59	1.63	2.34	0.70	2	3	0.67

Table 3.5 Performance comparison on medium size examples for "And", "Substitute", "Existential Quantification , and "SwapVars" operations: Long's BDD package (A) vs. our package (B).

t.o.: Time out after 10,000 CPU seconds.

Example	# Page Faults		CPU		Elapsed		Ratio
	w/o SS	w/ SS	w/o SS	w/ SS	w/o SS	w/ SS	
C6288_4M	485.82	436.94	2214	2452	53261	63272	0.85
C6288_5M	630.37	602.50	5333	4648	172205	137004	1.26
C6288_6M	815.91	724.99	18354	7840	712868	252323	2.83
C6288_7M	956.89	831.30	24747	10267	970192	328086	2.96

Table 3.6 Performance improvement using superscalarity for creating output BDDs.

3.9.4 Performance Enhancement Due to Superscalarity

We demonstrate the power of superscalarity on three different applications: creating output BDDs, ARRAY AND, and QUANTIFICATION.

Creating Output BDDs

In section 3.6.1 we described how superscalarity can be exploited for creating output BDDs. In Table 3.6, we show the performance improvement achieved by employing superscalarity. We observe that in all the cases, employing superscalarity results in better performance. Also in all the cases except C6288_4M, we observe that the number of page faults decreases with the use of superscalarity and we achieve a better performance by a factor of more than 2.

ARRAY AND and QUANTIFICATION

In Table 3.7, we present the results of using superscalarity for ARRAY AND and QUANTIFICATION. In ARRAY AND we are given an array of operand BDD pairs and we need to compute the AND of each of the operand pair. These operands were randomly chosen from the set of output BDDs of the circuit. For the quantification operation, we perform OR operations one at a time during its REDUCE phase (see Section 3.7.2) to illustrate the effect of superscalarity. For small circuits, superscalarity improves the performance due to inter-operation caching. For large circuits, superscalarity improves the performance due to increased locality of memory accesses, resulting in fewer page faults. We observe from Table 3.7 that for examples which fit in the main memory,

Example	BDD Operation									
	Array And						Quantify			
	CPU in secs		Elapsed in secs		# Page Faults in 1000		CPU in secs		Elapsed in secs	
	W	X	W	X	W	X	Y	Z	Y	Z
C1355	134.03	119.41	137	123	0	0	13.40	12.30	13	12
C6288_1M	411.35	403.90	847	740	0	0	5.02	4.27	5	5
C6288_2M	290.55	283.41	595	581	0	0	18.17	16.16	18	17
s1423	35.39	18.02	37	18	0	0	20.63	19.13	31	30
C6288_3M	682.57	655.21	21005	7178	406	109	12.60	11.59	13	12
minmax10	810.32	679.88	19304	1619	326	9.2	66.0	55.8	91	81

Table 3.7 Performance improvement using superscalarity for ARRAY AND and QUANTIFICATION BDD operations.

W: ARRAY AND performed iteratively.

X: ARRAY AND performed in superscalar manner.

Y: In REDUCE phase of QUANTIFICATION, OR operations performed one be one.

Z: In REDUCE phase of QUANTIFICATION, OR operations performed in superscalar manner.

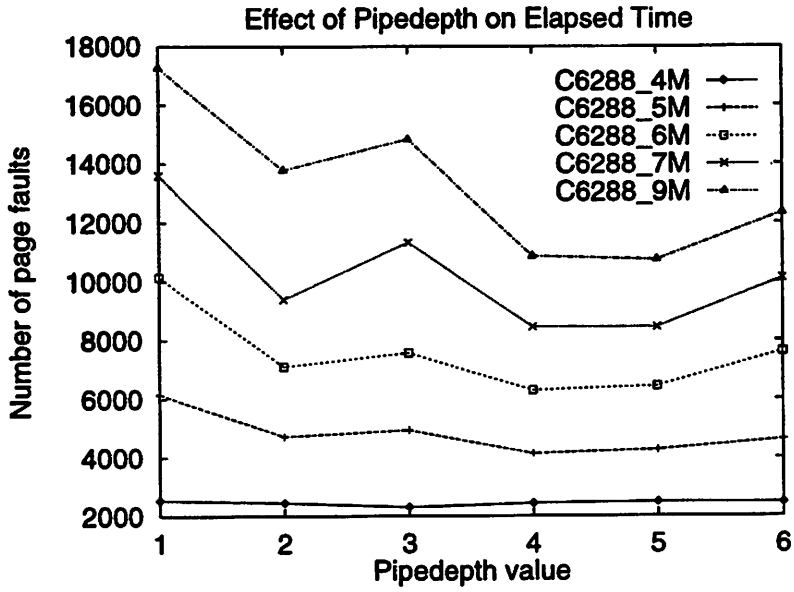


Figure 3.26 Variation of elapsed time with pipedepth in creating output BDDs.

superscalarity helps with improved CPU time. For large examples a performance improvement of upto a factor of 10 (minmax10) is obtained.

3.9.5 Performance Enhancement Due to Pipelining

In this section, we demonstrate the effect of pipelining on two different applications.

Creating Output BDDs

We demonstrate the effect of pipelining on the performance of creating BDDs for outputs. We have described in Section 3.6, how pipelining technique can be exploited. Figures 3.26 and 3.27 depict the effect of pipedepth on the elapsed time and the number of page faults for a series of C6288 sub-networks. As pipedepth is increased, we see a decrease in the number of page faults (hence the decrease in elapsed time). However, the memory overhead increases with increase in pipedepth since we are working with unreduced BDDs. Hence, after a certain value of pipedepth, the decrease in page faults due to pipelining is offset by the increase in page faults due to the memory overhead.

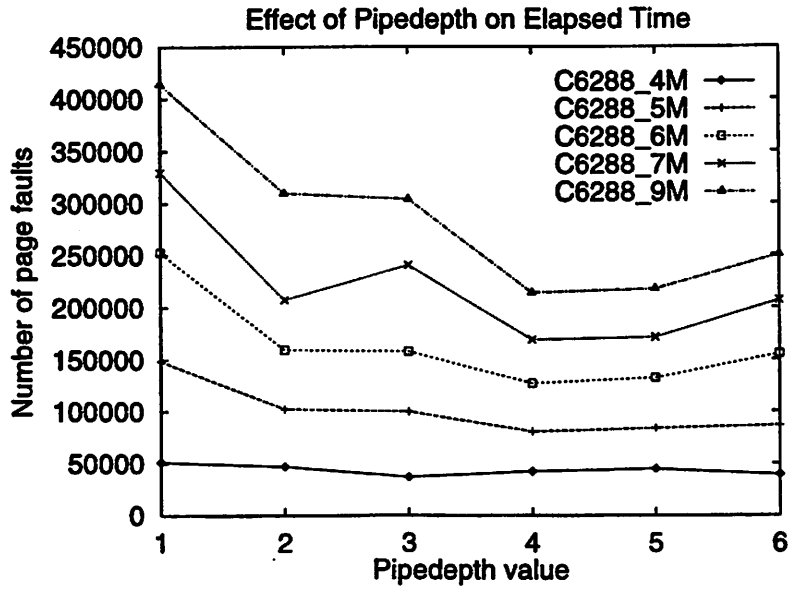


Figure 3.27 Variation of number of page faults with pipedepth in creating output BDDs.

In most cases, a pipedepth of four gives the best results.

MULTIWAY AND

As described in Section 3.6.1, the the MULTIWAY AND operation computes the conjunction of an array of BDDs. We compute the result using i) the pipelined computation approach and ii) the iterative approach (computing the product by successive AND operations). For a MULTIWAY AND operation with n operands, the depth of the pipeline is $\lceil \log_2 n \rceil$. In Table 3.8, we present results for various values of n, and hence various pipeline depths.

Table 3.8 shows that the pipelining technique does not improve the performance for "s1423". This is due to the fact that output BDDs for "s1423" fit in the main memory and hence pipelining cannot improve the memory accesses. However, for "C6288_3M" and "C6288_4M", we observe a performance improvement of up to a factor of 2. This is attributed to the reduction in the number of page faults due to pipelining.

Example	# Nodes	n	CPU Time		Elapsed Time		# Page Faults	
			Iterative	Pipelined	Iterative	Pipelined	Iterative	Pipelined
s1423	57524	2	0.53	0.51	0	0	0	1
		4	4.17	4.03	4	4	0	0
		8	6.00	5.99	7	6	0	0
C6288_3M	3×10^6	2	0.57	0.58	23	26	603	762
		4	100.88	97.10	3449	802	83967	12796
		8	112.68	169.36	8578	2304	212484	45537
C6288_4M	4×10^6	2	0.80	0.63	97	66	2932	2213
		4	107.80	102.99	4602	2307	113325	53077
		8	121.76	183.34	12808	5540	326127	130237

Table 3.8 Effect of pipelining on the performance of MULTIWAY AND.

3.9.6 Memory Overhead in the Breadth First Approach

As noted in Section 3.1, in the breadth-first (BF) technique the isomorphism checks cannot be performed in the APPLY phase. As a result, some temporary BDD nodes are created which are freed during the REDUCE phase. This results in memory overhead inherent in the BF technique. Furthermore, due to operations on unreduced BDDs (pipelining) the isomorphism check reduces even further. In Table 3.9, we give the memory overhead involved in computing "AND" operation on two random BDDs selected from BDDs for the outputs. The memory overhead is computed as the ratio of extra nodes created during the APPLY phase (E) to the total number of nodes in the unique table (U). In Table 3.10, we provide the overhead for the "Multiway-And" operation, for various values of pipe-depth. We observe the memory overhead incurred increases with increase in pipedepth. But generally the overhead is small, on average it is about 5% (for pipedepth value of four).

3.9.7 Repacking After Garbage Collection

We performed some experiments to evaluate the viability of the repacking algorithm as described in Section 3.8.4. In Table 3.11 we notice that up to 30% improvement in memory consumption is achieved. The associated computational overhead is not significant. As a matter of fact, due to repacking the locality of memory access in-

Example	# Nodes	E/U for "AND"
apex6	3312	0.0000
i9	12423	0.0000
minmax5	4812	0.0009
s1196	5537	0.0047
s1238	5687	0.0046
s1494	2028	0.0142
s298	225	0.0103
s344	406	0.0051
s349	406	0.0034
s420	1039	0.0415
s641	2003	0.0025
tlc	310	0.0372
x1	4305	0.0025
cbp.32.4	8234	0.0008
s1423	57254	0.0000
sbc	3008	0.0060
C1355	242732	0.0458

Table 3.9 Memory overhead involved with breadth first manipulation technique in performing "AND" operation.

creased, leading to better memory access times which offset the computational cost of repacking.

3.9.8 Some Results with CAL-2.0

We performed a selected set of experiments to compare the performance of the latest CAL package (version 2.0) with Long's package (CMU) and Colorado package (version 2.1.1, denoted as CU in the table). These experiments were run on SUN Ultra SPARC, with a 200MHz processor, 16KB/1MB L1/L2 caches, 256MB main memory and 300MB swap space. A time limit of 3000 seconds was used.

From Table 3.12 we observe the following:

Example	# Nodes	E/U ratio for various pipe-depths				
		1	2	3	4	5
i9	12423	0.0000	0.0004	0.0005	0.0011	0.0033
minmax5	4812	0.0006	0.0207	0.0356	0.0450	–
s641	2003	0.0100	0.0058	0.0122	0.0177	0.0295
cbp.32.4	8234	0.0008	0.0023	0.0040	0.0109	0.0157
s1423	57254	0.0009	0.0043	0.0664	0.0733	0.3465
sbc	3008	0.0043	0.0061	0.0106	0.0769	0.0651
C1355	242732	0.0465	0.0531	0.1333	0.1743	0.3566

Table 3.10 Memory overhead as a function of pipe-depth in "Multiway And" operation

1. For the examples which fit in the main memory CAL and CU have the similar performance, both of which outperform CMU.

2. When the BDD size does not fit in the main memory, CAL outperforms both CMU and CU. In particular for C6288_11M and C6288_12M, CAL takes little over 200 seconds to build output BDDs, whereas both CMU and CU run out of time (3000 seconds).

3.10 Conclusions, Related Work, and Future Directions

We have presented new techniques targeting the memory access problem for manipulating very large BDDs. These include i) an architecture independent customized memory manager and new BDD data structures, ii) performing multiple BDD operations concurrently (superscalarity), and iii) performing a BDD operation even when the operand(s) are yet to be computed (pipelining). A complete package consisting of an entire suite of BDD operations based on these techniques has been built. We demonstrate the performance of our package by i) comparing with state-of-the-art BDD package [Lon93], and 2) performing a comprehensive set of experiments to substantiate the capability of our package. We show that our package provides competitive performance on small examples and a performance ratio of more than 100 on large examples.

Related work: Due to the nature of breadth-first traversal, sometimes memory blow-

Example	Memory Consumed (in 8KB)		Elapsed Time (in secs)	
	No Repacking	Repacking	No Repacking	Repacking
every	1627	1306	6.95	6.74
C1355	1971	1971	11.56	12.16
C2670	12582	12580	77.40	74.17
C3540	13671	13607	361.12	346.98
C5315	11419	11227	326.14	323.33
C880	1733	1733	14.62	14.16
abs_bdlc	4069	2833	62.06	59.21
biu	4094	2920	77.78	75.69
bdlc	4153	3109	83.60	80.60
minmax12	1276	1144	16.34	15.72
bigkey	6524	4913	13.36	8.20
s1423	2027	1509	41.18	39.34
s4863	7614	4978	105.33	98.65
s5378	5895	3486	56.94	54.57
s6669	9483	5580	175.11	168.27

Table 3.11 Reduction in memory consumed due to repacking after garbage collection.

up can occur during the APPLY phase. In [YCBO97], a technique was proposed which is based on partial breadth-first expansion. This technique tries to avoid the memory blow-up by controlling the working set size. More specifically, in their approach, BDDs are traversed in a breadth-first manner until the number of temporary nodes reaches a threshold. At that point, these nodes are divided into several groups and further breadth-first traversal is performed on a group-by-group basis. The power of this method comes from the fact that it can leverage the locality of access of breadth-first traversal while controlling the amount of memory used. It seems that with proper implementation, this approach may outperform either the pure depth-first or breadth-first approaches for BDD manipulation.

Many researchers have looked into optimizing cache misses during BDD operations. In [KR97], a new data structure is presented which minimizes cache misses during the

Example	# Nodes	CPU Time (secs)			Elapsed Time (secs)			# Memory Usage (MBytes)		
		CMU	CU	CAL	CMU	CU	CAL	CMU	CU	CAL
C6288_1M	1001950	15	9	8	15	10	8	22.81	28.10	27.27
C6288_2M	2066979	33	20	20	35	22	21	46.66	57.21	27.38
C6288_3M	3123431	56	33	33	58	35	34	71.17	96.36	65.41
C6288_4M	4273617	75	45	46	78	48	48	109.18	117.07	85.81
C6288_5M	5337118	105	56	61	108	59	63	134.35	135.17	105.28
C6288_6M	6381609	128	71	73	131	135	76	149.03	195.79	122.51
C6288_7M	7489177	144	85	93	149	107	98	171.77	215.29	144.96
C6288_9M	9193337	174	104	113	178	274	117	218.23	244.43	180.22
C6288_10M	10414238	220	t.o.	130	226	t.o.	134	238.49	–	208.20
C6288_11M	11812054	t.o.	t.o.	149	t.o.	t.o.	153	–	–	219.63
C6288_12M	12828721	t.o.	t.o.	167	t.o.	t.o.	228	–	–	238.51

Table 3.12 Comparison between CMU, CU, and CAL-2.0 for creating output BDDs.

t.o. : Time out after 3000 seconds.

unique table and computed table look-ups. However, their technique is highly specialized and not amenable for integration in a general purpose BDD package. In [MGS97], it has been empirically established that breadth-first manipulation does not have any advantage over depth-first manipulation in terms of cache locality. A different result is reported in [MQRK97] in which a study is done to benchmark various computer architectures for CAD applications. One of the outcome shown in that work is that cache miss rate for CAL package is 50% less than that in Long's package.

Future directions: In our work, we have observed that most of the non-local accesses occur during the REDUCE phase of breadth-first manipulation. By reallocating nodes for individual unique tables as described in Section 3.8.5, we can expect to reduce the number of cache misses significantly. Interestingly, this technique of minimizing cache misses can also be applied for depth-first traversal.

In the current work, we allow only one kind of operator for multiple BDD operations (say NAND). This requires that all Boolean operations be decomposed as NAND operations. This can have significant overhead if there are many XOR operations (each of which leads to two NAND operations). By allowing multiple operations types during superscalarity and pipelining, we can avoid this overhead. This can be implemented by maintaining different REQUEST QUEUEs for different operations.

Chapter 4

BDDs on a Network of Workstations

In Chapter 3, we proposed techniques to exploit the secondary memory of a workstation for efficient BDD manipulation. In this chapter, we propose a technique to manipulate BDDs on a network of workstations (NOW). A NOW provides a large amount of collective memory resources, both main memories and disks. The collective memory resources of NOW provide a potential to manipulate very large BDDs. To make efficient use of memory resources of a NOW, while completing execution in a reasonable amount of wall clock time, extension of breadth-first technique is used to manipulate BDDs. BDDs are partitioned such that nodes for a set of consecutive variables are assigned to the same workstation. We present experimental results to demonstrate the capability of such an approach and point towards the potential impact for manipulating very large BDDs.

The rest of the chapter is organized as follows. We explain the relevant attributes of the network of workstations in Section 4.1 and that of the BDD algorithm in Section 4.2. After explaining the characteristics of the available resources and the algorithmic requirements, we propose a new BDD algorithm on a network of workstations in Section 4.3. We present the implementation details in Section 4.4 and experimental results in Section 4.5. Most of the work presented in this chapter was first reported in [RSBS96].

4.1 Network of Workstations

A network of workstations is a computing resource that uses as its building block, an entire workstation. These building blocks are interconnected by a local area network such as ethernet, FDDI, switched ethernet, or ATM. Using a network of workstations as a large computer system to solve large scale problems is attractive, since it uses the existing infrastructure as opposed to buying a dedicated scalable parallel computer,

a server, or a shared-memory multiprocessor machine. Further, when the system is upgraded to use faster processors, faster network, larger capacity DRAMs, or larger capacity disks, a network of workstations leverages each of the enhancements.

Let us first understand the nature of NOW computing resource to exploit it fully to match the requirements of BDD algorithms. An existing computing infrastructure with two year old technology may consist of a network of workstations, each with 50 MHz processor, 64KB cache, 64MB main memory, and 200MB of disk space. It takes about 0.1–0.6 microseconds to access data from the main memory and about 6 milliseconds to move a page of memory from the disk to the main memory. The software overhead and latency for a local area network is of the order of about 10 milliseconds and bandwidths are 10 Mbits per second for ethernet and 100 Mbits per second for FDDI network.

From the above performance analysis, the time taken to access the data from the disk or from across the network is about 10000–50000 times more than the time to access the data from the main memory. Over the next few years, the networks are expected to become faster in terms of the latency, the software overhead, and the bandwidth [ACP94]. However, the ratio of time to access the remote memory which involves a network transaction vs. the time to access the main memory is still expected to be the order of 1000. This qualitative analysis has important implication when distributing the BDD nodes across the several workstation memories.

For developing distributed BDD algorithms on NOW, the message passing model of computation is assumed for the following reasons: 1) it closely resembles the underlying NOW architecture and 2) easy availability of robust message passing software in the public domain. The message passing programming model makes the cost of communication explicit. The programmer has to worry about resource management, sending and receiving messages, and overall orchestration of the collection of processes spread across several workstations.

4.2 BDD Algorithms

To implement BDD algorithms using the message passing model over a NOW, we need to design distributed BDD data structures. However, it is important to understand the requirements of BDD algorithms on a uniprocessor to help guide our design decision about distributing the data and scheduling the interprocessor communication.

The conventional depth-first recursive BDD manipulation algorithm performs a Boolean operation by traversing the operand BDDs on a path-by-path basis (see Figure 3.4), which results in extremely disorderly memory access pattern. The random memory access pattern with no spatial locality of reference translates into severe page faulting behavior when the BDD does not fit the available main memory.

Since the access to the main memory of an another workstation involves a network transaction, the aforementioned disk access behavior of the depth-first algorithm translates to a large number of network transactions for any distribution of the BDD nodes among main memories of a NOW. Since it is very expensive to access the data across the network compared to the workstation main memory, any attempt to use depth-first manipulation algorithm on a NOW will meet limited success.

The breadth-first iterative algorithm [OYY93, AC94, SRBS96] (see Figures 3.10, 3.11, and 3.12) attempts to regularize the memory access pattern by traversing the operand BDDs on a level-by-level basis and by using a customized memory allocator that allocates the BDD nodes for a specific variable id from the same page. We make the following observations to guide the implementation of breadth-first search algorithm for a NOW.

1. We need a mechanism to determine the variable id from the BDD node pointer without accessing the BDD node.

2. While processing the REQUEST for a specific variable id during the APPLY phase, we need to access only those BDD nodes that have the same variable id.

3. The forwarding mechanism, which allows temporary creation of redundant nodes can facilitate the creation of a REQUEST on one workstation and servicing of that REQUEST on an another workstation.

4.3 BDDs on Network of Workstations

4.3.1 Issues:

The following issues need to be resolved before we can implement the breadth-first BDD manipulation algorithm on a NOW.

Node Distribution How to distribute the BDD nodes among the workstations on a network? The number of nodes assigned per workstation should be proportional to the memory resources available on the workstation. The high overhead and latency of accessing a remote memory by performing network transaction implies that performing a large number of communications which involve small messages would result in unacceptably high performance penalty. Therefore, a distribution that results in exchanging information at the level of a BDD node would not be satisfactory.

Naming BDD Nodes How to uniquely identify each BDD node regardless of where it resides on the network, i.e., regardless of workstation address space it belongs to? For a single address space, each BDD node is uniquely identified by its pointer; we need to extend the pointer mechanism to have a generalized address for a BDD node.

Variable Id Determination How to determine the variable id of a BDD node given its generalized address? In the breadth-first algorithm, we need to determine the variable id from the BDD "pointer" to avoid random access to the BDD node. However, the BDD node to index lookup table solution proposed by Ashar *et al.* is unattractive for NOW case for three reasons: 1) each workstation will need to maintain a private copy of the lookup table to determine the variable id from a generalized address for *all* the nodes in the BDD, 2) this private copy will have to be updated every time any workstation allocates a page of memory, and 3) since generalized address would augment 32-bit address space, it may be necessary to implement the node to index lookup table as hash table instead of an array.

We have designed a generalized addressing scheme that works in conjunction with a partitioning scheme to solve these problems, while resulting in a very compact representation for the BDD nodes.

4.3.2 Solutions:

Node Distribution The breadth-first algorithm constructs the result BDD one level at a time by accessing the operand BDD nodes on a level-by-level basis, the natural choice for the decomposition of the BDD is to partition it by levels. To

make number of nodes in a partition (BDD section) proportional to the amount of memory resources per workstation, we can use the flexibility of determining the location of and the number of levels in the partition. For example, a BDD section closer to the root nodes can have more levels than a BDD section at the halfway between root and leaf nodes.

Naming BDD Nodes By assigning nodes for a set of consecutive variables to the same workstation, it is possible to determine the workstation on which a BDD node resides by knowing its variable id. Hence a (variable id, memory address) tuple can serve as a generalized address that uniquely identifies each node in the BDD.

Variable Id Determination We could have used the pair (workstation number, memory address) to represent a generalized address that uniquely identifies each node in the BDD. However, the reason for choosing (variable id, memory address) tuple to represent the generalized address under the constraint of specific levelized partitioning scheme is to solve the variable id determination problem for free. Further, this choice of generalized address results in very compact representation for a BDD node.

Given the partitioning scheme and the mechanism to determine the variable id, we need to address one more issue before we can perform computations related to the BDD sections assigned to a workstation. Servicing a REQUEST in the APPLY phase may result in creation of an another REQUEST, for which the corresponding variable id belongs to another workstation. The newly created REQUEST with a specific top variable id should now be serviced on the workstation that owns the BDD section containing that variable id. REQUEST can be generated on a source workstation and processed on a destination workstation, as long as the source workstation receives a correct generalized address that should result from processing the REQUEST. It is an easy matter to use forwarding mechanism in the APPLY phase for the source workstation by forwarding the generated REQUEST to the generalized address. Since the REQUEST node that gets generated on the source workstation is a *shadow* of the REQUEST node that gets processed on the destination workstation, we call this as *shadow node forwarding*. By using *shadow* nodes, a node which creates the *shadow* node can now be processed in the APPLY phase without accessing the remote memory. Using the same *shadow* node

```
now_bdd_op(op,F,G)
    if(NOT a terminal case (op, F, G))
        if(processor id = 0)
            min_id = minimum variable id of (F, G)
            create a REQUEST (F, G) and insert in request_queue[min_id];
        for(proc_id = 0; proc_id < processor id; proc_id++)
            bf_apply_recv(proc_id, set of requests);
        bf_apply(op, first_var_id, last_var_id);
```

Figure 4.1 Breadth-first BDD algorithm on NOW.

forwarding concept, a set of REQUEST, which belong to the set of consecutive variables assigned to the processor, can be processed without accessing remote memories. The mechanism of *shadow node forwarding* also helps to separate the computation and the communication for the collection of sequential processes. The separation helps simplify the development of the NOW BDD package. The algorithm for manipulation of BDDs on a NOW is presented in Figure 4.1.

The breadth-first BDD manipulation algorithm on a NOW is obtained by suitable modifications of APPLY and REDUCE phase of the breadth-first algorithm for a single address space. The assignment of BDD sections imposes a total order on the workstations. Each workstation receives a set of REQUEST from all its predecessor workstations before the beginning of the APPLY phase. The APPLY phase is now modified to process only those REQUESTs, for which the variable ids belong to the workstation. The set of generated *shadow* requests are sent to appropriate successor workstations for processing. The workstation then waits to receive from the successor workstation, the generalized address to which each shadow REQUEST gets forwarded to. After the REDUCE phase, the workstation sends a set of generalized addresses to each of its predecessor workstations. The overall procedure can be viewed as top-down APPLY phase followed with bottom-up REDUCE phase for a distributed BDD which is partitioned into set of sections each of which is made up of set of consecutive levels. A graph-

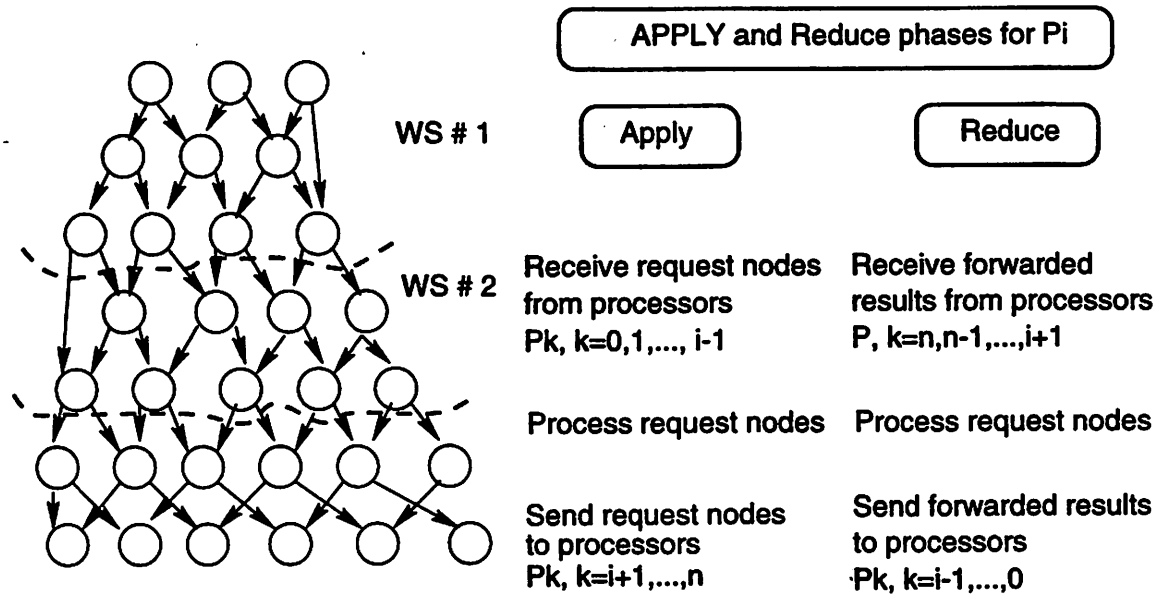


Figure 4.2 Illustration of BDD manipulation algorithm on a NOW.

ical representation of this concept which also illustrates the algorithm in Figure 4.1 has been given in Figure 4.2. The communication serves as a glue to hold together the computations performed in different memories by using *shadow node forwarding* concept.

4.4 Implementation Issues

The following issues are unique to the breadth-first implementation on NOW.

1. **Shadow REQUEST duplication:** Shadow REQUEST may have multiple shadow REQUESTs on different workstations. However, the multiple shadow REQUEST are identified before the REQUEST is processed, hence, only a single REQUEST gets processed and the resulting generalized address is sent to all the workstations with its shadow REQUEST.

2. **Reference count management for nonlocal BDD nodes:**

   (a) Even if a REQUEST can be simplified without accessing the remote memory

(e.g. F AND F), it is important to create a new shadow request and process it on appropriate workstation so that the reference count of the node in the unique table is maintained correctly.

(b) During the REDUCE phase if a redundant node is found for which one of the THEN and or the ELSE generalized addresses point to a node on a successor workstation, we need to adjust reference count of that remote BDD node. This can be achieved by delayed evaluation to avoid communication to all successor workstations after completion of REQUEST phase on a workstation. The delayed evaluation can be performed during the garbage collection step when reference count for the remote nodes can be adjusted appropriately.

3. **Caching shadow REQUEST vs. on-line issue of remote requests:** If the shadow request are not cached, we need a network transaction for every shadow REQUEST created during the APPLY phase. Given the high network overhead and latencies this may not be acceptable. However, this may change if communication can be overlapped with computation and low latency, low overhead networks, which can pipeline several small messages, become available.

4.5 Experimental Results

We have used a heterogeneous network of workstations as the computing environment to perform our experiments. This environment contains approximately 60 workstations with 64MB (about 40MB available) main memory and 256MB (about 200MB available) disk space and MIPS-R4000 processor.

We have used PVM [GBD+94] (Parallel Virtual Machine) software to provide the communication between the workstations in the cluster during a BDD operation. This software permits a network of heterogeneous UNIX computers to be used as a single large parallel computer by providing user level routines to send and receive messages among clusters of workstations.

To evaluate the performance we integrated our BDD package with SIS [SSL+92]. In order to systematically analyze the performance of our algorithms with increase in the BDD size, we have used a series of sub-networks of the ISCAS benchmark C6288. We

Examples	# Nodes	Uniprocessor Scheme		NOW Scheme	
		# Page Faults	Elapsed Time (in secs)	# Maximum Page Faults	Elapsed Time (in secs)
C6288_1M	1×10^6	0	877	0	2589
C6288_2M	2×10^6	0	1918	0	3743
C6288_3M	3×10^6	4392	3587	280	4818
C6288_4M	4×10^6	70184	7234	450	5530
C6288_5M	5×10^6	187843	10676	3060	6454
C6288_6M	6×10^6	780361	15844	8090	10397

Table 4.1 Exploiting collective main memories.

have suitably taken sub-networks of this benchmark such that the shared BDD sizes of the outputs are roughly multiple of one million. For instance, C6288_3M is one of such examples, for which creating the BDD's of all its outputs will involve creating about three million nodes.

In the following subsections we describe the experiments that highlight the salient features of our approach.

4.5.1 Exploiting Collective Main Memories

The main emphasis of our approach is to exploit the collective main memory available across all the workstations. This would lead to less page faults and hence reduced wall clock time to complete the computation. To observe this phenomenon, we have used a virtual machine consisting of 4 workstations. The results have been given in Table 4.1.

From Table 4.1, we observe that for small BDDs, performance on uniprocessor outperforms that on multiple workstations by about a factor of 2-3. The reason being small examples did not result in significant number of page faults for a single processor and network transaction overhead incurred in the network of workstations approach resulted in large elapsed time. However, as the number of BDD nodes increase, causing the uniprocessor implementation to page fault enormously, the multiple workstations scheme outperforms uniprocessor scheme.

C6288 subckts	Elapsed Time		
# Nodes	One WS	Two WS	Four WS
9×10^6	26098	24074	*n.p.*
10×10^6	*s.o.*	24853	21617
11×10^6	*s.o.*	36802	*n.p.*
12×10^6	*s.o.*	49801	35652
13×10^6	*s.o.*	47521	*n.p.*
14×10^6	*s.o.*	58383	*n.p.*
15×10^6	*s.o.*	60139	*n.p.*

Table 4.2 BDDs on multiple workstations.

s.o.: could not complete due to disk space limitation
n.p.: data could not be collected due to time constraint

4.5.2 Exploiting Collective Disk Space

Table 4.5.2 indicates the potential of a NOW in manipulating large BDDs. In this experiment, we increased the number of BDD nodes to be manipulated increased to the extent that it did not fit the disk space of a single workstation. We observe that we are able to manipulate BDDs of much larger size using the collective disks of many workstations.

The breadth-first algorithm implemented using PVM demonstrates the basic advantages of using a NOW. However, remote memory paging using operating system support – network RAM (NRAM) – is a better alternative to managing network memory using user-level message passing. We show results for creating output BDDs on Myrinet connected cluster of four SPARC10 workstations in Table 4.3. All the comparisons are with the fastest breadth-first code running on the MIPS DEC 5000 workstations. The NRAM uses four workstations; the client main memory is 30 MBytes and 3 servers have main memory of 40 MBytes each. The remote memory pager for NRAM uses active-messages [Eic93] as the basic communication primitive.

For C6288 sub-circuits with less than three million nodes, the speedup is due to faster processing speed on SPARC10 used as a building blocks for NOW. For more

Example	Elapsed Time		
	One WS	NRAM	Ratio
C6288_1M	94	39	2.41
C6288_2M	258	119	2.17
C6288_3M	933	372	2.51
C6288_4M	2535	576	4.9
C6288_5M	6156	1003	6.13
C6288_6M	10146	1469	6.19
C6288_7M	13598	2058	6.61

Table 4.3 Exploiting remote memory using network RAM (NRAM).

than 3 million nodes, we start seeing the effect of the disk accesses on the elapsed time for MIPS code and achieve about 6-7x speedup. The CPU time for large problems is a small fraction (< 10% and decreasing) of the elapsed time. Therefore, the speedups on larger problems are mainly due to collective main memories of servers as a secondary store and not due to faster processing speeds of SPRAC10s.

4.5.3 Analysis of Experiments

In previous two subsections we have presented results which demonstrate the two key advantages of manipulating BDDs on a NOW, namely, exploiting collective main memory for improved performance and using collective disk space to build large BDDs. However, we note that the time taken to manipulate BDDs on a NOW is large. We monitored the elapsed time in our algorithm and found that a large part of the elapsed time is due to the network transaction. Hence, the performance of our approach is significantly dominated by the penalty incurred during message transfers. The hope is that with the ongoing research in NOW community [ACP94] which includes using asynchronous transfer mode, parallel file server, and active message passing will result in low network latency and overhead. Our approach will take advantage of performance enhancements achieved by NOW research community.

4.6 Related Work

Arunachalam *et al.* [ACM96] have presented a technique to manipulate BDDs on a network of workstations. They also target alleviating the memory consumption problem by exploiting the memory available in a cluster of workstations. However, in their approach, the distribution of BDD nodes on the workstations is random. Since every dereferencing of node for reading and writing purposes requires message passing, the random nature of BDD node accesses leads to large amount of communication amongst workstations. In order to optimize the number of message exchanges caching, pipelining, and pre-fetching is used.

4.7 Conclusions

We presented an algorithm for manipulation of binary decision diagrams (BDD) on a network of workstations (NOW). A NOW provides a collection of main memories and disks which can be used effectively to create and manipulate very large BDDs. We use a breadth first manipulation technique to exploit the memory resources of a NOW efficiently. The prototype implementation points to the potential impact this approach can have in manipulating very large BDDs.

The effectiveness of our approach was demonstrated with experiments. This chapter serves as a proof of concept for our approach. The current work can be extended with following features:

1. Utilizing the computation power of a NOW: In the current approach, the computations are carried out one processor at a time. Hence, we have only exploited the memory resources of workstations in the network. This approach can be extended to utilize the parallel computation power offered by NOW. This will be achieved by *pipelined* processing of REQUEST's during the APPLY and REDUCE phase. In the pipelined scheme, REQUEST's are processed on more than one processor concurrently. Hence, a processor need not wait to collect REQUEST's from predecessor processors during the APPLY phase and from successor processors during the REDUCE phase. This will result in improved computation time of the processing of the REQUESTs. However this scheme has two drawbacks: i) on-line issue of remote requests will result in significant increase in the network

transaction. We observed in Section 4.5 that the performance of our approach was significantly hampered by the network latency and overhead. ii) it will result in duplication of effort due to inability to recognize a REQUEST after it is processed in the APPLY phase. Consequently, it will also increase the working memory requirements and amount of work during the REDUCE operation.

The benefits of this approach in view of these two drawbacks need to be investigated. A plausible solution could be to adopt a scheme in between the two extremes and issue remote requests in a group only.

2. Dynamic load balancing: In the current scheme the variable indices are statically distributed over several processors. This has the disadvantage that if the number of nodes in certain levels grow very large then it leads to uneven distribution of BDD nodes. A better approach would be to dynamically change the distribution of a set of variables among the processors to balance the number of nodes on each processor.

Chapter 5

Parallel BDD Manipulation

In Chapters 3 and 4, we discussed how memory hierarchy in a computer system can be exploited to expedite BDD manipulation when the data size exceeds the main memory capacity. In particular Chapter 3 addressed the efficient use of the secondary memory of a workstation to increase the available data memory without incurring a significant memory access penalty. In Chapter 4, we extended this notion to a network of workstations where the collective main memory of multiple workstations was used for efficient BDD manipulation.

In this chapter, we consider an orthogonal technique to expedite BDD manipulations – the use of concurrent computation. Parallelization offers a way to complement graph reduction research efforts for enabling verification of larger problem sizes. In particular we consider the use of shared-memory multiprocessors for efficient BDD manipulation. We identify the key elements needed for a successful parallel implementation of a BDD package. We argue that by combining the locality of access of a breadth-first manipulation approach with the parallel computing power of a shared-memory multiprocessor, one can achieve a high degree of performance improvement over a conventional BDD package.

The organization of the chapter is as follows. Section 5.1 provides brief overview of the popular parallel architectures and highlights their features.* In Section 5.4, we give a brief introduction on multi-threaded programming and their advantages and disadvantages.† In Section 5.2, we discuss the basic idea behind parallel computer application in BDD manipulation and outline the requirements for efficient manipulation. In Section 5.3, we discuss previous works on parallel manipulation of BDDs and analyze their trade-offs in a common framework. We present our technique in Section 5.5. Due to

*Introductory material on parallel architectures and their performance has been obtained from [HP90, San96, CSG97].

†Most of the material in this section has been taken from [Eng95, SS94].

lack of time we did not implement our technique. However, in Section 5.6 we present a detailed analysis of closely related work by Yang *et al.* [YO97] that has been fully implemented. Finally, we conclude by indicating the potential for further improvements.

5.1 Parallel Computer Architectures

The classic definition of a parallel computer is captured in following quote from [AG89]: A *parallel computer* is a collection of processing elements that cooperate and communicate to solve large problems fast.

These days parallel architectures have become the mainstay of scientific computing, including physics, chemistry, material science, biology, astronomy, earth sciences, and others. In computer-aided design applications, parallel architectures are extensively used for simulations at various levels (device, transistor, logic, etc) in the design flow.

The general taxonomy of parallel architecture is presented in Figure 5.1. In the next few sections we briefly describe various categories of parallel architectures.

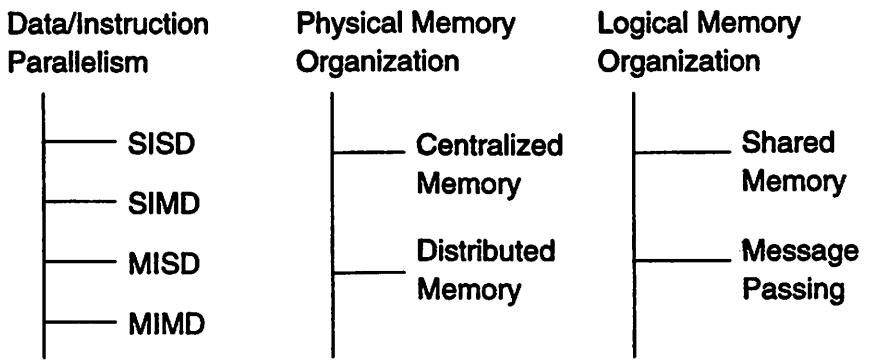


Figure 5.1 Taxonomy of parallel architectures along various dimensions.

5.1.1 Parallelism in Instruction and Data Streams

Based on the parallelism in the instruction and data streams operated on, computers can be classified into four categories: i) Single instruction stream / single data stream (SISD) – single processor computers; ii) single instruction stream / multiple

data stream (SIMD) – multiple processors executing the same instruction stream on different data streams; iii) Multiple instruction stream / single data stream (MISD) – no present day computer falls in this class; iv) Multiple instruction stream / multiple data stream (MIMD) – each processor has its own instruction stream and operates on its own data.

SIMD machines are also known as processor arrays or data parallel architectures. The key characteristics of the programming model is that operations can be performed in parallel on each element of a large regular data structure, such as an array or matrix. Some *vector processors* which operate on vectors of data out of a common memory, also fall in this category.

MIMD is the most general parallel computer. In recent years it has emerged as the architecture of choice for general-purpose multiprocessors. We discuss the various manifestations of the MIMD architectures in the next section.

5.1.2 Memory Organization in Parallel Architecture

Based on the physical memory organization, MIMD architectures can be broadly divided into following categories:

Centralized shared-memory architecture

In this architecture, processors share a single centralized memory and the processors and memory are interconnected by a bus. Since there is a single main memory that has a uniform access time from each processor, these machines are sometimes called *uniform memory access* (UMA) machines. Currently this type of centralized shared-memory architecture is by far the most popular MIMD architecture. An illustration of such architecture is shown in Figure 5.2. Parallel machines in this category are popularly known as *shared-memory multiprocessors*. Examples of machines in this category are Intel quad-processor Pentium-Pro multiprocessor and SGI Challenge multiprocessor.

Physically distributed memory architecture

In distributed memory architecture multiple memories are physically distributed with the processors. A generic distributed memory architecture is shown in Figure 5.3. Distributing the memory among the nodes has two major benefits. First, if most of the

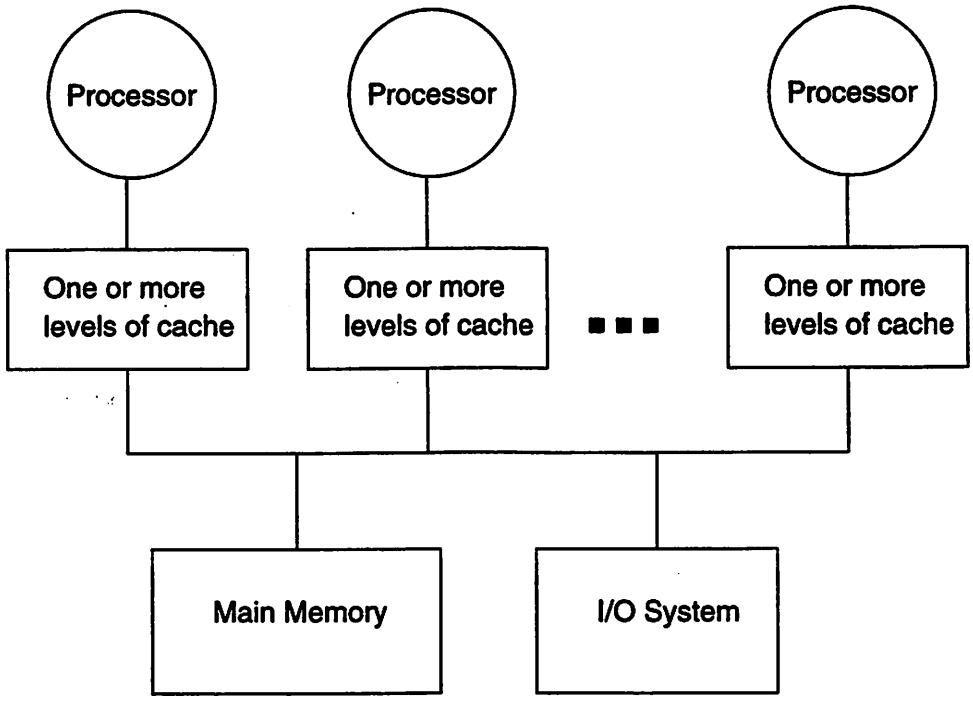


Figure 5.2 Basic structure of a centralized shared-memory multiprocessor.

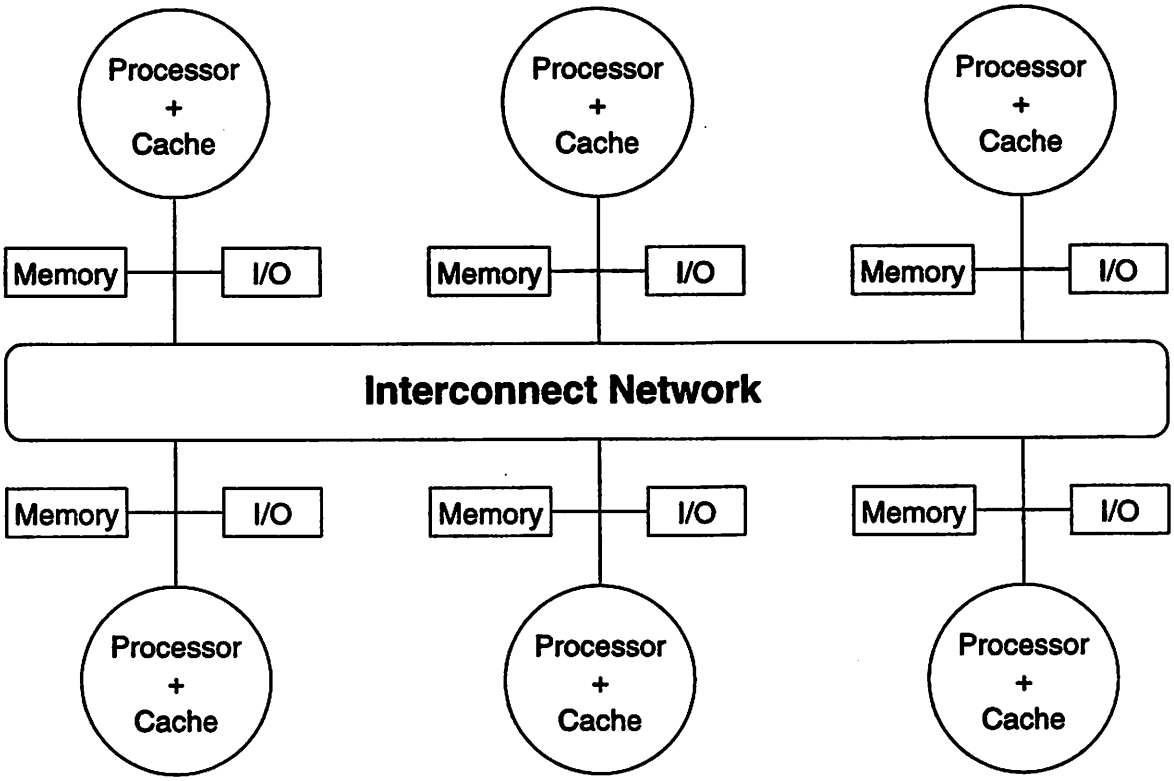


Figure 5.3 Basic architecture of a distributed-memory machine consists of individual nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes. Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology.

accesses are to the local memory in the node, this provides a cost-effective way to scale the memory bandwidth. Second, it reduces the latency for accesses to the local memory. *Massively parallel processors* are the most popular machines in this category. Examples include Cray T3E and IBM SP-2. The major architectural differences between distributed-memory machines come from the logical organization of the memory and the communication paradigm.

The physically separate memories can be addressed as one logically shared address space, meaning that a memory reference can be made by any processor to any memory location, assuming it has the correct access rights. These machines are called *distributed shared-memory (DSM)* or *scalable shared-memory* architectures (as opposed to shared-memory multiprocessors). The term *shared memory* refers to the fact that the *address space* is shared; that is, the same physical address on two processors refers to the same location in memory. Shared memory does not mean that there is a single, centralized memory. In contrast to the centralized memory machines, also known as UMAs (uniform memory access), DSM machines are also called NUMAs, *non-uniform memory access*, since the access time depends on the location of a data word in memory.

Alternatively, the address space can consist of multiple private address spaces that are logically disjoint and cannot be addressed by remote processors. In such machines, the same physical address on two different processors refers to two different locations in two different memories. These employ complete computers as building blocks – microprocessor, memory, and I/O system.

5.1.3 Communication Paradigms in Distributed Memory Machines

Distributed memory machines call for a specific communication mechanism based on the address space organizations. For a machine with a shared address space, that address space can be used to communicate data implicitly via load and store operations. In machines with multiple address spaces, communication of data is done by explicitly passing messages among the processors.

In distributed shared memory machines the multiple processes can be configured so that portions of their address space are shared. Threads within a process cooperate and coordinate by reading and writing shared variables and pointers referring to shared

addresses. Writes to a logically shared address by one thread are visible to reads of other threads. The advantages of shared-memory communication include:

1. Compatibility with well-understood mechanisms in use in centralized shared-memory multiprocessors.

2. Ease of programming when the communication patterns among processors are complex or vary dynamically during execution.

3. Lower overhead for communication and better use of bandwidth when communicating small items.

4. The ability to use hardware-controlled caching to reduce the frequency of remote communication by supporting automatic caching of all data, both shared and private.

In message passing machines, communication occurs by sending messages that request an action or deliver data similar to simple network protocols. Compared to shared memory, this architecture has a greater distance between the programming model and the communication operations at the physical hardware level. Typically, user communication is performed through operating system or library calls. The major advantages for message-passing communication include:

1. The hardware can be simpler, especially by comparison with a scalable shared-memory implementation that supports coherent caching of remote data.

2. Communication is explicit, forcing programmers and compilers to pay attention to communication.

5.1.4 Performance Issues in Parallel Computing

Performance of a parallel program for a specific data set on a particular parallel computer is defined as the time required to complete the program on that machine. The performance is usually quoted in terms of relative speedup, which is defined as the ratio of time required to complete the program for a specific data set on a single processor identical to one compute node of the parallel machine to the time required to complete the program on a parallel machine with N compute nodes.

Usually, the speedup is less than N. However, for some programs it is even possible to obtain speedup greater than N. The reason for this super-linear speedup is availability of multiple memory systems. If the program has very high degree of parallelism and very low or no communication overhead, there is performance gain due to availability of multiple caches and main memories when the total data exceeds size of a particular memory level (cache or main memory) but fits inside the aggregate size for N processors. A program running on a single processor requires much larger effective cache and main memory to offset the advantage of multiple caches and main memories.

Every program has some serial part and some parallel parts. Ignoring the effect of multiple caches and main memories, the total available speedup S is limited by Amdahl's law [Amd67] which roughly states that the speedup cannot exceed the reciprocal of fraction of serial computation

$$S = \frac{(T_s + T_p)}{(T_s + T_p/N)}$$

where T_s and T_p represent time for serial and parallel computations, respectively. Therefore,

$$S < \frac{1}{T_s/(T_s + T_p)}$$

The amount of parallelism in a given application, therefore, is the most obvious and the most important issue that limits the performance of a parallel program. It should be noted that the amount of parallelism exhibited by a parallel program is also a function of the data size. The amount of serial work may be constant or increasing very slowly with the problem size, hence increasing the problem size decreases the fraction of serial computation, thereby improving the chances for obtaining high speedups [Bai91].

The amount of parallelism in the application depends on the level of abstraction that determines the granularity. Usually, the finer the granularity the higher the parallelism. However, finer granularity, usually results in higher volume of communication. Effective parallelization requires a good balance between amount of things that can be done in parallel and associated overhead cost.

For a given abstraction level, it is important to make sure that each processor does roughly the same amount of work. Each processor may do an equal amount of total

work, but may spend a lot of time waiting for results from other processors. It is important for processors to do an equal amount of work at about the same time. Duplicating computation may also help reduce dependencies and communication.

The next important issue is communication, which is an artifact of parallel computing and therefore, is an additional overhead over and above the work specified by the underlying sequential algorithm. The startup cost of communication (latency), cost per unit communication (bandwidth), and the frequency of communication determines the overall communication cost. It is important to minimize the communication cost as much as possible. Overlapping computation is one means of hiding the communication cost.

Another issue that is important from the practical stand-point is related to management of parallelism. Parallel programs incur overheads in issuing computations and migrating data so as to efficiently utilize the parallel computing resources. The management of parallelism needs to be effective – overheads should be far outweighed by the efficiency achieved in utilizing parallel computing resources.

5.2 Using Parallel Architecture in BDD Manipulation

The basic idea behind parallel manipulation of BDD is the processing of multiple nodes in parallel, i.e., the accessing memory locations in parallel. However, it is nontrivial to parallelize BDD manipulation efficiently, because the manipulation process involves numerous memory references to small data structures with little computational work to amortize the cost of each reference. In addition, a conventional BDD manipulation scheme has irregular control flows and memory access patterns. The control flow is irregular because the recursive expansion can terminate at any time when a terminal case is detected or when the operation is cached by the computed table. The memory access pattern is irregular because a BDD node can be accessed due to expansion on any of its many parents and, since the BDD is traversed in the depth-first manner, expansions on the parents are scattered in time citebwolen97.

Keeping these facts in mind and the performance issues discussed in Section 5.1.4, we present various aspects of parallel BDD manipulation that can potentially affect the gains from a parallel architecture.

Selection of the hardware: What is the underlying hardware – a vector processor, a massively parallel processor, a distributed shared-memory multiprocessor, a centralized shared-memory multiprocessor etc. The selection of hardware restricts the communication mechanism and also determines the cost in dollars.

Distribution of data: How are BDD nodes representing various functions distributed among the processing nodes and how uniform is the distribution?

Generation of computation loads: How is the work generated for different processors?

Distribution of computational load: How the computational load is distributed amongst the different processors? How balanced is the load distribution?

BDD traversal scheme: How the operand BDDs are traversed during computation? Conventional BDD construction algorithms based on depth-first traversal of the BDDs has poor memory behavior because of irregular control flows and memory access pattern. Breadth-first traversal alleviates some of these problems.

Sharing of global data structures: How is the sharing of various global data structures, e.g., unique table, computed table, etc. done amongst processors? What data structures are duplicated for each processor?

Communication paradigm: How is the validity of the shared data structures maintained? How do processors communicate with each other – message passing vs. synchronizing primitives (lock, barrier, mutex, etc.)?

With this perspective, we analyze previous research efforts to parallelize BDD manipulation in the following section.

5.3 Previous work

[KC90] used a shared-memory multiprocessor (Encore multimax). They used a two-phase (Apply and Reduce) BDD traversal technique to generate the result. Parallel jobs were created through either multiple Boolean operations or through unrolling of recursive calls for few steps. Interlocks were used for process synchronization. Due

to the nature of their job distribution amongst processors, they achieved very coarse grain load balancing (a simple BDD operation on one processor could finish much earlier than some complex operation on another processor), and even though their results indicated up to 10x speedup with 15 processors on multiplier examples, it would be unlikely to observe the similar performance on general circuits.

[OIY91] used a vector processor (HITAC S-820/80). They chose a breadth-first traversal scheme for BDDs (without any notion of locality of access). This allowed them to vectorize the processing of temporary nodes at each index. They reported a vector acceleration ratio of up to a factor of 14 in creating output BDDs.

[PSC94] used a distributed shared-memory multiprocessor (CM-5). They chose a two-phase BDD traversal scheme. Unlike previous approaches which used a shared-memory architecture, they used a distributed stack to achieve fine-grain balancing. Task synchronization was done using global broadcast and the communication was implemented using the active messages library. They presented results on small ISCAS examples with speed-ups of 20 to 32 on a 32-node CM-5.

[CGRR94] used a massively-parallel computer (CM-200) in a data parallel approach where BDD nodes were distributed to the processing elements. Parallelism was achieved by allocating one processing element for each BDD node. They did not give any results comparing the performance with a uniprocessor and indicated up to 5x speedup in going from 4K processes to 32K processes configuration.

[SB96] used a distributed memory approach on Meiko CS-2 with 64 scalar nodes (distributed shared-memory multiprocessor). The unique table was distributed across all processors. Multiple threads of computation were used on a distributed BDD. Up to 7x speed-up was obtained for a 32-node configuration.

Reference	Hardware Selection	Traversal Scheme	Load Generation	Load Distribution	Communication Paradigm	Communication Volume
[KC90]	Shared-memory multiprocessor ($)	Apply–Reduce	High level BDD operations	Highly coarse	Locks	Light
[OIY91]	Vector processor ($$$$$$$$$)	Breadth-first	Temporary nodes processing	Fine grain	Locks	Medium
[PSC94]	Distributed Shared Memory ($$$$)	Apply–Reduce	BDD nodes	Fine grain	Message passing	Heavy
[CGRR94]	Massively parallel processor ($$$$$$)	Apply–Reduce	BDD nodes	Fine grain	Message Passing	Heavy
[SB96]	Distributed shared-memory multiprocessor ($$$$)	Apply–Reduce	BDD nodes	Fine grain	Message passing	Heavy

Table 5.1 Previous work in parallel BDD manipulation: a summary.

$: Indicates the relative cost of the hardware.

A summary of the discussion on previous work has been provided in Table 5.1. We analyze this summary below:

Load generation: Concurrency in BDD manipulation is achieved in three ways:

1. High level operations: If there are multiple Boolean operations to be performed, they are started on different processors [KC90].
2. Concurrent processing of the cofactors: In this case a place holder for the results of the cofactors is created and processing of the cofactors continues in parallel on different processors [PSC94, CGRR94, SB96].
3. Breadth-first processing: In this case, breadth-first manipulation scheme is used. The set of temporary nodes (similar to REQUESTs) are maintained at each index, processors divide the set of temporary nodes equally.

Load distribution: The first approach of load generation leads to very coarse grain parallelism which could be highly unbalanced. All other approaches apply parallelism at the level of processing of a node. This leads to fine grain distribution.

Amount of communication: This is related to the load distribution. Coarser distribution leads to less communication and vice-versa.

Sharing of data structures: All of the approaches share the UNIQUE TABLE as the global data structure. Except [OIY91], others also share their work queues.

Distribution of BDD nodes: In both the distributed memory based approaches, the BDD nodes are divided randomly amongst the processors' memories.

Cost of the hardware: The specialized hardwares (MPP, vector processors) cost the most. Small- to medium-scale distributed memory multiprocessors cost lot less. Small scale shared-memory multiprocessors have the best cost advantage.

5.4 Using Multi-threading on a Shared-Memory Multiprocessor

In this section, we provide a brief introduction on threads and multi-threaded programming.

A *process* is a program whose execution has started but is not yet complete (i.e., a program in execution). A process has a single address space and a single thread of control to execute a program within that address space. To execute a program, a process has to initialize and maintain state information. The state information typically is comprised of page tables, swap images, file descriptors, outstanding I/O requests, saved register values, etc. This information is maintained on a per program basis, and thus, a per process basis. The volume of this state information makes it expensive to create and maintain processes as well as to switch between them. To handle situations where creating, maintaining, and switching between processes occur frequently (e.g., parallel applications), threads or lightweight processes have been proposed.

A *thread* (also called thread of control) is a sequence of instructions executed by a program. In the traditional UNIX model, a process contains a single thread. Threads execute independently of each other and share a common address space. In a multi-threaded system, two or more threads share a common UNIX process. Threads share process instructions, process data, and process resources: open files, signals, user data. These threads are managed by the threads library routines in the user space.

Computers with more than one processor provide multiple simultaneous paths of execution. Multiple threads are an efficient way for application developers to utilize the parallelism of the hardware. Multi-threading (MT) is the set of programming paradigms, tools, and techniques that enable applications to take advantage of multi-processing. It provides a powerful way for software developers to speed up applications on uniprocessor or multiprocessor systems, transparently leveraging parallel hardware.

Threads share process instructions and most of the process data. A change in data by one thread can be seen by the other threads in the process. Threads also share most of the operating system state of a process. Unique to every thread are: thread ID, register state (including PC and stack pointer), stack, signal mask, scheduling priority, and thread specific data.

Creating multiple threads within a process is inexpensive compared to forking new processes. The reason is that creating a process requires creating a new address space. The time needed for creating a new thread is typically 30 times less than for creating a new process.

Synchronization primitives, also known as locking mechanisms, are necessary for

multi-threaded programming. They are variables in memory which are used to coordinate threads and control the access to the memory shared by threads.

5.4.1 Bottlenecks in Multi-threading

Performance hit when not enough done in a thread: Compute bound programs with coarse-grained parallelism can benefit from multi-threading when run on multiprocessor hardware. While the overhead involved in synchronizing and context switching threads is significantly smaller than using processes, it is not zero. Programs in which each thread does not execute enough code between synchronizations or context switches will have performance problems.

Difficulty in programming multi-threaded applications: Multi-threaded applications can be difficult to design and to debug. The memory, process state, and address space are shared between threads. Since these shared resources are easy to access or corrupt by any thread in the process, programming with multiple threads requires more care and discipline than does single-threaded programming. The errors in multi-threaded programming are caused by i) accessing global memory without the protection of a synchronization mechanism and ii) using a local or global variable for assigning an argument to a new thread. However, the improved performance and scalability are worth the effort.

5.5 Our Approach: Combining Locality of Access with Parallel Computing

We propose a parallel BDD manipulation technique based on breadth-first travesal and the use of multi-threading on a shared-memory multiprocessor.

Before we proceed to explain our strategy for parallel breadth-first BDD manipulation, for ease in understanding, some important aspects of uniprocessor version of breadth-first technique are briefly reviewed below.

- The breadth-first algorithm proceeds in two phases – top-down APPLY phase followed by bottom-up REDUCE phase.

- During APPLY phase, place holders are created to represent the results of co-factors. These place holders, termed REQUESTs, are processed on an index-by-index basis.
- At each index, all the REQUESTs belonging to that index are stored in a hash table. This is to allow fast check for the REQUESTs to avoid duplication.
- During the REDUCE phase, the REQUESTs are processed on an index-by-index basis, from bottom to top. If the REQUEST is not redundant, it is inserted in the UNIQUE TABLE of the corresponding index.

Based on this basic technique, the parallel implementation is described below.

After the BDD manager is initialized by the main thread, several auxiliary threads are created and bound to different CPUs for concurrent processing. The main thread (the process itself), takes care of synchronizations amongst threads. The auxiliary threads wait for a signal from the main thread to start processing. The amount of private data associated with each thread is very small .

When a BDD operation needs to be performed, the main thread performs some initialization which sets up the type of the operation, the minimum index of the operand BDDs, etc. This initialization is visible to all the auxiliary threads. Next the main thread wakes up the auxiliary threads and the APPLY and REDUCE phases are performed by all of them.

During the APPLY step, the work load is divided equally amongst processors. This is achieved by dividing the set of bins of each REQUEST QUEUE amongst processors. Empirically, we have observed a fairly equal distribution of REQUESTs amongst bins of the REQUEST QUEUE.

Since the REQUEST QUEUE for a given index is shared amongst various threads, we need to create a lock on the appropriate REQUEST QUEUE to avoid concurrent modification of a REQUEST QUEUE by two different threads. Using a lock at the level of each BDD node or REQUEST is not a feasible option. In our technique we prevent the concurrent modification of the same data item by using a lock for each "id".

During the REDUCE phase, the REQUEST QUEUE bins are again divided equally amongst the different threads and the REQUESTs are processed accordingly. If the REQUEST is not redundant, we need to update the UNIQUE TABLE of the corresponding

index. Since the REQUESTs are processed concurrently by various threads, to avoid concurrent updating of the UNIQUE TABLE, each thread needs to lock it before making any modification. This would result in significant locking overhead. To avoid this, we adopt a strategy where threads do not update the UNIQUE TABLE on an incremental basis. Instead, each thread collects all the REQUESTs which are not redundant in a list, which is later processed by the main thread.

Our algorithm is outlined in Figures 5.4, 5.5, 5.6, 5.7, and 5.8.

```
pre_process{
    k = number of processors in the system;
    Create and initialize the BDD manager;
    Initialize k − 1 threads and attach them to different CPUs;
    Initialize thread related data and store in the BDD manager;
    Threads wait for the signal from the main thread;
}
```

Figure 5.4 Pre-processing step in multi-threaded BDD manipulation on a shread-memory multi-processor.

5.5.1 Analysis

In our algorithm, we have made specific decisions with regard to the various issues presented in Section 5.2. Below we analyze the impact of these choices.

1. Locality of access: Since the underlying manipulation paradigm is based on breadth-first traversal of BDDs, we can benefit from the same locality of access as in the case of uniprocessors (described in Chapter 3). In addition, the local memory accesses leads to better cache locality and hence less bus contention.

2. Distribution of load: During APPLY phase, our load distribution strategy leads to roughly equal amount of work for each thread. Hence the proposed technique leads to fine grain load balancing.

```
multi_thread_apply(op){
    thread_num = unique self thread id;
    minIndex = minimum index of operands supplied by main thread
    for (index = minIndex; index ≤ numVars; index++){
        req_que = REQUEST QUEUE[index];
        req_que_array = REQUEST QUEUE;
        multi_thread_apply_aux(thread_num, req_que, req_que_array, op);
        Synchronize;
    }
}
```

Figure 5.5 APPLY step in multi-threaded BDD manipulation.

```
multi_thread_apply_aux(thread_num, req_que, req_que_array, op){
    Calculate the set of bins to be processed by the thread based on thread_num.
    Process REQUESTS belonging to these bins in req_que .
    Create lock on appropriate req_que before creating a new REQUEST.
}
```

Figure 5.6 Request processing during APPLY step in multi-threaded BDD manipulation.

```
multi_thread_reduce{
    thread_num = unique self thread id;
    for (index = numVars; index ≥ 0; index--){
        req_que = REQUEST QUEUE[index];
        unique_table = UNIQUE TABLE[index];
        multi_thread_reduce_aux(thread_num, req_que, unique_table);
        Synchronize;
        Processing of the REQUESTS in the auxiliary lists by the main thread;
    }
}
```

Figure 5.7 REDUCE step in multi-threaded BDD manipulation.

```
multi_thread_reduce_aux(thread_num, req_que,unique_table){
    Calculate the set of bins to be processed by the thread based on thread_num.
    for each REQUESTS belonging to these bins in req_que {
        Update THEN and ELSE cofactors of the REQUEST;
        If current REQUEST is redundant, put appropriate forwarding information;
        Otherwise, put REQUEST in auxiliary list (to be processed by main thread);
    }
}
```

Figure 5.8 Request processing during REDUCE step in multi-threaded BDD manipulation on a shread-memory multiprocessor.

3. All the data structures are shared. Hence there is no replication of data or computation.

4. Multi-threaded programming: Instead of using processes to perform operations concurrently on different processors, we make use of threads. This allows us to share the global data structures without the overhead of remote memory access or message passing.

5. The proposed strategy is best suited for a shared-memory multiprocessor where global data structures can be accessed without any communication overhead.

6. During REDUCE phase, the updating of UNIQUE TABLE at each index is done by the main thread which keeps all other threads idle. However, in all the previous works proposed so far for parallel BDD manipulation, the same limitation arises in one form or the other.

An important point to note is that unlike its counterpart in the uniprocessor domain, the proposed breadth-first algorithm for shared-memory multiprocessor does not have any memory overhead over conventional depth-first algorithm. This is because in the parallelization of depth-first algorithm, the cofactors need to be processed in parallel (otherwise there would not be any concurrency). This is done by following some variant of two phase scheme – APPLY followed by REDUCE – as given in [Bry86]. This implies that some place holder is used to store the information about the child nodes (identical to the approach in breadth-first technique).

5.6 Related Work

In this section, we describe a related work presented in [YO97].‡ In that work the authors propose a parallel algorithm for BDD construction targeted at shared-memory multiprocessor. The algorithm focuses on improving memory access locality through specialized memory managers and partial breadth-first expansion, and on improving processor utilization through dynamic load balancing.

The algorithm uses partial breadth-first expansion (equivalent to APPLY phase in our terminology) that improves locality of reference by controlling the working set size

‡All the material in this section has been obtained from [YO97].

and thus reducing the overhead due to page faults, communication, and synchronization. The algorithm also incorporates dynamic load balancing to deal with the fact that the processing required for a BDD operation can range from constant to quadratic in the size of the BDD operands, and is impossible to predict before runtime. BDD construction is parallelized by distributing operations among processors during the APPLY phase. Once the operations are assigned, each processor independently constructs corresponding BDDs using partial breadth-first expansion, which carefully controls the working set size in order to minimize accesses beyond a processor's own local memory. When a processor becomes idle, work loads are redistributed to keep the load balanced. In the sequential world, good memory access locality results in good hardware cache locality and reduced page faults. In the parallel world, memory access locality has the additional advantage of minimizing communication and synchronization overhead.

5.6.1 Parallel Apply and Reduce

For BDD construction, both the APPLY and the REDUCE phases have a large degree of parallelism. However, to efficiently utilize this parallelism, careful memory layout is necessary to reduce synchronization cost. In this algorithm, each process independently maintains its own copy of the BDD node managers, operator node managers, and the compute cache. This data layout allows each process to proceed independently of each other during the APPLY phase. During the REDUCE phase, synchronization is necessary to prevent concurrent modification to the BDD unique tables.

In the APPLY phase, only operator nodes and their corresponding compute cache entries are created. By requiring each process to maintain its own operator nodes and compute cache, a process can expand its assigned share of operations without synchronizing or communicating with other processors.

In the REDUCE phase, new BDD nodes are created and inserted into the unique tables. To avoid concurrent modification of the unique tables, one semaphore lock is associated with each variable's unique table. Before creating a new BDD node and inserting in into its unique table, a process must acquire the corresponding lock.

The main drawback of this data layout is that since the compute cache is not shared, a process will not be able to take advantage of another's compute cache. Thus, the same work might be duplicated in different processes. Another drawback of the per-process

data structures is that memory is used less efficiently as the free space in the blocks allocated by one process is not available to other processes.

5.6.2 Work Distribution

As processing required for a BDD operation can range from constant to quadratic in the size of the BDD operands, it is impossible to distribute the load evenly through static allocation. In this algorithm, the load is dynamically balanced based on stealing unexpanded operations from processes' context stacks. The processes' context stacks serve as distributed work queues. When a process is idle, it tries to steal unexpanded operations from busy processes' context stacks. If an idle process fails to find any work, it notifies busy processes to create more sharable work by context switching. Upon successfully stealing work, the thief process produces the results for the stolen operations and return these results to the original owner. During the reduction phase, a process stalls if the results it needed for the reduction have not yet been returned by the thief processes. This stalled process then becomes a thief and tries to steal work from other processes' context stacks.

5.6.3 Results

	Elapsed Time (seconds)			
# Procs	C2670	C3540	mult-13	mult-14
Seq	208	215	256	935
1	204	220	293	1092
2	120	132	173	633
4	76	81	114	383
8	52	58	96	301

Table 5.2 Elapsed time for building BDD for each circuit with different number of processors. "Seq" represents the sequential case.

Performance results were obtained on a twelve processor SGI Power Challenge with 1GB of shared memory. Each processor is a 195MHz MIPS R10000.

Table 5.2 shows the elapsed time for building BDDs with different number of proces-

	Total # of Operations (in millions)			
# Procs	C2670	C3540	mult-13	mult-14
Seq	92.5	68.1	72.8	245
1	92.5	68.1	83.5	296
2	98.4	68.8	84.2	294
4	110.1	71.6	86.7	297
8	125.1	76.2	87.8	305

Table 5.3 Total number of operations in millions. "Seq" represents the sequential case.

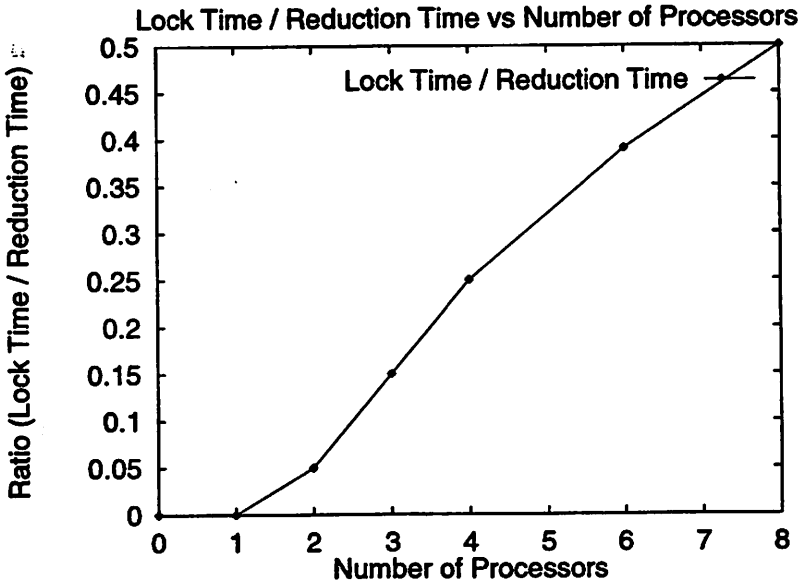


Figure 5.9 Ratio of lock acquiring time to the total time of the reduction phase for the mult-14 circuit.

sors. The algorithm is able to achieve speedups of over a factor of 2 on four processors and up to a factor of 4 on eight processors.

Table 5.3 shows the total number operations (i.e., number of the Shannon expansion steps) for different circuits. Results show that despite the compute cache not being shared, the total number of operations does not increase in the same proportion as number of processors.

To understand the effect of lock contention during the reduce phase, consider the Figure 5.9. This figure plots the lock acquiring overhead as a fraction of the total cost of the reduction phase. For the 8 processors case, the lock acquiring time is 50% of the total reduction phase; i.e., over 20% of the total running time. This represents a significant bottleneck in the performance improvement due to parallel manipulation.

5.7 Conclusion

We draw following conclusions from this study.

1. A trade-off can be made between avoiding duplicate computation and minimizing communication while sharing global data structures. However, duplicating the computation without reducing the communication is a lose-lose proposition.

2. The REDUCE step is the main bottleneck in all implementations of parallel BDD manipulation.

3. A better distributed-hashing algorithm is necessary to reduce the unavoidable synchronization cost incurred during REDUCE step.

4. In some BDD-based applications the non-canonicity of BDD nodes can be allowed leading to better parallelization.

Chapter 6

Dynamic Ordering

An important aspect of BDDs is that for a given variable ordering, they represent Boolean functions canonically. The ordering plays a significant role in determining the size of BDDs, which is crucial for their efficient manipulation. The problem of finding an optimal ordering is intractable and several heuristics have been proposed which statically determine a good ordering of variables based on the circuit information [FFK88, MWBS88]. Recently, dynamic ordering [Rud93] has become a popular alternative to static ordering. The salient features of this technique are the transparent nature of the algorithm (user need not be aware of it) and the in-place swapping of variables. In fact, dynamic ordering is established as a critical component for any BDD package to be used for practical-sized problems.

In the last three chapters (Chapters 3, 4, and 5), we saw how the breadth-first manipulation [OYY93, AC94, SRBS96] techniques exploit memory hierarchy in a computer system to efficiently manipulate very large BDDs which do not fit in the main memory. Although breadth-first schemes outperform a conventional depth-first based scheme when reordering is not performed, lack of reordering leads to its inability to finish the application when the initial variable ordering results in very large BDDs [Sen96]. At first glance, it might seem that the reordering algorithm in a breadth-first scheme would be identical to the one in a depth-first scheme. However, this would destroy the locality of reference of BDD nodes which is critical to the performance of breadth-first based packages. In this chapter we provide efficient techniques to address this issue. We propose techniques to preserve the locality of reference during reordering. After identifying the problems with implementing variable swapping (the core operation in dynamic reordering) in breadth-first based packages, we propose techniques to handle the computational and memory overheads. We believe that combining dynamic reordering with the powerful manipulation algorithms of a breadth-first based scheme

can significantly enhance the performance of BDD-based algorithms.

The organization of this chapter is as follows. In Section 6.1, we briefly touch upon the background material on dynamic reordering. In particular, we present the basic variable swapping technique. In Section 6.2, we discuss the problems and issues in implementing the variable swapping algorithm in a breadth-first manipulation scheme. In Section 6.3, we present our first solution approach. In Section 6.4, we discuss overhead optimization techniques. Section 6.5 describes the "sifting" algorithm, Section 6.6 presents the "window" technique. We discuss the node packing scheme in Section 6.7. We present our second approach in Section 6.8. Experimental results are given in Section 6.9. Most of the work presented in this chapter was first reported in [RGBS97].

6.1 Dynamic Reordering: Background

Terminology: Variables in BDDs are ordered such that the variable at level i has index i, i ranges from 0 to $n-1$ (starting from root to the constant nodes, n being the total number of variables). A variable with a lower value of index is "higher" in the order and vice versa. An *identifier (id)* is associated with each variable, and it also ranges from 0 to $n-1$. The identifier value for a variable remains constant throughout the life of the variable, however its index changes during reordering.

The basic operation in reordering is that of variable swapping. Variable swapping involves moving all BDD nodes at level i to level $i+1$ and nodes at level $i+1$ to i. Consider a node $F = (x_i, F_1, F_0)$ at level i, where x_i is the variable associated with level i, F_1 is the positive cofactor with respect to x_i, and F_0 is the negative cofactor. Similarly, let F_{11} and F_{10} be the two cofactors of F_1 with respect to x_{i+1} and F_{01} and F_{00} be the two cofactors of F_0. Using Shannon cofactor expansion,

$$\begin{aligned} F &= x_i F_1 + \bar{x}_i F_0 && (6.1)\\ &= x_i(x_{i+1}F_{11} + \bar{x}_{i+1}F_{10}) + \bar{x}_i(x_{i+1}F_{01} + \bar{x}_{i+1}F_{00}) && (6.2)\\ &= x_{i+1}(x_i F_{11} + \bar{x}_i F_{01}) + \bar{x}_{i+1}(x_i F_{10} + \bar{x}_i F_{00}) && (6.3) \end{aligned}$$

In short,

$$(x_i, F_1, F_0) = (x_{i+1}, (x_i, F_{11}, F_{01}), (x_i, F_{10}, F_{00})) \quad (6.4)$$

The Eqn. 6.4 forms the basis of swapping variables between levels i and $i+1$. Variable swapping is the core operation of reordering and is the dominating factor in memory and time consumption during reordering. The problem in swapping x_i and x_{i+1} with only local modifications lies in preserving the functionality of all nodes at indices i and $i+1$. This is necessary to avoid updating the references to those nodes by their parent nodes higher in the order. The updating of references is not efficient because the "in-degree" of a node could be arbitrarily large, making it infeasible to maintain a back pointer to all the parent nodes, and thereby making it necessary to traverse the graph.

6.1.1 Variable Swapping in Depth-First Implementation

Rudell [Rud93] proposed a scheme which forms the key idea for efficient variable swapping. The success of his technique lies in the ability to overwrite the contents of a node to maintain its functionality. This is illustrated in the Figure 6.1. Notice that

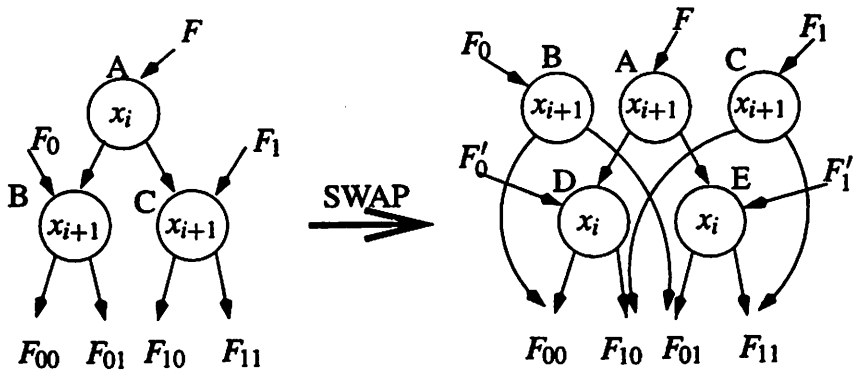


Figure 6.1 Variable swapping: node distribution before swapping (left), after swapping (right).

by swapping variables x_i and x_{i+1}, the contents of the original node (A) representing function F are overwritten as the cofactor pointers are changed. This results in the creation of two new nodes F_0' (D) and F_1' (E) (these may already be present in the unique table). The important point to note is that this change is completely local since the nodes above index x_i and below index x_{i+1} need not be changed and only nodes of levels i and $i+1$ need to be traversed. This warrants an array of unique tables rather than a single unique table for the whole BDD. In summary, the conventional variable

swapping involves creating at most two new BDD nodes and overwriting an existing BDD node.

6.2 Variable Swapping in Breadth-First Implementation: Problems

Let us now see if the same paradigm could be applied to a breadth first based package. Breadth-first schemes rely on two salient features: i) the increased locality of reference due to nodes belonging to an index beling located on the same page or set of pages, and ii) the ability to determine the indices of the cofactor nodes without fetching them from memory. To benefit from the locality of reference of the breadth-first scheme, it is crucial that the BDD node allocations are still local after reordering.

The first implementation of breadth-first manipulation ([OYY93]) used padding nodes to overcome the cofactor index determination problem, i.e., their algorithm manipulated quasi-reduced BDDs and indices of cofactors were always one more than the indices of the corresponding nodes. As reported in [AC94], this approach has significant memory and computational overheads. Moreover, an arbitrary overwriting of nodes during variable swapping would result in the loss of locality (nodes of an index would belong to the several different pages in the memory, sharing with nodes of other indices).

The approach in [AC94] uses a table which maps the page address of a node to its index, i.e., each page address is uniquely associated to an index. This mapping breaks down if we arbitrarily overwrite the content of an address during variable swapping. Consider Figure 6.2, which represents the node distribution before and after swapping x_i and x_{i+1}. Before swapping, nodes for variable x_i reside on page #10 and those for variable x_{i+1} reside on page #20. In the process of swapping, if we simply overwrite the node F, page # 10 contains nodes corresponding to both variables x_i as well as x_{i+1} and the index of a node is no longer uniquely identified by its page address. In addition, by arbitrary overwriting a node, the locality of nodes is no longer preserved.

Next we look at the breadth-first implementation in CAL [SRBS96]. In this approach, a node is represented by an $\{id, node\}$ pair (please refer to Figure 3.20 on page 68). Also, the node contains the id as well as the address of its cofactor nodes.

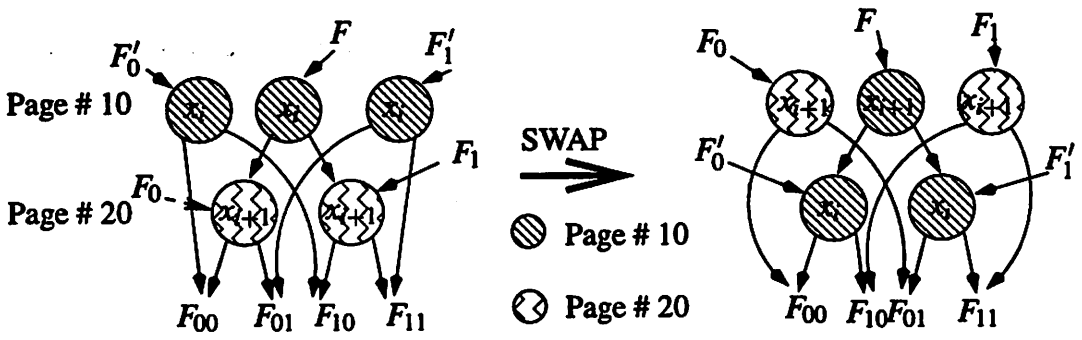


Figure 6.2 Variable swapping : node distribution before swapping (left), after swapping (right) in the breadth-first implementation of *Ashar et al.*

This data structure avoids the need for cofactor fetching or for looking up any table to determine the cofactor indices. However, like previous implementations, arbitrary overwriting of a BDD node suffers from two problems. First, it would result in the loss of locality. Second, since CAL uses a data structure for a node that stores both the *id* as well as the address of the cofactor nodes, we need to update the *id* field of all parent nodes of nodes at x_i and x_{i+1} (similar problem arises if the node stores the *index* of the cofactor nodes instead of their *id*).

We see that directly employing the depth-first reordering technique leads to problems in all breadth-first implementations. In the next section we propose our solutions.

6.3 Solution Approach A

Our first scheme is based on delayed updating, similar to garbage collection, to deal with the problems discussed in the previous section. This avoids the non-local computation involved in updating the cofactors, We delay the cofactor updating by overwriting the node with a "forwarding address" to the new node at an appropriate location and with proper functionality. Since the original node contains a forwarded address, we call it a *forwarding node*. This approach results in both memory and computational overhead: memory overhead because we cannot immediately reuse the node being forwarded and the memory consumption increases temporarily; and computational overhead because the information in the parent nodes of the forwarding node must be

updated at some point which requires traversing the nodes at higher indices and updating the contents if necessary. Since variable swapping is the core operation, we would like to optimize the cost of memory and computation overhead as much as possible. In sections 6.3.1 and 6.3.2, we explore two methods for variable swapping which preserve the locality of nodes and discuss their memory and computational overhead.

6.3.1 Method 1: Keeping *index* $\leftrightarrow$ *page* mapping constant

In the first method, pages in the memory are associated with the index, i.e., if the index of a variable changes, all the nodes corresponding to that variable need to be reallocated on the pages corresponding to the new index. Figure 6.3, shows an example of the node distribution before and after the swapping. Nodes representing cofactors F_0 and F_1 (nodes B and C respectively before swapping) are reallocated on page #10 in order to maintain the *index* $\leftrightarrow$ *page* mapping. This approach has overhead of 2 forwarding nodes (B, C) over the depth-first based swapping.

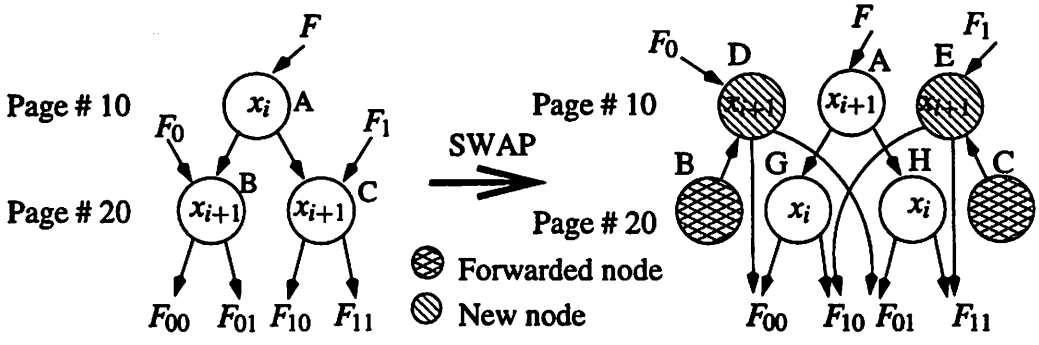


Figure 6.3 Variable swapping (keeping *index* $\leftrightarrow$ *page* mapping constant) : node distribution before swapping (left), after swapping (right).

In the worst case, nodes corresponding to (x_i, F_{10}, F_{00}) and (x_i, F_{11}, F_{01}) may already be present before swapping. After the swapping, both of them will become forwarded nodes on page #10 and will point to G and H respectively. In general, suppose there are N_i and N_{i+1} nodes at indices i and $i+1$ respectively. Further, suppose N_i' of N_i nodes have both the cofactors below index $i+1$. Then the memory overhead in this approach consists of $N_{i+1} + N_i'$ forwarded nodes.

6.3.2 Method 2: Keeping *id* ↔ *page* mapping constant

In this method, pages in the memory are associated with an *id*, i.e., variable. Hence, even if the index of a variable changes, all the nodes corresponding to that variable remain on the same set of pages. Again, consider the node for function F in Figure 6.4.

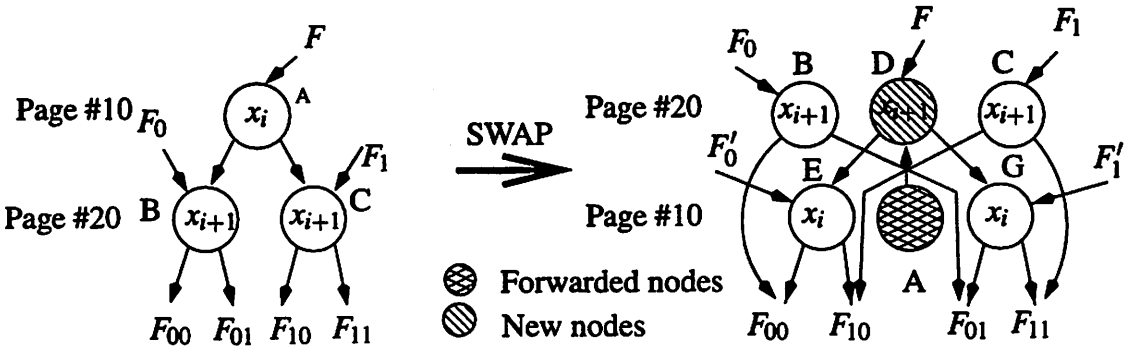


Figure 6.4 Variable swapping (keeping *id* ↔ *page mapping* constant): node distribution before swapping (left), after swapping (right).

In this approach, as a result of swapping, a new node (D) is created on page # 20 to represent the function F. Node A which represented the function F before swapping, becomes a forwarding node. Nodes F_0 and F_1 are simply moved to level i. We observe that there is one new node created (D) and one node is forwarded (A) after the swapping. In general for N_i nodes at level i, with N_i' of them having at least one cofactor at level $i+1$, we have memory overhead of N_i' forwarded nodes as compared to depth-first implementation.

Analysis

From the above two approaches we make the following observation. In Method 2, swapping variables at level i and $i+1$, requires traversing only nodes at level i which is not true for Method 1. This is an important advantage for Method 2 because traversing all nodes at a level is computationally expensive. Moreover, the memory and computation overhead in Method 1 is significantly more than that in Method 2. For these reasons we chose Method 2 for variable swapping in this approach (we refer to this as

```
swap_variable(i, checkFlag){
    foreach node in the unique table for index i {
        get then and else cofactors;
        if (checkFlag) update the cofactors;
        if (thenCofactorIndex > i+1 && elseCofactorIndex > i+1) continue;
        put the node in the processingList;
    }
    foreach node in the processingList {
        get F00, F01, F10, F11 cofactors (as shown in Figure 6.1);
        create or find F0' and F1';
        create a new F' and put a forwarding pointer in F;
        Append the list of forwarded nodes for index i;
    }
    Perform some book-keeping;
}
```

Figure 6.5 Algorithm for swapping two variables.

the CAL-A approach).

6.4 Memory and Computational Overhead Minimization

Since forwarding nodes do not represent any Boolean functions, they are essentially the memory overhead of the swapping algorithm. They also bring in computational overhead, since at some point during reordering BDD nodes need to be traversed and the forwarded cofactors updated. In order to reduce memory overhead, we can reuse the forwarded nodes. This requires fixing the cofactors of all BDD nodes which could be pointing to those forwarding nodes resulting in computational overhead. Hence, during variable swapping we need to be careful about fixing the cofactors (so as not to perform unnecessary computations) and in controlling the number of forwarding nodes (so that reordering does not run out of memory) resulting in memory and computational overhead trade-off. This makes the process of overhead minimization quite complex.

The algorithm for swapping two variables is given in Figure 6.5. We notice that all the nodes for a particular index are traversed during swapping. Some of these nodes become forwarding and are appended to the list of forwarding nodes for that index. At the end of each variable swapping we monitor the number of forwarding nodes. If this number exceeds a threshold, we traverse the BDDs appropriately, fixing the cofactor pointers and freeing the forwarding nodes.

Ideally, we would like to reduce memory overhead with little extra computation and mitigate the computation overhead by integrating the cofactor updating operation inside the swapping operation. In the following we discuss various observations about variable swapping (x_i and x_{i+1}, at indices i and $i+1$ respectively) which let us minimize the overheads. These observations are heavily used in optimizing the reordering algorithms.

Observation 1: Consider a node F at level i which is independent of the variable at level $i+1$. When the variables at level i and $i+1$ are swapped, no new node is created by the algorithm. This is illustrated in Figure 6.6. In all other cases

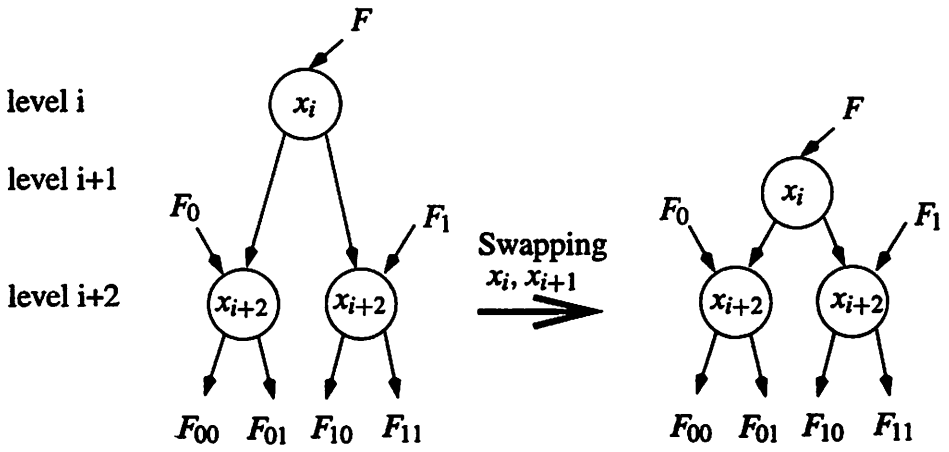


Figure 6.6 Variable swapping: if the cofactors of a node are independent of x_{i+1}, no new nodes are created in swapping.

(when at least one cofactor of F is at index $i+1$), we need to create a new node and the original node for F becomes forwarded (Figures 6.4 and 6.7). .

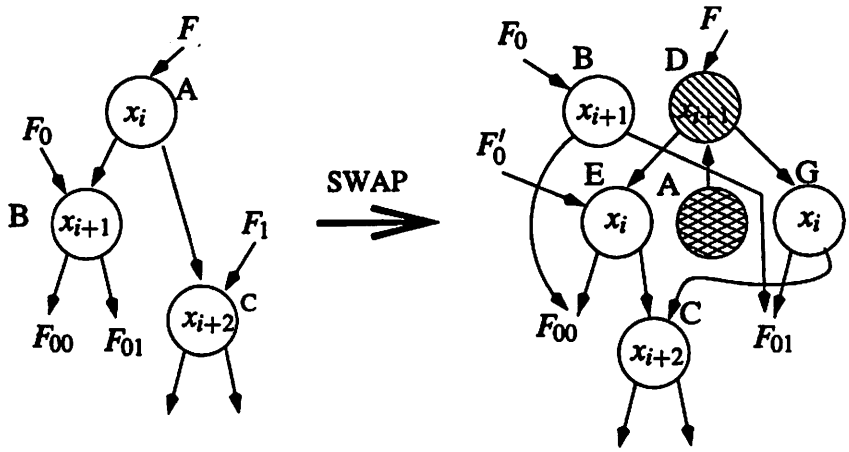


Figure 6.7 Variable Swapping : If any cofactor of a node is dependent on x_{i+1}, a new node and a forwarded node are created in swapping.

Observation 2: We need to traverse the unique table of variable x_i only. In some cases, it would be necessary to update the cofactors of nodes in x_i (illustrated in Figure 6.8). (some of the cofactor pointers could be pointing to forwarded nodes). However, upon careful analysis we can determine various stages in reordering algorithms where checking for forwarded nodes (i.e., updating the cofactors) would not be necessary, thereby avoiding the computational overhead.

Observation 3: Nodes corresponding to the variable x_i (the one going down) can either remain unchanged (Figure 6.6) or can become a forwarded node if they have cofactors at the next index $(i+1)$ (Figures 6.4 and 6.7).

Observation 4: Since the nodes at index i are cofactors of nodes at indices $i-1$ or lower, it implies from observation 1 that cofactors for some of the nodes at indices $i-1$ or lower become forwarded as illustrated in Figure 6.9).

Observation 5: The only change possible for the nodes corresponding to variable x_{i+1} (the one moving up in the order) is that their reference count might get decremented by 1 (as shown in Figure 6.9 for nodes C and D).

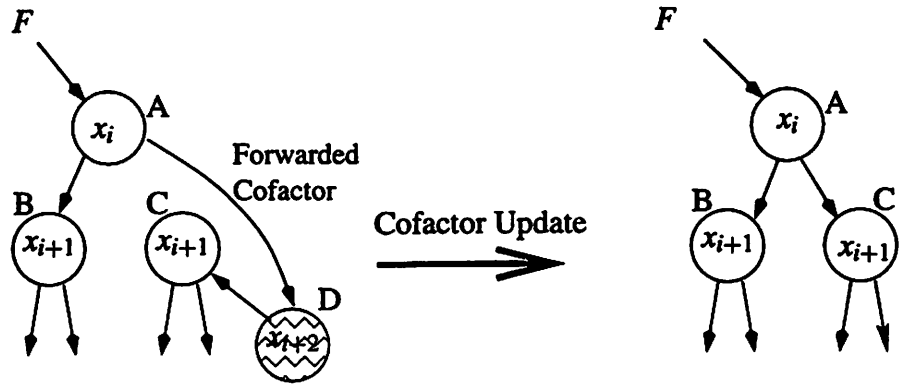


Figure 6.8 Variable swapping: if any cofactor of a node is forwarded, it needs to be updated before swapping.

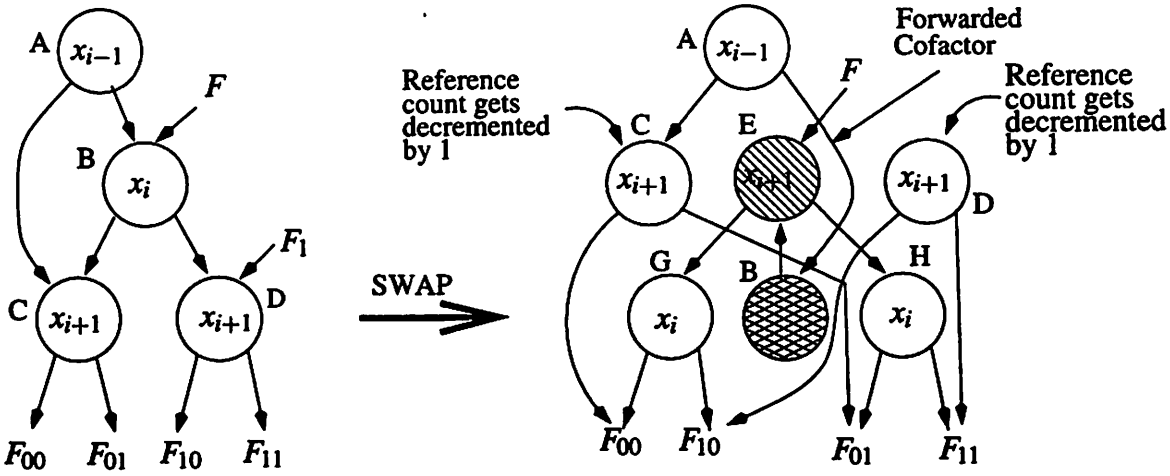


Figure 6.9 Variable swapping: when x_i and x_{i+1} are swapped, a node at higher index can get a forwarded cofactor.

Observation 6: When the nodes at levels i and $i+1$ are being swapped, none of the nodes for index $i+1$ or higher get affected.

Observation 7: During the swapping process, we need to traverse all nodes for variable x_i and update the cofactors if needed. Hence at the end of swapping, all nodes for x_i have valid cofactors.

Observation 8: Once a node becomes a forwarding node, we do not have to consider that node anymore for swapping purposes. Once all the pointers to the forwarding node are fixed, the node can be freed.

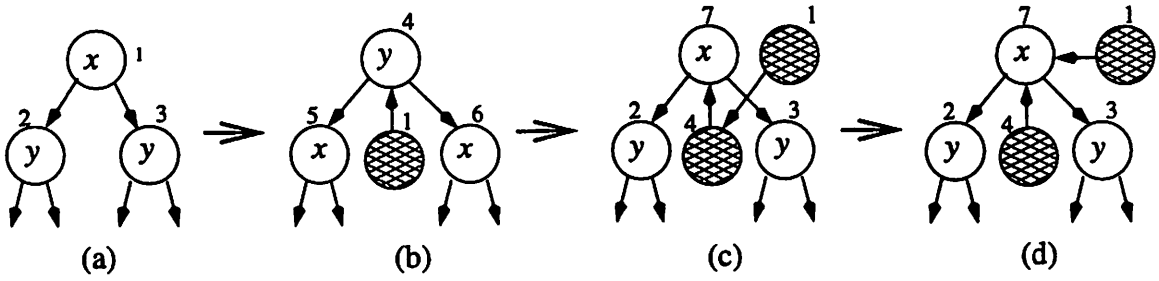


Figure 6.10 Double forwarding of nodes.

Observation 9: Sometimes a node, being pointed at by a forwarded node, becomes a forwarded node. This occurs when two variables x and y are swapped (x going down, y going up, Figure 6.10b), and at some later stage, y and x are swapped (y going down, x going up, Figure 6.10c). This results in *double forwarding*. By updating the first forwarded node we can get rid of the double forwarding (Figure 6.10d). For efficiency reasons we would like to fix them as they occur.

In conclusion, we discussed the variable swapping algorithm in breadth first scheme and related memory and computational overhead issues. We also looked at various properties of variable swapping. We will use these properties in the different parts of two reordering algorithms – sifting and window – to optimize the overheads.

6.5 Dynamic Reordering: Sifting Technique

This "sifting" algorithm is based on finding the optimum position for a variable, assuming all other variables remain fixed [Rud93]. The basic algorithm for sifting remains the same as the one given in [Rud93], but we have paid keen attention to minimizing the overhead as much as possible. First, we make use of a simple observation in deciding the initial direction (up or down) of swapping for the variable being sifted. Then, we divide the sifting process in four phases and use the properties from the previous section to optimize the overheads.

If the initial position of the variable is in the upper half then it is advantageous to sift the variable upwards first. As an example, consider the case when a variable's initial position is in the upper half and its best position in the bottom half. In Figure 6.11, we

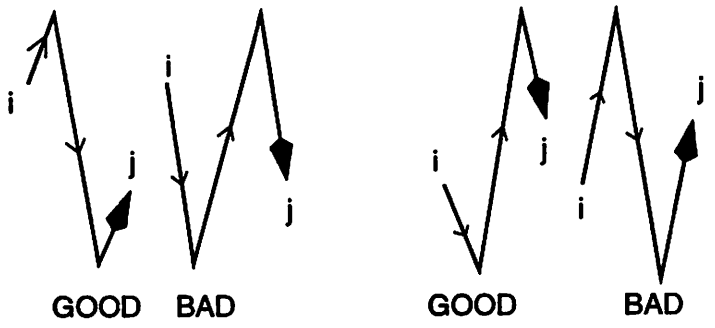


Figure 6.11 Strategies for sifting variables.

illustrate the amount of variable swapping needed in moving the variable for upwards first strategy as well as for downwards first strategy. Suppose i is the initial position of the variable ($i < n/2$). And j is the best position of the variable ($j > n/2$). The number of variable swappings needed if the variable is sifted upwards first is equal to $(i-1)+(n-1)+(n-j) = 2n-(j-i+2)$. However, if the variable is sifted downwards first, the number of swappings needed = $(n-i)+(n-1)+(j-1) = 2n-(i-j+2)$. Since $(j-i) > 0$, the number of swapping saved in a first approach is $2(j-i)$. It is clear that moving the variable upwards first results in the computational advantage. Similarly, it can be shown that moving the variable in the lower half downward first can result

in a computational advantage. Note that the same optimization can be achieved in the depth-first implementation as well.

To minimize the overhead of fixing the cofactors during the variable swapping and to minimize the traversal of BDDs during fixing of forwarded nodes, we have divided each variable "sifting" into four phases. These phases are shown in Figure 6.12. In each phase we have established certain invariants which cut down on the computation overhead, which we discuss below.

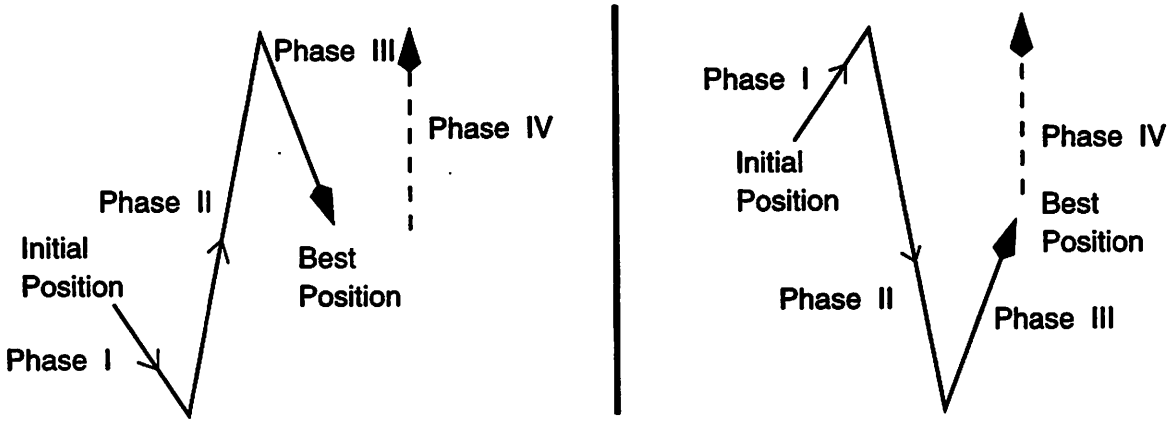


Figure 6.12 Various phases of sifting a variable.

Case 1 Starting position of the selected variable, x, is in the bottom half: in this case, the four phases of variable swapping are shown on the left in Figure 6.12.

Phase I: In this phase the variable x is successively swapped down to the bottom. Hence for all the swappings only nodes corresponding to variable x are traversed. At the start of phase I, there are no forwarded nodes and the only forwarded nodes created are for variable x. Hence, in this phase we need not update any cofactors. At the end of this phase: i) nodes corresponding to x do not have any forwarded cofactors (from observation 3) and ii) nodes corresponding to every other variable possibly have forwarded cofactors (from observation 4).

```
reorderingSift(startIndex)
    if (startIndex > numberVariables/2){
        /* Phase I */
        for(index = startIndex; index ≤ numVariables-2; index++)
            swap_variables(index, 0)
            Perform book-keeping
        endfor
        /* Phase II */
        for (index = numVariables-2; index ≥ startIndex; index--)
            Fix cofactors of nodes
            swap_variables(index, 1)
            Fix the double forwarding
        endfor
        for (index = startIndex-1; index ≥ 0; index--)
            Fix cofactors of nodes
            swap_variables(index, 1)
            Perform book-keeping
        endfor
        Reclaim forwarding nodes
        /* Phase III (need to move the variable to the best location)*/
        for (index = 0; index ≤ bestIndex-1; index++)
            swap_variables(index, 0)
        /* Phase IV (need to update the cofactors of variables) */
        for (index = bestIndex-2; index ≥ 0; index--)
            Fix cofactors of nodes
        endfor
    }
    else { /* Variable's starting position is in the top half */
        /* Phase I */
        for (index = startIndex; index ≥ 0; index--)
            Fix cofactors of nodes
            swap_variables(index, 1)
            Perform book-keeping
        endfor
        Fix the cofactors of all the nodes and reclaim forwarding nodes
        /* Phase II */
        for (index = 0; index ≤ startIndex-1; index++)
            swap_variables(index, 0)
            Fix the double forwarding of variable moving up
        endfor
        for (index = startIndex; index ≤ numVariables-2; index++)
            swap_variables(index, 0)
        endfor
        /* Phase III (need to move the variable to the best location)*/
        for (index = numVariables-1; index ≤ bestIndex; index--)
            swap_variables(index, 1)
            Fix the double forwarding of variable moving up
        endfor
        /* Phase IV (Fix the cofactors of higher indices) */
        for (index = bestIndex-1; index ≥ 0; index--)
            Fix the cofactors
        endfor
    }
```

Figure 6.13 Pseudo-code for *sifting* algorithm.

Phase II: In this phase, the variables which are going down potentially can have cofactors pointing to the forwarded nodes of x. Hence we need to update the cofactors of the nodes being swapped down. Also, until x is brought back to the original position i, it is reverse swapped with a variable it was swapped in phase I. Hence, we need to fix the double forwarding (from observation 9). At the end of this phase, all the variables have moved down at some point. From the observations 2, 3, 5, 6, and 7, the cofactors of all the nodes have been updated. At this point, we can reclaim forwarded nodes belonging to each index.

Phase III: In this phase, x is successively swapped down to the best position found at the end of phase II. Since, by the end of phase II, we have performed clean-up (updated forwarded cofactors, reclaimed forwarded nodes) this phase is similar to phase I.

Phase IV: This is the final clean-up phase. We want to make sure that all the nodes in the manager have proper cofactor nodes. For that purpose, we need to fix the cofactors of nodes between 0 and *best index.*

Case 2 Starting position of the selected variable, x, is in the top half: in this case, the four phases of variable swapping are shown to the right in Figure 6.12.

Phase I: In this phase, the variable x moves all the way to the top. From observation 4, the variable going down in the swapping may become forwarded nodes. Hence we would need to update the cofactors at each variable swap. At the end of this phase, we can reclaim all forwarded nodes.

Phase II: This phase is exactly similar to phase I of the previous case.

Phase III: Similar to phase II of the previous case. However, we need to perform swapping only till x gets to the best position.

Phase IV: Same as previous case.

The resulting pseudo-code for the *sifting* algorithm is given in Figure 6.13.

6.6 Dynamic Reordering : Window Technique

The window permutation algorithm proceeds by choosing a level i in the DAG and exhaustively searching all $k!$ permutations of the k adjacent variables starting at level i [FMK91, ISY91]. Typically this operation is repeated starting from each level until no improvement in the size is seen.

Based on the observations in Section 6.3.2, we make the following optimizations:

1. When variables corresponding to indices i, $i+1$ and $i+2$ are being permuted in a window, some of the cofactors of nodes belonging to indices $i-1$ and lower get forwarded.

2. At the end of one full pass (i.e., when the permutation window has been brought from top to bottom), nodes corresponding to all the variables may have forwarded cofactors.

3. Instead of starting the next pass again from the top, we start it at the bottom, i.e., window slides from bottom to top (Figure 6.14). At the end of the bottom-

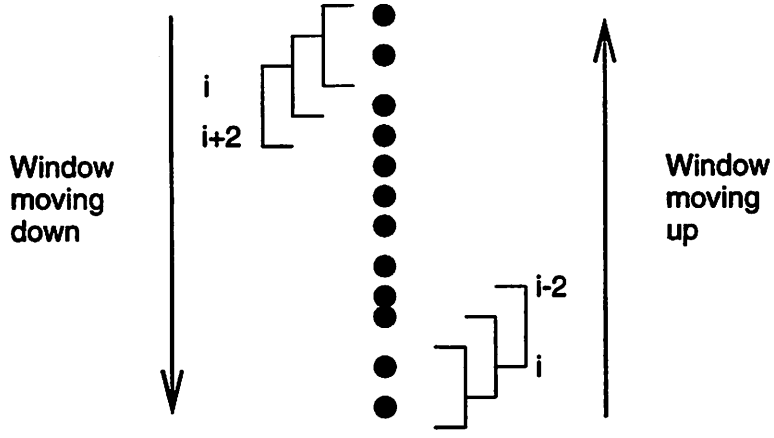


Figure 6.14 Alternate top-down and bottom-up swapping.

up pass, all the cofactors of all the variables are updated (as observed in Section 6.3.2).

4. For each top-down and bottom-up pass, we save the overhead of traversing all the nodes fixing the cofactors.

5. If the reordering converges in a top-down pass, we perform a clean-up phase, where all the cofactors are updated.

6.7 Node Packing

In the sifting and window algorithms for reordering, one or more variables are moved up and down in the ordering. As a result, the number of BDD nodes corresponding to those variable(s) change drastically (from one node at the bottom of the order to tens of thousands of nodes in the middle of the order). To accommodate the large number of nodes, a corresponding number of pages gets allocated for the variable being moved in order. As a result, if the final position of a variable does not require too many nodes, these nodes are scattered over many pages, i.e., memory fragmentation occurs. This phenomenon has two side-effects. The first one is a loss of locality of reference, because even for a small number of nodes, we need to access several pages. The second problem is excessive memory consumption: if a large number of pages are left allocated for a variable with few nodes, these pages cannot be used for other memory allocations and it is possible that the application runs out of memory. To address these problems, we use the technique of intermittent repacking of nodes. This technique is transparent to the user (like garbage collection). We define the term *Utility Ratio* (*u.r.*) for a variable at index i as $u.r. = \frac{n_i}{p_i * N_p}$ where, n_i is number of nodes at index i, p_i is number of pages allocated for nodes at index i, N_p is number of nodes which can fit in a page. In other words, *Utility Ratio* captures the memory fragmentation. During dynamic reordering, we monitor the *Utility Ratio* of the variable moving in the order. Whenever this ratio drops below a threshold, we stop the operation and repack the nodes. This process requires copying the set of nodes onto a new set of pages, leaving a forwarding address at the old nodes. Next, we traverse the BDD nodes at the higher levels (lower indices) to fix their cofactors.

In the *window* scheme of reordering, we perform this packing for three (the width of the window) consecutive variables at a time.

6.8 Solution Approach B

After implementing the techniques described in approach A, we still noticed significant computation and memory overhead compared to reordering schemes used in the depth-first approaches. In approach B, we evaluated the use of depth-first reordering technique in a breadth-first based package. This approach consists of following steps:

1. Change the data structure to conform to conventional BDD nodes, i.e., a node contains – reference count, id, then cofactor, else cofactor, and next pointers.
2. Perform a conventional reordering.
3. Reallocate nodes on pages to maintain the locality.
4. Change back to the original data structure.

The overhead in this approach is involved in (a) changing the data structures in steps 1 and 4, and (b) reallocating the nodes in step 3. We have formulated a strategy to perform an in-place reallocation of nodes to achieve the local distribution of nodes in memory (shown in Figure 6.15). This strategy has no memory overhead (the reallocation is done in-place) and requires three passes of the BDD. The first pass is done on a page-by-page basis, leading to local memory accesses. The second pass requires tracing the next pointers of the nodes which could be on multiple pages. The third pass again is done on a page-by-page basis leading to local accesses.

However, in the current implementation we simply duplicate the final set of nodes into new memory locations, freeing the old space later on. This results in a temporary memory overhead.

6.9 Experimental Results

6.9.1 Experimental Setup

We implemented our reordering scheme in the CAL BDD package [RS97]. This package has been chosen for three main reasons: (i) in [SRBS96], it has been clearly shown to outperform other breadth-first based implementations, (ii) CAL is integrated with the synthesis tool SIS [SSL+92] and verification tool VIS [BSA+96a] making it easier to

```
reallocate_nodes_in_place{
    /* First pass, get the new addresses of the nodes */
    initialize the pointers for each index;
    traverse all the nodes on page-by-page basis;
    for each node{
        get the new address for the node by looking at the corresponding pointer;
        put the new address in the next field;
        update the pointer;
    }
    /* Second pass, update the cofactors of the nodes */
    while there are still nodes to be updated {
        tmp = content of next pointer of node;
        store the content of the node in the next pointer;
        node = next pointer of the new location;
    }
    /* Third pass, update the next pointer of the nodes by hashing them appropriately */
    traverse all the nodes on page-by-page basis; for each node{
        get the value of the next pointer by hashing it to appropriate unique table;
    }
}
```

Figure 6.15 In-place reallocation of nodes to maintain locality.

perform benchmarking, and (iii) as opposed to other implementations, CAL provides a comprehensive set of high performance BDD algorithms. This makes the integration of our work on dynamic reordering worthwhile, making it directly applicable to practical problems.

For comparison purposes we have used the CMU package developed by Long [Lon93] and the CU package [Som97] developed at University of Colorado at Boulder. Both of these packages are publicly available and are being widely used in industry and academia. We have used the standard ISCAS and MCNC benchmark examples for our experiments. In addition we have also used some industrial examples collected from various sources. All our experiments were performed on a 250MHz DEC Alpha 21164 with 4MB L2 cache, 1GB RAM, and 3GB swap space. A time limit of 3000 CPU seconds was used.

In the first experiment we used an artificially created example given below:

$$f = \sum_{i=0}^{i=\frac{n}{2}-1} a_i a_{i+\frac{n}{2}}$$

We start with an initial variable ordering of $a_0, a_1, \ldots, a_{n-1}$. For this variable ordering, the function f has exponential size in n. By changing n, we can do a performance comparison with gradual increase in BDD size. In Table 6.1, we give the elapsed time in performing the reordering to obtain the optimal variable order $(a_0, a_{\frac{n}{2}}, a_1, a_{\frac{n}{2}+1} \ldots)$. In all the cases the optimal order was obtained by all the BDD packages. We observe that the approach B (column CAL-B), significantly outperforms approach A (column CAL-A). Comparing CAL-B with other packages we observe that CAL-B outperforms CMU package in all the cases; CU performs the best.

In the second experiment, we created output BDDs for combinational circuits and partitioned transition relations for sequential circuits, and invoked dynamic reordering to reduce the BDD sizes. In Table 6.2 and Table 6.3, the second column gives the total number of nodes present in the manager when the reordering is invoked. We compare the quality and the run time of reordering for both sift and window schemes. Under the individual package columns, we give the number of nodes at the end of dynamic ordering and the elapsed time in seconds. We observe that the CU package performs the best. The performance of CAL-B is comparable in most cases. For window based reordering we observe that due to different terminating conditions during reordering,

# Variables	Initial Size	Sifting				Window			
		CMU	CU	CAL-A	CAL-B	CMU	CU	CAL-A	CAL-B
32	131087	3	1	2	2	3	2	5	2
34	262160	8	3	4	4	9	6	10	7
36	524305	19	7	11	9	22	15	23	17
38	1048954	43	16	27	21	47	32	54	40
40	2097171	95	36	62	48	105	71	128	91
42	4194324	208	78	141	108	223	157	279	206
44	8388629	431	168	323	246	471	334	629	454
46	16777238	928	376	s.o.	524	s.o.	743	s.o.	982

Table 6.1 Direct reordering performance comparison for an artificial set of examples.

s.o. : Space out

the quality and time for reordering for different packages differ, making the comparisons a bit difficult. For instance, CU takes longer for some examples and also results in better quality. CAL-B performs better than CMU for similar quality results.

Example	Initial Size	Final Size				Elapsed Time			
		CMU	CU	CAL-A	CAL-B	CMU	CU	CAL-A	CAL-B
C1908	30933	7912	7355	7960	7955	6	3	4	3
C1355	239030	13438	11129	13286	13286	17	6	15	11
C2670	1169361	3990	2705	5821	5821	156	39	123	78
C3540	2581494	67648	60327	68568	68733	349	228	530	364
C5315	1980487	7819	3759	9674	9904	650	173	546	336
C880	186493	9043	8940	9123	9123	24	10	19	14
every	47390	3746	4038	5475	3679	11	6	12	9
abs_bdlc	132096	38787	37189	55997	39851	115	43	94	59
biu	124015	26353	27165	31877	26501	136	39	98	73
bdlc	136002	38425	42951	39818	38504	153	55	118	81
minmax12	43315	8607	9520	6617	8652	25	6	21	15
bigkey	10464	7689	7614	7562	7729	23	10	25	12
s1423	61392	14882	15972	15890	14951	65	20	42	39
s4863	306070	149723	165132	143855	135856	258	88	173	100
s5378	204598	35007	34674	21934	34904	131	43	72	54
s6669	368457	147723	132823	120616	158379	565	84	252	168

Table 6.2 Sifting based reordering: performance and quality comparison.

Example	Initial Size	Final Size				Elapsed Time			
		CMU	CU	CAL-A	CAL-B	CMU	CU	CAL-A	CAL-B
C1908	30933	8614	8617	7483	7483	3	2	4	2
C1355	239030	12206	12209	13835	13836	7	5	11	7
C2670	1169361	1139930	1139905	1159905	1159905	54	23	39	20
C3540	2581494	562182	562148	438378	438378	425	372	564	294
C5315	1980487	1531110	1531113	1435721	1435721	103	44	131	52
C880	186493	35926	35928	35239	35239	17	10	19	10
every	47390	8555	4038	24580	11406	5	6	4	3
abs_bdlc	132096	112583	37189	122401	108736	14	42	4	11
biu	124015	106227	27165	109633	104126	8	39	7	3
bdlc	136002	107853	42951	127151	109324	12	55	8	7
minmax12	43315	18779	9552	32520	15359	9	6	2	4
bigkey	10464	8009	7614	10272	10420	2	10	1	1
s1423	61392	52597	15996	54550	51014	6	20	2	3
s4863	306070	230602	161242	249523	240563	45	78	46	25
s5378	204598	141568	42858	156425	130650	17	47	7	9
s6669	368457	290373	146078	347897	298430	42	61	8	16

Table 6.3 Window based reordering: performance and quality comparison.

In Table 6.4 we compare the memory consumption in each BDD package. These numbers are reported in terms of number of pages (8KB size). For CAL-B we report the peak memory usage (which includes the memory allocated for reallocating nodes), even though the final memory usage is much less. Notice the significant reduction in the memory used in CAL-B compared to CAL-A. Also note that the memory consumption in CAL-B is comparable with that in CU, both of which are larger than that in CMU. In some cases, CAL-A has better memory performance than CAL-B. Upon investigation, we found that this was due to the memory overhead incurred during the reallocation of nodes in new memory space. We would like to mention here that in general this memory overhead was not found to be significant compared to memory used before reordering. The larger memory consumption in CAL can be taken care of with minor changes in its implementation.

Example	Sifting				Window			
	CMU	CU	CAL-A	CAL-B	CMU	CU	CAL-A	CAL-B
C1908	566	955	1391	827	566	955	1141	827
C1355	1264	1907	3669	1971	1264	1706	2218	1971
C2670	8651	11972	21818	12582	8651	12237	14793	17062
C3540	14828	22696	40455	13735	14828	22971	18069	15143
C5315	10587	15842	37200	11419	10587	16309	12764	16923
C880	1243	1650	3308	1733	1243	1673	2099	1797
every	384	1155	2835	1378	384	1155	1624	1378
abs_bdlc	912	2651	14969	2773	912	2651	3119	3029
biu	811	2419	11473	2927	811	2419	3089	3183
bdlc	1067	2748	14801	3111	1067	2748	3460	3367
minmax12	309	714	4918	1016	309	714	1590	1016
bigkey	2013	5160	4143	4981	2013	5160	4197	4981
s1423	597	1254	8425	1514	597	1254	1893	1642
s4863	2802	5528	22159	5025	2802	5544	4488	5409
s5378	1349	4255	8128	3945	1349	4277	4064	4329
s6669	2820	8476	28952	5591	2820	8486	4749	6103

Table 6.4 Memory consumption comparison for various packages.

6.9.2 Analysis

From the above set of experiments, we observe a mixture of performance by our reordering scheme. We make the following observations:

1. In the current experiments, we have compared the raw performance of reordering in the two schemes. For practical purposes, however, it is more important to compare the overall performance of the packages embedded in applications like sequential verification, design verification, etc. By leveraging the locality of access, a breadth-first technique can manipulate large BDDs more efficiently than a depth-first technique. Since invoking dynamic reordering introduces a performance overhead in the breadth-first scheme, reordering techniques are needed which have lower performance overhead, probably at the cost of smaller reduction in BDD sizes. However, this loss of quality can be offset by the higher efficiency of breadth-first manipulation.

2. In the current reordering schemes, it is possible to get better performance by tuning two parameters: the threshold value for invoking dynamic reordering (minimum number of nodes needed to invoke reordering) and the maximum number of forwarding nodes allowed during reordering (point at which the relevant BDD nodes are traversed to update the cofactors and to free the forwarding nodes). The optimum values of these parameters are system dependent. For instance, for larger available memory we can set them at higher values. This leads to less frequent invocation of reclaiming forwarded nodes (leading to better performance) without getting into memory overflow problem. The behavior of the first parameter (when to invoke dynamic reordering), is a complex one. On the one hand we do not want to invoke dynamic reordering too frequently (setting the limit higher) to avoid the computational and memory cost; on the other hand invoking dynamic reordering when the BDD size gets too large makes it expensive.

6.10 Conclusions

The goal of this work was to establish the feasibility of dynamic reordering in breadth-first packages. In particular, we wanted to demonstrate that the memory and computation overhead in the core operation of variable swapping can be reduced with proper

implementation. Our experimental results show that the reordering inside the CAL package has comparable performance as that of CUDD package. The difference in the performance can be explained by the fact that the CUDD package has better heuristics for dynamic reordering algorithms. However, these heuristics are not specific to the package and can be implemented in other packages. In the current implementation, we have only incorporated the "interaction matrix" optimization as in [Som97]. It is our belief that by adding other heuristics, the performance of CAL package can be further improved and would be on a par with any other package. In the current experiments, we have compared the raw performance of reordering in the two schemes. For practical purposes, however, it is more important to compare the overall performance of the packages embedded in applications like sequential verification, design verification, etc. By leveraging the locality of access, a breadth-first technique can manipulate large BDDs more efficiently than a depth-first technique. Since invoking dynamic reordering introduces a performance overhead in the breadth-first scheme, there is a need to investigate reordering schemes which have lower performance overhead, probably at the cost of smaller reduction in BDD sizes. However, this loss of quality can be offset by the higher efficiency of breadth-first manipulation.

Part II

State Transition Graph Representation and Traversal

Chapter 7

Efficient Techniques for State Space Traversal

So far we have presented the techniques which target the efficient manipulation of binary decision diagrams as a data structure to represent Boolean functions. In particular, we considered computer architecture based solutions for BDD manipulation (Chapters 3, 4, and 5). In this chapter, we investigate application-specific solutions for BDD-based verification algorithms. In particular, we address the issue of state transition graph representation and state-space traversal of finite-state systems. We establish that the core computation in BDD-based state-space traversal is that of forming the image and pre-image of a set of states under the transition relation characterizing the system. In this chapter, we consider several solution approaches order to make this step as efficient as possible.

Th chapter is organized as follows: Section 7.1 presents the preliminaries and motivates the need for symbolic techniques for state enumeration. In the next five sections, we describe various techniques which are targeted towards state-transition graph representation and state enumeration. In Section 7.2, we describe our method for compact FSM representation of a network. Section 7.3 discusses our technique for early variable quantification for efficient image/pre-image computation. In Section 7.4, we discuss a technique to compactly represent the behavior of the finite-state machine by partitioning the underlying combinational network. We describe our findings on efficient usage of don't cares in image/pre-image computation in Section 7.5. In Section 7.6 we present a new technique to eliminate redundant latches, thereby improving the efficiency of state-space traversal algorithms. We provide experimental results in Section 7.7. Most of the work presented in this chapter was first reported in [RAP+95].

7.1 Motivation

In this section we motivate the importance of efficiently computing the image and pre-image of a set of states in various formal verification paradigms. We also introduce the problem of state explosion to motivate the need for using BDDs for representing and manipulating state space.

We start by defining the concepts of the image and pre-image of a set of states and also the set of reachable states in a finite state system. Following definitions assume a transition relation $T(\vec{x},\vec{u},\vec{y}) : \mathbb{B}^n \times \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}$.*

Definition 1 *Let $P(\vec{x}) \in \mathbb{B}^n$. The* **forward image** *(also called image) of P under the transition relation T is the set Q, such that,*

$$\vec{y} \in Q \;\Leftrightarrow\; \exists \vec{x} \in P, \exists \vec{u} \text{ s.t. } T(\vec{x},\vec{u},\vec{y}) = 1 \tag{7.1}$$

Definition 2 *Let $P(\vec{y}) \in \mathbb{B}^n$. The* **pre-image** *of P under the transition relation T is the set Q, such that,*

$$\vec{x} \in Q \;\Leftrightarrow\; \exists \vec{y} \in P, \exists \vec{u} \text{ s.t. } T(\vec{x},\vec{u},\vec{y}) = 1 \tag{7.2}$$

Definition 3 *Let $I(\vec{x})$ be the set of initial states of the system. The set of* **reachable states** *of the system, $R(\vec{y})$ is the least fixed point of*

$$\begin{aligned} R_0(\vec{y}) &= I(\vec{y}) \\ R_{k+1}(\vec{y}) &= R_k(\vec{y}) \cup \exists \vec{x}, \vec{u}\,[R_k(\vec{x}) \wedge T(\vec{x},\vec{u},\vec{y})] \end{aligned} \tag{7.3}$$

7.1.1 Formal Design Verification

In this section we informally describe the CTL model checking and language containment approaches to formal design verification and indicate the core operations in the underlying computation. In both cases the underlying design is characterized by a *Kripke structure* [CES86].

*For a brief background on FSM and the terminology used in this chapter, please refer Section 2.3.

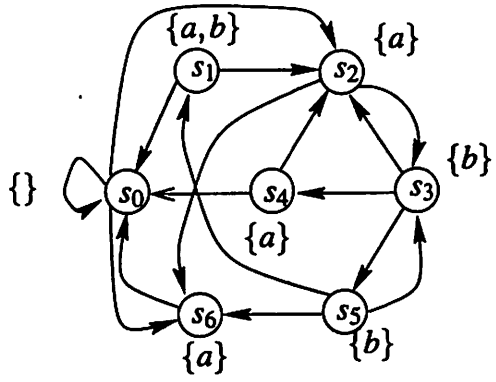


Figure 7.1 An example illustrating a Kripke structure. $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6\}$, $AP = \{a, b\}$. An edge from state s_i to s_j indicates that $(s_i, s_j) \in T$. States are labeled with the subset of APs true at the state. A path through K is a sequence of states $\sigma_1, \sigma_2, \cdots$ such that $\forall i (\sigma_i, \sigma_{i+1}) \in T$.

Definition 4 *A* **Kripke structure** $\mathcal{K}$ *is a triple* (S, T, L)*, where* S *is a finite set of states,* $T \subset S \times S$ *is the transition relation, and* $L : AP \rightarrow 2^S$ *is the labeling function mapping* **atomic propositions (AP)** *(we view outputs as atomic propositions) to sets of states. A pictorial representation of a Kripke structure is given in Figure 7.1.*

CTL Model Checking Paradigm

In the CTL model checking paradigm, properties are expressed as formulae from an inductively defined syntax. Truth of the formulae is interpreted over states in Kripke structures; determining the truth value of a formula over a state in the structure is referred to as model checking and can be algorithmically performed using fixed point calculations. Precise syntax and semantics are given in [Eme90]. As an example, state s_0 in the Kripke structure of Figure 7.1 models the formula $EF(a \wedge b)$ ("there exists a path to a state where both a and b hold"). This is because s_1 is labeled by a and b, and there is a path from s_0 to s_1, namely $s_0 \rightarrow s_2 \rightarrow s_3 \rightarrow s_5 \rightarrow s_1$. This result can be mathematically obtained by finding the least fixed point of

$$\begin{aligned} R_0(\vec{x}) &= p \\ R_{k+1}(\vec{x}) &= R_k(\vec{x}) \cup EX R_k(\vec{x}) \end{aligned} \tag{7.4}$$

where p denotes the set of states which satisfy the formula $(a \wedge b)$, i.e. set of states labeled with "a" and "b". Note that the set of states satisfying $EX R(\vec{x})$, i.e. the set of

states that can reach some states in $R(\vec{x})$ in one step, can be found by computing the *inverse image* of $R(\vec{x})$, with respect to the transition relation. Similarly, evaluation of other CTL formulae require repeated image and pre-image computations.

Language Containment Paradigm

In the language containment paradigm, the design is identified by the set of generated output traces L_D, and a property is given by a set of acceptable traces L_P (typically specified by a set of states, also called *fair states*). Verification consists of checking whether all design behavior is acceptable, i.e., checking $L_D \subset L_P$, which in turn is equivalent to checking that $L_D \cap \bar{L}_P$ is empty. Kurshan [HK90] observed that for certain classes of properties (namely deterministic L-automata) the set $\bar{L}_P$ is efficiently computable. Both L_D and L_P are modeled as Kripke structures and a new Kripke structure is formed by composing them appropriately. Verification consists of finding a path in a Kripke structure which starts at the initial state and leads to a fair cycle i.e. a cycle which includes at least one state from a designated subset of *fair* states F [EL85]. Conceptually, this check may be performed by first finding the set of states F^* which reach a fair cycle. Thus the property fails if and only if the initial state lies in F^* (since we want $L_D \cap \bar{L}_P = \phi$, i.e., no fair cycles).

Suppose $R_\infty(\vec{x})$ represents the set of reachable states. Limit the transition relation to the set of reachable states by $\tilde{T}(\vec{x},\vec{y}) = T(\vec{x},\vec{y})R_\infty(\vec{x})$.

The algorithm to find set of states F^* is as follows:

1. Initialize $F_0(\vec{x}) = R_\infty(\vec{x})$.

2. Compute A_∞ using the following fixed point computation:

$$\begin{aligned} A_0(\vec{x}) &= F_0(\vec{x}) \\ A_{k+1}(\vec{x}) &= A_k(\vec{x}) \cap \exists y[A_k(\vec{y}) \wedge \tilde{T}(\vec{x},\vec{y})] \end{aligned} \tag{7.5}$$

   $A_\infty(\vec{y})$ gives the set of states which can reach some states of $F_0(\vec{y})$ which lie on a cycle.

3. Compute B_∞ using the following fixed point computation:

$$\begin{aligned} B_0(\vec{x}) &= A_\infty(\vec{x}) \cap F(\vec{x}) \\ B_{k+1}(\vec{x}) &= B_k(\vec{x}) \cup \exists \vec{y}[B_k(\vec{y}) \wedge \tilde{T}(\vec{x},\vec{y})] \end{aligned} \tag{7.6}$$

$B_\infty(\vec{x})$ gives the set of states which can reach some states of $A_\infty(\vec{x}) \cap F(\vec{x})$.

4. $F_0(\vec{x}) = B_\infty(\vec{x})$.

5. Repeat (2-4) until convergence.

6. F^* is given by the least fixed point of

$$\begin{aligned} C_0(\vec{x}) &= F_0(\vec{x}) \\ C_{k+1}(\vec{x}) &= C_k(\vec{x}) \cup \exists \vec{y} [\tilde{T}(\vec{x},\vec{y}) \wedge C_k(\vec{y})] \end{aligned} \tag{7.7}$$

It is apparent from Equations [7.4, 7.5, 7.6, 7.7] that the core computation in verification consists of taking the image or inverse image of sets of states under the transition relation.

7.1.2 State Explosion

Often designs are constructed by linking components together. The synchronous product of components defines a single Kripke structure (also referred to as the *product machine*), the state space of which is the product of the components' state spaces. Hence algorithms that directly manipulate states will have time and space complexity that is exponential in the size of the system description. Indeed, the computational complexity of state transition graph related problems is known to be PSPACE–complete [ASB93]. The complexity introduced by concurrent interaction is popularly referred to as the *state explosion problem*. The quest for heuristic solutions to this problem forms the forefront of research in formal verification [ASS+94, BCMD90, CHM+93, CM90b, Gra94].

Transition relations and sets of states can be represented using BDDs of their characteristic functions, which can be used for efficient fixed-point computations [BCMD90, CM90b, Pix90]. BDDs are now extensively used for both design and implementation verification of hardware systems and many non-trivial design examples have been verified using BDDs [CYF94, McM93]. Still, there are many instances of medium sized circuits that cannot be verified using existing BDD techniques.

In the next five sections, we describe various techniques which enable state transition graph representation and state-space enumeration for large designs. These are:

- Use of clustered transition relations – grouping parts of the design to reduce the number of iterations required for each image and pre-image computation.
- Ordering of clustered transition relations for efficient image and pre-image computation.
- Network partitioning for compact state-transition graph representation
- Use of don't cares to reduce BDD sizes and computation times.
- Removal of redundant latches via constant propagation and retiming.

7.2 Clustered Transition Relations

A transition relation can be represented either as a monolithic relation or as a partitioned one as described below.

Monolithic Transition Relation: In this representation, the transition relation of the system is represented by a single BDD [BCMD90] which is the conjunction of the transition relations of the individual latches. As the circuit complexity grows, the size of the transition relation usually explodes. Hence this approach becomes infeasible for large, complex circuits.

Partitioned Transition Relation: In this case, a vector of transition relations is used; each element of the vector represents the next-state relation for a latch. Coudert [CM90b] proposed reducing image computations to range computations by exploiting the property of the constrain operator; the range computation is performed by recursive co-factoring. The efficiency of this approach comes from caching intermediate results and exploiting disjoint support. Touati [TSL+90] suggested a similar approach based on forming the product as a balanced binary tree. Image computation or pre-image computation is carried out iteratively using transition relations for individual latches. Heuristically speaking, as the number of latches in the system grows, the computation time increases.

We present a simple extension of these two approaches which overcomes some of their shortcomings. We represent the transition relation of the system by a vector of *clustered transition relations*. First, the next-state relation of each latch is computed. Next,

a group of transition relations are clustered together to form a vector of clustered transition relations. The idea is illustrated below.

Suppose the original vector of transition relations corresponding to individual latches is given by $T_i = T_i(\vec{x}, \vec{u}, y_i)$ for $i = 1, 2, \ldots n$. Then the image of a set of states $A(\vec{x})$ is given by,

$$Image(A(\vec{x})) = \exists \vec{x}, \vec{u} [A(\vec{x}) \prod_i T_i(\vec{x}, \vec{u}, y_i)] \tag{7.8}$$

The transition relation of a clusters of latches is the product the transition relations of corresponding latches. If there are K clusters $C_1, C_2, \cdots C_k$ of latches, then the image computation can be equivalently written as,

$$Image(A(\vec{x})) = \exists \vec{x}, \vec{u} [A(\vec{x}) \prod_{i=1}^{i=k} T_{C_i}] \tag{7.9}$$

where $T_{C_i} = \prod_{j \in C_i} T_j(\vec{x}, \vec{u}, y_j)$.

In [BCL91a], Burch also proposed the use of clustered transition relations to represent circuits more efficiently. Latches were grouped together to form clusters but no automatic way to form clusters was given. Their technique possibly required user expertise, based on circuit structure.

In our approach the user specifies a limit on the BDD size of individual clusters (partition size limit). The next-state relations of latches are ordered using the heuristics given in Section 7.3. Then the next-state relations of latches are conjoined in this order until the product size surpasses the user specified limit. At this point the current cluster is complete and is stored in an array. Then, the clustering continues starting from the next latch. This is illustrated in Figure 7.2.

7.3 Ordering of Clustered Transition Relations

Since the system behavior is represented in terms of clusters of transition relations, the core verification operations (image and reverse image computation) are performed iteratively, one cluster at a time. Suppose $A(\vec{x})$ represents the set of states, and $T_i(\vec{x}, \vec{u}, \vec{y}_i)$ represents the transition relation of the i^{th} cluster. Equation 7.9 can be rewritten as

$$\begin{aligned} Image(A(\vec{x})) \;=\; & \exists \vec{x}, \vec{u} \, [A(\vec{x}) \wedge T_1(\vec{x}, \vec{u}, \vec{y}_1) \wedge T_2(\vec{x}, \vec{u}, \vec{y}_2) \wedge \\ & \ldots \wedge T_k(\vec{x}, \vec{u}, \vec{y}_k)] \end{aligned} \tag{7.10}$$

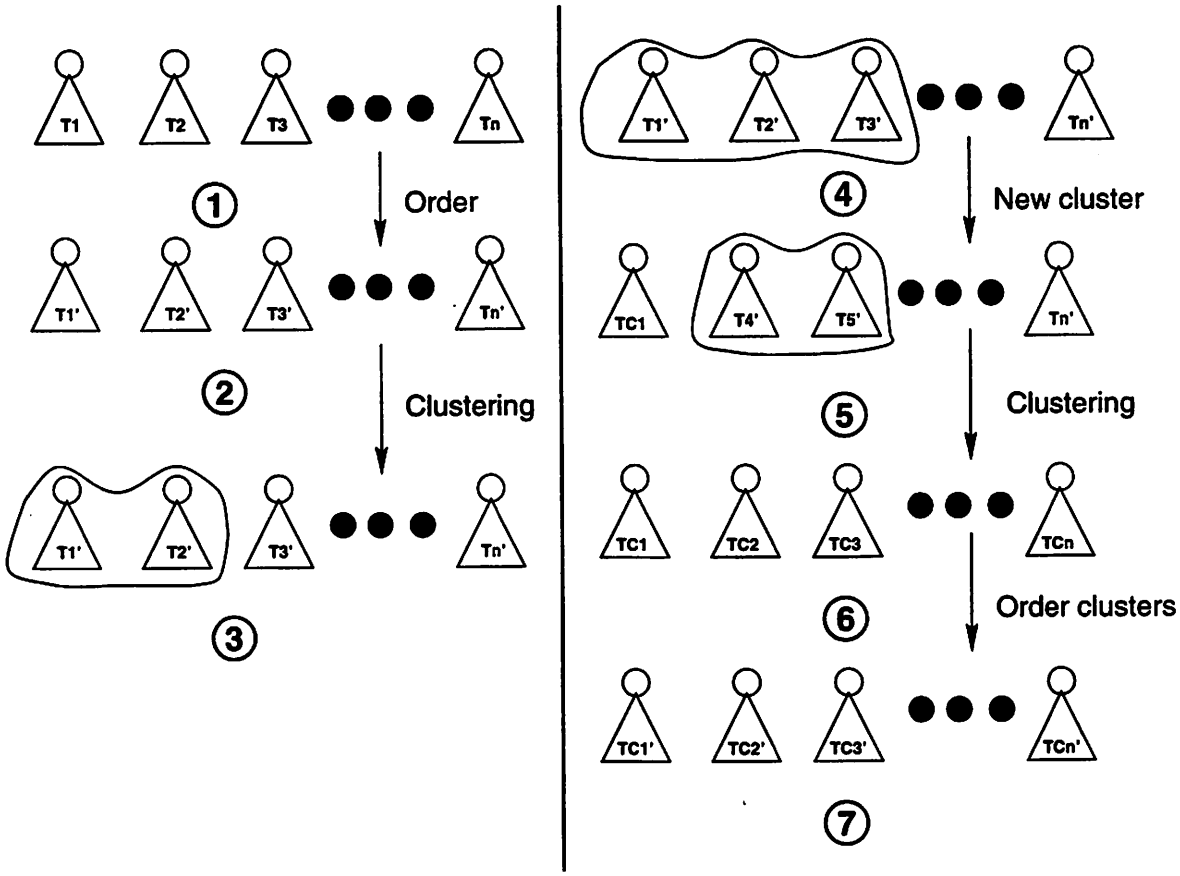


Figure 7.2 Illustration of ordering and clustering.

Notice that if we were to take the product of individual clustered transition relations T_i's and finally with the set of states given by $A(\vec{x})$ before quantifying out any variable, the computation will amount to using a monolithic transition relation. As a result it will suffer from the similar problems as the monolithic transition relation approach.

It has been empirically observed that quantifying out variable(s) from a function leads to smaller BDD sizes. But existential quantification does not distribute over conjunctions, i.e.,

$$\exists_x (T_1(x,y) \cdot T_2(x,y)) \neq \exists_x T_1(x,y) \cdot \exists_x T_2(x,y)$$

Hence, we cannot simplify individual clustered transition relations before taking the product.

However, existential quantification does distribute over conjunction when one of the conjuncts does not depend on the variable being quantified. Some examples are shown below:

$$\begin{aligned} \exists_{w,x,y,z} A(w,x) \cdot B(y,z) &\equiv (\exists_{w,x} A(w,x)) \cdot (\exists_{y,z} B(y,z)) \\ \exists_{x,y,z} A(x,y,z) \cdot B(x,y) &\equiv \exists_{x,y} (B(x,y) \cdot (\exists_z A(x,y,z))) \end{aligned}$$

The technique of distributing the quantification over conjunction appropriately is popularly known as *early variable quantification*. In some cases there may be more than one way to perform the computation as shown in the following examples:

$$\begin{aligned} &\exists_{w,x,y,z} T_1(w,x,y) \cdot T_2(y,z) \cdot A(w,x,y,z) \\ &\qquad \equiv \exists_{w,x,y} T_1(w,x,y) \cdot (\exists_z T_2(y,z) \cdot A(w,x,y,z)) \\ &\qquad \equiv \exists_{y,z} T_2(y,z) \cdot (\exists_{w,x} T_1(w,x,y) \cdot A(w,x,y,z)) \\ &\exists_{w,x,y,z} T_1(u,v,w,x) \cdot T_2(v,x,y,z) \cdot A(w,x,y,z) \\ &\qquad \equiv \exists_{w,x} T_1(u,v,w,x) \cdot (\exists_{y,z} T_2(v,x,y,z) \cdot A(w,x,y,z)) \\ &\qquad \equiv \exists_{y,z} T_2(v,x,y,z) \cdot (\exists_{w,x} T_1(u,v,w,x) \cdot A(w,x,y,z)) \end{aligned}$$

In both the examples above, the existential quantification can be done in two ways corresponding to two different permutations of T_1 and T_2. For a problem with n components, there are $n!$ permutations possible. The complexity of a computation of any of

these permutations will depend on the amount of simplification performed during the process.

We can apply this technique to Equation 7.10 by moving transition relations out of the scope of the existential quantification if they do not depend on any of the variables being quantified. For a given ordering of transition relations this equation can be rewritten as,

$$\begin{aligned} Image(A(\vec{x})) \;=\; & \exists\, \vec{x}_k, \vec{u}_k \; (\, T_k(\vec{x}, \vec{u}, \vec{y}_k) \wedge (\exists \vec{x}_{k-1}, \vec{u}_{k-1} \; T_{k-1}(\vec{x}, \vec{u}, \vec{y_{k-1}}) \wedge \\ & \cdots \wedge (\exists \vec{x}_1, \vec{u}_1 \; T_1(\vec{x}, \vec{u}, \vec{y}_1) \wedge A(\vec{x}))) \,) \end{aligned} \tag{7.11}$$

7.3.1 Previous Work

Touati [TSL+90] computes the image of a set of states by exploiting the property of the generalized cofactor in converting the image computation into range computation given by

$$\exists \vec{x}, \; \vec{u} \left[\prod_{i=1}^{i=k} T_{iA(\vec{x})}(\vec{x}, \vec{u}, \vec{y}_i) \right]$$

where $T_{iA(\vec{x})}$ denotes the generalized cofactor of $T_i(\vec{x}, \vec{u}, y_i)$ with respect to $A(\vec{x})$. This range computation is performed using a balanced binary tree – leaves correspond to terms and variables at nodes of the tree that do not appear in the support of nodes elsewhere are existentially quantified. Burch [BCL91a] criticized this approach on the grounds that generalized co-factor may introduce new variables in the supports of the terms and delay the ability to quantify out variables. Heuristically, this would lead to larger BDD size of the intermediate product terms. Note that if $T_i(\vec{x}, \vec{u}, \vec{y}_i)$ is conjoined with the product term obtained so far, it introduces $|\vec{y}_i|$ new variables (the corresponding next-state variables). We argue that the number of the variables that are existentially quantified from the product term and the number of variables that are introduced in the product term determine the computational efficiency of this operation. Thus the space requirement and the efficiency of image and pre-image computations become dependent on the order in which these clusters are processed. In [BCL91a], an ordering scheme of the partitioned transition relation is proposed, based on the semantics of the underlying model. However, this requires detailed understanding of the semantics of

the model and hence is not easily automated. [CC93] introduces a new "exist" generalized cofactor which allows for distribution of conjunction and quantification. Their technique for the early quantification problem is quadratic in the number of relations, and becomes impractical when the number of relations is large. [GB94] give a simple automated way to order the relations when each relation consists of the next-state function of a single latch. The primary criterion used is to choose the relation next in ordering for which the maximum number of variables can be quantified out from the new product (unique variables belonging to that partition). In case of a tie, the relation with the maximum support is chosen.

Since, in our approach, clusters do not necessarily consist of a single latch, the ordering criteria should also take into account the number of next-state variables introduced, while choosing the next cluster in the order. It was found that the maximum depth in the BDD ordering of any variable in a partition, referred to as the *index* of the variable, also affects the performance. The reasoning behind this is that existentially quantifying a variable from a function becomes computationally less expensive as the depth of the variable in the ordering increases.

7.3.2 Our Heuristic

In our heuristic, four different factors were used to decide the order of the partitions. We maintain two sets of clusters P and Q. The set P denotes the set of clusters which have already been ordered and the set Q contains the clusters which are not yet ordered. Initially, P is an empty set and set Q contains all the clusters. In the following expressions, PS, PI and NS denote the set of present state, primary input, and next-state variables, respectively, a variable is denoted by v, $S(T)$ represents the set of support variables of T and $\| A \|$ denotes the cardinality of the set A. For each cluster C_i in the set Q, we compute the following parameters:

1. $v_{C_i} = \| \{ v \mid (v \in S(T_{C_i})) \wedge (v \in PS \cup PI) \wedge (v \notin S(T_{C_j})\ C_j \neq C_i, C_j \in Q) \} \|$, i.e. the number of variables which can be existentially quantified when T_{C_i} is multiplied in the product.

2. $w_{C_i} = \| \{ v \mid (v \in PS \cup PI) \wedge (v \in S(T_{C_i})) \} \|$, i.e. the number of present state and primary input variables in the support T_{C_i}.

3. $x_{C_i} = \| \{ v \mid (v \in PS \cup PI) \wedge (v \in S(T_{C_j}), C_j \in Q) \} \|$, i.e. the number of present state and primary input variables which have not yet been quantified.

4. $y_{C_i} = \| \{ v \mid (v \in S(T_{C_i})) \wedge (v \in NS) \} \|$, i.e. the number of next-state variables that would be introduced in the product by multiplying T_{C_i}.

5. $z_{C_i} = \| \{ v \mid (v \in NS) \wedge (v \in S(T_{C_j}), C_j \in Q) \} \|$, i.e. the number of next state variables not yet introduced in the product.

6. $m_{C_i} = \max\{\text{index}(v), v \in S(T_{C_i}) \wedge v \in (PI \cup PS) \}$, i.e., the maximum BDD index of all variable to be quantified in the support of T_{C_i}.

7. $M_{C_i} = \max\{m_{C_j}, C_j \in Q \}$, i.e. the maximum BDD index of a variable to be quantified out in the remaining clusters.

In order to normalize the effect of parameters 1, 2, 5, and 6, we form the following ratios.

1. $R^1_{C_i} = (v_{C_i}/w_{C_i})$.

2. $R^2_{C_i} = (w_{C_i}/x_{C_i})$.

3. $R^3_{C_i} = (y_{C_i}/z_{C_i})$.

4. $R^4_{C_i} = (m_{C_i}/M_{C_i})$.

The cost function is defined as a weighted sum of these four ratios. The order of the clusters is obtained by greedily choosing the cluster with the best cost function value at each step. The chosen cluster is moved from set Q to set P and the process is repeated until all the clusters are ordered (set Q becomes empty). The weights used can be interactively varied. We performed a series of experiments to find a good combination of these weights. The above algorithm has a straightforward implementation with $O(k^2 \cdot n)$ complexity, where k is number of clusters and n is number of latches. This complexity can be reduced to $O(k \cdot n)$ with appropriate book-keeping. Note that the cost of finding an optimal ordering is paid only once. The same ordering can be used for successive image and pre-image computations inside some fixed-point computation.

7.4 Network Partitioning

The analysis in Sections 7.2 and 7.3 assume as a first step that the next-state functions of the finite-state machine have been obtained. For large designs, it is often not possible to build monolithic next state functions for latches in terms of primary inputs and present state variables. In these cases, either the BDDs grow too large to be built or their manipulation becomes extremely inefficient.

To contain the BDD size while building the next-state function, we adopt a similar strategy as in building the clusters of next-state relations. We build the next-state functions by composing appropriate BDDs for the nodes in the network. After each composition, we monitor the size of the BDDs and if the size of the BDD for that node becomes larger than a threshold, we introduce an intermediate variable. The algorithm is shown in Figure 7.3. The resulting structural partition is shown in Figure 7.4. A related technique is also presented in [JNC+96]. The idea of controlling BDD sizes by introducing variables has appeared in literature before [CM90a, CCQ94].

```
CreateNetworkPartition(N) {
    nodeList = network nodes sorted in topological order;
    foreach node in nodeList {
        build function f for the node by composing the fan-ins;
        if BDD size of f > threshold {
            instantiate a new BDD variable and assign it to the node;
        }
    }
}
```

Figure 7.3 Algorithm for creating partitioned representation of the network.

After building the next-state functions of all the latches as described in Figure 7.3, we assemble the next-state relations for all the latches as well as the relations for the intermediate variables. This whole set of relations is then appropriately combined to form clusters which are ordered as described in Section 7.3. For clustering and ordering

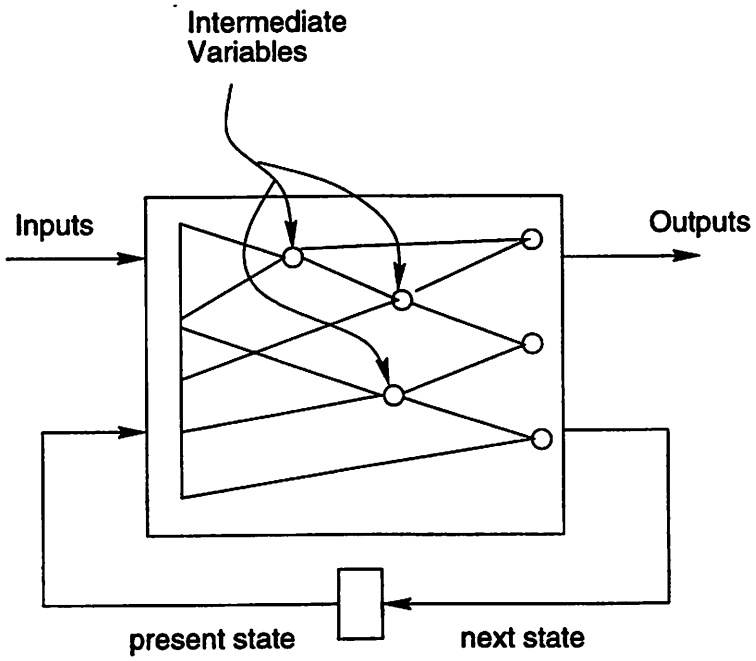


Figure 7.4 Using intermediate variables to represent transition relation.

purposes, we treat the intermediate variables as primary inputs.

In the following theorem, we establish that even for the partitioned network Equation 7.11 correctly computes the image of a set of states.

Theorem 2 *Suppose, $z_1, z_2, \ldots, z_k$ are the intermediate variables introduced during building of next-state functions. Suppose $\hat{T}_j(\vec{x},\vec{u},\vec{z}) = \overline{f^{z_j} \oplus z_j}$ represents the transition relation corresponding to the j^{th} intermediate variable and $\tilde{T}_i(\vec{x},\vec{u},\vec{z},y_i) = \overline{f^{y_i} \oplus y_i}$ represents the transition relation corresponding to the i^{th} state variable. Further, suppose $T_i(\vec{x},\vec{u},y_i)$ represents the partitioned transition relation for the i^{th} state variable in the finite state machine. Then,*

$$\exists_{\vec{u},\vec{x}} \left(\prod_i T_i(\vec{x},\vec{u},y_i) A(\vec{x}) \right) = \exists_{\vec{u},\vec{x},\vec{z}} \left[\left(\prod_j \hat{T}_j(\vec{x},\vec{u},\vec{z}) \prod_i \tilde{T}_i(\vec{x},\vec{u},\vec{z},y_i) \right) A(x) \right]$$

Proof:

$$\exists_{\vec{u},\vec{x},\vec{z}} \left[\left(\prod_j \hat{T}_j(\vec{x},\vec{u},\vec{z}) \prod_i \tilde{T}_i(\vec{x},\vec{u},\vec{z},y_i) \right) A(x) \right] =$$
$$\exists_{\vec{u},\vec{x}} \left[\exists_{\vec{z}} \left(\prod_j \hat{T}_j(\vec{x},\vec{u},\vec{z}) \prod_i \tilde{T}_i(\vec{x},\vec{u},y_i) \right) A(x) \right]$$

For the result of the image computation to be equal we need to show that

$$\exists_{\vec{z}} \left(\prod_j \hat{T}_j(\vec{x},\vec{u},\vec{z}) \prod_i \tilde{T}_i(\vec{x},\vec{u},y_i) \right) = \prod_i T_i(\vec{x},\vec{u},y_i)$$

This follows from the composition rule $f(\vec{x})|_{x_i=g} = \exists_{x_i}(f(\vec{x})(\overline{x_i \oplus g}))$. ■

7.5 BDD Minimization Using Don't Cares

At various steps of the verification algorithm, we often encounter flexibility in representing a set of states or the transition relation. In particular, we might have a choice of adding or removing elements of a set S, between a given lower bound and an upper bound. This can be expressed by the expression $S_L \subseteq S \subseteq S_U$, where S_L and S_U are the given lower and upper bounds respectively. Any S which respects these bounds will be acceptable. Similarly, we might have a choice of adding or deleting a transition in

the transition relation, i.e., given $T_L(\vec{x},\vec{u},\vec{y})$ and $T_U(\vec{x},\vec{u},\vec{y})$, any $T(\vec{x},\vec{u},\vec{y})$ satisfying the condition,

$$T_L(\vec{x},\vec{u},\vec{y}) \subseteq T(\vec{x},\vec{u},\vec{y}) \subseteq T_U(\vec{x},\vec{u},\vec{y})$$

will be acceptable for the computation at that step.

These flexibilities can be alternatively represented as a pair (f,c), where f is the function and c is the care-set of minterms, i.e., those points in Boolean space where we care about the value of f. Essentially, we have the freedom to choose any function g such that for every minterm in c, g matches the value of f. All other minterms (those in $\bar{c}$) belong to what is called the *don't-care set*. On those minterms, we can pick any value of g. For instance, the flexibility to represent a set given as $S_L \subseteq S \subseteq S_U$, can alternatively be represented as $(S_L, S_L + \bar{S}_U)$.

Since the size of a BDD representing a set is not directly correlated to the number of elements in the set, the presence of don't care minterms leads to a BDD optimization problem. This is illustrated in Figure 7.5. Figure 7.5(a) shows the original function. The don't care minterm is represented as terminal labeled "DC". By choosing a value of 1, we get the BDD shown in Figure 7.5(b) and the BDD node count remains unchanged. However, by choosing a value of 0 for the don't care minterm, we get the BDD in Figure 7.5(c), and the BDD size reduces by 1.

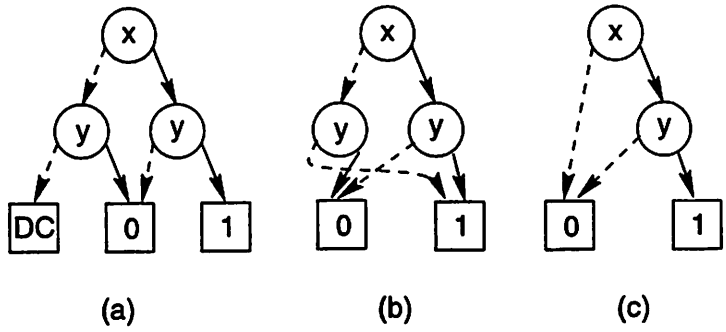


Figure 7.5 BDD optimization using don't care minterms.

The problem of finding the optimal assignment to the don't care points is NP-complete [TY93]. A lot of research work has gone into simplifying the BDD rep-

resentation of a function f with respect to a care-set c [TSL+90, CM90b, SHSB94, HBBM97]. By reducing the BDD sizes via usage of don't cares, the computation time of various BDD operations improves.

Here we outline various scenarios where the don't cares arise in BDD-based formal verification.

1. Model checking: In general, model checking algorithm consists of multiple fixed-point computations. These computations in turn make use of image/pre-image operations. If a reachability operation is performed on a system a priori, the unreachable states could be used as don't cares to simplify the operands for image/pre-image operations. For instance, the least fixed point computation for verifying the CTL formula $EF(p)$.† Suppose, P represents the set of states where proposition p is true and R represents set of reachable states, $EF(p)$ can start with the set of states S, such that $P \cdot R \subseteq S \subseteq P$.

2. Reachability: Consider the following fixed point computation for finding the set of reached states:

$$\begin{aligned} R_0(\vec{x}) &= \text{Init}(\vec{x}) \\ R_{k+1}(\vec{y}) &= R_k(\vec{y}) \cup \exists \vec{x},\vec{u}\,[T(\vec{x},\vec{u},\vec{y}) \wedge R_k(\vec{x})] \end{aligned} \tag{7.12}$$

   Observe that in Equation 7.12, any set of states between $R_k \setminus R_{k-1}$ and R_k can be used in place of R_k while preserving the result for R_{k+1} [CM90b]. Thus R_{k-1} can be used as a don't care set to minimize the BDD size of the states of which the image is taken.

3. Simplification of the transition relation during image computation. Consider Equation 7.11 which can be rewritten as

$$\mathit{Image}(A(\vec{x})) = \exists \vec{x},\vec{u}\,[A(\vec{x}) \wedge T_1(\vec{x},\vec{u},\vec{y}_1) \wedge T_2(\vec{x},\vec{u},\vec{y}_2) \wedge \ldots \wedge T_k(\vec{x},\vec{u},\vec{y}_k)]$$

   Note that each T_i is a relation over the entire Boolean space of $\vec{x}$. We can make use of the following fact:

$$\exists \vec{x}(T(\vec{x},\vec{u},\vec{y})A(\vec{x})) = \exists \vec{x}\ \left(T(\vec{x},\vec{u},\vec{y})|_{A(\vec{x})}A(\vec{x})\right) \tag{7.13}$$

†For a detailed description on the syntax and semantics of CTL, please refer to [CES86].

where $T(\vec{x},\vec{u},\vec{y})|_{A(\vec{x})}$ denotes the *generalized cofactor* of $T(\vec{x},\vec{u},\vec{y})$ with respect to $A(\vec{x})$. The generalized cofactor of a function f with respect to another function g results in an incompletely specified function $h = f|_g$, for which the onset, offset and don't care sets are given by $f \cdot g$, $\bar{f} \cdot g$, and $\bar{g}$ respectively. For our purposes, we will overload the "$|$" operator and use $f|_g$ to indicate the minimization of BDD representation of the function f with respect to the don't care set $\bar{g}$. The "constraint" method (proposed in [CM90b, TSL+90]) is "image" preserving, i.e., if "constraint" is employed to simplify the BDD of $T(\vec{x},\vec{u},\vec{y})$ in Equation 7.13 and the simplified BDD is given as $\tilde{T}(\vec{x},\vec{u},\vec{y})$, then Equation 7.13 simplifies to the following:

$$\exists \vec{x}(T(\vec{x},\vec{u},\vec{y})A(\vec{x})) \;=\; \exists \vec{x}\left(\tilde{T}(\vec{x},\vec{u},\vec{y})\right)$$

Note that, there is no need to multiply the set $A(\vec{x})$ with the simplified transition relation. However, BDD simplification my introduce new variables in the support of the function being simplified. The set of new variables will depend on $A(\vec{x})$.

For a transition relation vector with k components, the corresponding simplification is given as:

$$\exists \vec{x},\vec{u}\,[A(\vec{x}) \wedge T_1(\vec{x},\vec{u},\vec{y}_1) \wedge T_2(\vec{x},\vec{u},\vec{y}_2) \wedge \ldots \wedge T_k(\vec{x},\vec{u},\vec{y}_k)] \;=$$

$$\exists \vec{x},\vec{u}\,\left[A(\vec{x}) \wedge T_1(\vec{x},\vec{u},\vec{y}_1)|_{A(\vec{x})} \wedge T_2(\vec{x},\vec{u},\vec{y}_2)|_{A(\vec{x})} \wedge \ldots \wedge T_k(\vec{x},\vec{u},\vec{y}_k)|_{A(\vec{x})}\right]$$

If we apply the "constraint" method to simplify the individual transition relations, we would not need to multiply $A(\vec{x})$ before performing the quantification. However, since our cluster ordering technique described in Section 7.3 makes use of support variables of each cluster and the "constraint" method can potentially introduce new variables in the supports of relations, we cannot have a static schedule of early variable quantification. That means, we would need to pay the price of finding the obtaining the optimal schedule for each different value of $A(\vec{x})$. This is found to be computationally expensive. Hence, in our approach we make use of another BDD minimization technique – "restrict" (proposed in [CM90b]). This method does not introduces any new variables. The overall motivation is

that co-factoring and simplifying each transition relation will result in simpler BDDs and hence faster manipulation.

4. Simplification of the transition relation with respect to range care set: Sometimes during an image (pre-image) computation, a care set is specified in the co-domain (domain) which can be used to further simplify the transition relation. Suppose we would like to compute the image of $A(\vec{x})$ and the range care set is specified as $B(\vec{y})$. We can perform the following simplification of transition relations:

$$\exists \vec{x},\vec{u} \left(\prod_i T_i(\vec{x},\vec{u},\vec{y}_i)\ A(\vec{x}) \right) =$$
$$\exists \vec{x},\vec{u} \left(\prod_i T_i(\vec{x},\vec{u},\vec{y}_i)|_{A(\vec{x})B(\vec{y})}\ A(\vec{x}) \right) \quad (7.14)$$

5. Simplification of image/pre-image with respect to range/domain care set: After obtaining the image (pre-image), the range (domain) care set can be used to simplify it even further in the following way. Suppose, $C(\vec{y})$ is the image of $A(\vec{x})$ with the range care set $B(\vec{y})$ as obtained in Equation 7.14. We can further simplify $C(\vec{y})$ as $C(\vec{y})|_{B(\vec{y})}$.

6. Simplification of the transition relation using approximate reachability analysis: Conservative approximations to the unreached states also yield don't cares. Given a set of clusters $\{C_1, C_2, \ldots, C_k\}$, we can compute an upper bound on the projection of reachable states in the product space to a component C_i. Assignments to the latches in component C_i not corresponding to the above states can never be attained in any environment.

In Section 7.7.4, we present results indicating the performance improvement achieved via usage of don't cares in minimizing BDDs.

7.6 Removing Redundant Latches

To a first approximation, the BDD sizes of transition relations and state sets depend on the number of variables. The basic motivation behind removing redundant latches is to simplify BDDs for transition relations and reached state sets by removing variables.

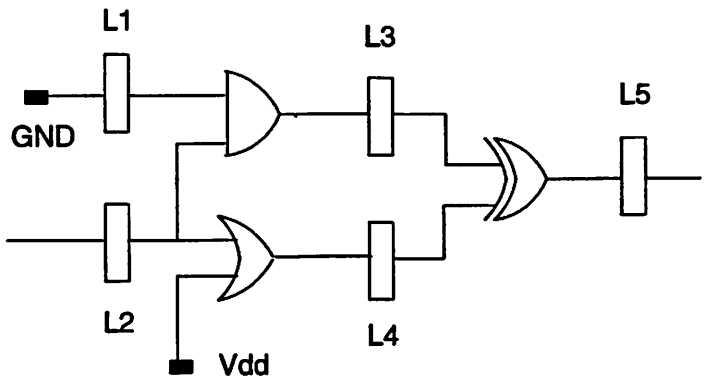


Figure 7.6 Propagation of constants through latches: L_1, L_3, L_4 and L_5 are redundant.

A latch is *redundant* if it can be replaced by a wire without changing the functionality of the circuit. Replacing a latch by a wire, reduces the number of BDD variables by two and, heuristically speaking, would also reduce the size of the BDDs that depend on these variables.

We describe two methods of finding redundant latches and removing them.

7.6.1 Constant Propagation

Sometimes latch inputs are tied to either *VDD* or *GND*. In our algorithm we detect such latches and propagate their constant values to their fan-outs (hence the term "constant propagation"). An example is shown in Figure 7.6.

A recursive algorithm for removing redundant latches is the following:

1. The input to the algorithm is the next-state functions of the latches.
2. Each latch has two flags – value ("constant" or "variable") and status ("processed" or "unprocessed").
3. Mark all latches as "unprocessed".
4. While there exists unprocessed latches, pick an "unprocessed" latch L.
5. Call the function *find_redundant*(L).

6. return

find_redundant(L):

1. If L is processed, then return its value ("constant" or "variable").
2. Mark L as "processed"
3. If L is tied to *VDD* or *GND* assign value "constant" and return.
4. Assign value "variable" to L.
5. For each variable v in the support of the next-state function of the L,
   (a) If v corresponds to a primary input, mark L as "variable" and return.
   (b) If v represents a wire with constant value "0" or "1", continue.
   (c) Find latch L_i for which v is the present state variable, and call *find_redundant*(L_i).
6. Modify the next-state function of L by propagating the constant value of fan-ins.
7. If the next-state function becomes a constant, change the value of L to "constant".
8. return value.

At the end of the algorithm, the next-state functions of latches do not contain present-state variables corresponding to redundant latches. These variables are not considered for further BDD manipulations.

In a related work Beer *et al.* [BBDG+94] also mentioned a "constant-elimination" technique to reduce the number of inputs and memory elements.

7.6.2 Latch Removal by Retiming

Retiming rearranges the storage elements in a circuit to reduce its cycle time or to reduce the number of storage elements, without changing its functionality. We use retiming to reduce the number of storage elements. A simple example to demonstrate this is shown in Note that, the inputs of these two latches are fed by the same combinational logic. Hence the next-state values of these latches will always be the same. The new

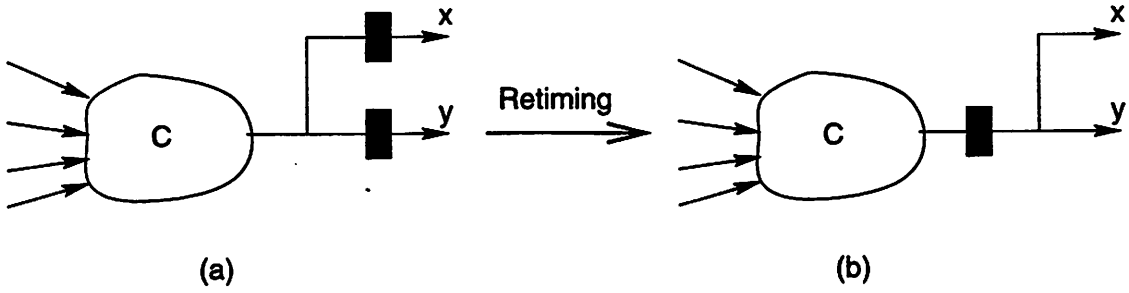


Figure 7.7 Removing latches by retiming.

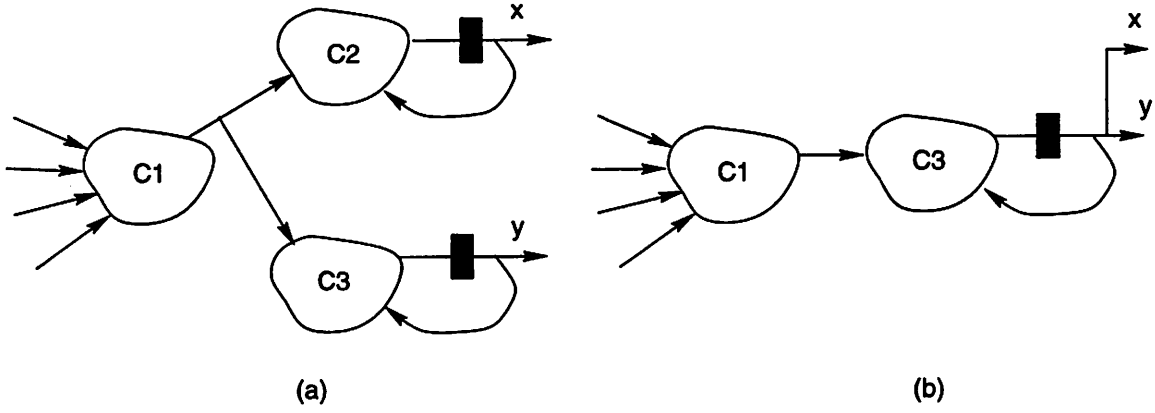


Figure 7.8 Removing latch by retiming, a more general case.

circuit with the redundant latch removed is presented in Figure 7.7(b). The example in Figure 7.7 is a special case of one in Figure 7.8. In the more general case, latches have feedback paths. However, if the combinational logic blocks feeding to the latches have identical functionality (if $C_2 \equiv C_3$ in the Figure 7.8), we can substitute one of the latches by wire.

Figure 7.7(a).

The following algorithm detects such general cases and removes redundant latches:

1. Sort all latches in increasing order of the support size of the corresponding next-state functions.

2. For each pair of latches L_i and L_j, with equal support size do the following:

   (a) Suppose x_i, x_j denote the corresponding present-state variables (outputs of latches) and F_i, F_j are the corresponding next-state functions.

   (b) Find the co-factors F_{ix_i}, $F_{i\bar{x}_i}$, F_{jx_j}, and $F_{j\bar{x}_j}$.

   (c) If $F_{ix_i} \equiv F_{jx_j}$ and $F_{i\bar{x}_i} \equiv F_{j\bar{x}_j}$ then remove L_j from the circuit and replace it by a "wire" instead (as illustrated in Figure 7.8).

The correctness of this algorithm is explained below:
Suppose F and G is a pair of functions such that all the support variables except one are common to them. For example, $F(x_1, x_2, \ldots, x_n, y)$ and $G(x_1, x_2, \ldots, x_n, z)$ will be such a pair. Now from Shannon decomposition,

$$\begin{aligned} F &= yF_y + \bar{y}F_{\bar{y}} \\ G &= zG_z + \bar{z}G_{\bar{z}} \end{aligned}$$

$F_y, F_{\bar{y}}, G_z$, and $G_{\bar{z}}$ have common support variables. Now, if $F_y = G_z$ and $F_{\bar{y}} = G_{\bar{z}}$, then except for the variable labeling, F and G are computing the same function. Hence, if we substitute one variable (say y) by another variable (say z), we can replace F by G. As shown in Figure 7.8, we are able to share the logic if $C_2 \equiv C_3$.

A similar approach was proposed in [Lin91] who described an algorithm to remove a maximal set of state variables without affecting the uniqueness of the reachable states. The problem with that approach is that the set of reachable states has to be pre-computed. In many big designs computing the reachable states becomes infeasible due to the size of the BDD. In our technique, redundant latches are removed once the next-state functions are calculated. Hence the size of the reached state set is reduced before we need to compute it.

It is interesting to see that our approach is orthogonal to Lin's. After minimizing the transition relation using this approach, we can still apply Lin's method to possibly remove more latches and get a further reduction in BDD size after computing the set of reachable states.

Notice that retiming, in its more general form, can reduce the number of latches significantly while preserving the I/O functionality of the circuit. However, since in

this case, latches can change locations arbitrarily, the functionality of a latch may not be preserved. By performing only simple retiming transformations, as shown in Figures 7.7 and 7.8, we preserve the functionality of each "latch".

7.7 Experimental Results

To illustrate the effectiveness of these algorithms and to compare them with some previous approaches, we performed experiments using sequential ISCAS/MCNC benchmarks and some industrial examples. Important characteristics of some of these examples along with a brief description are presented in Table 7.1. We perform reachability analysis and model checking on these examples to demonstrate the effectiveness of various techniques proposed in this chapter.

Unless otherwise noted the experiments were performed on a DEC5900/260 workstation with 440 MBytes memory and a limit of 10000 seconds on CPU time and 400 MBytes on data size were used while running the experiments.

Example	# Latches	# Gates	Description
sbc	28	927	ISCAS'89 sequential benchmark (a snooping bus controller).
Gigamax	45	994	Cache coherency protocol description for hardware implementation of Gigamax distributed multiprocessor [McM93].
BDLC*	144	4775	Abstracted Byte Data Link Controller (BDLC); Manages the transmit-receive protocol between microprocessor and a serial bus. Contains the abstract description of BIT module. Part of a commercial chip.
BDLC	172	6639	Unabstracted version of the previous example.
2MDLC	83	2596	Two BIT modules interacting via a serial bus using BDLC protocol.
BIU	154	3018	Abstracted version of a Bus Interface Unit from a commercial microprocessor.
Every	63	838	Cache flush controller module of a commercial microprocessor.

Table 7.1 Description of industrial examples.

7.7.1 Clustering

Table 7.2 shows our results on clustering by BDD size. We make the following observations. Setting higher limits obviously leads to fewer clusters but the total number of

BDD nodes taken by the clusters increases. As shown in Equation 7.9, image computation is performed by taking the product of transition relations of clusters sequentially (we will refer to them as sequential iterations). The time taken in forming this product is a function of the number of clusters as well as the cluster sizes. This results in total CPU time being a convex function of partition-size limit. This can be reasoned as follows.

Using a threshold limit of 1, results in a procedure which uses the least amount of space but results in maximum number of clusters (equal to the number of latches in the system) implying maximum number of sequential iterations. As the threshold is raised, the number of iterations is reduced, while BDD sizes of the operands increase, potentially leading to greater computational complexity. In the beginning, the effect of reduction in the number of iterations dominates over the effect of increasing BDD sizes. As a result, initially run time is reduced as the cluster size is increased. Later, the effect of increasing BDD sizes (greater computational complexity) dominates the savings due to decreasing number of iterations and we observe an increase in runtime. This is true for all the examples, except ones for which the monolithic transition relation is not very big (e.g. 2MDLC).

Partition Size Limit	Examples											
	2MDLC (L=83)			BDLC* (L=144)			BDLC (L=172)			BIU (L=154)		
	N	|T|	Time	N	|T|	Time	N	|T|	Time	N	|T|	Time
1	83	2744	728	144	5114	432	172	7042	9248	154	3943	564
100	16	2943	348	49	8454	235	57	13259	3905	48	9950	430
1000	5	3434	203	14	19613	125	20	24966	2014	18	30511	266
2000	3	3238	171	11	27762	115	14	34774	1662	15	35746	245
5000	2	6612	167	8	44033	106	8	46994	1443	10	76563	228
10000	1	6853	142	6	56410	106	6	61704	1243	9	180914	227
20000	–	–	–	5	88984	116	5	90179	1121	7	217395	171
30000	–	–	–	4	111867	129	4	99020	1185	7	284450	198

N: Number of partitions

L,|T|, Time

Table 7.2 Results on space-time trade off in clustering by the BDD size approach.

7.7.2 Cluster Ordering

Table 7.3 compares the performance (CPU time in seconds) of our ordering heuristic with the heuristics proposed in [GB94, TSL+90]. Specifically we report the time taken in the reached state computation. The weights chosen after some experimentation in our heuristic were $W_1 = 2, W_2 = 1, W_3 = 1, W_4 = 1$.

	Various Heuristics		
Example	**[TSL+90]**	**[GB94]**	**Proposed**
BIU	305	326	315
Every	6087	5857	5788
2MDLC	176	244	179
BDLC*	140	191	144
BDLC	*space out*	3023	2231
Gigamax	4.8	7.4	4.8
sbc	116	135	118

Table 7.3 Comparison of CPU time (in seconds) for different cluster ordering heuristics.

The above results indicate that the proposed approach always outperforms that in [GB94]. Improvements up to 25% were achieved.

Although in some examples (BIU, BDLC*, 2MDLC, sbc) Touati's heuristic [TSL+90] performs marginally better than ours, on BDLC, Touati's approach ran out of memory.

7.7.3 Network Partitioning

In Table 7.4, we present results on network partitioning based on BDD-size heuristic. We make following observations from the data presented in the Table 7.4.

1. The number of partitions monotonically decreases with increase in threshold value. This is obvious, since increasing threshold value allows us to compose larger BDDs before creating a partition.

2. The total number of BDD nodes in all partitions increases with the threshold value. This is because at lower threshold values, intermediate variables are cre-

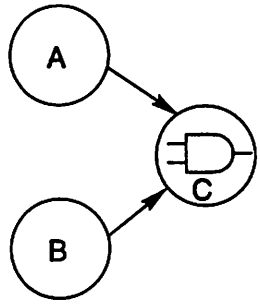


Figure 7.9 Illustration of effect of partition threshold on the overall BDD size.

ated more often which keeps the BDD size in control. However, in some cases, the shared size of the BDDs might reduce with increased threshold value because of the simplification resulting due to reconvergent signals. This is illustrated in Figure 7.9. Consider the following scenario for the Figure 7.9.

(a) Two different threshold values: t_L and t_H, with $t_L < t_H$.

(b) The BDD sizes for nodes A and B is greater than t_L, but less than t_H.

(c) Due to reconvergent fanout the AND of signals A and B consists of a single minterm. The number of BDD nodes required to represent this minterm is linear in the number of inputs.

(d) Since the nodes A and B do not fanout to any other node in the network, BDDs for nodes A and C can be freed after building the BDD for node C.

For the smaller threshold value, t_L, we create two new variables a and b for network nodes A and B, respectively. The correlation between the BDDs for A and B is lost when we compute the BDD for node C. As a result, even though the BDD size for C is small (2 BDD nodes), the total shared size which combines the BDD sizes for A and B is large.

For the larger threshold value, t_H, the composition of the BDDs for A and B and later on freeing them results in overall smaller shared BDD size.

3. With minor anomaly, the time required to build BDDs for the partitions increases monotonically with increasing in threshold. This is because as the threshold value is increased, larger BDDs are composed to compute the functionality of each node in the network, leading to increased computation time.

7.7.4 Usage of Don't Cares

In Table 7.5, we give results on computational and memory performance improvement achieved by using don't care during reachability analysis. A performance improvement of a factor from 2 to 15 and a memory usage improvement of up to factor of 2 is observed.
In Table 7.6, we give results on usage of don't cares during model checking. We make following observations:

1. Without any don't care usage (Column D), we were unable to complete model checking on 6 out of 11 examples (exceeded time limit of 1000 CPU seconds).

2. With simplification of pre-image using unreachable states as don't cares (Column C) during model checking we could complete one more example (8-arbit).

3. The simplification of transition relation using don't cares allows us to complete all the examples (Column B).

4. Comparing the CPU times we observe that using don't cares appropriately can enable us to achieve performance improvement of up to 100 and more.

5. Using don't cares also improves the memory usage during model checking. For the cases where it was not possible to complete model checking in given time, we observed memory usage improvement by a factor of 20.

6. The primary source of don't cares during model checking is that derived from unreachable states. Computing this set of don't cares requires reachability analysis, thereby incurring some computational cost. In some cases, the performance achieved by the don't care set obtained via unreachable states cannot offset the

		Parameters		
	Threshold	**P**	**N**	**T**
8085	100	175	400110	10.49
	1000	40	469501	11.12
	5000	30	504546	14.62
	10000	13	509471	14.68
	50000	8	500738	15.07
C5315	100	93	25061	1.06
	1000	29	182724	3.82
	5000	19	949573	24.27
	10000	13	1979381	48.91
	50000	8	3589854	109.54
C2670	100	90	122942	2.44
	1000	44	1057659	53.03
	5000	32	1187959	57.54
	10000	22	1169287	56.66
	50000	–	–	t.o.
C3540	100	88	13580	0.46
	1000	59	192882	4.13
	5000	38	328742	8.58
	10000	50	2581451	54.91
	50000	39	8914776	315.67

Table 7.4 Partitioning of the network based on BDD size threshold.

P: Number of partitions

N: Total number of shared BDD nodes in all partitions

T: CPU time to build the partitions

	CPU Time (Secs)		Memory (MB)	
	A	B	A	B
ethernet412	14.9	4.9	5.8	4.9
biu	105.8	22.3	21.5	14.9
abs_bdlc	474.3	31.1	18.0	10.7
ethernet213	696.5	98.9	29.6	14.9
sbc	611.8	361.7	26.5	25.7
slider	595.2	257.4	52.5	33.5
every	1998.0	1140.8	28.0	13.9

Table 7.5 Performance improvement with don't care usage during reachability analysis.

A: Image computation for the set R_k.

B: Image computation for the simplified frontier set $(R_k|_{\overline{R_{k-1}}})$.

computational cost of reachability analysis. For example, in minMax30, we observe that model checking time is insignificant, but, reachability analysis takes ten times as much time and leads to worse performance.

7.7.5 Redundant Latches

The results of redundant latch removal techniques on various examples are shown in Table 7.7. We observe up to 30% reduction in the BDD size of the transition relation. Also, a reduction of up to 25% in the BDD size of the reached set was obtained.

In the above analysis, reset values of the latches were ignored, i.e., we did not check for consistency of the reset values. However, these optimization techniques can be applied even if the reset values of the latches are taken into account. In the constant propagation approach, the reset value of the latch must match the constant next-state value it takes, for it to be made redundant. In the retiming approach, the reset values of the latches must be identical for either of them to be removed. This analysis can be done very easily.

	CPU Time (Secs)				Memory Usage (MBytes)			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**
eisenberg	27.3	28.3	15.2	15.1	1.6	1.7	1.5	1.5
bakery	18.6	13.5	9.9	9.5	1.6	2.6	1.7	1.7
ethernet112	**t.o.**	**t.o.**	63.4	5.6	–	–	7.7	4.3
ethernet212	**t.o.**	**t.o.**	222.0	8.5	–	–	9.1	4.4
ethernet312	**t.o.**	**t.o.**	526.6	19.9	–	–	11.1	5.5
ethernet412	**t.o.**	**t.o.**	567.0	60.6	–	–	24.1	6.9
elevator	**t.o.**	**t.o.**	182.7	111.3	–	–	15.9	8.2
4-arbit	11.2	3.5	1.2	0.6	3.3	2.1	1.5	1.4
8-arbit	**t.o.**	5.3	5.3	5.0	–	4.3	4.3	4.3
abp	6.0	3.6	1.7	1.8	1.7	1.6	1.6	1.6
minMax30	2.4	19.7	19.9	21.7	8.5	22.0	22.0	22.1

Table 7.6 Performance improvement with don't care usage during model checking.

α: Using reachable state set for creating don't cares.

β: Simplification of the pre-image w.r.t. care set.

γ: Simplification of T_i during backward image computation w.r.t. care set.

A: No usage of don't care.

B: α

C: $\alpha+\beta$

D: $\alpha+\beta+\gamma$

t.o. : Time out after 1000 CPU seconds.

examples	A	B	C	D	E	F		G	
						$\|T\|$	$\|R\|$	$\|T\|$	$\|R\|$
gigamax	45	9	0	0	9	2018	402	1389	301
BDLC*	144	1	0	5	6	24275	12208	23441	9984
BIU	154	6	2	26	34	30834	25276	20088	20956

A: Total # of latches

B: # of constant latches removed without constant propagation

C: # of latches removed after constant propagation

D: # of latches removed by re-timing

E: Total # of latches removed

F: Redundant latches not removed

G: Redundant latches removed

$|T|$, $|R|$: Sizes of transition relation and reached states

Table 7.7 Effects of redundant latch removal on BDD sizes.

7.8 Summary

We have discussed a series of algorithms for efficient state-space traversal of a finite-state machine. We established that the core computation in BDD-based formal verification is that of forming the image and pre-image of a set of states under the transition relation characterizing the system. To make this step efficient, we addressed use of partitioned transition relations, use of clustering, network partitioning, use of don't cares, and removal of redundant latches. The efficacy of these algorithms was demonstrated on ISCAS/MCNC sequential circuits as well as some industrial designs. Almost all the algorithms described in this chapter have been implemented in the verification tool VIS [BSA+96a]. In particular, the image/pre-image computation techniques form the core engine for model checking. A brief description of the tool is given in Appendix B.

Other BDD-based techniques which look promising include the "exists–cofactor" of [CC93], and the "implicitly conjoined invariants" of [HYD94]. Certain limitations of BDD-based formal design verification cannot be solved by the techniques described in this work. For example, the BDD size of the reached set may be large under any

variable ordering. Other data structures might be useful in these cases. There is also a wide class of heuristics, orthogonal to the approaches we have taken, for coping with the state explosion problem; such as property-specific reductions [ASS+94], abstractions [Gra94], and conservative approximations to reached state sets [CHM+93].

Part III

Sequential Circuit Verification

Chapter 8

Retiming and Resynthesis: Complexity Issues

So far we have looked at the techniques which primarily target the functional representation and manipulation of the designs for efficient verification – high performance BDD manipulation for various Boolean operations, compact state transition graph representation, and efficient state-space traversal. An orthogonal set of techniques, called structure-based techniques, use structural information about the circuit and the nature of transformations performed on it to obtain efficient verification algorithms. It has been established that a combination of functional and structure-based techniques provides a robust methodology for combinational verification [KK97, Mat96, RWK95, Bra93]. For sequential circuits, however, attempts to combine structural and functional techniques have been limited to smaller size examples or relatively minor transformations [HCC96b, HCC97].

In the next two chapters we investigate the implementation verification problem for circuits which have undergone repeated retiming and combinational synthesis transformations. In this chapter, we attempt to formalize the notions of the optimization capability of retiming and resynthesis operations. Also, we formally establish the computational complexity of the corresponding implementation verification problems. Our goal is to benefit from these results in establishing practical retiming and resynthesis logic optimization and verification methodologies. In the next chapter we propose a practical algorithm for this implementation verification problem.

8.1 Introduction

In combinational synthesis [BRSW87, SSL+92], the positions of the latches are fixed and the logic is optimized. In retiming [LRS83, LS91], the latches are moved across fixed combinational gates. The effects of retiming are – changes in the number of latches (thereby leading to increase/decrease in area) and increase/decrease in the cycle

time (leading to slower/faster clock rate). A side effect of retiming is that it enables interaction between different combinational logic blocks. Hence retiming followed by combinational synthesis enables logic optimization which is not possible by combinational optimization alone. Combinational synthesis generates new possibilities for the latch locations perhaps leading to further optimization. A sequence of retiming and combinational resynthesis steps can provide powerful optimization of a sequential circuit. In [MSBS91], it has been shown that retiming combined with synthesis can be used to optimize sequential networks.

After the initial retiming algorithm proposed in [LS91] for a simple circuit containing single clock edge-triggered latches, many advancements have been made in terms of efficient implementation and applicability of retiming with more complex memory elements. In particular, techniques given in [SR94, MS97] can be applied to large sequential circuits. Retiming of level-sensitive latches was addressed in [SBS93a, LE92]. Recently, Legl *et al.* proposed retiming techniques for edge-triggered circuits with multiple clocks and load enables [LVW97]. They introduced the notion of a latch class $cl = (CLK, LE)$, which is all latches connected to clock signal CLK and load signal LE. The retiming problem for multiple-class sequential circuits was reduced to an equivalent retiming for single class sequential circuits, thereby exploiting performance enhancements made in that domain. Since most industrial designs contain latches with different load signals and multiple clocks, their technique further improves the applicability of retiming to such designs.

Even though both retiming and synthesis techniques have been around for over a decade, the optimization capability of a combination of these transformations and the corresponding verification complexity have not been formally established. This chapter addresses this issue.

The rest of the chapter is organized as follows: in Section 8.2, we establish the optimization potential of various combinations of retiming and synthesis transformations. In Section 8.3, we discuss simple extensions to traditional notions of combinational optimization and retiming which can improve their optimization capability without a significant increase in the algorithm complexity. The verification complexities of various combinations of transformations are discussed in Section 8.4. In Section 8.5, we summarize our results and discuss the open problems in this area. In this and the next

chapters, we will use the term "combinational optimization", "combinational synthesis", "synthesis", and "resynthesis" interchangeably.

8.2 Optimization Power

We attempt to characterize the optimization power of sequential synthesis using only retiming and combinational synthesis transformations. In that direction, we first start with basic transformation steps and gradually move towards general retiming-resynthesis transformations.

8.2.1 Synthesis

In synthesis, the latches in the circuit are untouched and the latch inputs and outputs are treated as circuit outputs and inputs respectively. Since I/O functionality is preserved during the optimization, the logic function feeding into each latch also remains the same. The combinational logic block can be optimized for area or delay of the circuit as shown in Figure 8.1 (taken from [Smi97]). The critical path is shown with bold lines.

8.2.2 Retiming

Retiming moves the latches across combinational blocks, leading to possible increase or decrease in the number of latches. The effects of retiming are the following.

1. Change in cycle time: Due to movement of latches, the delay along the combinational path between two latches can change. This might increase or decrease the clock rate at which the circuit can function correctly (the delays through combinational gates along any path should be less than the cycle time). The reduction in cycle time is shown in Figure 8.2(b).

2. Change in area: The increase/decrease in the number of latches leads to the increase/decrease in the circuit area. An example of area optimization using retiming is shown in Figure 8.2(c).

   However, reducing the number of latches arbitrarily can increase the maximum allowed clock period, leading to reduced clock speeds. Hence the typical use of retiming has been for constrained area optimization. In this case, the area is

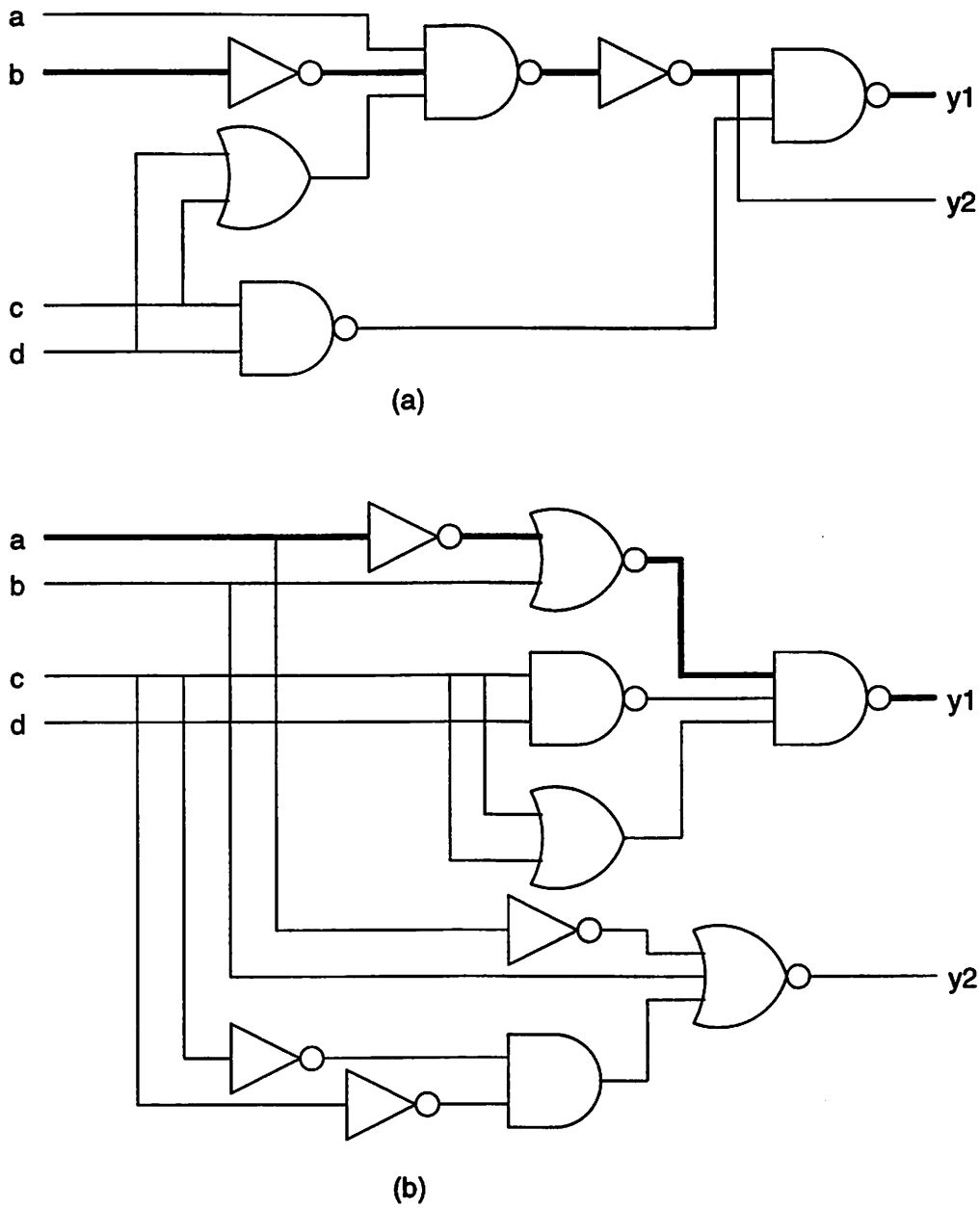


Figure 8.1 Combinational optimization: area vs. delay trade-off – (a) circuit with minimum area (b) circuit with minimum delay. The delay is measured in terms of levels of logic.

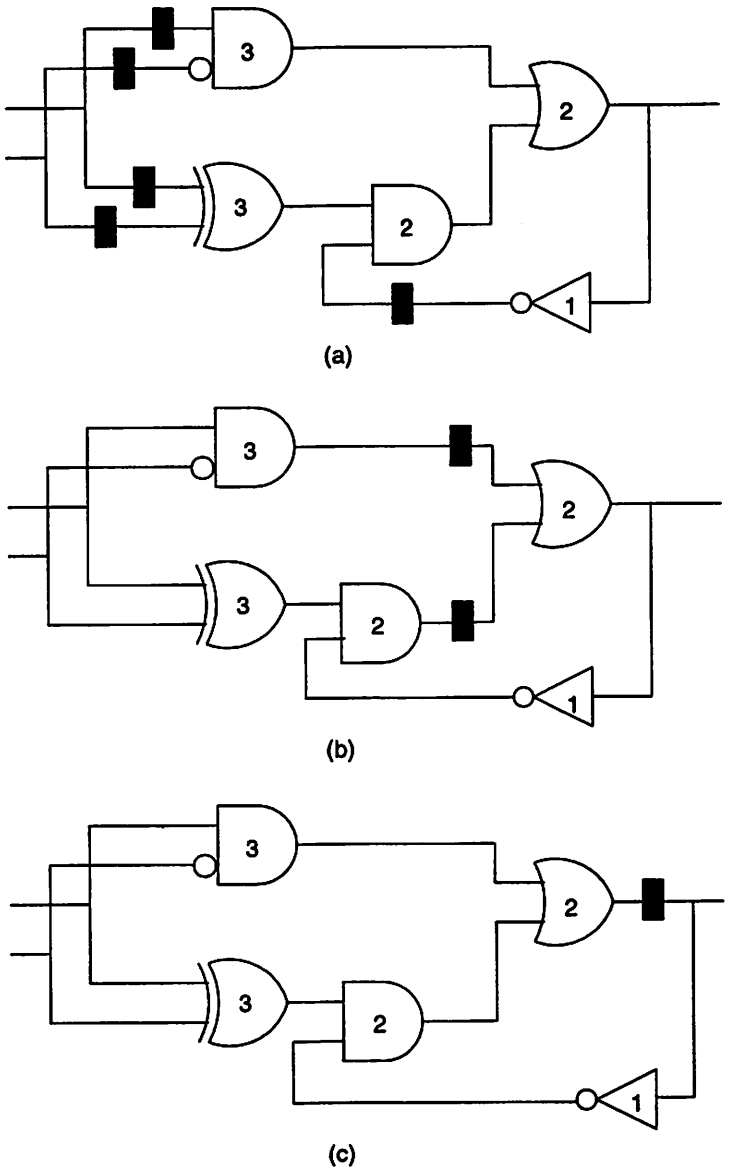


Figure 8.2 Retiming: area vs cycle time trade-off – (a) original circuit (b) circuit with minimum cycle time (c) circuit with minimum area.

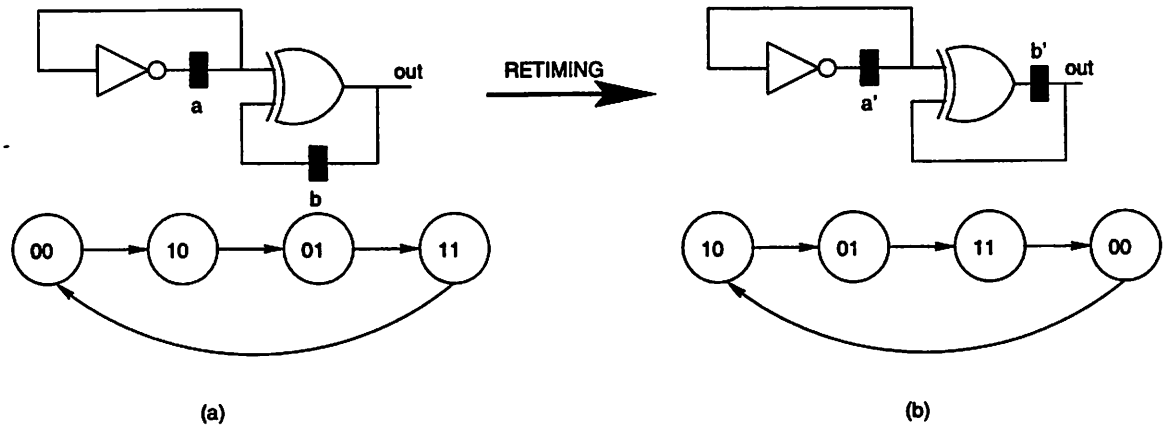


Figure 8.3 Retiming changes the state encoding. Circuit diagram and state-transition graph for (a) Original circuit and (b) Retimed circuit.

minimized while keeping the maximum cycle delay below a given value. The trade-off between reducing cycle time and area is seen in Figures 8.2(a),(b), and (c).

3. Change in state encoding: Movement of latches across a combinational gate results in different encodings of the state transition graph. Retiming across a multi-fanin/fanout can also lead to change in the number of state bits. Figure 8.3 shows the state-encoding before and after retiming. In this case, retiming has performed the following state encoding transformation:

00	10
10	01
01	11
11	00

Retiming can also change the number of state-bits as illustrated in Figure 8.4. This fact can be exploited to optimize state-space-exploration based verification and synthesis methods where number of state-bits plays a crucial role in the complexity of the problem.

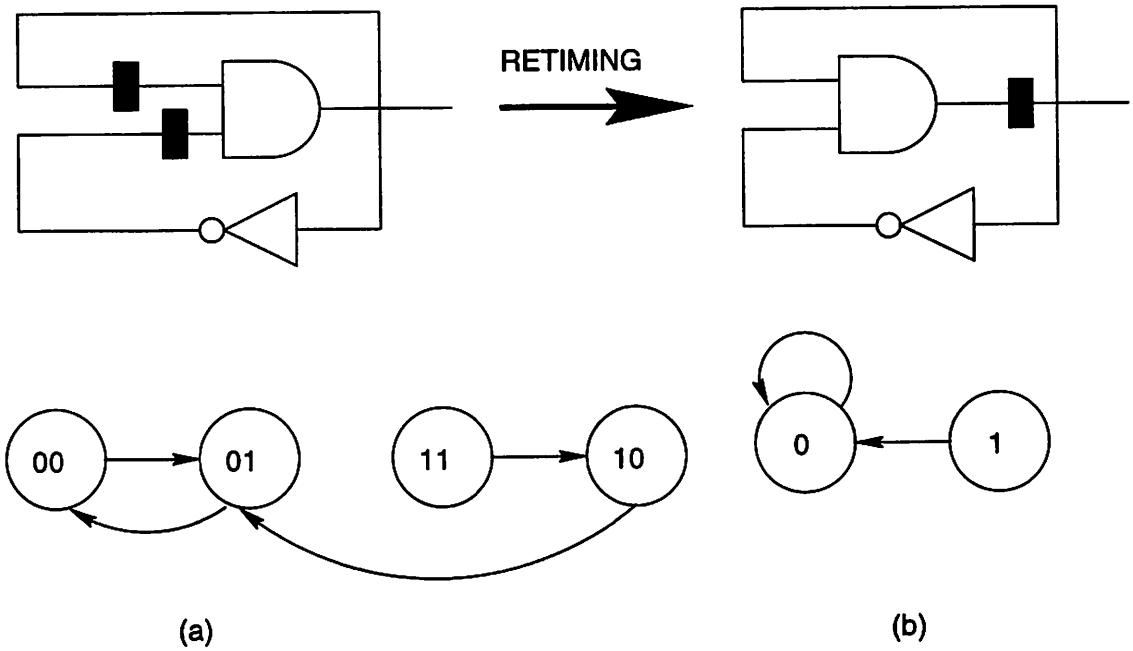


Figure 8.4 Change in the number of state bits due to retiming: 2 state bits in the original circuit (a) vs. 1 state bit in the retimed circuit (b).

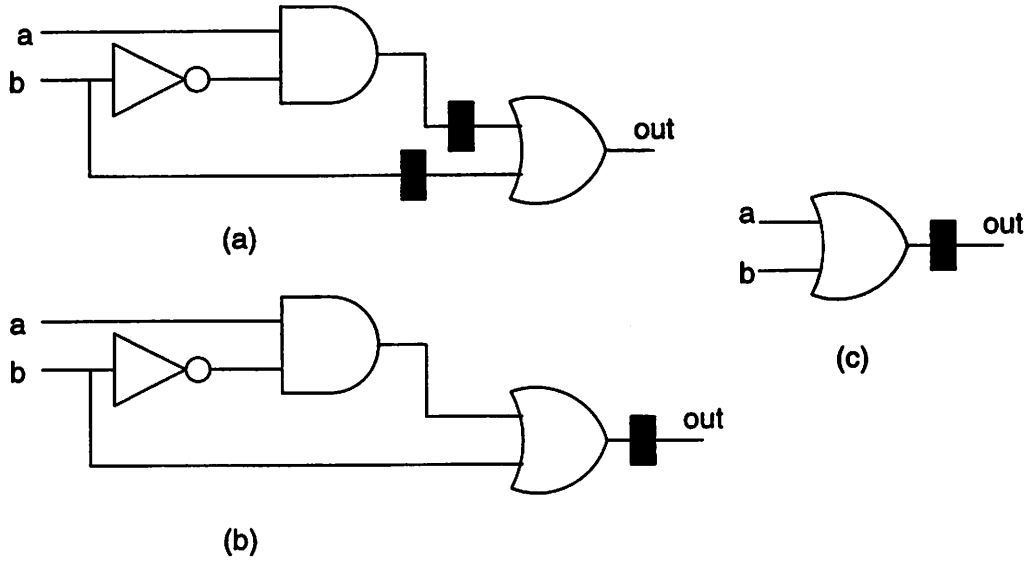


Figure 8.5 Optimization power of retiming followed by resynthesis: (a) original circuit (b) retimed circuit (c) resynthesized circuit.

8.2.3 Retiming – Resynthesis

In this optimization, resynthesis is performed after retiming the latches. This method is more powerful than simple retiming or synthesis because retiming exposes logic, thus enabling logic optimization via synthesis which would not have been possible without retiming.

For illustration purposes, consider the circuit given in Figure 8.5(a). By retiming alone we obtain the circuit in Figure 8.5(b). Further logic optimization results in circuit in Figure 8.5(c). The final circuit cannot be obtained by performing any amount of synthesis on the original circuit.

8.2.4 Resynthesis – Retiming

In this optimization, retiming is performed after synthesis. This method is more powerful than simple retiming or synthesis because resynthesis can enable latch movement by creating additional points for latch transfer or by adding or removing redundant logic. Consider the circuit given in Figure 8.6(a). By resynthesis alone we obtain the circuit in

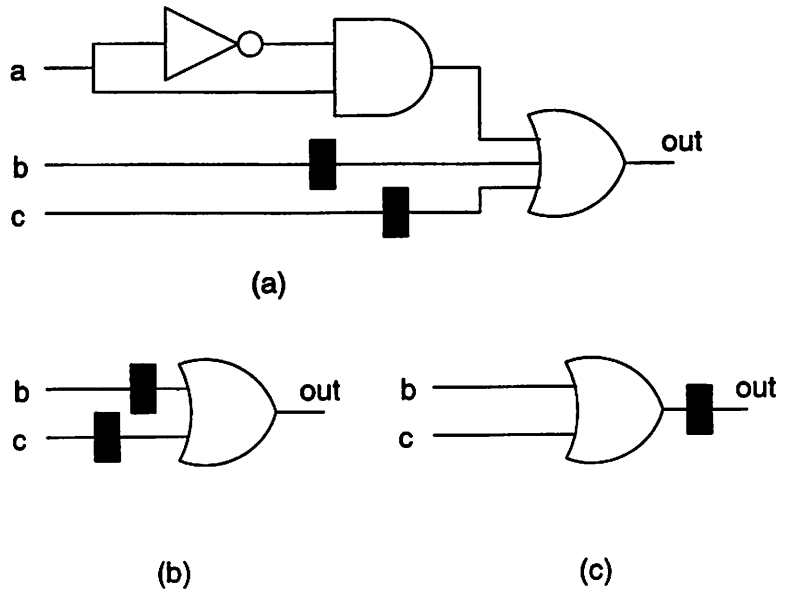


Figure 8.6 Optimization power of synthesis followed by retiming: (a) original circuit (b) synthesized circuit (c) retimed circuit.

Figure 8.6(b) while applying retiming after synthesis results in circuit in Figure 8.6(c). The final circuit cannot be obtained by performing any amount of retiming operation on the original circuit.

Retiming followed by resynthesis and synthesis followed by retiming have different optimization power. For example, the circuit in Figure 8.6(a) cannot be optimized by the former transformation sequence, whereas the circuit in Figure 8.5(a) cannot be optimized by the latter.

8.2.5 Synthesis – Retiming – Synthesis

In Figure 8.7, we show an example of an optimization that can be obtained by two synthesis steps with a retiming step in between. Essentially, the first synthesis step duplicates the logic for the second input to the XOR gate which facilitates the backward retiming of latches. The second synthesis step optimizes the logic. Note that trying to perform retiming as the first step will be futile, since both the forward and backward

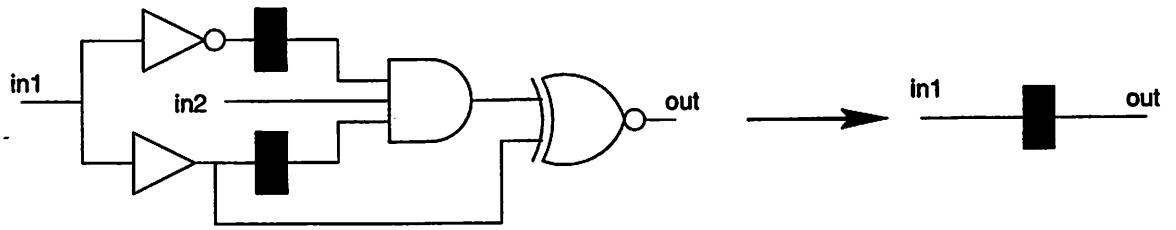


Figure 8.7 Optimization power of synthesis–retiming–synthesis: (a) original circuit (b) optimized circuit. This optimization cannot be done by retiming–synthesis–retiming.

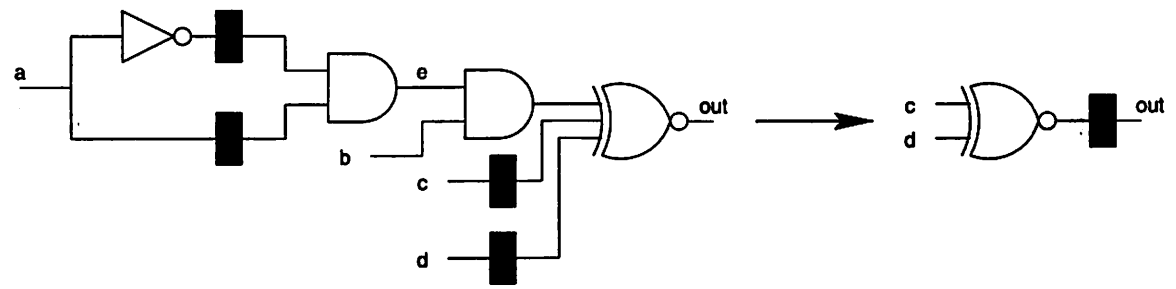


Figure 8.8 Optimization power of retiming–synthesis–retiming: (a) original circuit (b) optimized circuit. This optimization cannot be done by synthesis–retiming–synthesis.

movements of latches are blocked.

8.2.6 Retiming – Synthesis – Retiming

In Figure 8.8, we show an example of an optimization that can be obtained by two retiming steps with a synthesis step in between. Essentially, the first retiming step allows the simplification of Signal e (which reduces to constant 0). In second retiming, latches are moved across the ex-nor gate at the output. Note that combinational optimization at the first step will not achieve anything, since no simplification can be made.

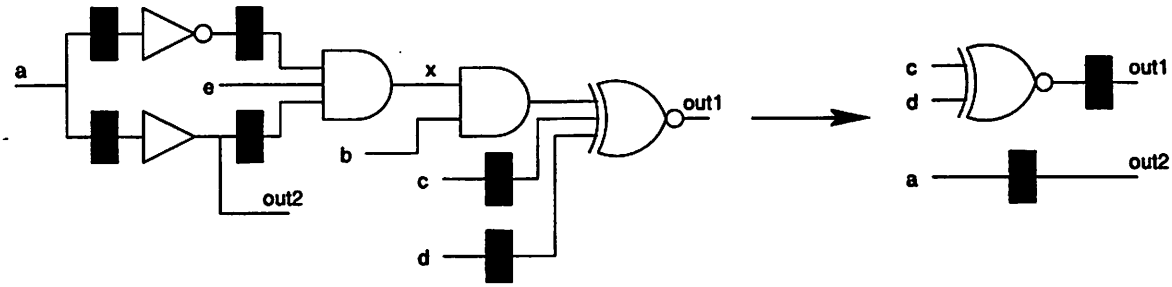


Figure 8.9 A circuit requiring "retiming – synthesis – retiming – synthesis – retiming" transformations.

8.2.7 Iterative Retiming and Resynthesis

The next question arises whether a finite number of retiming and resynthesis moves is sufficient, i.e., can the transformation obtained by arbitrary number of retiming and resynthesis operations always be captured by a finite number of such moves?

If we can come up with a parameterized version of the examples in the previous section, that would suffice as a counter example to the above hypothesis. Unfortunately, we have not been able to come up with such an example. The example shown in the Figure 8.9 indicates that the sequence of "retiming – synthesis – retiming – synthesis – retiming" transformations will be necessary to get the optimized design shown on the right. The first four transformations are necessary to optimize Signal X to 0.

Analysis

Upon analysis of the illustrations we observe the following:

1. Forward movement of latches can get blocked due to an input.
2. Backward movement of latches can get blocked due to an output.
3. Retiming facilitates synthesis by exposing logic.
4. Synthesis facilitates retiming by:
   (a) making some inputs redundant and allowing forward movement

(b) duplicating the logic for the output and allowing backward movement

(c) creating new cut-points for latches to move across

We now compare the optimization power of an arbitrary number of retiming-resynthesis steps to general sequential optimization techniques.

8.2.8 Retiming–Resynthesis vs. General Sequential Optimization

General sequential-circuit optimization makes use of various techniques including:

1. State encoding
2. State minimization
3. Logic optimization using information about unreachable states
4. Logic optimization using input/output don't care sequences

Let us now see which ones of these can be implemented using retiming and synthesis transformations alone. In [Mal90], an attempt has been made to characterize the optimization power of retiming and resynthesis transformations, which we discuss next.

8.2.9 Exposition in Malik's Thesis

The following theorem asserts the state encoding power of retiming and resynthesis operations.

Theorem 3 **[Mal90]** *Given a machine implementation M_1, corresponding to a state transition graph G, with a state assignment S_1, it is always possible to derive a machine M_2 corresponding to the same state transition graph G, and a state assignment S_2 by applying only a series of resynthesis and retiming operations on M_1.*

The proof of the theorem makes use of one-to-one mapping between the states of M_1 and M_2, thereby transforming one state assignment to another using appropriate logic. [Mal90] also discusses the case where the STGs of M_1 and M_2 are different. It is asserted that under restricted state-transformations of the STG, the final circuit can be obtained from the initial circuit using retiming and resynthesis operations.

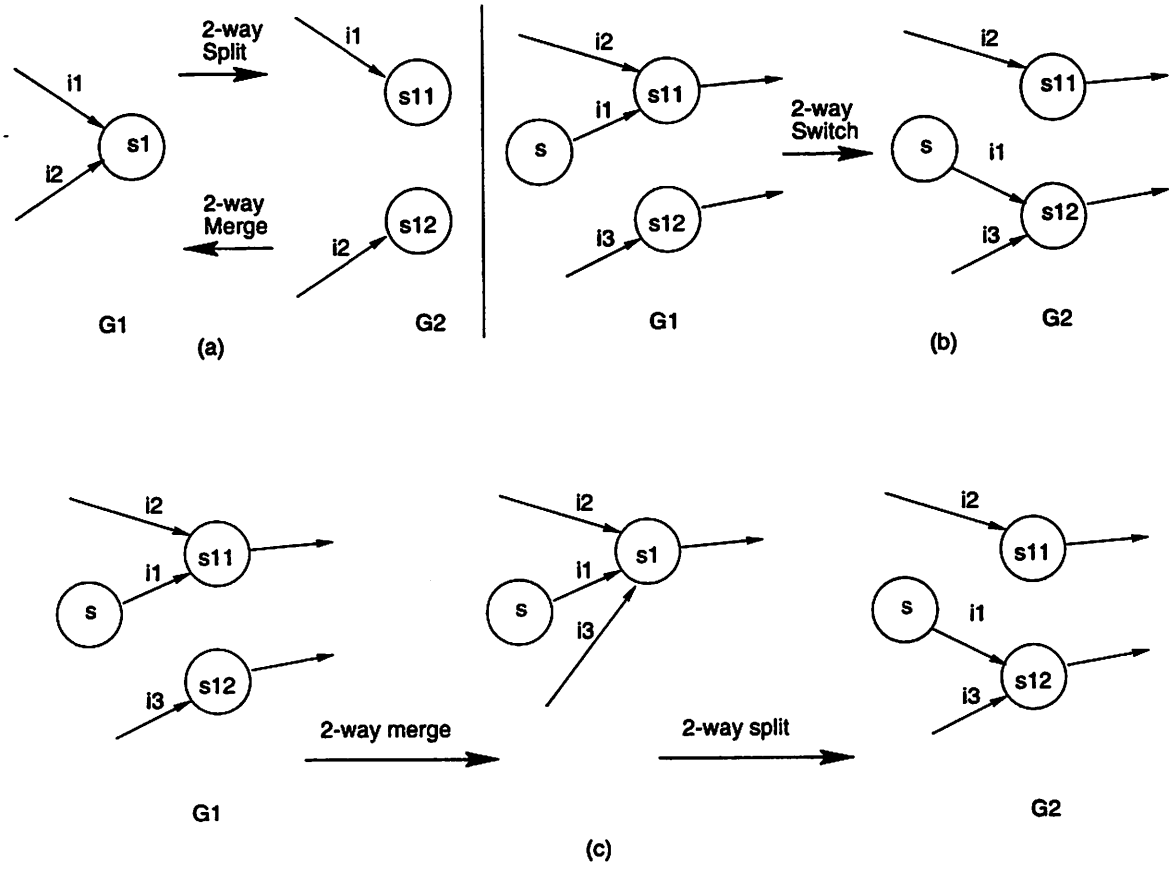


Figure 8.10 State-graph transformations (a) 2-way split and 2-way merge (b) switch (c) switch using 2-way split and merge.

Suppose G_1 and G_2 are STGs corresponding to M_1 and M_2 respectively. G_1 may be modified to obtain G_2 through a series of three basic transformations. These transformations may create states that are equivalent to existing states, merge states that are equivalent to each other, and modify state transitions to go to states equivalent to the original destinations. The definitions of basic transformations are given below:

2-way split A state s_1 in G_1 is equivalent to two states in G_2 (Figure 8:10(a)).

2-way merge Two equivalent states s_{11} and s_{12} in G_1 are merged to a single state s_1 in G_2 (Figure 8.10(a)).

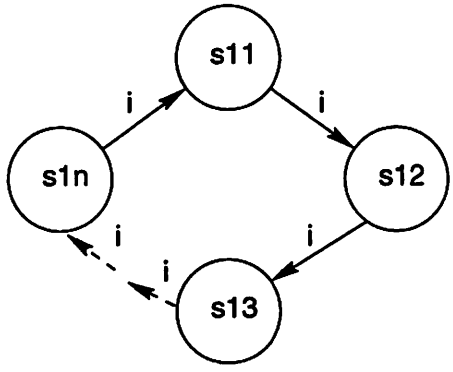


Figure 8.11 Labeled cycle of equivalent states.

Switch A transition in G_1 to a state s_{11} is modified to go to an equivalent state s_{12} in G_2 (Figure 8.10b).

The 2-way split and 2-way merge constitute *primitive transformations*, a 2-way switch, multi-way splits and merges can be accomplished by a sequence of 2-way splits and merges (Figure 8.10c).

Definition 5 *A* **labeled cycle of equivalent states** *in an* STG *is a directed cycle such that all state vertices in the cycle are equivalent and all transition predicate vectors on the edges in the cycle have the same label (Figure 8.11).*

Definition 6 *A* **cycle preserving** (CP) **transformation** *does not create or destroy a labeled cycle of equivalent states.*

A *non cycle preserving transformation* (NON-CP) creates or destroys a labeled cycle of equivalent states.

Theorem 4 **[Mal90]** *Let M_1 be an implementation corresponding to the state assignment S_1 and* STG *G_1 and M_2 be an implementation corresponding to the state assignment S_2 and* STG *G_2. If G_2 is obtained from G_1 using only* CP *transformations then M_2 can be obtained from M_1 using only a sequence of retiming and resynthesis operations.*

The proof considered G_2 to contain a CP 2-way split of some state s_1 in G_1. A transition to s_1 in G_1 corresponds to a transition to either s_{11} or s_{12} in M_2 depending on the primary input vector. It was stated that the primary input vector and state s_1 uniquely determine which of s_{11} or s_{12} is the destination state in M_2. Thus, the one-to-many mapping between the state codes for M_1 and the state codes for M_2 is actually a one-to-one mapping between the M_1 state codes plus the primary input and M_2 state codes. This can be accomplished through a combinational circuit C. Circuit C' performs many-to-one mapping from M_2's state codes to M_1's state codes. The proof was illustrated with a figure that is reproduced in Figure 8.12. The figure shows how the circuit may be retimed resulting in a circuit that corresponds to G_2. This may be further resynthesized to any circuit M_2 that corresponds to state assignment S_2.

8.2.10 Interpretation and Extensions

In the exposition given in [Mal90] of the synthesis capability of retiming and synthesis, we came across some aspects that needed either correct interpretation or correction. These are enumerated below:

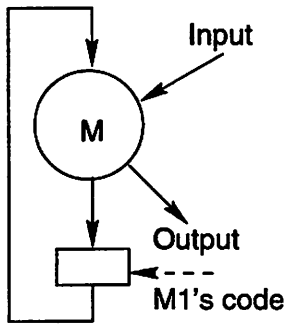
1. The conditions under which merger of two states can be implemented with retiming and synthesis was not clearly stated.

2. An assertion is made that all valid transformations are some sequence or combinations of splits, merges and switches. In particular all valid transformations can be obtained using 2-way switch and 2-way merge.

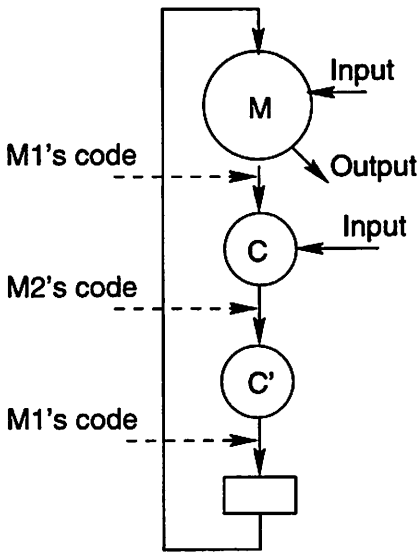
3. The proof of the Theorem 4 states that the primary input vector and state s_1 uniquely determine which of s_{11} or s_{12} is the destination state in M_2.

4. The proof is given for the case where G_2 contains a 2-way split of some state in G_1. The proof states that since each step in retiming and resynthesis is reversible, 2-way merges (obtaining M_2 given M_1) can be handled using retiming and resynthesis. While this is theoretically possible, it is not possible to give a constructive algorithm to obtain this transformation as shown later.

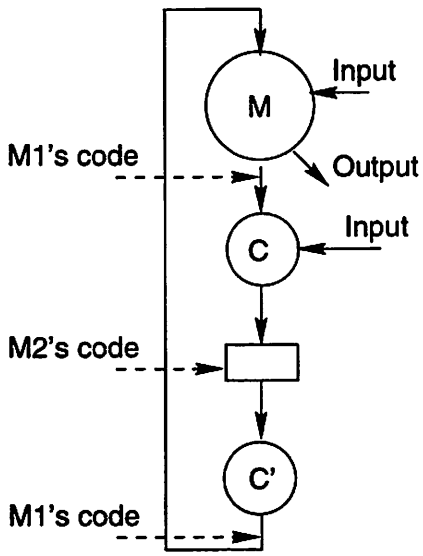
5. No condition is given for splitting a state with a self loop.



(a) Machine M1



(b) Resynthesize



(c) Retime to get M2

Figure 8.12 Obtaining equivalent FSM implementations (proof for Theorem 4).

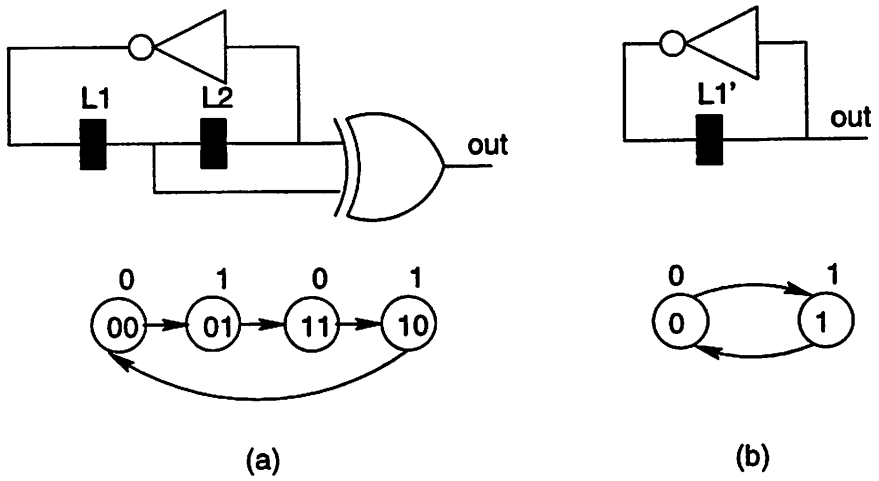


Figure 8.13 Counterexample to the assertion in [Mal90]: Original circuit in (a) cannot be transformed to the final circuit in (b) using retiming and resynthesis.

6. The equivalence of old and new destination states is given as the condition for making the "switch" transformation. This is not completely correct.

7. The illustrations of non-CP transformations are not valid.

Below we make our clarifications and provide extensions wherever possible.

Definition 7 *Two states s_1 and s_2 are* **1-step equivalent**, *if for all inputs i, the next state of s_1 on i is the same as the next state of s_2 on i and vice versa.*

1. Upon investigation we found that the merger of two equivalent states s_0 and s_1 can be implemented using retiming and synthesis only if the states are 1-step equivalent. Similarly, a switch can occur only if the new destination state is 1-step equivalent to the original state.

2. With this restriction on the merger of two states, not all valid state-transition graph transformations can be modeled. An example is shown in Figure 8.13. The original circuit is shown on the left with the associated state transition graph G_1. A sequentially equivalent circuit is shown on the right with corresponding

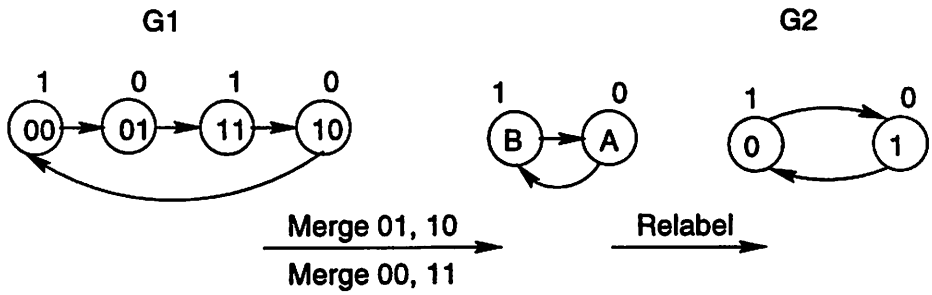


Figure 8.14 Using CP transformations to obtain the final STG from initial STG.

state transition graph G_2. A quick analysis shows that neither the latches can be retimed nor can the logic be optimized indicating that a sequence of retiming and resynthesis moves cannot make this circuit transformation. However, in Figure 8.14, we show how G_2 can be obtained from G_1 by a sequence of CP transformations. The transformation in Figure 8.14 involves merging the state "01" with "10" and state "00" with "11". However, since these states are not 1-step equivalent (they are 2-step equivalent), the STG transformation cannot be implemented with retiming and synthesis transformation. This violates the assertion [Mal90] that all valid transformations are some sequence or combination of splits, merges and switches, where these terms are interpreted appropriately.

3. In the proof of the Theorem 4, the assumption of the unique determination of the destination state in M_2 (one of s_{11} or s_{12}) given the primary input vector and state s_1 is not correct. Consider the splitting of s_1 in G_1 as shown in the Figure 8.15. Given i and s_1, we do not know which of s_{11} or s_{12} is the next state in G_2.

   We can fix this problem by considering the transformation as shown in Figure 8.16. The main difference between the transformations shown in Figure 8.16 and Figure 8.12 is that we also make use of previous state information of M_1 in evaluating the state codes for M_2. By using information about previous state in M_1, next state in M_1, and the input, we uniquely determine the next state for M_2. The combinational logic C' performs many-to-one mapping from M_2's state

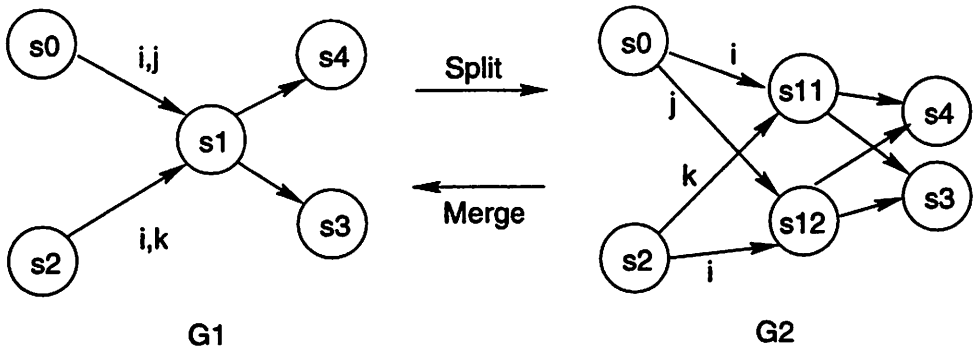


Figure 8.15 Counter-example to the proof of the Theorem 4. The next state in G_2 cannot be determined solely by the next state in G_1 and the input vector.

codes to M_1's state codes. We notice that the counter-example shown in Figure 8.15 is trivially handled by the approach shown in Figure 8.16.

4. It is not straightforward to implement merger of two equivalent states. As shown in Figure 8.17(b), the new next-state bits can be directly obtained using combinational logic. To get back the next-state bits for M_1 from the next-state bits of M_2, however, requires information about the present state of M_1. As a result, we cannot retime the latches across the logic D to put them at the boundary of M_2's code. This is because of the feedback path from the latch output to D'. To overcome this problem, first we need to make the transformation as shown in Figure 8.17(c). Notice that this transformation requires the knowledge of the next-state function for M_2. Now we can retime the latches appropriately to obtain Figure 8.17(d). Performing a final resynthesis step results in machine N as shown in Figure 8.17(e). We should note, however, that this is a practical rather than theoretical problem.

5. In [Mal90], no condition was given for splitting a state with a self-loop. In Figure 8.18 we show the transformation for splitting a state with a self-loop. The rationale behind this transformation is that we want to be able to perform 2-way merge of the states obtained via 2-way splitting. As shown in Figure 8.18, states s_{11} and s_{12} on the right are 1-step equivalent and hence we can perform 2-way

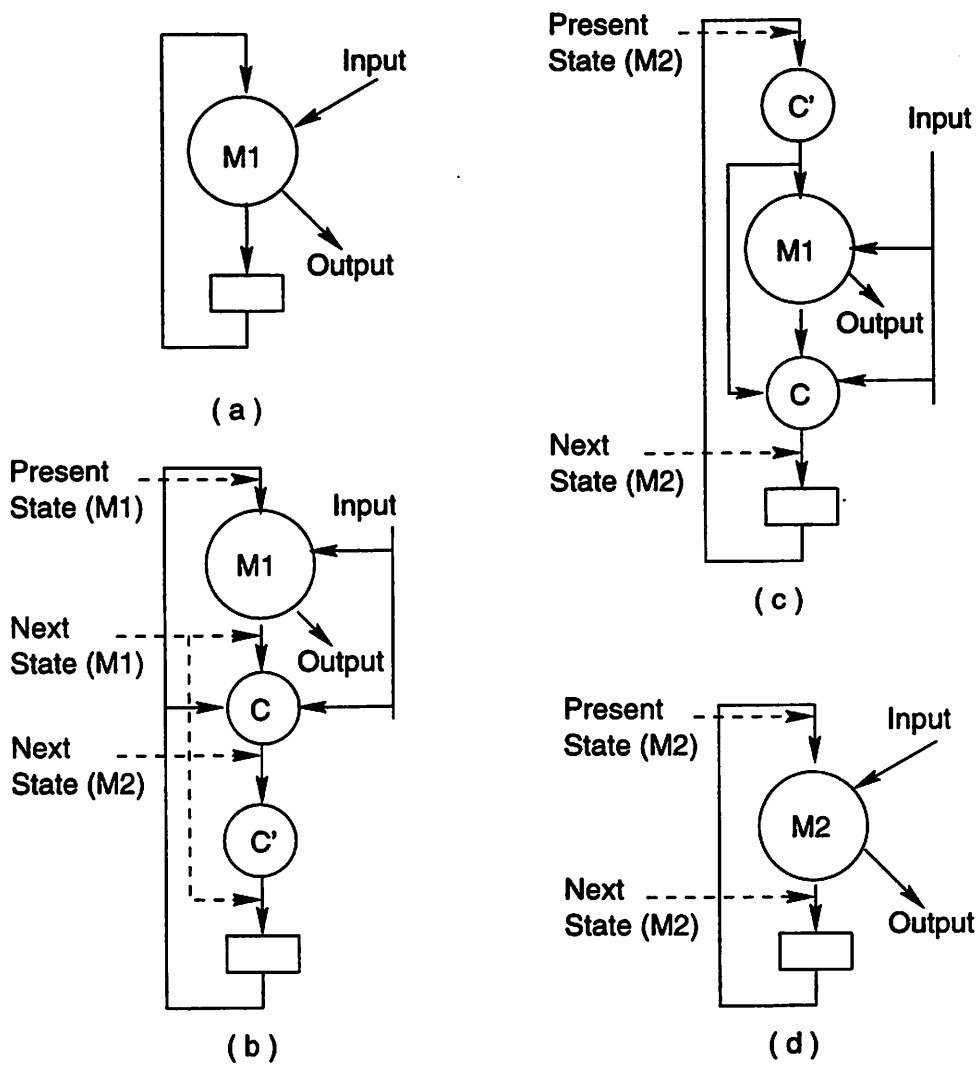


Figure 8.16 Illustration of STG transformation (splitting of states) which can be implemented by retiming and combinational optimization: (a) Original machine M_1 (b) Generation of next state bits for the new machine (c) Retiming to generate next state bits (d) Combinational optimization to obtain new machine M_2.

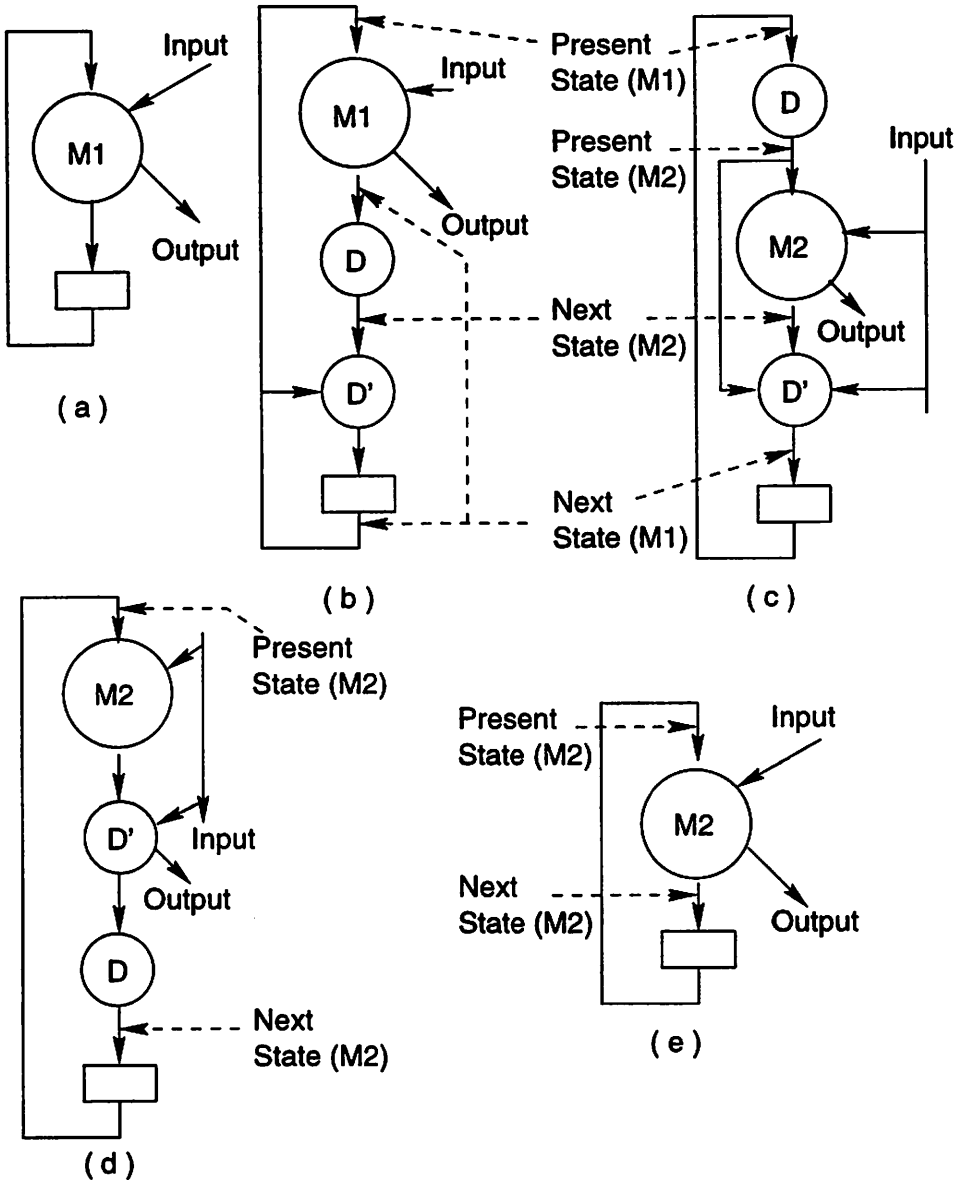


Figure 8.17 Illustration of STG transformation (merger of states) which can be implemented by retiming and combinational optimization: (a) Original machine M_1 (b) Generation of next-state bits for the new machine (cannot be retimed to get M_2) (c) Reverse transformation to generate M_2 and the encoding and decoding logic (d) Retiming to generate next-state bits (e) Combinational optimization to obtain new machine M_2.

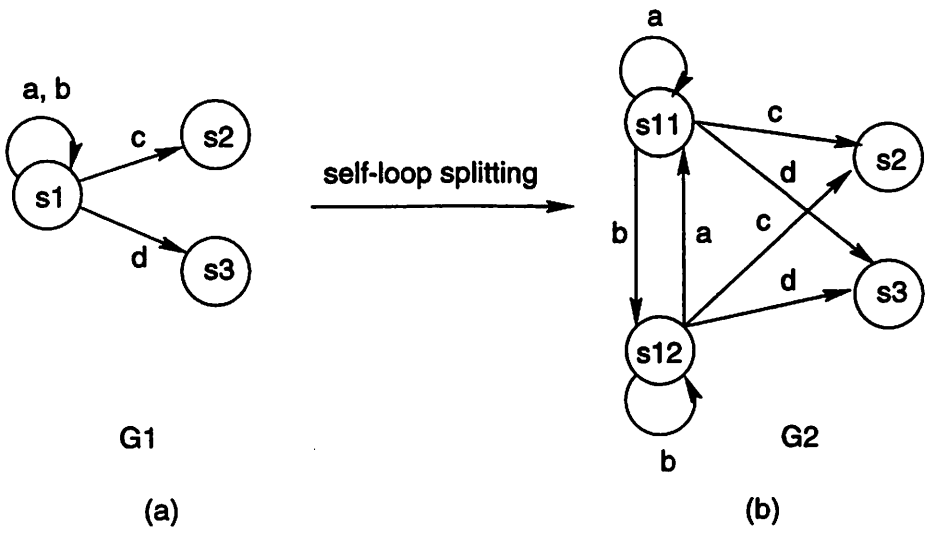


Figure 8.18 STG transformations involving splitting a state with a self-loop.

merge to obtain the STG shown on the left.

The splitting of states with self-loops as shown in Figure 8.18 can be implemented using retiming and resynthesis with the transformations given in Figure 8.16.

Note that multi-way splitting of a state with self-loop can be achieved by further splitting the states s_{11} and s_{12}. Applying the condition of merging 1-step equivalent states, it is obvious that we can merge two states each with a self-loop only if they are of the form shown in Figure 8.18.

6. Since 2-way switch can be implemented using a combination of a 2-way merge and a 2-way split, an obvious outcome of the correct interpretation of "merger" of two states is that a "switch" can take place only if the new destination state is 1-step equivalent to the old destination state. A switch between two 1-step equivalent states can be obtained using similar transformations.

7. An interesting observation can be made that with the correct interpretation on the merging of two states – we do not need the condition of CP preserving trans-

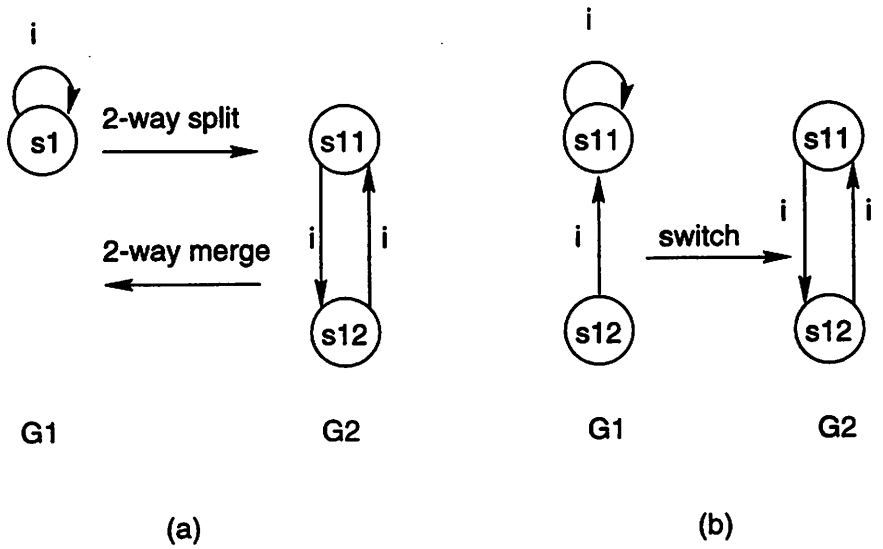


Figure 8.19 Non-CP transformations, as illustrated in [Mal90].

formations. The counter-example can be seen in Figure 8.18. In STG G_1, the self-loop on s_1 is a *labeled cycle of equivalent states* (there is just one state in the cycle). However, in STG G_2, due to the self-loops on s_{11} and s_{12}, we have two labeled cycles of equivalent states, i.e., this STG transformation is non-CP. However, as discussed earlier, we can implement this STG transformation using retiming and resynthesis.

Next we look at the examples of non-CP transformation given in [Mal90] shown in Figure 8.19. The merger of states s_{11} and s_{12} shown in Figure 8.19a is not a valid 2-way merge because states s_{11} and s_{12} are not 1-step equivalent. This invalidates the classification of this transformation as non-CP. In Figure 8.19b, the transformation involves a switch. Notice that states s_{11} and s_{12} are 1-step equivalent. However, after the switch, states s_{11} and s_{12} are no longer 1-step equivalent, making the switch transformation invalid.

Based on the above observations, we state the modified version of Theorem 4.

Definition 8 *A transformation of an* STG G_1 *into another* STG G_2 *is a* **1-step equivalent transformation** *if* G_2 *has been obtained from* G_1 *by either splitting of a state, or merger of two 1-step equivalent states, or switching between two states which are 1-step equivalent.*

Theorem 5 *Let* M_1 *be an implementation corresponding to state assignment* S_1 *and* STG G_1 *and* M_2 *be an implementation corresponding to state assignment* S_2 *and* STG G_2. M_2 *can be obtained from* M_1 *using only a sequence of retiming and resynthesis operations if and only if* G_2 *is obtained from* G_1 *using only* 1-STEP EQUIVALENT TRANSFORMATIONS.

Proof:

$\Leftarrow$

Figures 8.16 and 8.17 illustrate how splitting and merging of 1-step equivalent states transformations can be implemented using retiming and resynthesis. Switching between two 1-step equivalent states can be implemented by a combination of merging the two states and splitting as shown in Figure 8.10.

$\Rightarrow$

As shown in Figure 8.14, the merger of 2-step equivalent states cannot be implemented using retiming and resynthesis. We can extend this counter-example to indicate that a merge of k-step equivalent states, $\forall k > 1$ cannot be implemented using retiming and resynthesis. ■

8.2.11 Sequential Optimization Using Unreachable States

The general sequential circuit optimization extracts don't cares by computing the set of unreachable states. The don't cares obtained in this manner are used to optimize the logic. This optimization is not possible by retiming and resynthesis alone. This is illustrated in in Figure 8.20. In Figure 8.20(a), we present a circuit, with the corresponding state transition graph in Figure 8.20(b). State "00" is the initial state. We observe that state "11" is unreachable from the initial state. We can make use of this information in optimizing the circuit resulting in circuit given in Figure 8.20 (c). Since the above transformation uses information (the set of reachable states) not available

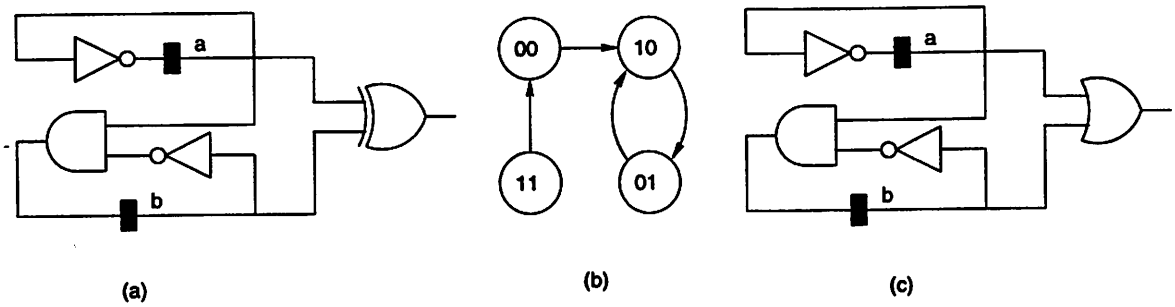


Figure 8.20 Logic optimization using don't cares derived from unreachable states.

to retiming and resynthesis we can conclude that the optimization power of retiming and resynthesis transformations is strictly less than that of general sequential circuit optimization.

8.3 Extending Notions of Retiming and Synthesis

The examples given in the previous section illustrated the limitations of retiming and combinational transformations. By suitably extending the notions of conventional retiming and combinational optimization, we can increase the optimization capability of these transformations.

8.3.1 Eliminating Floating Latches

The current combinational optimization techniques do little manipulation of latches (e.g., latch removal via constant propagation). While gates which do not transitively fanout to any primary output are eliminated during combinational optimization, latches are treated as pseudo primary inputs and outputs and hence are not eliminated even if they do not transitively fanout to any primary output. Such latches are not eliminated during a retiming operation either. We can extend the notion of combinational optimization to one which trivially gets rid of such latches before proceeding to regular combinational optimization. The process of removing latches that do not fanout to any primary outputs is termed as *floating latch elimination*. It does not add to the complexity of the synthesis algorithm. With this extended notion of synthesis, the cir-

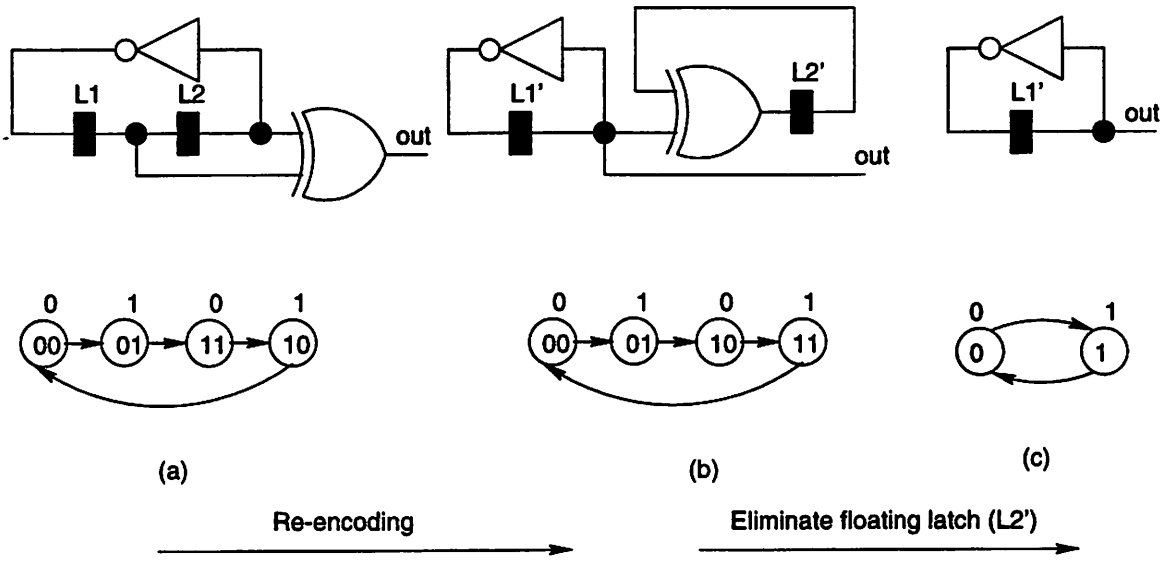


Figure 8.21 Circuit transformation using floating latch elimination.

cuit transformation shown in Figure 8.13 can be obtained. The transformation process is shown in Figure 8.21. Essentially, the first transformation re-encodes the circuit, which can be implemented by retiming and resynthesis as explained in Theorem 3. This is followed by floating latch elimination.

In general, this transformation will allow us to implement STG transformations, where some redundant state bits are removed and the STG is reduced in size.

8.3.2 Allowing Negative Retiming

Retiming can be extended by introducing the concept of "negative" latch [Mal90]. Allowing a negative edge weight n on a peripheral edge (an edge that connects either an input pin to a logic block or connects a logic block that computes the value of an output to the corresponding output pin) is equivalent to "borrowing" n latches from the environment. The latches may be "returned" by a subsequent retiming step. Using this concept, we will be able to deal with some of the examples presented in Section 8.2. In particular, in Figure 8.22, we show how the circuit transformation of Figure 8.9, can be done with synthesis – retiming – synthesis step (as opposed to retiming – synthesis

– retiming – synthesis – retiming).

The question arises, whether negative retiming adds any optimization power to conventional retiming, i.e., are there circuit instances where optimization using negative retiming combined with synthesis cannot be implemented with conventional retiming and synthesis? In [Mal90], it is claimed that allowing negative edge weights on the peripheral edges allows retiming operations and subsequent optimizations that would otherwise not be possible. To illustrate this assertion, an example was given which is reproduced in Figure 8.23. However, the same optimization can be achieved without resorting to any negative retiming as shown in Figure 8.24. Essentially, instead of borrowing a latch from the environment, we duplicate the logic that produces the output, thereby allowing the backward propagation of latches. After combinational optimization, the duplicate logic can be appropriately removed. It is our conjecture that in general, any optimization made possible by negative peripheral retiming can be implemented by suitably duplicating the output logic that prevents the backward movement of latches.

8.4 Verification Complexity

In the previous section, we discussed the optimization potential for various sequences of retiming and synthesis transformations. In this section, we formalize the implementation verification complexity for each of these transformations. Intuitively, the verification complexity should be proportional to the optimization power of the corresponding transformation.

8.4.1 Verification After Retiming

The retiming moves the location of the latches leaving the topology of the combination blocks unchanged. The implementation verification problem definition is:
Given two circuits C_1 and C_2, decide whether C_2 can be obtained from C_1 by simple retiming moves.

This verification involves following two steps [SSBS92]: i) establish that the underlying graphs of the two circuits are isomorphic and ii) establish that the latch counts in corresponding cycles in the two circuits remains the same. These steps are explained in detailed below.

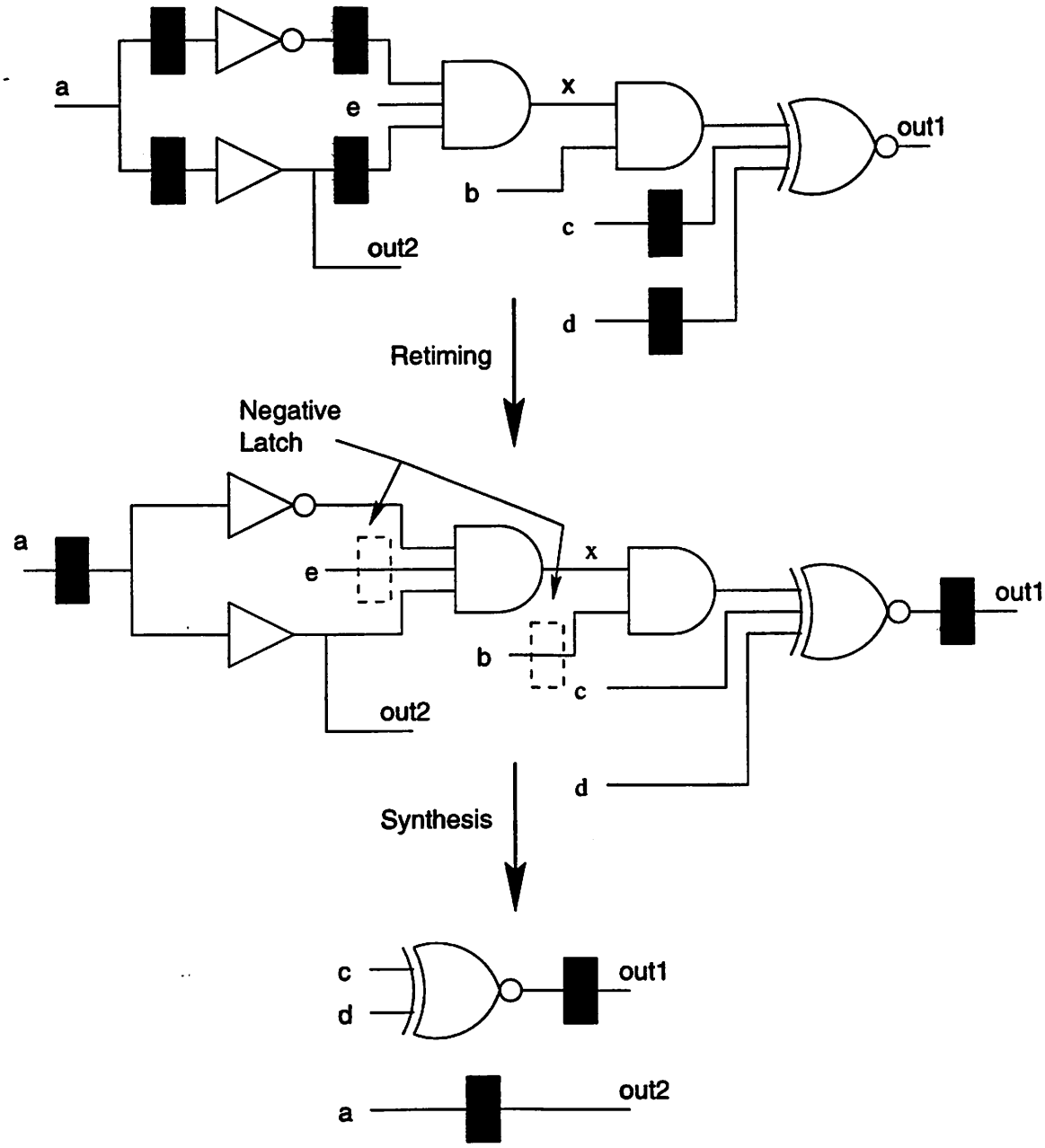


Figure 8.22 Retiming using negative latches.

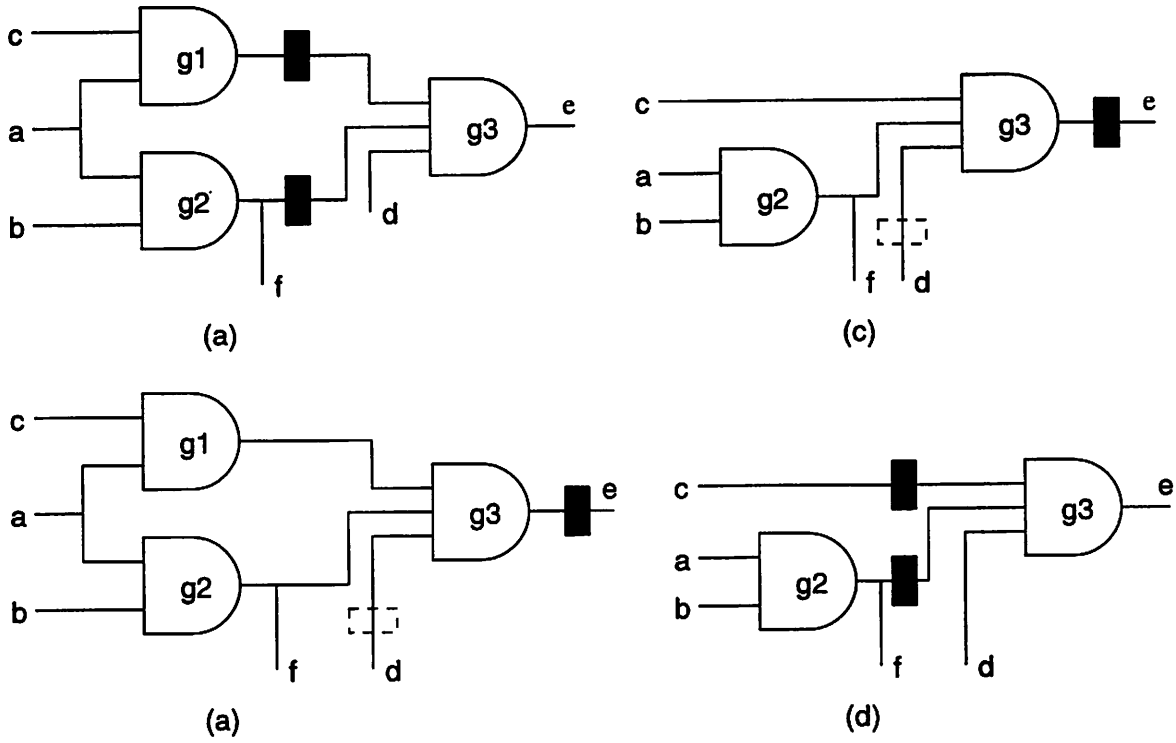


Figure 8.23 Example illustrating optimization using negative latches: (a) original circuit (b) peripheral retiming with negative edge-weights (c) optimization (d) final retiming.

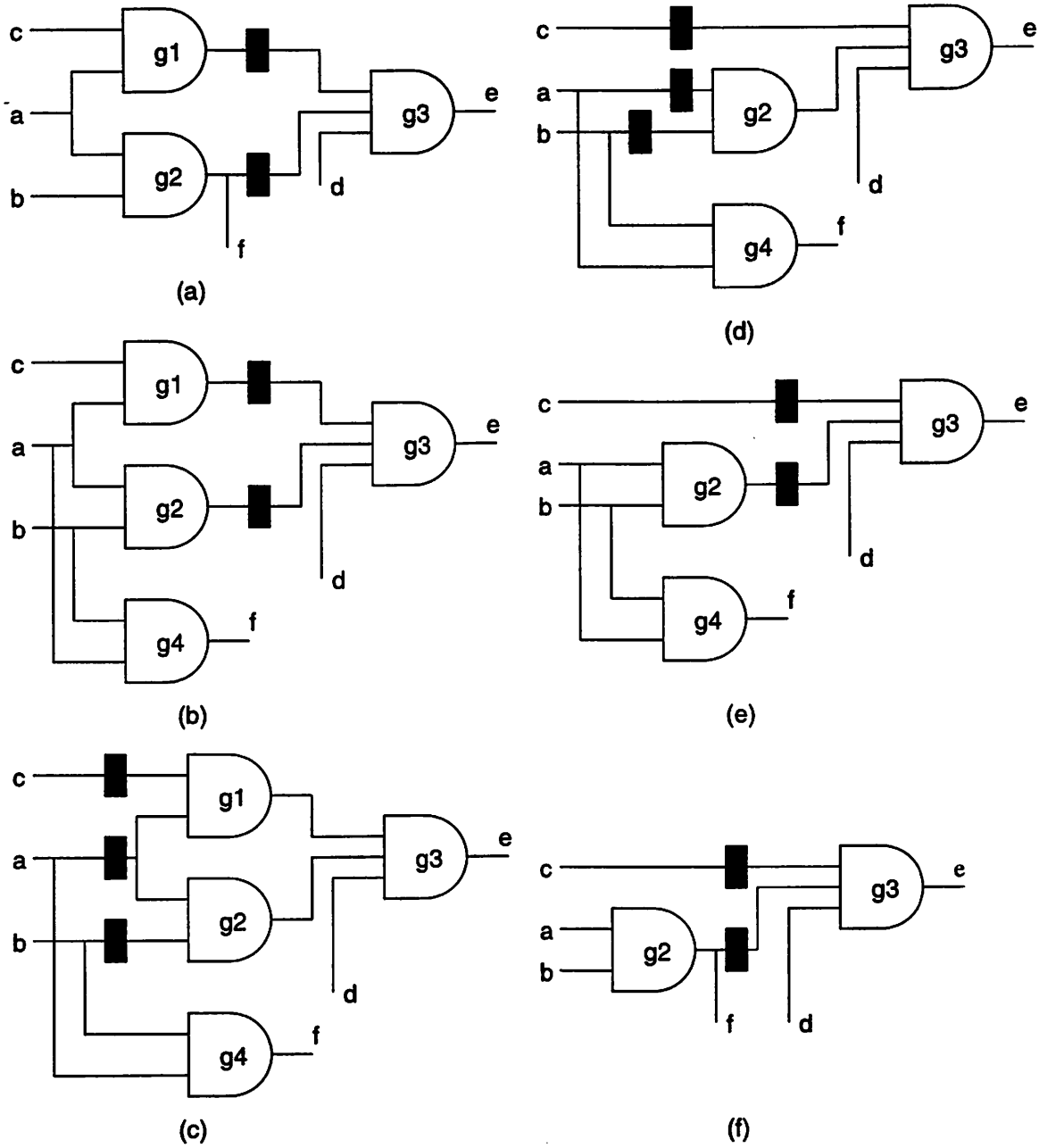


Figure 8.24 Normal retiming has same optimization power as negative retiming (negative latches are shown as dashed boxes): (A) original circuit (B) duplicating output logic (C) retiming (D) optimization (E) retiming (F) removing duplicate logic.

1. First we need to verify that the two circuits have the same graph structure (assuming the correspondence of primary inputs and outputs). This is an instance of directed graph isomorphism.

   Formally, given a circuit C with combinational gates and latches, we create a directed graph $G = (V, E)$. We create a node corresponding to each combinational gate. For simplicity, assume that all combinational gates are of the same kind. Create an edge e_{ij} between node v_i and v_j if the corresponding gate c_i is a fanin to gate c_j (possibly through latches). To each edge e_{ij}, assign a weight w_{ij} equal to the number of latches between the gates c_i and c_j. Suppose G_1 and G_2 are two circuit graphs obtained from circuits C_1 and C_2, respectively.

   Note that even though we have points of correspondences between G_1 and G_2, i.e., nodes corresponding to input/output ports are matched, still the structural matching between G_1 and G_2 remains a general isomorphism problem. This fact is proved in following theorem.

   Theorem 6 *Given two circuits C_1 and C_2 with input/output correspondences, to determine whether C_2 is structurally equivalent to C_1 is as hard as graph isomorphism problem.*

   Proof: We prove this by reducing a general graph isomorphism problem to structural equivalence problem of retimed circuits. From a given graph G we create a circuit in the following way:

   (a) For each node in the graph, we create a combinational gate.

   (b) For an edge e_{ij} in the graph, we create a net from gate i to gate j.

   (c) We create a dummy input node I and a dummy output node O.

   (d) We create a net from node I to each gate in the circuit.

   (e) Finally, we create a net from each gate in the circuit to the output node O.

   For two graphs G_1 and G_2, we create two circuits C_1 and C_2. It is trivial to observe that G_1 and G_2 are isomorphic *iff* the circuits C_1 and C_2 are structurally equivalent. ■

2. We also need to validate the number of latches along each net. A necessary and sufficient condition has been given in [SSBS92]. The condition states that for a valid retiming, the number of latches in each cycle should be preserved. Instead of enumerating all the cycles in the cyclic component (which could be exponential in the circuit size), one can check only the fundamental cycles. An $O(m^2 \log(n))$ algorithm for enumerating the fundamental cycles is presented in [SSBS92], where m is the number of edges and n is the number of nodes.

8.4.2 Verification After Retiming–Resynthesis

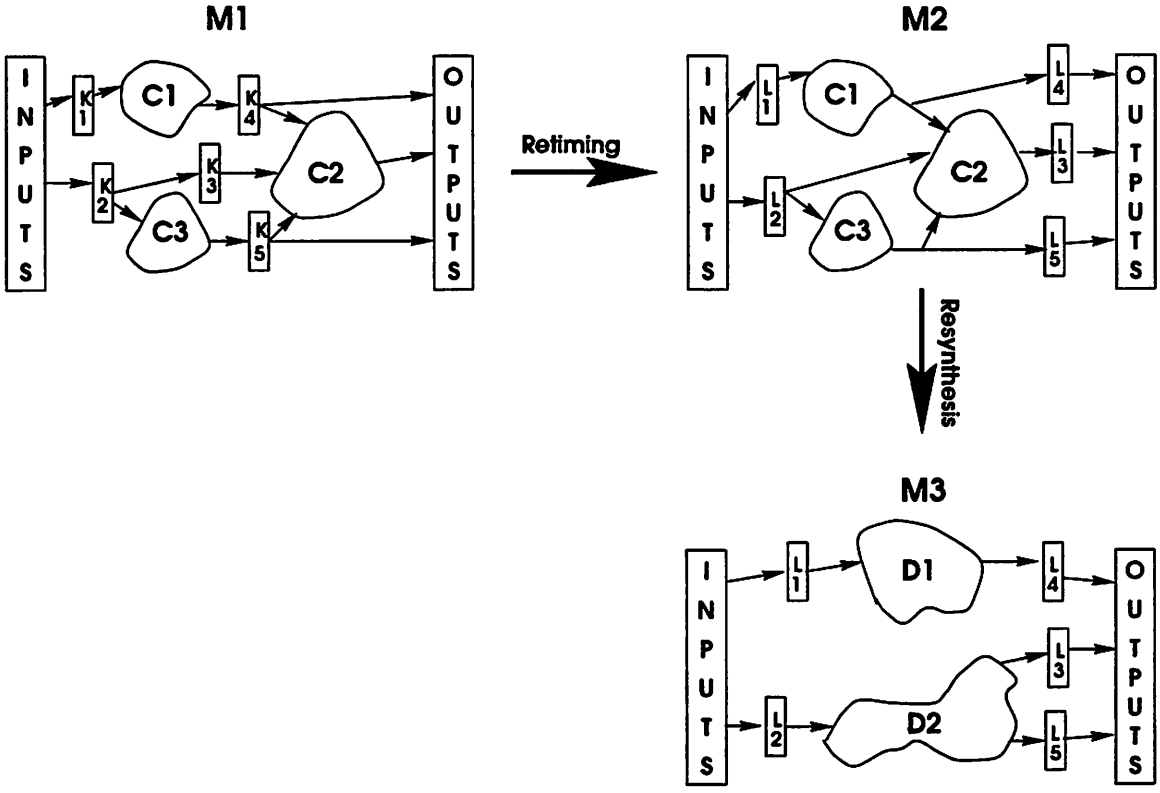


Figure 8.25 Transformation sequence for retiming followed by resynthesis.

The transformation obtained using one retiming followed by resynthesis is shown in Figure 8.25. The original circuit M_1 is retimed to obtain M_2, which is then resynthe-

sized to obtain M_3. The problem at hand is the implementation verification between M_1 and M_3. In the figure we notice that for both retiming and resynthesis, the input/output ports function as fixed anchors. Moreover, for retiming, the locations of the combinational blocks remain fixed (as shown in M_2). And for resynthesis the location of latches remain fixed (as shown in M_3). Notice that the new latch location in M_2 are also present in M_1 (which obviously has initial latch locations). Hence, given M_1 and M_3, we can appropriately retime the latches in M_1 to obtain M_2. In order to determine the new latch locations in M_1, we will need to solve a variant of graph isomorphism problem. The verification between M_2 and M_3 is equivalent to a combinational equivalence check. The complexity of these two steps is dominated by the combinational equivalence check, hence the verification complexity is NP-complete.

8.4.3 Verification After Resynthesis–Retiming

The transformation obtained using resynthesis followed by retiming is shown in Figure 8.26. The original circuit M_1 is synthesized to obtain M_2, which is then retimed to obtain M_3. The problem at hand is the implementation verification between M_1 and M_3. Notice that the initial latch location in M_1 are also present in M_3 (which also contains the new latch locations). Hence, given M_1 and M_3, we can appropriately retime the latches in M_3 to obtain M_2. The verification between M_1 and M_2 is equivalent to a combinational equivalence check. Hence the overall complexity is NP-complete. Another way to look at this is that the transformation $M_1 \rightarrow M_2 \rightarrow M_3$ is just reversed $(M_3 \rightarrow M_2 \rightarrow M_1)$ and hence result obtained Section 8.4.2 applies.

8.4.4 Verification After Resynthesis–Retiming–Resynthesis

The transformation obtained via resynthesis-retiming-resynthesis is shown in Figure 8.27. The original circuit M_1 is synthesized to obtain M_2, which is then retimed to obtain M_3. Finally M_3 is synthesized to obtain M_4. The problem at hand is the implementation verification between M_1 and M_4. Notice that, unlike the previous two cases, the initial and the final latch locations are not simultaneously present either in the original circuit or in the final circuit. To verify M_1 against M_4, we need to guess an intermediate circuit (perhaps M_2) which is obtained after resynthesis of M_1. After performing combinational equivalence check between M_1 and the guessed circuit, we can

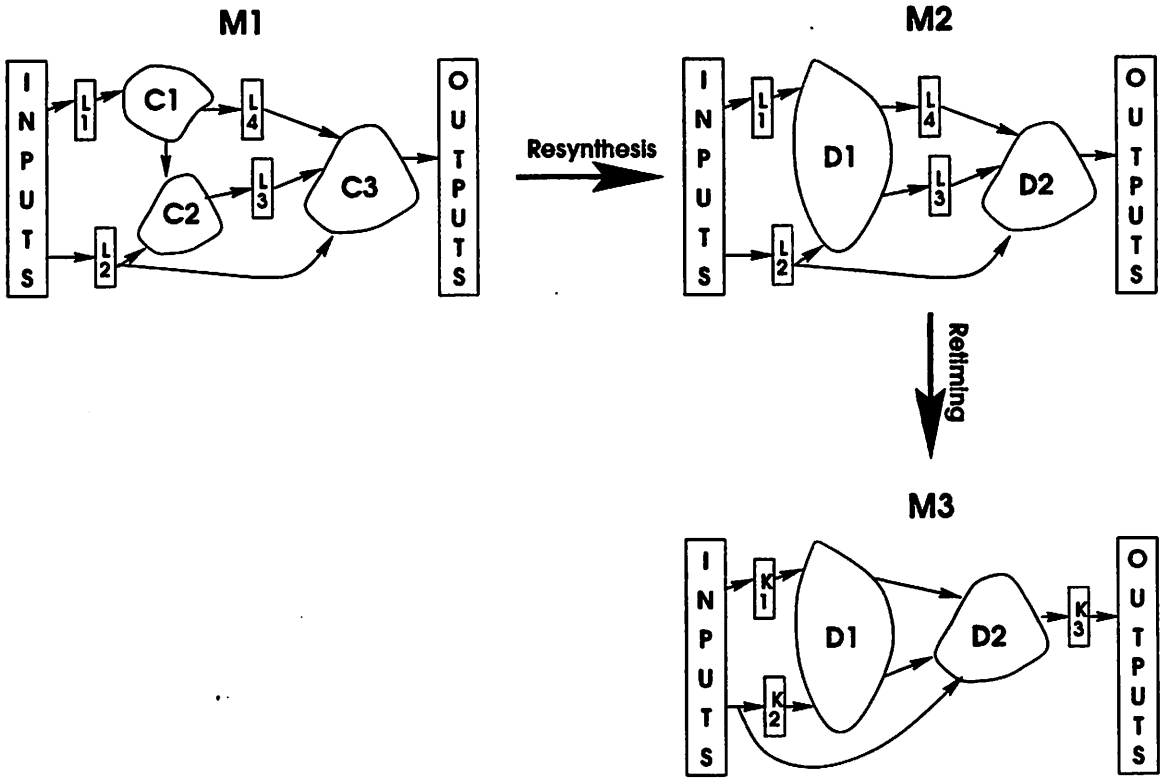


Figure 8.26 Transformation sequence for synthesis followed by retiming.

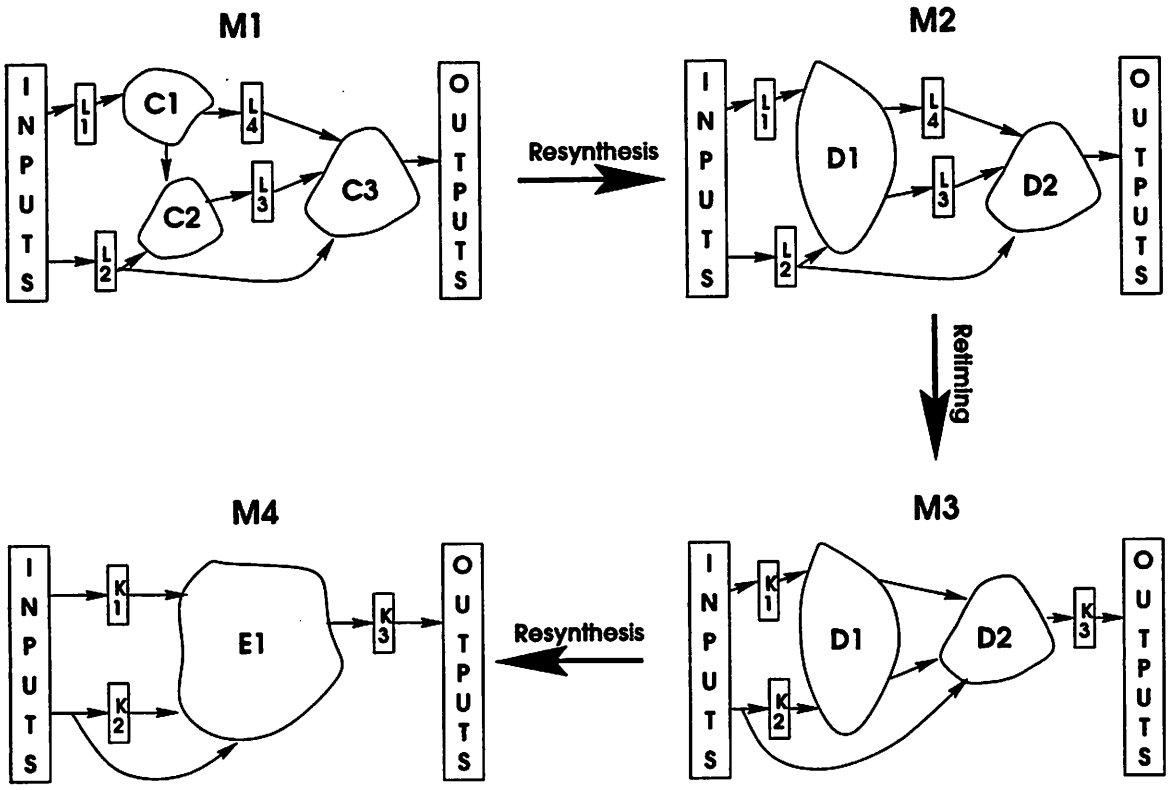


Figure 8.27 Transformation sequence for synthesis followed by retiming and resynthesis.

apply the technique given in Section 8.4.2. The complexity of the overall verification problem falls in Σ_2 class [GJ79] in the polynomial hierarchy.

8.4.5 Verification After Retiming–Resynthesis–Resynthesis

The transformation obtained via retiming–resynthesis–resynthesis is shown in Fig-

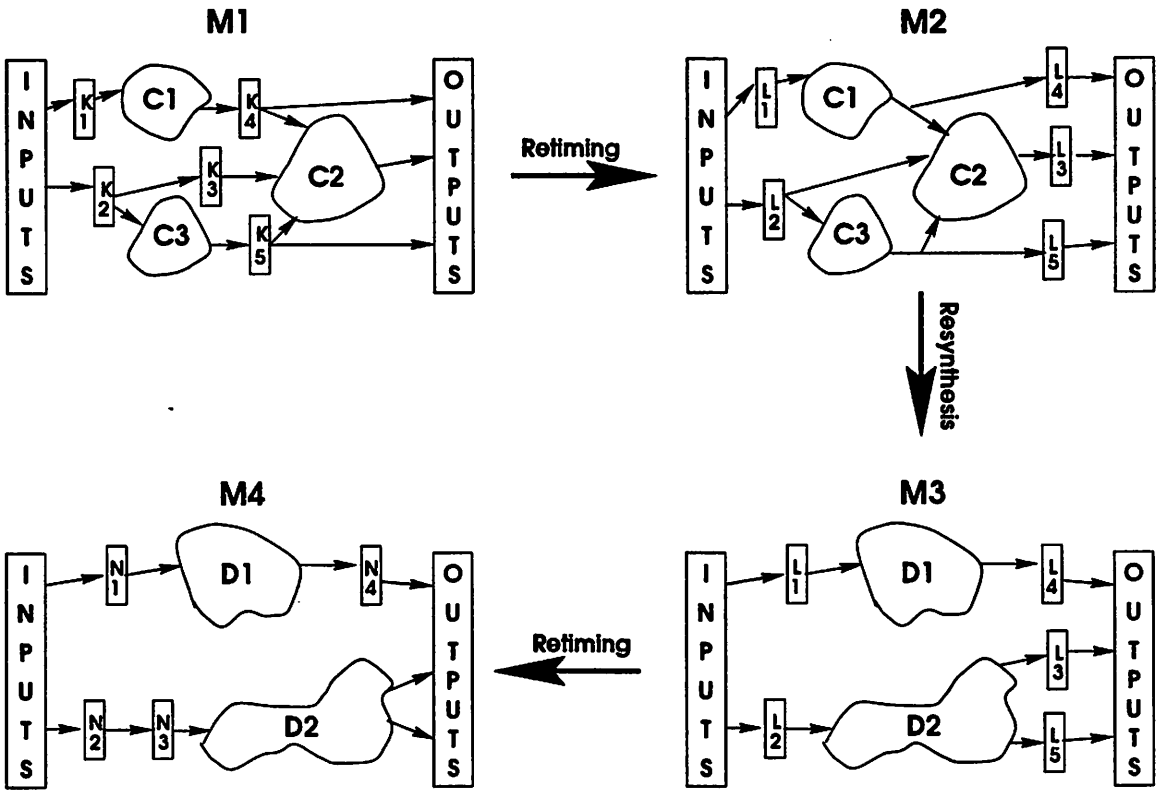


Figure 8.28 Transformation sequence for retiming followed by synthesis and retiming.

ure 8.28. The analysis follows along the lines of that in Section 8.4.4.

8.5 Summary and Open Issues

In summary we have been able to establish the following:

1. STG transformations which involve splitting a state into equivalent states can be implemented using retiming and resynthesis transformations.

2. Merging of two equivalent states can be implemented by retiming and resynthesis only if these states are 1-step equivalent.

3. It is proven that STG transformations can be implemented using retiming and synthesis if and only if they are 1-step equivalent.

4. The traditional notions of retiming and combinational synthesis can be modified leading to improved optimization capability without any increase in the complexity.

A number of issues are still unresolved. In particular:

1. The exact optimization potential of an arbitrary number of retiming and synthesis transformations is unknown except for the fact that it is less than full sequential optimization.

2. The number of retiming and synthesis transformations required to obtain the most optimum circuit possible is unknown. More precisely: suppose S indicates the optimization space of retiming and synthesis transformations; does there exist a finite number k, such that k transformations of retiming and resynthesis can explore all of S.

3. The complexity of establishing if a circuit C_2 has been obtained from C_1 using only retiming and synthesis transformations is unknown. We conjecture that this complexity will be dependent on Point 2.

8.6 Conclusion

Retiming and resynthesis are powerful tools to optimize a sequential circuit. However, so far their exact optimization potential and the complexity of the corresponding verification problem has not been investigated. In this chapter, we have made an attempt to formally characterize the optimization power of various flavors of retiming and resynthesis transformations and also to characterize the exact complexity of the

corresponding implementation verification problem. We have established and clarified some results and have indicated some open issues.

Chapter 9

Verifying Retimed and Resynthesized Circuits

In Chapter 8, we made an attempt to formally establish the sequential optimization capability and corresponding verification complexity of retiming combined with combinational optimization transformations. In this chapter, we propose a practical verification technique for such transformations. We start with the notion that although retiming combined with combinational optimization is a powerful sequential synthesis method, this methodology has not found wide application because formal verification of sequential circuits is not practical and current simulation technology requires the correspondence of latches for ease in the detection of errors. We present a practical verification technique which enables such sequential synthesis for a class of circuits. In particular, we require certain constraints to be met on the feedback paths of the latches involved in the retiming process. For a general circuit, we can satisfy these constraints by fixing the location of some latches, e.g., by making them observable. We show that implementation verification after performing repeated retiming and synthesis on this class of circuits reduces to a combinational verification problem. We also demonstrate that our methodology covers a large class of circuits by applying it to a set of benchmarks and industrial designs.

This chapter is organized as follows: after a brief introduction in Section 9.1 we present previous research works in the area of sequential verification in Section 9.2. We establish the notation, terminology, and our notion of equivalence in Section 9.3. In Section 9.4 we describe the basic idea behind our work and give appropriate definitions. In Section 9.5 we discuss our technique for a circuit with no feedback latches. In Section 9.6 we present the extension to include circuits containing feedback latches. The details of the experimental setup and results are given in Sections 9.7 and 9.8, respectively. Most of the work presented in this chapter was first reported in [RSSB97].

9.1 Introduction

Retiming and resynthesis, though less powerful in theory than full sequential optimization (based on unreachable states, input/output don't care sequences), cover a wide part of the optimization space. However, this technique has not had much success in obtaining a place in traditional synthesis methodology. One of the main bottlenecks has been the lack of efficient verification tools to verify the functionality of the optimized design. The verification complexity of a retimed and resynthesized design is not formally known. It is conjectured to be harder than the NP-hard class of problems. On the other hand, the verification problem for combinational logic optimization is a relatively easier problem in practice. Much work has gone into combining structural and functional techniques to obtain verification algorithms that can deal with reasonably large industrial circuits [Mat96, JMF97, KK97].

We propose a methodology which reduces a sequential verification problem into an equivalent combinational verification problem for a class of circuits. This allows exploitation of the advancements made in the field of combinational verification and use of its powerful techniques to perform verification. Our method requires that for each latch with a feedback path, its next-state function should be positive unate in the latch variable. Later we will show that the scope of this methodology allows i) the presence of self-loops on latches, ii) pipelined circuits where the latches cannot be retimed to the periphery, iii) the presence of latches trapped inside combinational blocks, iv) circuits with load-enabled latches, and v) circuits where latches conditionally update their contents.

Typically, industrial designs consists of two kinds of latches. The first kind constitutes small finite state machines. Each such state machines are strongly connected. These machines interact with each other via the acyclic network of latches of the second kind. In general, designers want to preserve the locations of the latches that hold the states of FSM (the first kind), since they want to monitor simulation results. Fixing some latch locations breaks the feedback paths and as a result the circuit might satisfy our constraint. In case the given circuit still fails our constraint, we expose a minimum number of latches making their locations fixed. Then we perform retiming and resynthesis optimizations on the modified circuit. In general, making a latch observable

can restrict some optimization transformations thus incurring a penalty in optimization quality. In practice, our approach does not incur a significant optimization penalty due to this modification.

9.2 Previous Work

Many researchers have investigated the problem of sequential equivalence checking and in particular verification of retimed circuits. A popular approach is to compose the machines together and traverse the state space of the product machine. The composed circuit is modeled as a finite state machine and the outputs are evaluated as functions of the present state and primary inputs. Equivalence between two circuits implies identical values of corresponding outputs in all reachable states. Explicit state enumeration techniques perform an explicit traversal of the state space [DMN88, DDHY92]. Due to the explicit nature of this technique, it is limited to only a small number of state elements.

Symbolic techniques [CBM89, BCMD90] implicitly perform state-space traversal on the product machine. A salient feature of these techniques is that the size of the underlying decision diagram data structures does not depend on the number of states or the state elements in the circuit. Although, the state-of-the-art symbolic methods can deal with circuits with up to a few hundred latches, their capability falls below the smallest size designs being optimized in industry.

In [AGM96], a technique is described where sequential optimization is performed on a modified circuit (where each pair of states can be distinguished by applying an input in a single clock cycle). The modified circuit is obtained by making some latches observable which in turn restricts the amount of optimization that can be performed. The theoretical complexity of their verification problem remains PSPACE–complete (the complexity of an arbitrary sequential equivalence check). However, on a practical note, their technique requires state space traversal of individual machines as opposed to the product machine. They produced results on relatively small MCNC and ISCAS benchmarks because it was not possible to perform single machine state space traversal for large ones.

In [HCC96b], a technique for verifying the equivalence of two circuits after retiming and synthesis transformations was given. Their technique relies on finding the corre-

spondence between latches by retiming them appropriately. They presented results for ISCAS benchmarks by comparing circuits which have undergone one step of retiming after combinational optimization.

In [HCC97], a combination of BDD-based and ATPG-based technique is presented. This approach relies on finding equivalent points in the two circuits and the symbolic justification requires the computation of the transition relation for the product machine. They gave results on circuits optimized by "script.rugged" inside SIS. This optimization is mostly combinational and only redundancy identification and removal leads to minor sequential changes.

In [SK97], a structural technique for sequential verification is presented. The equivalence is performed by expanding the circuit into an iterative array and by proving equivalence of each time frame by well-known combinational verification techniques. Their technique relies on finding the logic transformations at each time frame. They show results on ISCAS benchmarks, where an optimized circuit is obtained by just one step of combinational optimization (using fx in SIS) followed by retiming. Application of their approach to optimized circuits obtained by a sequence of retiming and resynthesis operations seems difficult.

Proposed solutions to the sequential equivalence problem can be broadly divided into two categories. The solutions in the first category attempt to solve the general sequential equivalence problem [TSL+90, CM90b, HCC97, SK97]. However, due to the complexity of the problem, the proposed solutions are either limited to relatively small size circuits or to circuits which have undergone relatively fewer optimization transformations.

The second approach is to trade off the optimization capability with the verification complexity. In this approach, the sequential optimization is constrained in order to reduce the verification complexity. In the limit, by making all the latches observable, the sequential synthesis reduces to combinational optimization leading to combinational verification problems. The solution proposed in [AGM96] falls in this category. Our methodology can also be viewed as offering another point in the tradeoff curve between constraints-on-synthesis versus complexity-of-verification.

We propose a technique which reduces the sequential equivalence problem to an instance of combinational equivalence; hence it can be applied in practical verification

environments. For each latch, we impose certain constraints on the feedback path (if one exists). If the constraints are not met in the original circuit, we make a minimum number of latches observable in order to satisfy the constraints. We allow arbitrary sequences of retiming and synthesis operations for logic optimization. Also, unlike structure-based approaches [HCC97, SK97], our technique does not rely on the structural similarity between the circuits and we can deal with circuits which have gone through a sequence of retiming and synthesis optimizations. The techniques proposed apply to circuits containing edge-triggered latches (both regular and load-enabled).

In industrial design environments, combinational verification is applied to sequential circuits. However, this requires that little or no retiming be performed. These constraints limit the scope of retiming and synthesis transformations drastically. By contrast, our approach allows an arbitrary number of retiming and combinational synthesis transformations since it does not rely on structural similarity or matching state-bits.

9.3 Preliminaries

In this section we present our circuit model and notion of equivalence used in this work.

9.3.1 Circuit Model

A sequential circuit is an interconnection of combinational gates (no combinational cycles) and memory elements along with input and output ports. Typically various notions of sequential circuits differ in the definition of memory elements. We focus on sequential circuits where all the memory elements are edge-triggered latches driven by the same clock (single phase). However, these latches can have load-enable signals. A sequential circuit is given as $C = (I, O, G, L)$, where I, O, G, and L are sets of inputs, outputs, gates, and latches, respectively. Each latch $l \in L$ is a pair $l = (x, e)$, where x is the output signal of the latch and e is the load-enable signal. For a latch without any load-enable signal (also referred to as regular latch in this paper), we assume $e = 1$. Similar to the notion in [LVW97], we define a latch.class $cl = (e)$, which is all latches that have the the same load-enable signal e. This classification is important during retiming transformations, since latches can merge as the result of a move only if they belong to the same class.

9.3.2 Notion of Equivalence

Several notions of sequential equivalence are proposed in the literature. For circuits with a unique initial state, the "reset" equivalence is checked for all the states reachable from the respective initial states. For multiple initial states, the following notion of equivalence is used: two circuits C_1 and C_2 with initial states set S_{I_1} and S_{I_2} respectively are equivalent *if and only if* for each state $s \in S_{I_1}$, there exists a state $t \in S_{I_2}$ such that C_1 and C_2 are reset equivalent for these initial states and vice versa. For circuits with an unknown initial state, several notions of equivalence have been proposed: post-synchronization equivalence [Pix92], safe-replaceability [PSAB94], circuit-covering [HCC96a], and 3-valued equivalence, to name a few.

Similar to 3-valued equivalence, we do not assume a power-up initial state for the latches. Instead, we assume that at power-up, each latch has a non-deterministic Boolean value. Note that, this does not prevent the design from having a reset state for some latches which is activated when the reset line is pulled or a reset sequence is applied.

Since the power-up state is non-deterministic, the circuit behavior may not be deterministic for some input sequence. Given a circuit C with L latches and an input sequence π, the output function $O_C(\pi) = o$ if the circuit produces output o on input sequence π from every power-up state (in $2^{|L|}$); if the circuit produces two different outputs o_1 and o_2 on input sequence π from two different power-up states, we say that $O_C(\pi) = \perp$, where $\perp$ denotes an undefined value.

Definition 9 *Two circuits C_1 and C_2 are* **exact 3-valued equivalent** *if and only if for any input sequence π: $O_{C_1}(\pi) = O_{C_2}(\pi)$.*

Notice that $\perp$ is somewhat similar to the value 'X' used in conservative 3-valued logic simulation. However, $\perp$ gets rid of the conservative effects of 3-valued simulation: a 3-valued simulator may incorrectly say that a signal is 'X' because it does not have the ability to correlate the various instances of X values as illustrated in Figure 9.1. Circuits 9.1(a) and 9.1(b) are not 3-valued equivalent, but are exact 3-valued equivalent.

In the next section we present our technique to derive a combinational representation of the sequential circuits. In Sections 9.5 and 9.6 we apply it to sequential circuits without and with feedback respectively.

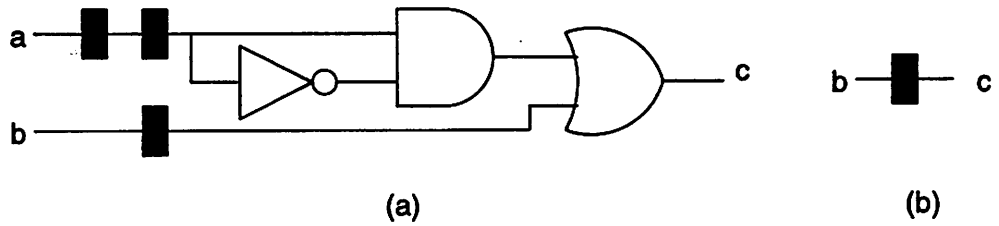


Figure 9.1 Example of circuits which are not 3-valued equivalent but are exact 3-valued equivalent.

9.4 From Sequential to Combinational

We reduce the problem of sequential verification to an extension of combinational verification. The goal of our technique is to obtain a canonical acyclic combinational circuit from a given sequential circuit. Towards that we introduce the following extensions of regular Boolean functions.

$$t \;:\; t \in \mathbb{Z} \quad \text{Represents current time}$$
$$\mathbb{T} \;:\; \{\tau \in \mathbb{Z} : \tau \leq t\}$$

9.4.1 Clocked Boolean function

Definition 10 *A* **clocked Boolean function (CBF)** *is defined for circuits containing combinational gates and regular latches. Given a circuit C, the CBF for the circuit represents the functionality of its outputs. This functionality is given in terms of input values in multiple (but finite) clock cycles. Formally, a CBF for the output of a circuit, with n inputs and latch depth d is a Boolean function* $F : \mathbb{B}^{n*d} \mapsto \mathbb{B}$. *For a signal s in the circuit, the CBF of the signal* $s(t)$ *at time t is defined inductively as follows:*

- *If s is the output of a gate G, the corresponding CBF is the functional composition of the CBFs of its fanins at the same time instant, i.e.,* $s(t) = f_g(y_1(t), y_2(t), \ldots, y_n(t))$, *where* $y_1, y_2, \ldots, y_n$ *are the fanin signals of G, and* f_g *represents its functionality.*
- *If s is the output of a latch, then the CBF is the value of its fanin after one clock cycle, i.e.,* $s(t) = y(t-1)$, *where y is the input of the latch.*

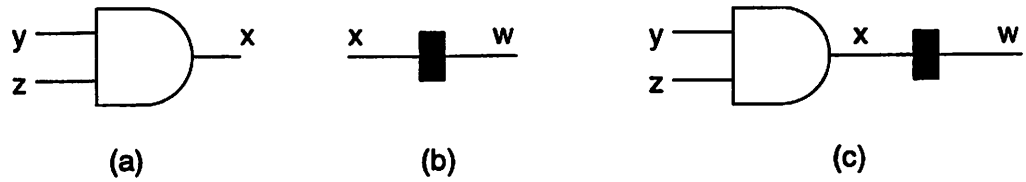


Figure 9.2 Functionality of AND gate and a latch.

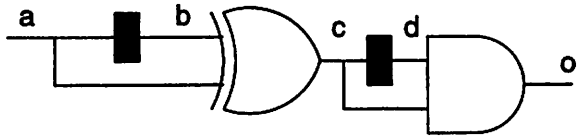


Figure 9.3 Example of a sequential circuit: latch trapped within a combinational block.

- *If s is the primary input of the circuit, its CBF is an independent input variable $s(t)$. Note that $s(t)$ and $s(t')$ for $t \neq t'$ are different independent variables.*

We illustrate this concept using the following examples. The function f_x for the output of the AND gate is nothing but the logical AND of the functions at the input, i.e., $x(t) = y(t)z(t)$. The function for the latch is interpreted as the function of the latch input signal at the previous clock cycle, i.e., $w(t) = x(t-1)$. If we put the latch and the AND gate together as shown in Figure 9.2(c), the functionality of the latch output in terms of the primary inputs is given by,

$$w(t) = x(t-1) = y(t-1)z(t-1)$$

Consider the circuit given in Figure 9.3. The output function is given as:

$$\begin{aligned} o(t) &= c(t)d(t) \\ d(t) &= c(t-1) \\ c(t) &= b(t) \oplus a(t) \\ b(t) &= a(t-1) \\ o(t) &= [a(t-1) \oplus a(t)] \wedge [a(t-2) \oplus a(t-1)] \end{aligned}$$

Essentially, the output function depends on the value of input a in three different clock cycles and we have obtained the CBF for the output.

Unlike the regular Boolean functions which give the value of a signal based on the assignment of input values for one time instant only, the CBF gives the value of a signal for input values delayed by a finite number of clock cycles. This notion is very similar to the notion of *Timed Boolean Function* given in [Lam93]. In [Lam93], similar expressions are obtained for the signals which integrate both timing and logical functionality and generalize the conventional Boolean functions to the temporal domain. These expressions were used in timing analysis, analysis and optimization of wave-pipelined circuits, and performance validation of circuits and systems. However, their usage in representing and verifying the functionality of sequential circuit has not been done before.

9.4.2 Event driven Boolean function

First we define some notation.

$$\begin{array}{rcll} p_i(\tau) & : & \mathbb{T} \mapsto \mathbb{B} & \text{Boolean predicates over time} \\ P & = & \{p_i(\tau)\} & \text{Set of Boolean predicates} \\ \mathbb{E} & = & \bigcup_{k\geq 0}\{E : E \in P^k\} & \text{Set of events} \end{array}$$

where elements of P^k are denoted by $[p_1, p_2, \ldots, p_k]$ and an event $E \in \mathbb{E}$ is an ordered set of timed Boolean predicates.

Next we establish the time instant defined by an event. We define the function $\eta : \mathbb{E} \mapsto \mathbb{T}$ as follows:

$$\begin{aligned} \eta([\,]) &= t \quad \text{empty event denotes the current time} \\ \eta([p_1, p_2, \ldots, p_n]) &= \begin{cases} -\infty \text{ if } A([p_1, p_2, \ldots, p_n]) = \phi \\ \max_\tau\{\tau \in A([p_1, p_2, \ldots, p_n])\} \text{ otherwise} \end{cases} \\ &\text{where} \\ A([p_1, p_2, \ldots, p_n]) &= \{\tau < \eta([p_2, p_3, \ldots, p_n]) : p_1(\tau)\} \end{aligned}$$

Intuitively, for an event $E \in \mathbb{E}$, consisting of Boolean predicates over time, $\eta(E)$ gives the most recent time instant after which all the Boolean predicates in E have been

active in the order in which they are listed. If the Boolean predicates in an event cannot be active in the order they are listed, $\eta(E) = -\infty$ indicating an undefined value.

Using the η notation, we now define the next extension to a regular Boolean function.

Definition 11 *An* **event driven Boolean function (EDBF)** *is defined for circuits containing combinational gates and enabled latches. The EDBF for the output of a circuit C, with n inputs and k distinct events, is a Boolean function* $f : \mathbb{B}^{n*k} \mapsto \mathbb{B}$. *For a signal s in C, and an event E, the functionality of s at time* $\eta(E)$ *is defined inductively as follows:*

- *If s is the output of a gate G, the corresponding EDBF is the functional composition of the EDBFs of its fanins values associated with the same event, i.e.,* $s(\eta(E)) = f_g(y_1(\eta(E)), y_2(\eta(E)), \ldots, y_n(\eta(E)))$, *where* $y_1, y_2, \ldots, y_n$ *are the fanin signals of G and* f_g *represents its functionality.*

- *If s is the output of a latch with fanin signal y and enable signal e, then it takes the most recent value of y at which e was active. This is given as* $s(\eta(E)) = y(\eta([e, E]))$.

- *If s is the primary input of the circuit, it represents an independent input variable.*

Intuitively, for a signal s and an associated event $E \in \mathbb{E}$, the EDBF $s(\eta(E))$ gives the value of s at the most recent time instant after which all the Boolean predicates in E were active in the time order consistent with the listed order.

The following examples illustrate the concept. In Figure 9.4, the value of signal y, can be represented as $y(\eta([e]))$, since the value of y is equal to the value of x at the time at which e was last active. In Figure 9.5, the functionality of signal z associated with

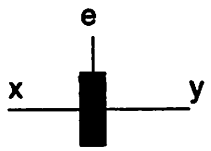


Figure 9.4 Combinational functionality in the presence of enabled latches (illustration I).

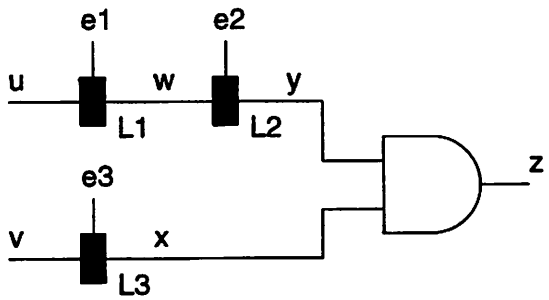


Figure 9.5 Combinational functionality in the presence of enabled latches (illustration II).

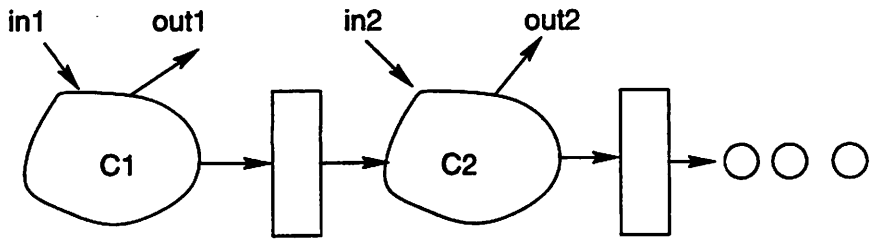


Figure 9.6 An example of acyclic sequential circuit: pipelined circuit.

an event E can be obtained as follows:

$$\begin{aligned} z(\eta(E)) &= y(\eta(E)) \cdot x(\eta(E)) \\ y(\eta(E)) &= w(\eta([e_2, E])) \\ w(\eta([e_2, E])) &= u(\eta([e_1, e_2, E])) \\ x(\eta(E)) &= v(\eta([e_3, E])) \\ z(\eta(E)) &= u(\eta([e_1, e_2, E])) \cdot v(\eta([e_3, E])) \end{aligned} \tag{9.1}$$

Equation 9.1 indicates that value of z is equal to the AND of value of u which has been propagated through both latches L_1 and L_2 and of v which has been propagated through L_3.

In the next section we show how we make use of CBF and EDBF to obtain combinational functions for sequential circuits.

9.5 Sequential Circuits without Feedback

We consider sequential circuits without feedback paths (also known as "acyclic sequential circuits"). The typical circuits in this category include: pipelined circuits (Figure 9.6); and acyclic circuits with latches trapped within a combinational block (Figure 9.3). We first explain our technique for circuits with regular latches (no load-enable signal) and then describe the case with load-enabled latches.

9.5.1 Circuits with Regular Latches

In this class of circuits the latches update their contents at each clock cycle. The functionality of the circuit depends on the input values possibly at multiple time instants.

We give the method to obtain the CBF for a general circuit. Given an acyclic sequential circuit C, in general, the value of a signal can be required for multiple time instants corresponding to different delays (depending on the number of latches along different paths between the signal and the primary outputs). Starting from primary outputs, we recursively obtain the CBF for each signal as shown in the Figure 9.7. The result of the CBF computation routine is a Boolean formula for each of the outputs in terms of values of inputs in multiple cycles. By treating the input values at different time instants as independent variables, we obtain a combinational function representation for the outputs of the circuit.

Definition 12 *For an acyclic sequential circuit C, the* **sequential depth** *d is equal to the largest delay for which an input affects the output. Note that d can be less than the topological latch depth (maximum number of latches along a path between an input-output pair) due to false dependencies.*

Lemma 1 *Given an acyclic circuit C with sequential depth d, suppose $\hat{C}$ is sequentially equivalent to C. Then the sequential depth of $\hat{C}$ is d.*

Proof: Suppose the depth of $\hat{C}$ is $\hat{d} > d$. Then there are sequences I_1 and I_2 of length $\hat{d}$ and identical in the last $\hat{d} - 1$ vectors such that some output of $\hat{C}$ differs on I_1 and I_2 after applying the last vector. However, the output of C will be the same. Hence C is not equivalent to $\hat{C}$, which leads to contradiction. The case when $d > \hat{d}$ is similar. ■

Canonicity of the Formula

Theorem 7 *Suppose C_1 and C_2 are two circuits and F_1 and F_2 their CBFs. Then $F_1 \equiv F_2 \Leftrightarrow C_1 \equiv C_2$, where equivalence between the circuits is exact 3-valued as defined in Section 9.3.2, and equivalence between the CBFs is combinational.*

Proof:

```
Compute_CBF(C){
    foreach primary output x {
        Compute_CBF_Recursively (x,0);
    }
}
Compute_CBF_Recursively(x,d){
    if x is a primary input, return x(t − d);
    if f(x,d) is already computed, return f(x,d);
    if x is output of a latch {
        y = corresponding latch input;
        f(x,d) = Compute_CBF_Recursively(y,d + 1);
    }
    else{
        G_x = Gate corresponding to signal x;
        foreach fanin y of G_x {
            Compute_CBF_Recursively(y,d);
        }
        f(x,d) = Compose the fan-in functions appropriately;
    }
    Cache the result of f(x,d);
    return f(x,d);
}
```

Figure 9.7 Computing CBF for outputs of a feedback free circuit.

$\Leftarrow$

Assume that $F_1 \not\equiv F_2$. Then there exists a CBF minterm m on the input values up to d clock cycles such that $F_1(m) \neq F_2(m)$. Since the circuit has finite depth, using this minterm m we can generate an input sequence of length d such that when applied to the two circuits, will produce different simulation results. This implies $C_1 \neq C_2$.

$\Rightarrow$

Assume that $C_1 \not\equiv C_2$. Then there exists an input sequence π such that $C_1(\pi) \neq C_2(\pi)$. Since the circuits are acyclic and have finite memory, π need not be longer than d. Using this sequence we can generate a CBF minterm such that when applied to the two formulae, will produce different results implying $F_1 \neq F_2$. Hence contradiction. ■

Note that the above result are stated for any two sequential equivalent circuits not just those obtained by retiming and combinational optimization.

9.5.2 Circuits with Load-enabled Latches

In the case where the latch output is controlled by an enable signal as well, the functionality is as follows: if the enable signal is high, the latch propagates the data value to the output, else the latch retains its old value. In [LVW97], a retiming technique was proposed to handle latches with different enable signals and different clocks. In this work, we propose a verification methodology where all the latches are driven by the same clock but can have different enable signals. Extension to circuits with multiple clocks is straightforward.

We obtain a Boolean function along the lines of the previous case (regular latches). However, in this case we make use of event driven Boolean functions (EDBF) as defined in Section 9.4.2. By instantiating separate Boolean variables for each unique combination of primary input and event, we create a combinational representation of the circuit.

Starting from primary outputs, we recursively obtain the EDBF for each signal as shown in the Figure 9.8.

Canonicity of the Formula

Lemma 2 *Given an acyclic sequential circuit with load-enabled latches, an input/output pair a path between the pair, the number of latches and the event associated with the*

```
Compute_EDBF(C){
    foreach primary output x
        Compute_EDBF_Recursively (x,[])
}
Compute_EDBF_Recursively(x,E){
    if x is a primary input, return (x,E).
    if F(x,E) is already computed, return F(x,E).
    if x is a latch output {
        y = latch input
        e = enable signal
        F(x,E) = Compute_EDBF_Recursively(y,[e,E]).
    }
    else{
        G_x = Gate corresponding to signal x.
        foreach fanin y of G_x {
            Compute_EDBF_Recursively(y,E)
        }
        F(x,E) = Compose the fan-in functions appropriately.
    }
    return F(x,E).
}
```

Figure 9.8 Computing EDBF for the outputs of a circuit.

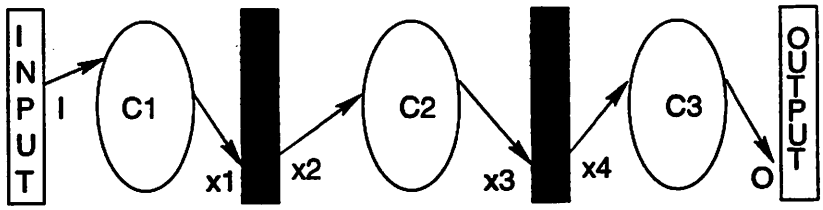


Figure 9.9 Topological arrangement of latches (black boxes) and combinational blocks (ovals).

sequence of enabling signals of the latches along the path is invariant during retiming (ala [LVW97]) and synthesis optimization steps.

Proof: Let us first consider the retiming transformation.

Suppose $\{G_1, G_2, ..., G_k\}$ is a path of gates between an input I and output O. Assume that the latches cannot be retimed across input and output ports. Suppose, during a retiming move, x latches move across gate G_i. If the latches are moved in the forward direction, then x latches are moved from each fanin of G_i (including G_{i-1}) to each fanout of G_i (including G_{i+1}). Hence the number of latches between G_{i-1} and G_{i+1} along the path remains the same. Suppose $e_1, e_2, \ldots, e_k$ is the sequence of enable signals of the latches along a path between input I and output O. For the forward or backward movement of load-enabled latches, the latches being moved must belong to the same enable class. Also, since the circuit is acyclic, a latch cannot jump over another latch during retiming thereby changing the order of enable signals. It implies that the sequence of the enable signals is preserved.

Since combinational synthesis keeps the latch positions fixed, the latch count and the sequence of enable signals along any path in the circuit does not change. To establish that a path pertaining to the true dependency is preserved during the transformation, we make use of illustration in the Figure 9.9. Since the circuit is acyclic, we can arrange the combinational logic and the latches as shown (for simplicity, only two layers of latches are shown in the figure). Now the path $I \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4 \rightarrow O$, from input I to output O is shown in the figure. For combinational optimization x_1, x_3 and x_2, x_4 are treated as primary outputs and primary inputs respectively. Hence to preserve

the functionality of combinational blocks, paths from I to x_1, from x_2 to x_3, and from x_4 to O, must be preserved. This implies that the number of latches and the sequence of enable signals along the path is also preserved. ■

Theorem 8 *Given two acyclic sequential circuits C_1 and C_2 with load-enabled latches, such that C_1 has been obtained from C_2 by retiming and combinational synthesis transformations. Suppose F_1 and F_2 are their EDBFs as computed by the algorithm of Figure 9.8. Then $F_1 \equiv F_2 \Leftrightarrow C_1 \equiv C_2$.*

Proof:

$\Rightarrow$

Assume $C_1 \neq C_2$. Then there exists an input sequence π, such that $C_1(\pi) \neq C_2(\pi)$. Without loss of generality, we assume that for some output k, $C_{1_k}(\pi) \neq \perp$ and $C_{1_k}(\pi) \neq C_{2_k}(\pi)$. Now, since $C_{1_k}(\pi) \neq \perp$, it implies that for π, all the enable signals for the k^{th} output must be active in the sequence they appear in the circuit. Since the number of latches and the enable sequence must be the same in C_1 and C_2 (from Lemma 2), $C_{2_k}(\pi) \neq \perp$. Hence using π, we can create an EDBF minterm m, such that $F_1(m) \neq F_2(m)$.

$\Leftarrow$

Assume $F_1 \neq F_2$. Since the number of latches and sequence of enable signals is same for C_1 and C_2 (from Lemma 2), the support variable set is identical for F_1 and F_2. Consider an EDBF minterm m such that $F_1(m) \neq F_2(m)$. The minterm m can be used to generate a sequence of events and input values such that when applied to the circuits, C_1 and C_2 will result in different outputs. ■

Unlike the regular latch case, the result does not hold for any two sequentially equivalent circuits. This is illustrated by following two examples.

In Figure 9.10, two sequentially equivalent circuits are presented. However, their EDBFs would be different since the enable signal of latch L_1 is different in the two circuits. The EDBF for the outputs O_1 and O_2 can be given as following:

$$O_1 = c(\eta[a(\tau), a(\tau-1)b(\tau-1)]) \tag{9.2}$$

$$\begin{aligned} O_2 &= c(\eta[1, a(\tau-1)b(\tau-1)]) \\ &= c(\eta[a(\tau-1)b(\tau-1)]) \end{aligned} \tag{9.3}$$

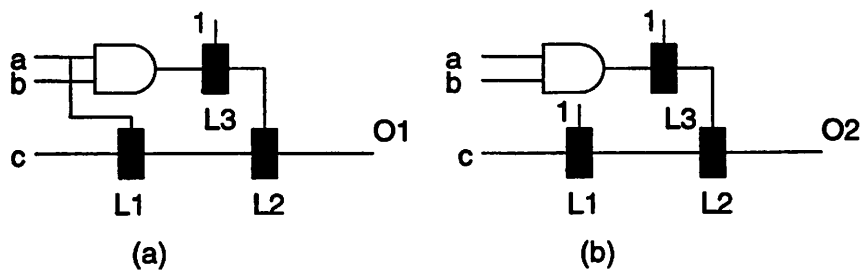


Figure 9.10 EDBF can lead to false negatives: illustration I.

Our technique will result in a false negative since the events defining the time instant for the value of c are syntactically different even though the definition is the same. We can work around this problem by rewriting our events. For example, it can be proven that,

$$p(\tau) \geq q(\tau) \;\Rightarrow\; \eta[p(\tau), q(\tau-1)] = \eta[q(\tau-1)] \tag{9.4}$$

Applying (9.4), on (9.2) (with $p(\tau) = a(\tau)$ and $q(\tau) = a(\tau)b(\tau)$), we get,

$$\begin{aligned} O_1 &= c(\eta[a(\tau-1)b(\tau-1)]) \quad \text{From (9.4)} \\ &= O_2 \end{aligned}$$

This rewriting rule extends the applicability of our technique. However, this rule is not complete, as shown by the next example. In Figure 9.11, (a) and (b) are two sequentially equivalent circuits. In this case, the enable signals to both the latches are the same. However, the data inputs to the latches are different. The EDBF representation for these two circuits are following:

$$\begin{aligned} O_1 &= b(\eta(\bar{a}+b)) \\ O_2 &= a(\eta(\bar{a}+b)) + b(\eta(\bar{a}+b)) \end{aligned}$$

This results in a false negative. Essentially, in this example there is some interaction between the enable and the data signals of the latch, resulting in equivalent sequential functionality even though the EDBFs are different. To handle these cases, we need

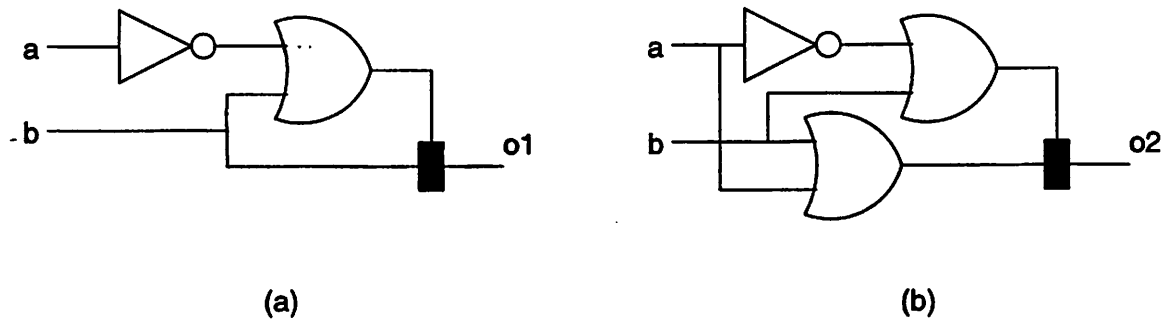


Figure 9.11 EDBF can lead to false negatives: illustration II.

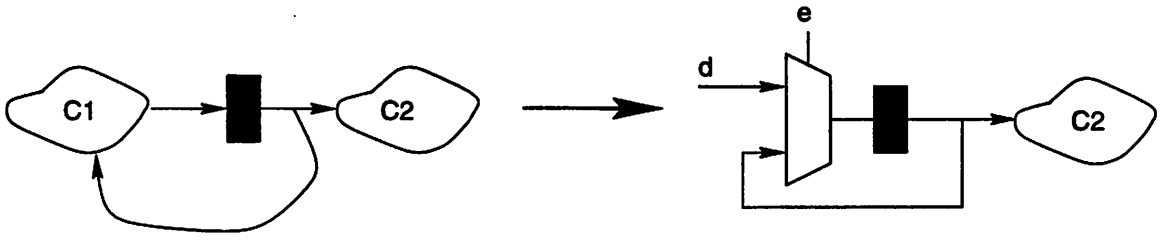


Figure 9.12 Modeling feedback path for a latch with enable and data signals.

to establish equivalence not only between different forms of events, but also between different forms of event/data interaction. Until then, our methodology for circuits with load enabled latches provides only a conservative check.

9.6 Sequential Circuits with Feedback

In these circuits there exists a feedback path for some latches. Our strategy is to model a latch with feedback in the form of an enabled latch with appropriate enable and data signals as shown in Figure 9.12. Next we derive the conditions under which this modeling is feasible.

Lemma 3 *Suppose the next-state function of a latch x given as* $F(x)$. *Then* $F(x) = e \cdot d + \bar{e}x \Leftrightarrow F_{\bar{x}} \subseteq F_x$, *i.e.,* $F(x)$ *can be decomposed in the form of Figure 9.12 if and only*

if $F(x)$ is positive unate in x. Note that e and d are independent of x.

Proof:

$\Rightarrow$

$F_x = ed + \bar{e} \supseteq ed = F_{\bar{x}}$.

$\Leftarrow$

Let $e = \bar{F}_x + F_{\bar{x}}$ and $d = F_{\bar{x}}$. Then,

$$\begin{aligned} ed + \bar{e}x &= (\bar{F}_x + F_{\bar{x}})F_{\bar{x}} + xF_x\bar{F}_{\bar{x}} \\ &= xF_x + \bar{x}F_{\bar{x}} + xF_{\bar{x}} \\ &= F + xF_{\bar{x}} \\ &= F \quad \text{(For a positive unate function)} \end{aligned}$$

■

As a matter of fact, any d, which satisfies,

$$F_{\bar{x}} \subseteq d \subseteq F_x \tag{9.5}$$

can be used as the data signal. The value of e, on the other hand, is unique as shown below.

Since F is a positive unate function in x, we can represent F as $F(x) = Ax + B$.

Now,

$$e \cdot d + \bar{e}x = Ax + B$$

Equating the cofactors with respect to x, we get,

$$\begin{aligned} \bar{e} + d &= A + B \\ \Rightarrow e\bar{d} &= \bar{A}\bar{B} \\ \Rightarrow e &\supseteq \bar{A}\bar{B} && (9.6) \\ e \cdot d &= B \\ \Rightarrow e &\supseteq B && (9.7) \\ \Rightarrow e &\supseteq \bar{A} + B \quad \text{Adding 9.6 and 9.7} && (9.8) \end{aligned}$$

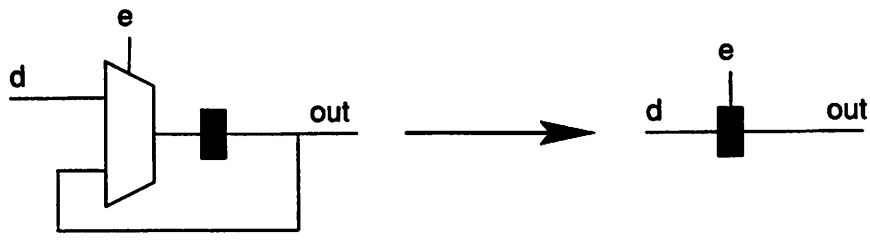


Figure 9.13 Modeling an enabled latch with extra logic.

Also, the flexibility in representing e is given by the following equation,

$$\begin{aligned} A\bar{B} \subseteq\ & \bar{e} \subseteq A+B \\ \Rightarrow \bar{A}\bar{B} \subseteq\ & e \subseteq \bar{A}+B \\ e\ =\ & \bar{A}+B \quad \text{From 9.8 and 9.9} \\ =\ & \bar{F}_x + F_{\bar{x}} \end{aligned} \tag{9.9}$$

Thus for latches whose next-state function is positive unate in the latch variable, the feedback can be modeled via a multiplexer. The advantage of the model shown in Figure 9.12, is that a latch fed by a multiplexer can be thought of as an enabled latch as shown in the Figure 9.13. This gets rid of the feedback path and for our purposes the circuit becomes acyclic. Now we can apply the analysis techniques developed in Section 9.5.2 for acyclic circuits with enabled latches. However, we need to be aware of following issues:

1. The data-input and the enable signal both need to be independent of the latch signal, else it will create a cycle.

2. The data value d obtained from the function $F = ed + \bar{e}x$ is not unique as shown in (9.5) since d has e as don't care. Hence for two circuits C_1 and C_2 we can come up with different decompositions leading to false negatives. This is the basis behind the counterexample in Figure 9.11, where the decomposition of the next-state function $ax+b$ is different for the two circuits. This can be handled in following ways:

(a) By fixing the latch modeling in the circuit, i.e., once we model the feedback path of a latch by an enabled latch, we restrict the logic optimization of the feedback logic by not using e as don't care and also, we move the latch in tandem with the logic for the enable signal. This will guarantee the event correspondence in two circuits. However, by preserving the multiplexor logic we incur some optimization penalty.

(b) By using the lower limit of the possible data signal, i.e., $d = F_{\bar{x}}$. This guarantees the matching of the enable signals, but an optimization penalty may be incurred.

(c) Perform a canonical decomposition of the enable and data signals. Below we give a sufficient condition for such decomposition.

Lemma 4 *Given a function $F = Ax + B$, suppose (e,d) and $(e,\hat{d})$ are two decompositions such that e and d have disjoint Boolean supports. Then $d = \hat{d}$, i.e., there is a unique decomposition of F such that d and e have different supports (if such decomposition exists).*

Proof: We have,

$$\begin{aligned} ed + \bar{e}x &= e\hat{d} + \bar{e}x \\ \Rightarrow ed\bar{x} &= e\hat{d}\bar{x} \\ \Rightarrow ed &= e\hat{d} \end{aligned} \tag{9.10}$$

Equation 9.10 follows from the fact that ed is independent of x. Since e and d have different supports (and so have e and $\hat{d}$), from (9.10) d and $\hat{d}$ must have the same support. Suppose X and Y are the support sets for e and $d, \hat{d}$ respectively.

Assume $d \neq \hat{d}$. Then there exists a minterm y on the Y variables such that $d(y) \neq \hat{d}(y)$. Choose an arbitrary minterm x on variables of X such that $e(x) = 1$. Suppose $(x \cup y)$ is the minterm on X and Y variables.

$$\begin{aligned} d(y) &= d(x \cup y) \\ \hat{d}(y) &= \hat{d}(x \cup y) \\ e(x) &= e(x \cup y) = 1 \end{aligned}$$

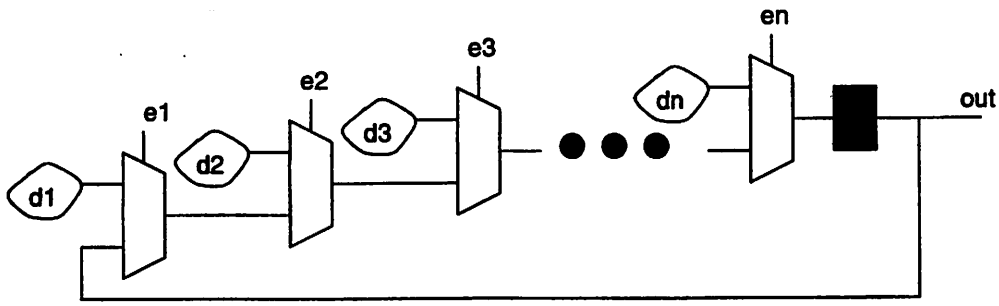


Figure 9.14 Conditional updating of the latch content.

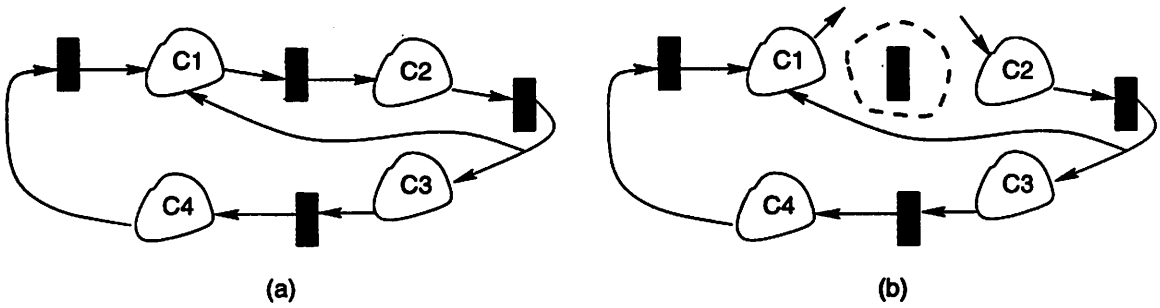


Figure 9.15 Making some latches observable to meet the feedback criterion.

Since $d(y) \neq \hat{d}(y)$, thus $e(x \cup y)d(x \cup y) \neq e(x \cup y)\hat{d}(x \cup y)$. This contradicts (9.10). ∎

The feedback modeling as derived in Figures 9.12 and 9.13 is best suited for the class of circuits where latches update their values when a set of conditions is met, else they keep their previous values. This is illustrated in Figure 9.14. The latches with feedback paths, for which we cannot derive the enabled latch model, are handled in the following way. We find a minimum number of latches that need to be exposed, i.e., need to be made observable, in order to remove the feedback path for these latches. This is illustrated in Figure 9.15. By exposing latches, we treat their outputs as primary inputs and hence the feedback paths are broken, i.e., we cut the latches from the circuit.

After finding the minimal set of latches to be exposed, we impose constraints on the synthesis step such that these latches cannot be moved during retiming.

9.7 Experimental Setup

9.7.1 Circuit Modification

Given a sequential circuit C, we create a directed graph $G = (V, \vec{E})$ in the following manner. For each combinational gate, latch, primary input and primary output we create a node. An edge from node v_i to v_j is created if there is a fanout from gate/latch/primary-input i to gate/latch/primary-output j. The graph in general has cycles due to feedback paths to latches. In the current work we have not implemented the technique to identify the latches with feedback paths that satisfy the criterion mentioned in the previous section. Instead, we obtain a minimal set of latches to expose such that the circuit becomes acyclic*. The problem of finding the minimum set of vertices to make the circuit acyclic "minimum feedback vertex set problem" which is NP-complete. We used a modified version of the heuristics given in [LR90].

9.7.2 Retiming

Retiming was done using *Minaret* [MS97]. This tool only supports the constant delay model (we could not find any efficient public domain retiming tools, which supported better delay models). Retiming was performed in two modes. First, the minimum feasible period was obtained and the area of the circuit was optimized for this period. In the second mode, the delay obtained through combinational optimization was used as the timing constraint and then constrained minimum area retiming was performed.

We could not find a public domain retiming tool which could handle latches with enable signals as proposed in [LVW97] and shown in Figure 9.16.

9.7.3 Combinational Optimization

We perform combinational optimization to obtain a minimum delay circuit. SIS [SSL+92] was used for synthesis purposes. A modified version of "script.delay" was used as

*Note that, in the presence of such a technique, we need to obtain the minimal set of latches to break cycles for only remaining set of latches.

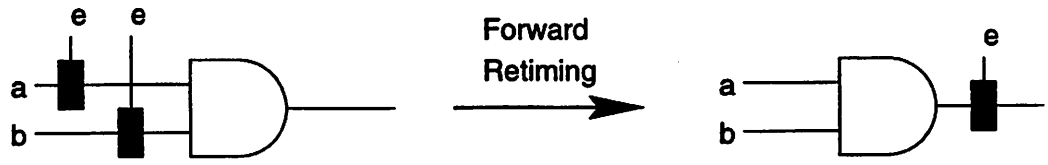


Figure 9.16 Retiming enabled-latch across gates.

shown in Figure 9.17. The modifications were made because the original script was not able to handle large designs (or took very long to complete).

As mentioned earlier, the unit delay model was used during retiming. Hence for consistency we used the unit delay model during synthesis steps as well. To keep the size of gates small, we created a library consisting of inverter, 2-input nand and 2-input nor gates only. Also, for reasonable optimization results we limited the number of fanouts for each gate to four. The delay models and the fanout limitation changes were achieved by appropriately modifying the library.

9.7.4 Generating Equivalent Combinational Equivalence Problems

In order to leverage from the existing combinational equivalence tools, we mapped the equivalence problem of CBF/EDBFs into combinational equivalence problems. This was done by creating a combinational circuit with appropriate variables which represents the CBF or EDBF. An illustration is shown in Figure 9.18. The combinational circuit Figure 9.18(b) represents the CBF for the sequential circuit Figure 9.18(a). Essentially, if the circuit outputs depend on the value of a signal at k different time instants (for a circuit with regular latches) or with k different enable signal paths (for a circuit with enabled latches), the cone of logic for the signal is replicated k times. The size of these circuits could become large due to replications. Note, however, that this step is performed only for convenience (to treat the combinational equivalence checker as a black box). In practice, a modified combinational equivalence checker could be used which would not require generation of such circuits and hence no blow-up would occur.

The combinational verification was performed by an in-house tool similar to the ones presented in [Mat96, KK97].

```
sweep
decomp -q
tech_decomp -o 2
resub -a -d
sweep
reduce_depth -b -r
eliminate -l 100 -l
simplify -l
sweep
decomp -q
fx -l
tech_decomp -o 2
rlib mylib2.genlib
rlib -a lib2_latch.genlib
map -s -n 1 -AFG -p -B -b 1000
print_delay -p1 -a -m unit
```

Figure 9.17 Script for synthesizing minimum delay circuit.

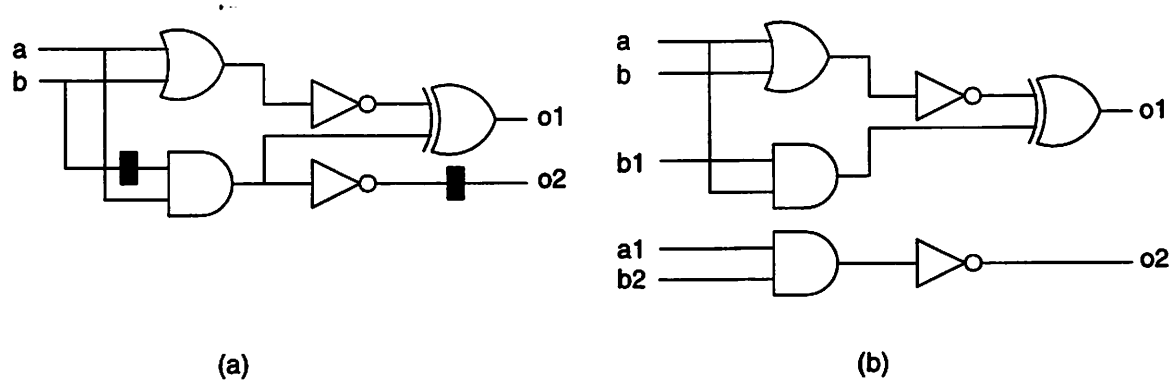


Figure 9.18 Generating equivalent combinational equivalence problems.

9.8 Experimental Results

Our experiment consisted of following steps (see also Figure 9.19).

1. Given the sequential circuit (A), modify it appropriately to satisfy constraints on all feedback paths to obtain a new circuit (B). This is done by creating a circuit graph and finding a minimal feedback vertex set. Due to lack of a retiming tool which could handle load-enabled latches, we did not model any latches with feedback path as load-enabled latches (as shown in Figure 9.12). In general, this leads to fewer latches that need to be exposed.

2. Perform synthesis for delay optimization and min-period retiming on the modified circuit (B) to obtain a new circuit (C).

3. To illustrate the advantage of combining retiming with combinational synthesis, we also performed pure combinational optimization (using the same script) on the original circuit (A) to obtain circuit (D).

4. We also compared the saving in area by performing constrained minimum area retiming. This was done on circuit (B) with the delay value of circuit (D) to obtain a new circuit (E).

5. To measure the loss in optimization due to modification in step 1, retiming and synthesis optimization on the original circuit (A) was performed to obtain an optimized circuit (F).

6. Step 5 was repeated to measure the loss of optimization in circuit (E). This was done by performing constrained minimum area retiming on (A) with the delay value of circuit (D) to obtain a new circuit (G).

7. Combinational circuits (H and J) were created (as described in Section 9.7.4) to obtain circuits (B) and (C) respectively.

8. Perform combinational verification between (H) and (J). Verifying equivalence of circuits (B) and (E) would be similar and is not done in the experiment.

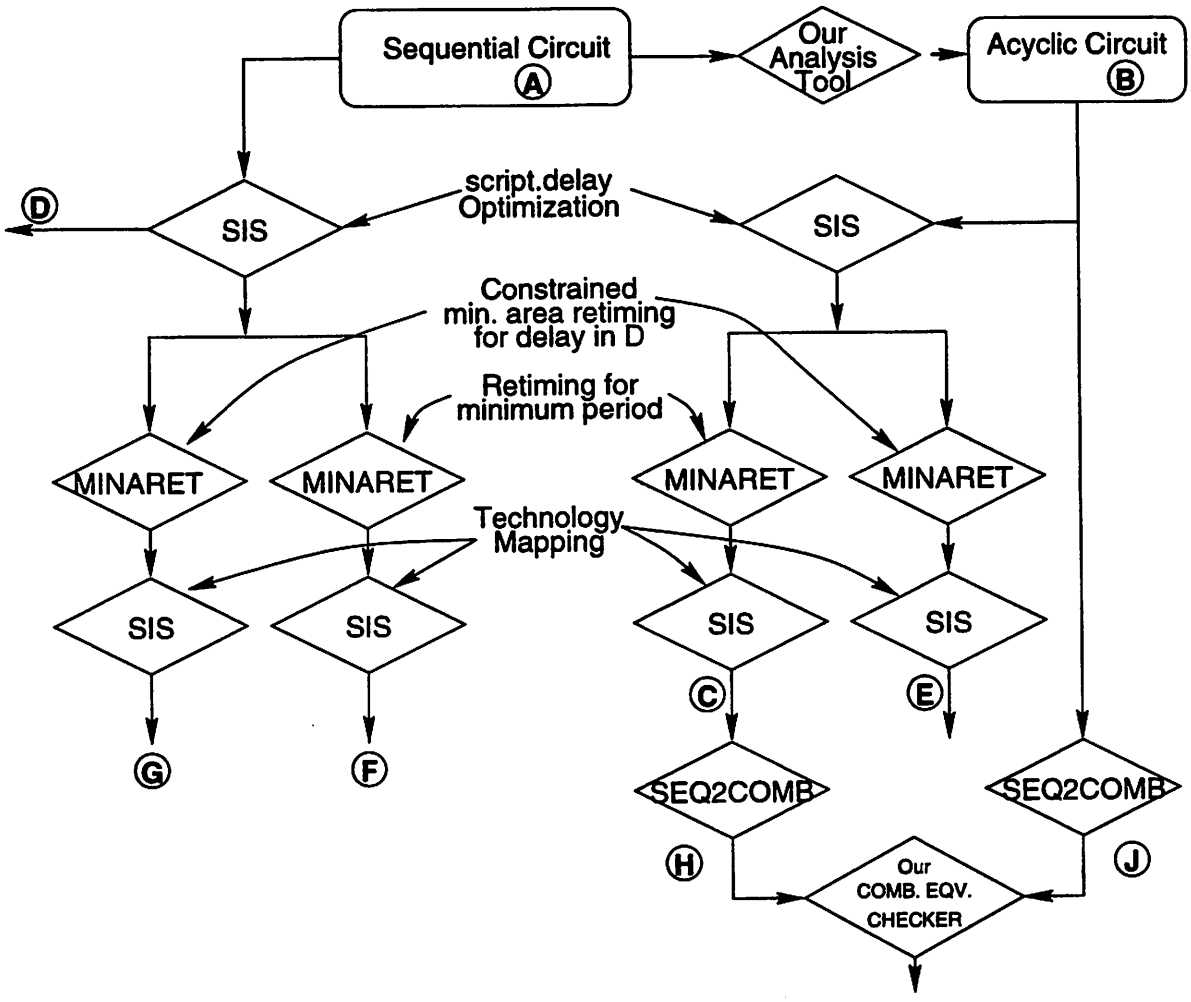


Figure 9.19 Flow chart indicating experimental set up.

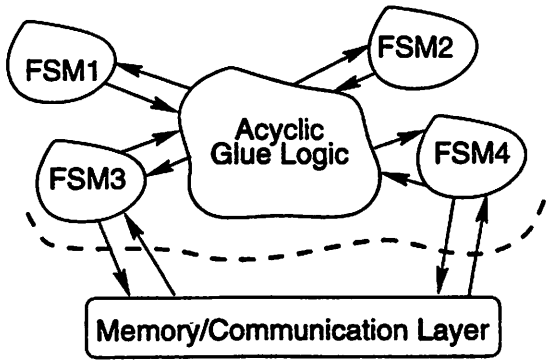


Figure 9.20 Feedback paths due to interaction with memory and communication layer.

The active area and delay numbers are obtained by the "map" command. The verification was performed on an UltraSparc-1 with 256MB of memory.

In Table 9.1, we have given results comparing the optimization potential of our strategy and also the corresponding verification times.

All the industrial circuits we investigated contained load-enabled latches. Since we did not have access to a retiming tool for circuits with load-enabled latches, we could not perform retiming on these circuits and hence could not get optimization and verification results. However, we did extensive analysis on them to understand the nature of feedback paths to latches. After analyzing a set of circuits we observed that most of the feedback paths exist due to interaction with memory and communication layer as shown in Figure 9.20. Typically, designers want to keep the boundary between the design and communication layer/memory preserved and they do not synthesize them together. We can take advantage of this fact and can assume for our purposes that these feedback paths do not exist. In Table 9.2, we have given the number of latches exposed in order to satisfy the feedback path constraint. Currently we do only structural analysis which can detect the kind of circuits as shown in Figure 9.14. A more detailed functional analysis (based on the next-state function of the latches as explained in Section 9.6) would lead to reduced number of exposed latches.

Circuit	A	F			C				D		G		E		Time
	#L	#L	Area	δ	%	#L	Area	δ	Area	δ	#L	Area	#L	Area	H vs J
minmax10	30	30	0.87	50	66	30	0.74	50	1.00	56	30	0.87	30	0.74	2
minmax12	36	36	0.87	54	66	36	0.75	54	1.00	57	36	0.87	36	0.75	2
minmax20	60	60	0.83	55	66	60	0.80	64	1.00	94	60	0.83	60	0.80	3
minmax32	96	96	0.81	88	66	96	0.72	96	1.00	145	96	0.81	96	0.72	5
prolog	65	85	1.06	15	43	65	0.97	17	1.00	18	65	1.00	65	0.97	7
s1196	18	18	1.00	21	0	18	1.00	21	1.00	21	18	1.00	18	1.00	5
s1238	18	18	1.00	19	0	18	0.99	19	1.00	19	18	1.00	18	0.99	7
s1269	37	69	1.12	24	75	37	0.98	32	1.00	33	37	1.00	37	0.98	6
s1423	74	78	1.01	40	95	74	1.00	44	1.00	45	74	1.00	74	1.00	6
s3271	116	221	1.18	16	94	116	0.99	25	1.00	26	116	1.00	116	0.99	7
s3384	183	174	0.97	30	39	154	0.95	56	1.00	57	154	0.95	154	0.95	34
s400	21	32	1.17	11	71	21	0.98	13	1.00	14	18	0.95	21	0.98	1
s444	21	32	1.17	10	71	21	0.99	13	1.00	14	18	0.95	21	0.99	1
s4863	88	146	1.05	32	18	142	1.04	32	1.00	58	78	0.99	83	0.99	4:25
s641	19	19	1.00	24	78	19	1.00	24	1.00	24	19	1.00	19	1.00	1
s6669	231	272	1.02	32	17	234	1.00	55	1.00	85	193	0.97	214	0.98	1:54
s713	19	19	1.00	25	78	19	0.96	24	1.00	25	19	1.00	19	0.96	1
s9234	135	169	1.05	21	66	144	1.00	27	1.00	30	128	0.98	135	0.98	22
s953	29	22	0.95	16	20	29	1.00	14	1.00	16	22	0.95	29	1.00	3
s967	29	22	0.95	15	20	29	1.00	14	1.00	15	22	0.95	29	1.00	3
s3330	65	78	1.04	15	43	65	0.96	17	1.00	19	65	1.00	65	0.96	7
s15850	515	537	1.01	34	72	515	1.00	46	1.00	46	495	0.99	515	1.00	11:24
s38417	1464	1285	0.96	33	70	1463	1.03	36	1.00	37	1248	0.95	1463	1.03	15:32

Table 9.1 Results on sequential optimization and verification.

A: Original circuit

B: Modified circuit (not shown in the table)

C: Obtained from B after retiming (for minimum period) and synthesis

D: Obtained from A after Combinational optimization only

E: Obtained from B after retiming (for delay in D) and resynthesis

F: Obtained from A after retiming (for minimum period) and synthesis

G: Obtained from A after retiming (for delay in D) and resynthesis

H: Combinational circuit for the CBF for circuit B

J: Combinational circuit for the CBF for circuit C

L: Latches

%: Percentage of latches exposed in B

Area/δ: Area (normalized against D) / Delay of the circuit

H vs J: CPU time (in minutes:seconds) for combinational verification between H and J

Example	# Latches	# Exposed
ex1	2157	934
ex2	100	16
ex3	146	56
ex4	1437	835
ex5	672	305
ex6	412	250
ex7	453	81
ex8	968	470
ex9	783	15
ex10	634	174
ex11	792	369
ex12	2206	691

Table 9.2 Number of latches exposed for some industrial circuits.

9.8.1 Analysis

By analyzing the data given in Tables 9.1 and 9.2 we make following observations:

1. Comparing δ values in columns C and D, for most of the circuits the delay values obtained through our approach is better than that by purely combinational optimization. In some cases delay values reduces by as much as 50%. The area penalty incurred in the process is negligible.

2. Comparing area numbers in columns D and E, for the same delay, retiming allows us to reduce the area.

3. The verification times were quite reasonable. Most of the examples took less than a minute to verify. The maximum time taken is fifteen minutes. Note that, for only a few of these sequential circuits the state-space can be traversed, and for fewer yet the state-space of the product machine can be traversed. This makes the proposed technique quite attractive.

4. Comparing the area numbers in columns D and E, we observe that the penalty paid in terms of reduced optimization capability was not significant in most of the cases.

5. Looking at the data for industrial circuits from Table 9.2, we observe that even though these circuits are highly control intensive implying a relatively tight interaction among latches, we did not need to expose more than 50% latches and sometimes as few as 2% latches were exposed. As mentioned, these numbers will decrease when positive unateness is used.

9.9 Conclusions, Related Work, and Future Directions

We proposed a practical verification technique for circuits which have undergone retiming and combinational synthesis transformations. In particular, we show that the corresponding sequential verification can be reduced to an extension of combinational verification. The proposed technique can deal with circuits with and without feedback paths, and with regular and load-enabled latches. We impose a constraint on the feedback path (if one exists) of latches. If these constraints are not met by the original circuit, we fix the position of some of the latches to cut the feedback paths. Experimental results indicate that imposing constraints does not result in significant optimization penalty. Our strategy can be used to obtain faster circuits by allowing the retiming transformations while performing fast verification as indicated by our experimental results.

In [BBJR97], implementation verification of the bus interface unit for the Alpha 21264 microprocessor is performed. More specifically, gate-level extraction of custom-designed transistor level schematics is verified against the RTL. They used a "retiming" comparison algorithm for verifying acyclic sequential circuits. In particular, they gave a similar algorithm as that in Figure 9.8 for computing the output functionality. However, no formal framework for such verification is presented and no technique to handle circuits with feedback paths is presented (as given in Section 9.6). Nonetheless, their work presents another application of our technique.

To make our approach exact for arbitrary sequential optimizations, we need to develop a complete technique to distinguish events and combination of events and signals.

Also, a better technique could be used to find the latches to be exposed. The strategy of finding the minimum latches may not always be optimum for area/delay optimizations. The need would be to identify latches, such that the least amount of area/delay penalty is paid by exposing them.

Chapter 10

Conclusions and Future Directions

THE design and implementation verification of digital circuits is becoming a critical aspect of the design methodology. As the circuit complexity is growing the need for efficient verification algorithms has increased dramatically. Simulation, which has traditionally been employed for verification purposes, cannot be relied on by itself due to its poor coverage of the system behavior. Formal verification, a technique which uses mathematical analysis to establish relationships between appropriate mathematical models of a design and its desired properties, has been emerging as a complementary alternative to simulation. However, the current state-of-the-art formal verification techniques are limited in their applicability to only small designs.

In this thesis we have presented a spectrum of techniques to radically improve the efficiency of various verification algorithms. This is critical to meet the performance demand on verification techniques and in particular to make formal verification a viable technology for practical applications. Below we summarize the key contributions of this work.

BDD manipulation For the underlying data-structure, binary decision diagram, we have presented computer architecture based techniques for efficient manipulation. In particular, exploiting memory hierarchy is shown to be a promising direction to achieve high performance. Due to the large difference in access times between various levels in memory hierarchy, the locality of access plays a critical role in the overall run time.

The basic idea is to reorganize the computation to achieve memory locality, in particular by converting a recursive procedure into an iterative one. The depth-first traversal of operand BDDs is replaced by the breadth-first traversal, which when coupled with customized memory management shows improved locality. Further performance improvements are obtained by identifying locality across

several independent and several dependent BDD operations using the notions of superscalarity and pipelining. Overall we observed a performance improvement of up to 100 for BDD sizes that do not fit the main memory. Even for the BDD sizes that fit within the main memory, about 50% performance improvement was observed.

We also demonstrated that it is possible to use a collection of main memories and disks on a network of workstations to improve the performance and build BDDs that do not fit in the memory of one workstation. We see a performance improvement by a factor of up to 6 on a Myrinet-connected cluster of four workstations by using remote memory as a paging device for the local memory.

We have argued that a multi-threaded breadth-based manipulation based BDD package running on a shared-memory multiprocessor has the highest potential for effectively utilizing the parallel architecture.

Since dynamic ordering constitutes an essential component of a BDD package, we have investigated dynamic ordering schemes inside a breadth-first manipulation based package and shown its performance to be at par with reordering in a depth-first manipulation based packges.

State space traversal State-traversal constitutes an important step in the verification of sequential circuits. Towards that, we have presented techniques for compact state-transition graph representation and traversal. We established that the core computation in BDD-based formal verification is that of forming the image and pre-image of a set of states under the transition relation characterizing the system. To make this step efficient, we addressed the use of clustered transition relations, ordering of clusters for early variable quantification, network partitioning, use of don't cares, and removal of redundant latches.

Sequential circuit verification The high complexity of general sequential circuit verification makes any sequential optimization in digital circuit unattractive. In this work, we have shown that with appropriate constraints, the implementation verification of circuits which have undergone iterative retiming and combinational synthesis transformations can be reduced to an extension of the combinational

verification problem. This enables exploitation of the advancements made in the field of combinational verification and use of its powerful techniques to perform verification.

10.1 Analysis and Future Directions

Efficient BDD evaluation: The evaluation of a BDD for a particular input minterm involves tracing one of the paths in BDD from the root node to the leaf. In other words, performing simulation on a BDD requires accesses to memory locations holding BDD nodes. The ideas behind localizing memory accesses during breadth-first BDD manipulation can be applied in this case as well. This can be achieved in following two ways:

Evaluation of multiple input vectors: In many simulation environments, it is required to simulate multiple input sequences. For a Boolean function on n variables, simulating one vector requires n BDD node accesses. Suppose we need to simulate k input sequences each of length m. If we simulate each of k input sequences one after another one vector at a time, we need to traverse paths in the BDD $k \cdot m$ many times. Each path traversal requires accessing one node from each level. If these accesses are non-local, there will be order of $O(n \cdot k \cdot m)$ non-local accesses in the complete simulation.

The node allocation scheme of breadth-first BDD manipulation technique combined with the simultaneous evaluation of all the input sequences will lead to better locality. Essentially, the BDD nodes are laid out in the memory such that nodes of an index are on the same page or set of pages. By maintaining an array of length k, and keeping track of evaluation nodes for all k sequences, we can obtain simulation results of one vector for all k sequences in one pass of the BDD. Assuming that we incur a non-local access in going from one level to another, there will be order of $O(n \cdot m)$ non-local accesses in complete simulation.

Note that, since there will be overheads associated with the book-keeping of multiple vectors, the technique will be useful if the number of input sequences to be simulated is large.

Evaluation of single input vector: In those cases, where we need to simulate one long input vector sequence, the above approach will lead to unnecessary overhead. In this case we propose the following strategy.

The goal is to optimally layout the BDD nodes in the memory such that some objective function based on non-local accesses along various paths of the BDD is optimized.

We can treat the optimal layout of BDD nodes in the memory as a graph problem by considering the whole BDD as a directed acyclic graph $\vec{G} = (V, \vec{E})$. In a simplistic view, we would like to assign each BDD node to a physical page in the memory. This corresponds to assigning a number to each node in the graph G. Assume the possible range of such numbers (based on the number of pages in the main memory) is N. Suppose $f : V \mapsto N$, is an onto function, mapping each node to a particular page. Since there is a limit on number of nodes that can fit in a page, appropriate constraints can be imposed on the mapping process.

We define a distance metric d_{ij} between two vertices v_i and v_j such that:

$$d_{i,j} = \begin{cases} 1 \text{ if } e_{ij} \in E \text{ and } f(v_i) \neq f(v_j) \\ 0 \text{ otherwise} \end{cases}$$

A path in the graph is defined by an ordered list of vertices $(v_0, v_1, v_2, \ldots, v_n)$, such that $\forall i, i = 1, \ldots n$, v_i is the child node of v_{i-1}. Suppose $P_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ is a path in the graph between vertices v_i and v_j. We define the path weight as

$$W_{P_{ij}} = \Sigma_{k=i}^{k=j-1} d_{k,k+1}$$

Note that there can be multiple paths between two vertices in the graph. The weight for a path indicates the number of times page change will occur in traversing the nodes along that path.

One plausible objective function is to minimize the maximum weight along any path, i.e.,

$$\text{Minimize} \left(\max_{i,j,k} W_{P_{ij}^k} \right)$$

The superscript k denotes the existence of multiple paths between the vertices v_i and v_j.

The other objective function (but not so practical) could be to minimize the weight along any path in the graph.

This paradigm of allocating nodes in the memory can be extended in two ways.

1. We can attach probabilistic attributes to the edges reflecting the probability of traversing that path during simulation.
2. We can target various levels of memory hierarchy simultaneously. In particular, we can formulate the node layout problem such that the number of cache misses is minimized. We can also formulate a combined objective function, which first tries to minimize the cache misses and then targets the optimal page assignment. We can associate different distance metrics to these assignments to reflect the large difference in the miss penalties.

Minimizing cache misses: As we discussed in Section 1.2.3 on page 5 in Chapter 1, the performance gap between the processor and the main memory is increasing over the years. To deal with this gap, one can devise algorithms which lead to fewer cache misses, reducing the need to access the main memory. In Section 3.8.5 on page 71 in Chapter 3, we discussed one possible way to reduce the cache misses during the REDUCE phase of the breadth-first BDD manipulation. Some other possible avenues are using a different BDD node data structure and unique table management.

Keeping BDD complexity under control: The performance demand on verification algorithms is growing at a rate that outpaces the improvements in the semiconductor technology. Roughly speaking, the complexity of BDD algorithms increase at least quadratically with increase in the number of variables. This is based on a conservative assumption that the sizes of BDDs grow linearly with the number of variables and typical Boolean operation complexity is of the order of the product of the sizes of the operands. The trend in the number of transistors has been that it quadruples every three years. This implies the number of

variables for BDD representation also quadruples every three years. The BDD complexity from a conservative estimation would grow at least sixteen times every three years. The performance of a microprocessor improves by a factor of only about 3.37 every three years. This indicates that a BDD based technique which does not scale with the size of the system, will run out of steam over years.

Therefore, the need is to find algorithms which apply the BDD-based techniques after suitably scaling down the problem size – either by abstraction or by appropriate partitioning – such that the BDD size and hence the algorithm complexity remains under control.

Complexity issues in retiming and synthesis transformations: A number of issues are still unresolved. In particular:

1. The exact optimization potential of arbitrary number of retiming and synthesis transformations is unknown.
2. The number of retiming and synthesis transformations required to obtain the most optimum circuit possible is unknown. More precisely: suppose S indicates the optimization space of retiming and synthesis transformations; does there exist a finite number k, such that k transformations of retiming and resynthesis can explore all of S.
3. The complexity of establishing if a circuit C_2 has been obtained from C_1 using only retiming and synthesis transformations is unknown. We conjecture that this complexity will be dependent on the point 2.

Hybrid methods: One can take a hybrid approach in two directions. The first one would be to combine theorem proving with automated techniques like model checking. Thereby we can hope to take advantage of the best of both techniques. Some effort has been made in this direction in the past however it did not gain mainstream success. For a successful integration of techniques the driving paradigm should always be to keep the methodology as simple as possible.

Another useful observation is that formal verification and random simulation present two extremes of the verification methodology. Both have serious draw-

backs making them infeasible for verifying large designs. Another hybrid method is to bring formal techniques into the world of simulation. Some effort in this direction has been made by few researchers [HYHD95, YSAA97].

Part IV

Appendix

Appendix A

CAL BDD Package

THE work on breadth-first BDD manipulation and dynamic ordering (described in Chapters 3 and 6 respectively) has been successfully implemented inside a comprehensive BDD package named CAL [RS97].

In the following we give brief descriptions of the functions which are unique to the CAL BDD package.

- Cal_BddMultiwayAnd(Or/Xor): Given n functions $f_1, f_2, \ldots, f_n$, this function computes $\prod_i f_i$ or $\Sigma_i f_i$ or $\oplus f_i$ respectively.

- Cal_BddPairwiseAnd(Or/Xor): Given a two arrays of BDD operands $f_1, f_2, \ldots, f_n$ and $g_1, g_2, \ldots, g_n$, this function computes $\prod_i (f_i g_i)$ or $\Sigma_i (f_i g_i)$ or $\oplus (f_i g_i)$ respectively.

- Cal_PipelineInit(op): This function initializes the pipeline engine with the designated operation. Until the current pipeline is executed and results are updated, all the operations need to be of the "op" type.

- Cal_PipelineCreateProvisionalBdd: Given the operands, this function creates a provisional BDD representing the result of performing the current pipeline operation on the operands.

- Cal_PipelineExecute: This function executes the current pipeline, evaluates all the operations.

- Cal_PipelineUpdateProvisionalBdd: Given the provisional BDD, it returns the actual resulting BDD. This function should be called after executing the pipeline for the provisional BDDs of interest.

- Cal_PipelineQuit: This function destroys the storage associated with the pipeline. Any un-updated provisional BDDs are freed.

So far three versions of the package have been released in the public domain. The latest version (2.0) is available at:
`http://www-cad.eecs.berkeley.edu/Research/cal_bdd`
The latest release of the package includes the dynamic reordering routines. This package has been successfully used inside a commercial FSM synthesis tool.

Appendix B

VIS: Verification Interacting with Synthesis

THE work on efficient state space representation and traversal (Chapter 7) was integrated inside a verification tool named VIS (Verification Interacting with Synthesis). * VIS was developed as a joint effort by researchers at University of California at Berkeley and University of Colorado at Boulder. This chapter gives a brief overview of this tool and discusses its features.

The motivating factors behind the creation of VIS were:

1. To provide an integrated environment for synthesis and verification.
2. To create a symbolic model checker using state-of-the-art algorithms.
3. To provide a solid platform for research in verification and synthesis.

The overview of the VIS architecture is given in Figure B.1. Roughly speaking, the front end for VIS allows the traversal in the hierarchy. The verification and synthesis parts respectively allow one to perform verification and synthesis on the sub-tree rooted at the current node in the hierarchy.

VIS was designed to be modular with well defined packages to handle various features of the tool. In Figure B.2, we outline the flow of a work session inside VIS and identify the corresponding packages (shown inside parentheses). The modularity allows a developer to integrate and investigate new ideas for various algorithms inside VIS in a clean fashion.

The VIS tool has been integrated with three different BDD packages [Som97, RS97, Lon93]. So far three releases of VIS have taken place. The latest release of VIS can be obtained from

`http://www-cad.eecs.berkeley.edu/~vis`

*For a detailed description of VIS architecture please refer to [BSA+96b, BSA+96a].

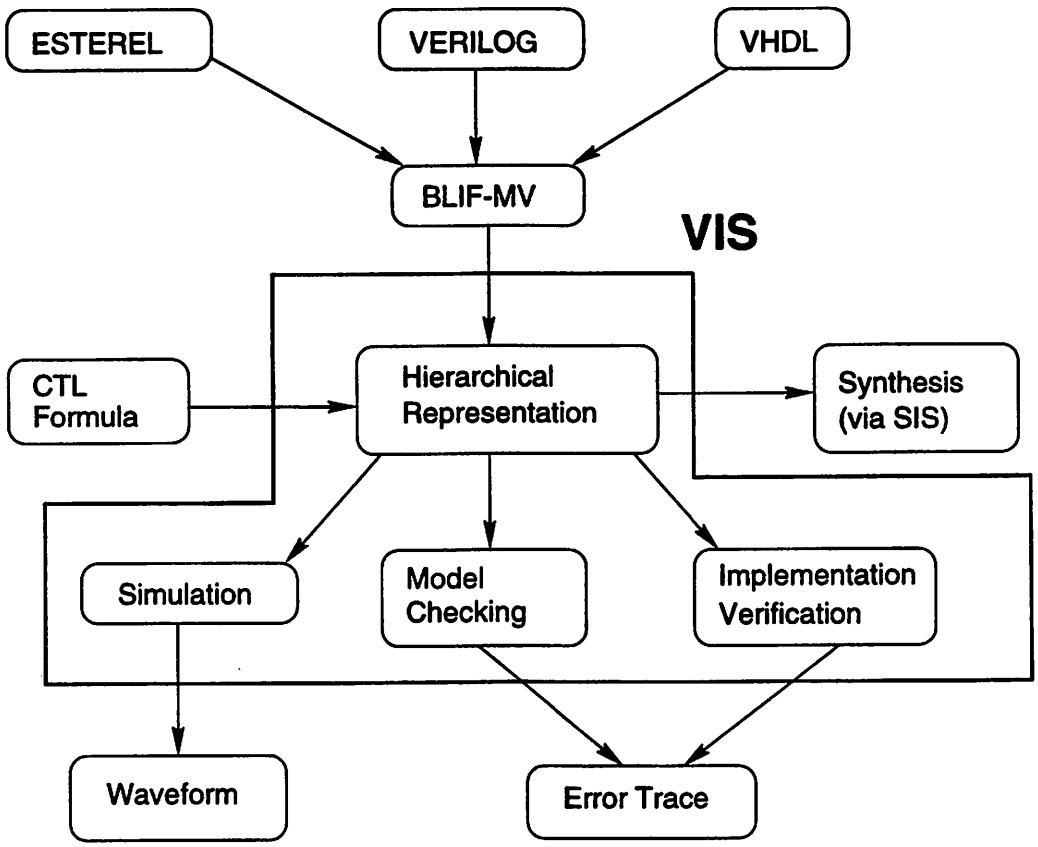


Figure B.1 VIS Overview.

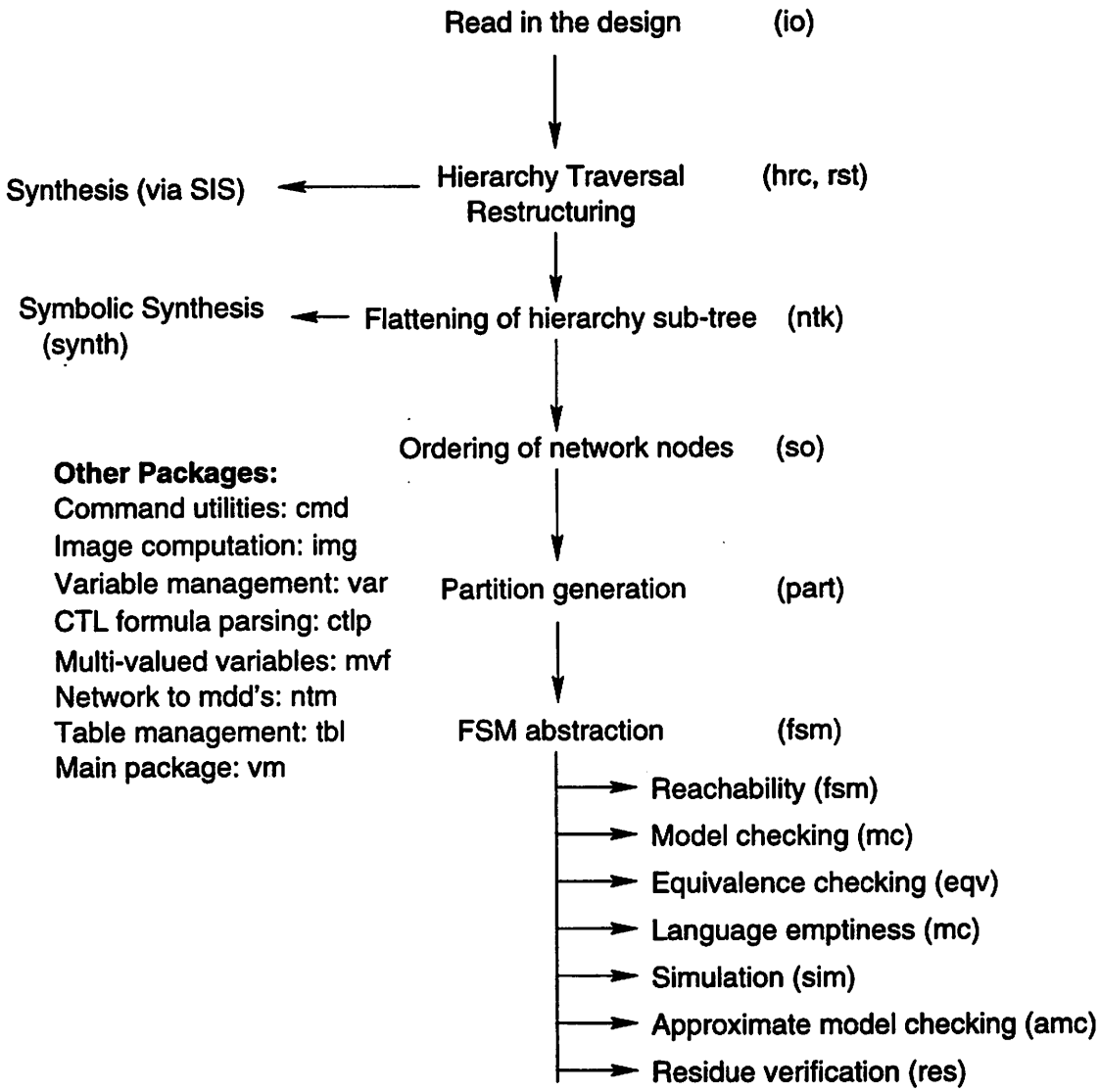


Figure B.2 Verification and Synthesis inside VIS.

VIS has found world-wide success in both industry and academic worlds. It forms the core engine for some commercial verification tools. It is also used as part of the verification courses at various universities. To date, there have been over 500 downloads from all over the world.

Bibliography

[AC94] P. Ashar and M. Cheong. Efficient Breadth-First Manipulation of Binary Decision Diagrams. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 622–27, November 1994.

[ACM96] P. Arunachalam, C. Chase, and D. Moundanos. Distributed Binary Decision Diagrams for Verification of Large Circuits. In *Proc. IEEE/ACM International Conference on Computer Design*, pages 365–70, 1996.

[ACP94] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for NOW: Network of Workstations. Technical Report UCB/ERL M94/58, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, November 1994.

[ADG91] P. Ashar, S. Devadas, and A. Ghosh. Boolean Satisfiability and Equivalence Checking Using General Binary Decision Diagrams. In *Proc. IEEE/ACM International Conference on Computer Design*, pages 259–64, 1991.

[AG89] G. S. Almasi and A. Gottleib. *Highly Parallel Computing*. Benjamin/Cummings, Redwood, CA, 1989.

[AGM96] P. Ashar, A. Gupta, and S. Malik. Using Complete-1-Distinguishability for FSM Equivalence Checking. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, November 1996.

[Ake78] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-37:509–16, June 1978.

[Amd67] G. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Spring Joint Computer Conference*, pages 483–85, 1967.

[ASB93] A. Aziz, V. Singhal, and R. K. Brayton. Verifying Interacting Finite State Machines: Complexity Issues. Technical Report UCB/ERL M93/68, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, September 1993.

[ASS+94] A. Aziz, T. R. Shiple, V. Singhal, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Formula-Dependent Equivalence for Compositional CTL Model Checking. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 324–337. Springer-Verlag, 1994.

[Bai91] D. Bailey. Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers. In *Supercomputing Review*, August 1991.

[BBDG+94] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and System for Practical Formal Verification of Reactive Hardware. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 182–93. Springer-Verlag, 1994.

[BBJR97] G. P. Bischoff, K. S. Brace, S. Jain, and R. Razdan. Formal Implementation Verification of the Bus Interface Unit for the Alpha 21164 Microprocessor. In *Proc. IEEE/ACM International Conference on Computer Design*, 1997.

[BC95] R. E. Bryant and Y.-A. Chen. Verification of Arithmetic Circuits with Binary Momemt Diagrams. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 535–41, June 1995.

[BCL91a] J. R. Burch, E. M. Clarke, and D. E. Long. Representing Circuits More Efficiently in Symbolic Model Checking. In *Proc. of the IEEE/ACM Design Automation Conf.*, June 1991.

[BCL91b] J. R. Burch, E. M. Clarke, and D. E. Long. Symbolic Model Checking with Partitioned Transition Relations. In *Proc. Intl. Conf. on VLSI*, August 1991.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proc. of the IEEE/ACM Design Automation Conf.*, June 1990.

[BFG+93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic Decision Diagrams and Their Applications. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 188–91, 1993.

[Bra93] D. Brand. Verification of Large Synthesized Designs. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 534–7, November 1993.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 40–45, June 1990.

[BRSW87] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-6(6):1062–81, November 1987.

[Bry86] R. Bryant. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35:677–91, August 1986.

[Bry87] R. E. Bryant. Boolean Analysis of MOS Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, pages 634–49, July 1987.

[Bry91] R. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the Association for Computing Machinery*, 38(2):299–328, April 1991.

[Bry95] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 236–43, November 1995.

[BSA+96a] R. K. Brayton, A. Sangiovanni-Vincentelli, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, S. Qadeer, R. K. Ranjan, T. R. Shiple, G. Swamy, T. Villa, G. D. Hachtel, F. Somenzi, A. Pardo, and S. Sarwary. VIS: A System for Verification and Synthesis. In *Proc. of the 8th International Conference on Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, 1996.

[BSA+96b] R. K. Brayton, A. Sangiovanni-Vincentelli, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, S. Qadeer, R. K. Ranjan, T. R. Shiple, G. Swamy, T. Villa, G. D. Hachtel, F. Somenzi, A. Pardo, and S. Sarwary. VIS: Tutorial. In *Proc. Formal Method in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 248–56. Springer-Verlag, 1996.

[BW97] B. Bollig and I. Wegener. Partitioned BDDs vs. Other BDD Models. In *Proc. IEEE/ACM Intl. Workshop on Logic Synthesis*, 1997.

[CBM89] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Based on Symbolic Execution. In J. Sifakis, editor, *Proc. of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–73, June 1989.

[CC93] G. Cabodi and P. Camurati. Exploiting Cofactoring for Efficient FSM Symbolic Traversal Based on the Transition Relation. In *Proc. IEEE/ACM International Conference on Computer Design*, pages 299–303, October 1993.

[CCQ94] G. Cabodi, P. Camurati, and S. Quer. Auxiliary Variables for Extending Symbolic Traversal Techniques to Data Paths. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 289–93, June 1994.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–63, 1986.

[CFZ95] E. Clarke, M. Fujita, and X. Zhao. Hybrid Decision Diagrams: Overcoming the Limitations of MTBDDs and BMDs. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, November 1995.

[CGRR94] G. Cabodi, S. Gai, M. Rebaudengo, and M. S. Reordea. A Data Parallel Approach to Boolean Function Manipulation using BDDs. In *International Conference on Massively Parallel Computer Systems*, 1994.

[CHM+93] H. Cho, G. D. Hachtel, E. Macii, B. Plessier, and F. Somenzi. Algorithms for Approximate FSM Traversal. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 25–30, June 1993.

[CHM+94] H. Cho, G. D. Hachtel, E. Macii, M. Poncino, and F. Somenzi. A Structural Approach to State Space Decomposition for Approximate Reachability Analysis. In *Proc. IEEE/ACM International Conference on Computer Design*, October 1994.

[CM90a] E. Cerny and C. Mauras. Tautology Checking Using Cross-Controllability and Cross-Observability Relations. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 34–37, November 1990.

[CM90b] O. Coudert and J. C. Madre. A Unified Framework for the Formal Verification of Sequential Circuits. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 126–9, November 1990.

[CM95] O. Coudert and J. C. Madre. The Implicit Set Paradigm: A New Approach to Finite State System Verification. In R. K. Brayton, E. M. Clarke, and P. A. Subrahmanyam, editors, *Formal Methods in System Design*, pages 133–145, 1995.

[CMZ+93] E. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J.-Y. Yang. Spectral Transforms for Large Boolean Functions with Application to Technology Mapping. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 54–60, 1993.

[CSG97] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Harware/Software Approach*. Morgan Kaufmann, 1997.

[CYF94] B. Chen, M. Yamazaki, and M. Fujita. Bug Identification of a Real Chip Design by Symbolic Model Checking. In *Proc. European Conf. on Design Automation*, Paris, France, February 1994.

[DB97] R. Drechsler and B. Becker. Overview of Decision Diagrams. In *IEE Proceedings-Computers and Digital Techniques*, pages 187–93, May 1997.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *Proc. IEEE/ACM International Conference on Computer Design*, pages 522–5, October 1992.

[DMN88] S. Devadas, H.-K. T. Ma, and A. R. Newton. On the Verification of Sequential Machines at Differing Levels of Abstraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, pages 713–22, June 1988.

[DST+94] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient Representation and Manipulation of Switching Functions Based on Ordered Kronecker Functional Decision Diagrams. In *Proc. of the IEEE/ACM Design Automation Conf.*, 1994.

[Eic93] T. H. V. Eicken. *Active Messages: An Efficient Communication Architecture for Multiprocessors*. PhD thesis, University of California Berkeley, 1993.

[EL85] E. A. Emerson and C. L. Lei. Modalities for Model Checking: Branching Time Strikes Back. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 84–96, 1985.

[Eme90] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, pages 996–1072. Elsevier Science, 1990.

[Eng95] S. D. Engineering. *Solaris Porting Guide*. Sunsoft Press, 1995.

[FFK88] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 2–5, November 1988.

[FMK91] M. Fujita, Y. Matsunaga, and T. Kakuda. On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis. In *Proc. European Conf. on Design Automation*, pages 50–54, March 1991.

[GB94] D. Geist and I. Beer. Efficient Model Checking by Automated Ordering of Transition Relation Partitions. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1994.

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, September 1994.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Co., 1979.

[GL91] O. Grümberg and D. E. Long. Model Checking and Modular Verification. In J. C. M. Baeten and J. F. Groote, editors, *Proc. of CONCUR '91: 2nd Inter. Conf. on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1991.

[GM94] J. Gergov and C. Meinel. Efficient Boolean Manipulation with OBDDs can be Extended to FBDDs. In *IEEE Transactions on Computers*, pages 1179–1209, 1994.

[Gra94] S. Graf. Verification of a Distributed Cache Memory by Using Abstractions. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 207–219. Springer-Verlag, 1994.

[Gup92] A. Gupta. Formal Hardware Verification Methods: A Survey. In *Formal Methods in System Design*, pages 151–238. Kluwer Academic Publishers, New York, 1992.

[HBBM97] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan. Safe BDD Minimization Using Don't Cares. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 208–13, June 1997.

[HCC96a] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen. An ATPG-based Framework for Verifying Sequential Equivalence. In *Proc. Intl. Test Conf.*, 1996.

[HCC96b] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen. On Verifying the Correctness of Retimed Circuits. In *Proceedings. The Great Lakes Symposium on VLSI*, pages 277–80, 1996.

[HCC97] S.-Y. Huang, K.-T. Cheng, and K.-C. Chen. AQUILA: An Equivalence Verifier for Large Sequential Circuits. In *Proc. of Asian and South Pacific Design Automation Conf.*, 1997.

[HDB96] A. Hett, R. Drechsler, and B. Becker. MORE: An Alternative Implementation of BDD Packages by Multi-Operand Synthesis. In *Proc. European Design Automation Conf.*, pages 164–9, 1996.

[HK90] Z. Har'El and R. P. Kurshan. Software for Analytical Development of Communication Protocols. *AT&T Technical Journal*, pages 45–59, January 1990.

[Hor96] S. Horeth. Compilation of Optimized OBDD-Algorithms. In *Proc. European Design Automation Conf.*, pages 152–7, 1996.

[HP90] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufman, 1990.

[HYD94] A. J. Hu, G. York, and D. L. Dill. New Techniques for Efficient Verification with Implicitly Conjoined BDD's. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 276–282, June 1994.

[HYHD95] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill. Architectural Validation for Processors. In *Proc. of the International Symposium on Computer Architecture*, June 1995.

[ISY91] N. Ishiura, H. Sawada, and S. Yajima. Minimization of Binary Decision Diagrams Based on Exchanges of Variables. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 472–5, November 1991.

[JMF97] J. Jain, R. Mukherjee, and M. Fujita. FLOVER: Filtering Oriented Combinational Verification Approach. In *Proc. IEEE/ACM Intl. Workshop on Logic Synthesis*, pages 263–68, May 1997.

[JNC+96] J. Jain, A. Narayan, C. Coelho, S. Khatri, M. Fujita, and A. L. Sangiovanni-Vincentelli. Decomposition Techniques for Rfficient ROBDD Construction. In *Proc. Formal Method in Computer-Aided Design*, pages 419–34, 1996.

[KC90] S. Kimura and E. M. Clarke. A Parallel Algorithm for Constructing Binary Decision Diagrams. In *Proc. IEEE/ACM International Conference on Computer Design*, pages 220–223, November 1990.

[Keu96] K. Keutzer. The Need for Formal Methods for Integrated Circuit Design. In *Proc. Formal Method in Computer-Aided Design*, November 1996.

[KHB96] S. Krishnan, R. Hojati, and R. K. Brayton. Early Quantification and Partitioned Transition Relation. In *Proc. IEEE/ACM International Conference on Computer Design*, pages 12–19, Austin, TX, October 1996.

[KK97] A. Kuehlmann and F. Krohm. Equivalence Checking Using Cuts and Heaps. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 263–8, June 1997.

[KOKD96] M. Kumanoya, T. Ogawa, Y. Konishi, and K. Dosaka. Trends in High-Speed DRAM Architectures. *IEICE Transactions on Electronics*, pages 472–81, April 1996.

[KR97] N. Klarlund and T. Rauhe. BDD algorithms and Cache Misses. In *Dagstuhl Seminar, Computer-Aided Design and Test*, January 1997.

[KSR92] U. Kebschull, E. Schubert, and W. Rosentiel. Multilevel Logic Bsed on Functional Decision Diagrams. In *Proc. European Conf. on Design Automation*, pages 608–13, 1992.

[Kuk89] J. H. Kukula. A Technique for Verifying Finite-state Machines. Technical Report 3A, IBM Technical Disclosure Bulletin, 1989.

[Kur93] R. P. Kurshan. *Automata-Theoretic Verification of Coordinating Processes*. Princeton University Press, 1993.

[Kur97] R. P. Kurshan. Formal Verification in a Commercial Setting. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 258–62, 1997.

[Lam93] W. Lam. *Algebraic Methods for Timing Analysis and Testing in High Performance Designs*. PhD thesis, University of California Berkeley, April 1993. Memorandum No. UCB/ERL M94/19.

[LE92] B. Lockyear and C. Ebeling. Retiming of Multi-phase, Level-clocked Circuits. In *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pages 265–280, March 1992.

[Lin91] B. Lin. *Synthesis of VLSI Design with Symbolic Techniques*. PhD thesis, University of California Berkeley, 1991.

[Lon93] D. E. Long. BDD Manipulation Library. June 1993. `ftp://emc.cs.cmu.edu/pub/bdd/bddlib.tar.Z`.

[LR90] D. H. Lee and S. M. Reddy. On Determining Scan Flip-Flops in Partial-Scan Designs. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 322–5, 1990.

[LRS83] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing Synchronous Circuitry by Retiming. In *Advanced Research in VLSI: Proc. of the Third Caltech Conf.*, pages 86–116. Computer Science Press, 1983.

[LS91] C. E. Leiserson and J. B. Saxe. Retiming Synchronous Circuitry. In *Algorithmica*, pages 5–35, 1991.

[LS92] Y.-T. Lai and S. Sastry. Edge-valued Binary Decision Diagrams for Multi-level Hierarchical Verification. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 608–613, 1992.

[LVW97] C. Legl, P. Vanbekbergen, and A. Wang. Retiming of Edge-Triggered Circuits with Mulitple Clocks and Load Enables. In *Proc. IEEE/ACM Intl. Workshop on Logic Synthesis*, 1997.

[Mal90] S. Malik. *Combinational Logic Optimization Techniques in Sequential Logic Synthesis*. PhD thesis, University of California Berkeley, November 1990. Memorandum No. UCB/ERL M90/115.

[Mat96] Y. Matsunaga. An Efficient Equivalence Checker for Combinational Circuits. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 629–34, June 1996.

[McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM94] K. L. McMillan. Fitting Formal Methods into the Design Cycle. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 316–19, 1994.

[MGS97] S. Manne, D. C. Grunwald, and F. Somenzi. Remembrance of Things Past: Locality and Memory in BDDs. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 196–201, June 1997.

[MQRK97] A. Mehrotra, S. Qadeer, R. K. Ranjan, and R. H. Katz. Benchmarking and Analysis of Architectures for CAD Applications. In *Proc. IEEE/ACM Intl. Conf. on Computer Design*, Austin, Texas, USA, October 1997.

[MS97] N. Maheshwari and S. S. Sapatnekar. An Improved Algorithm for Minimum-Area Retiming. In *Proc. of the IEEE/ACM Design Automation Conf.*, 1997.

[MSBS91] S. Malik, E. M. Sentovich, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Retiming and Resynthesis: Optimization of Sequential Networks with Combinational Techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 10(1):74–84, January 1991.

[MSSS95] P. McGeer, A. Saldanha, A. L. Sangiovanni-Vincentelli, and P. Scaglia. Fast Discrete Function Evaluation Using Decision Diagrams. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1995.

[Mur93] R. Murgai. *Logic Synthesis for Field Programmable Gate Arrays*. PhD thesis, University of California Berkeley, December 1993. Memorandum No. UCB/ERL M93/98.

[MWBS88] S. Malik, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 6–9, November 1988.

[NJFS96] A. Narayan, J. Jain, M. Fujita, and A. L. Sangiovanni-Vincentelli. Partitioned ROBDDs – A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1996.

[OIY91] H. Ochi, N. Ishiura, and S. Yajima. Breadth-First Manipulation of SBDD of Boolean Functions for Vector Processing. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 413–416, June 1991.

[OYY93] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-First Manipulation of Very Large Binary-Decision Diagrams. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 48–55, November 1993.

[Pix90] C. Pixley. A Computational Theory and Implementation of Sequential Hardware Equivalence. In E. M. Clarke and R. P. Kurshan, editors, *Proc. of the Workshop on Computer-Aided Verification*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 293–320. American Mathematical Society, June 1990.

[Pix92] C. Pixley. A Theory and Implementation of Sequential Hardware Equivalence. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 11(12):1469–1494, December 1992.

[Pnu86] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends. In *Lecture Notes in Computer Science*, volume 224, pages 510–84. Springer Verlag, 1986.

[PSAB94] C. Pixley, V. Singhal, A. Aziz, and R. K. Brayton. Multi-level Synthesis for Safe Replaceability. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 442–9, November 1994.

[PSC94] Y. Parasuram, E. Stabler, and S.-K. Chin. Parallel Implementation of BDD Algorithms using a Distributed Shared Memory. In *Hawaii Internation Conference on System Sciences*, pages 16–25, January 1994.

[RAP+95] R. K. Ranjan, A. Aziz, B. Plessier, C. Pixley, and R. K. Brayton. Efficient BDD Algorithms for FSM Synthesis and Verification. In *Proc. IEEE/ACM Intl. Workshop on Logic Synthesis*, Lake Tahoe, California, USA, May 1995.

[RGBS97] R. K. Ranjan, W. Gosti, R. K. Brayton, and A. Sangiovanni-Vincentelli. Dynamic Variable Reordering in a Breadth-First Based BDD Package: Challenges and Solutions. In *Proc. IEEE/ACM Intl. Conf. on Computer Design*, Austin, Texas, USA, October 1997.

[RS95] K. Ravi and F. Somenzi. High-density Reachability Analysis. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 154–8, November 1995.

[RS97] R. K. Ranjan and J. Sanghavi. CAL-2.0: Breadth-first Manipulation Based BDD Library. June 1997. `http://www-cad.eecs.berkeley.edu/Research/cal_bdd`.

[RSBS96] R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. Using Network of Workstations for Efficient Binary Decision Diagram Manipulation. In *Proc. IEEE/ACM Intl. Conf. on Computer Design*, Austin, Texas, USA, October 1996.

[RSSB97] R. K. Ranjan, V. Singhal, F. Somenzi, and R. K. Brayton. Using Combinational Verification for Sequential Circuit. Technical Report UCB/ERL, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, October 1997.

[Rud93] R. Rudell. Dynamic Variable Ordering for Binary Decision Diagrams. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 42–47, November 1993.

[RWK95] S. M. Reddy and D. K. P. Wolfgang Kunz. Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 414–9, June 1995.

[San96] J. Sanghavi. *High Performance Verification Algorithms*. PhD thesis, University of California Berkeley, December 1996.

[SB96] T. Stronetta and F. Brewer. Implementaion of an Efficient Parallel BDD Package. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 641–4, June 1996.

[SBS93a] N. Shenoy, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Minimum Padding to Satisfy Short Path Constraints. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 156–61, November 1993.

[SBS93b] T. Shiple, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Computing Boolean Expression with OBDDs. Technical Report UCB/ERL M93/84, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, December 1993.

[Sen96] E. M. Sentovich. A Brief Study of BDD Package Performance. In *Proc. Formal Method in Computer-Aided Design*, November 1996.

[Shi97] T. R. Shiple. Private communication, 1997.

[SHSB94] T. R. Shiple, R. Hojati, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Heuristic Minimization of BDDs Using Don't Cares. In *Proc. of the IEEE/ACM Design Automation Conf.*, June 1994.

[SK97] D. Stoffel and W. Kunz. A Structural Fixpoint Iteration for Sequential Logic Equivalence Checking Based On Retiming. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, 1997.

[SKMB90] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for Discrete Function Manipulation. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 92–95, November 1990.

[Smi97] D. J. Smith. *HDL Chip Design*. Doone Publications, 1997.

[Som97] F. Somenzi. *CUDD: CU Decision Diagram Package*. University of Colorado at Boulder, 1997.

[SR94] N. Shenoy and R. Rudell. Efficient Implementation of Retiming. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 226–33, November 1994.

[SRBS96] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High Performance BDD Package Based on Exploiting Memory Hierarchy. In *Proc. of the Design Automation Conf.*, June 1996.

[SRSB97] T. R. Shiple, R. K. Ranjan, A. L. Sangiovanni-Vincentelli, and R. K. Brayton. Deciding State Reachability for Large FSMs. Technical Report UCB/ERL, M97/73, Electronics Research Lab, August 1997.

[SS94] M. Singhal and N. Shivaratri. *Advanced Concepts in Operating Systems*. McGraw-Hill, 1994.

[SSBS92] N. Shenoy, K. J. Singh, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. On the Temporal Equivalence of Sequential Circuits. In *Proc. of the IEEE/ACM Design Automation Conf.*, pages 405–9, June 1992.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[SW95] D. Sieling and I. Wegener. Graph-driven OBDDs – A New Data Structure for Boolean Functions. *Theoretical Computer Science*, 1995.

[Swa96] G. M. Swamy. *Incremental Techniques for Logic Synthesis and Verification.* PhD thesis, University of California Berkeley, November 1996. Memorandum No. UCB/ERL M96/115.

[TSL+90] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines using BDD's. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[TY93] Y. Takenaga and S. Yajima. NP-Completeness of Minimum Binary Decision Diagram Identification. Technical Report COMP 92-99, Institute of Electronics, Information and Communication Engineers (of Japan), March 1993.

[YCBO97] B. Yang, Y.-A. Chen, R. E. Bryant, and D. R. O'Hallaron. Space- and Time-Efficient BDD Construction via Working Set Control. Technical Report CMU-CS-97, Department of Computer Science, Carnegie Mellon University, 1997.

[YO97] B. Yang and D. O'Hallaron. Parallel Breadth-First BDD Construction. In *Proc. of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Las Vegas, NV, June 1997.

[YSAA97] J. Yuan, J. Shen, J. Abraham, and A. Aziz. On Combining Formal and Informal Verification. In *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 376–87. Springer-Verlag, 1997.

Index