Logic Synthesis Techniques for High-Speed Circuits

David Bañeres

Advisors: Jordi Cortadella (UPC)
Mike Kishinevsky (Intel Corporation)

Disertation submitted in partial fulfillment
of the requirements for the Degree of
Doctor in Philosophy in Computer Science

Departament de Llenguatges i Sistemes Informàtics (LSI)
Universitat Politècnica de Catalunya

January 3, 2008

Acknowledgments

This thesis would not have been possible without the perseverance and guidance of my thesis advisors, Jordi Cortadella and Mike Kishinevsky.

Jordi with his continuous support, encouragement, supervision and constructive criticism made him a reliable reference in the most critical moments of my research. Mike showed me the real world inside the computer aided design. I would also thank him for his comments, advice and patience during my internship in Intel Corporation.

I must also acknowledge my colleagues Robert Clariso, Kyller Costa, Federico Heras, Nilesh Modi, Albert Oliveras, Enric Rodriguez at the LSI Department for making the stay especially enjoyable.

This work has been partially funded by a grant from Intel Corporation; a scholarship (FI) from Generalitat de Catalunya; and the projects CICYT TIC2001 2476 (MAVERISH), and CICYT TIN2004 07925 (GRAMMARS).

Finally, thanks to my beloved Maria Elena, for sharing this experience with me, and helping me in making it possible through the difficult times, and to my parents for their love, caring, and understanding.

Abstract

Complexity in the design of electronic systems is significantly increasing in DSM technologies. Synthesis requires more powerful techniques to meet the specification constraints and capable to run in affordable time in the larger designs. One of the phases in VLSI design is logic synthesis. This thesis introduces several methods in this phase to meet one of the primary objectives in circuit design: *timing optimization*.

Several contributions are presented. First, a solver of Boolean relations has been developed. A *Boolean relation* is able to capture more flexibility than conventional approaches based on *don't cares*. This work received the best paper award in the Design Automation Conference (DAC'04).

The second contribution is a new partitioning algorithm based on the concept of vertex dominator. When optimization algorithms are applied on these clusters, this partition offers more possibilities for restructuring towards delay minimization compared to other techniques based on min-cut.

A multi-level decomposition approach is also defined using the solver of Boolean relations: a time-driven n-way decomposition. Functions are decomposed to improve the performance (speed) using a small library of multi-input gates.

Finally, an integrated approach for layout-aware interconnect optimization is presented. This technique combines gate duplication and buffer insertion in the same framework with incremental placement. Similar to the principle of the *Engineering Change Order* (ECO), the circuit is incrementally improved by performing small modifications using fanout optimization techniques on top of the current placement.

The contributions of the thesis have been published in the following papers:

- D. Baneres, J. Cortadella, and M. Kishinevsky. A recursive paradigm to solve boolean relations. In *Proc. ACM/IEEE Design Automation Conference*, pages 416–421, June 2004
- D. Baneres, J. Cortadella, and M. Kishinevsky. Dominator-based partitioning for delay optimization. In *ACM Great Lakes Symposium on VLSI*, pages 67–72, 2006
- D. Baneres, J. Cortadella, and M. Kishinevsky. Layout-aware gate duplication and buffer insertion. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1367–1372, 2007
- D. Baneres, J. Cortadella, and M. Kishinevsky. A recursive paradigm to solve boolean relations. In *IEEE Transactions on Computers*, 2007 (Submitted)

Contents

Chapter 1

Introduction

Digital circuits are widely used in many technological products, such as computers, cellular phones or communications systems. Our society is currently dependant on these *small pieces of technology*. The Electronic Design Automation (EDA), which appeared forty years ago, is a category of tools for designing and producing electronic systems in the electrical engineering discipline. These tools aim at producing from printed circuit boards to integrated circuits. The importance of the EDA tools has quickly increased due to the continuous evolution of semiconductor technology, as Moore's Law predicted, and the new system designs. Currently, several millions of transistors can be put in a single circuit. For instance, the Intel Pentium 4 microprocessor at 3.2 GHz has 178 millions of transistors [1].

Before EDA, the designers were limited to deal with small- and medium-sized circuits since they were produced manually. However, this design methodology could not stand anymore when the size of the circuits grew to several thousands of transistors.

Currently, Very Large Scale Integration (VLSI)[1] requires other methodologies. VLSI is possible due to manufacturing and design technologies. Physical components have had a quick evolution in the recent years. For example, the feature size of a transistor in 1996 was 250 nm and nowadays it is around 65 nm [2]. This reduction on the size has helped to integrate a larger number of transistors in the same chip area. In design technologies, Computer Aided Design (CAD) tools have been developed since the early seventies to help in VLSI design. Initially, CAD tools were only used in small activities. Following the evolution of the manufacturing technologies, the CAD tools also progressed until complete design flows were automated.

Nowadays, automated CAD tools are the key in circuit design. The investments associated with the manufacturing of VLSI are very high, and the time-to-market limits the spent time. CAD tools

[1] Some experts on the area claim that the name of the current stage should be *Ultra-Large Scale Integration* (ULVI) since the current manufacturing processes produce chips larger than 1 million of transistors.

make the design of new systems cost-effective. The main objectives of CAD tools are:

- increase designers'productivity: the CAD tools help the designers to take the adequate decisions to develop a good design. Moreover, the cycle of production of new systems is sped up due to the automation of the design flows.
- reduce the number of components: functionally equivalent circuits with a smaller number of components are desirable. This implies less number of transistors and, therefore, a lower production cost.
- improve the performance (speed): the speed is an important factor in the current circuit designs mostly by the market requirements.
- reduce the power consumption: the density of the circuits increases significantly the temperature, and it generates a high dissipation of energy as heat. This drawback affects negatively performance of the circuit.
- validate: a design must be validated to check if the result of the design flow is functionally equivalent to the given specification. The objective is to avoid catastrophic mistakes, e.g. bug in the Pentium microprocessor in the floating-point division unit [51].

1.1 Design flow

The design of a circuit is a complex problem that can not be handled as a whole. The main objective is to optimize the implementation of the new design and meet the specification constraints. However, there are many objectives and constraints to take into account and it is difficult to deal with all the requirements at the same time.

The design flow is divided into different levels of abstraction, where each level typically tackles an objective. This division aims at reducing the complexity of the design problem. Note that, this is not the best approach to get the optimal solution. However, the development of optimization techniques for each subproblem is easier and these approaches usually have an affordable computational complexity.

1.1.1 Classical design flow

The design flow is usually divided into three stages: the RTL front-end, logic synthesis and physical synthesis, in which each stage focuses on a specific problem. The RTL front-end targets at specificate the *behavior* of the design. At this stage, the design is not described in the transistor or gate

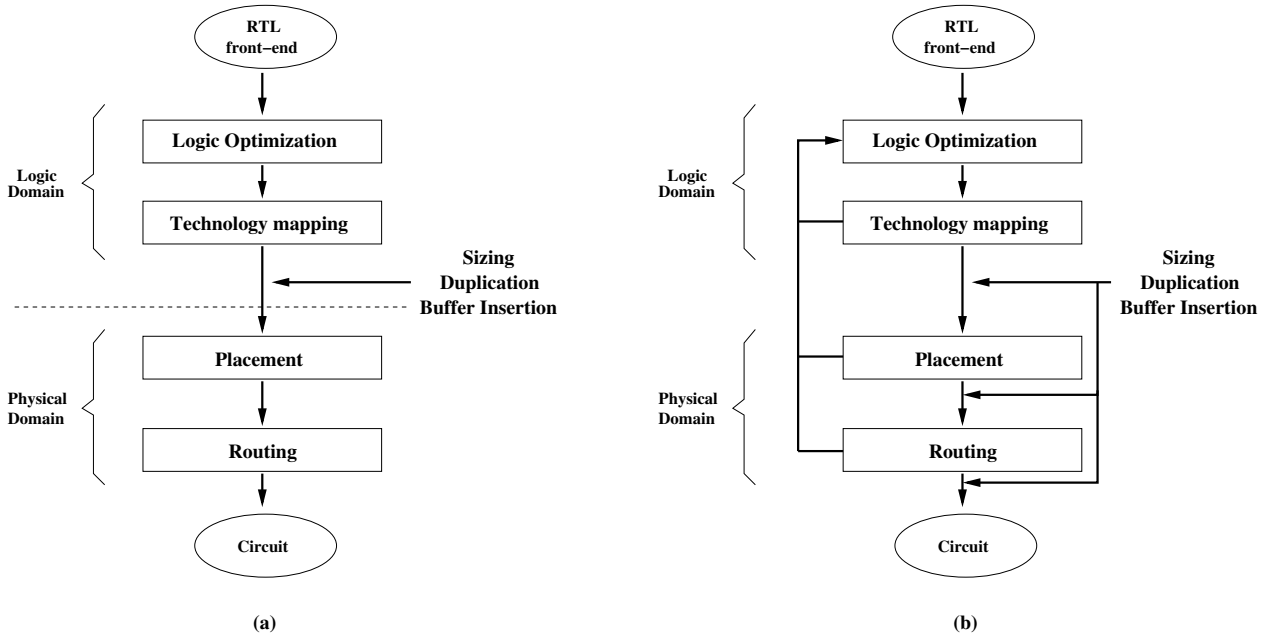


Figure 1.1: (a) Traditional design flow. (b) DSM design flow.

level. A higher level is used to detail the flow of the signals between registers. The Register Transfer Level (RTL) specification is used as input for the logic synthesis flow.

Figure 1.1-(a), extracted from [116], illustrates the traditional design flow with Logic and Physical synthesis. Several independent phases are distinguished depending on the targeted problem.

Logic optimization techniques and technology mapping are performed during the logic synthesis phase. Logic optimization covers sequential and combinational minimization. Sequential minimization uses *state-based* techniques, like FSM encoding, to obtain an initial structural representation of the data path. Other more modern techniques, like retiming, seek equivalent circuits with the same behavior targeting at minimizing some objective function, i.e. cycle time, number of sequential elements,... In combinational logic synthesis, two-level and multi-level optimization are the standard techniques applied to minimize the combinational blocks between the sequential components.

Technology mapping is the final step in logic synthesis that maps the Boolean network using a library of components (or logic gates). The output is a circuit with the same behavior as the original one but the circuit is implemented with the gates of the library. There are also some refining techniques after technology mapping such as sizing, buffer insertion and duplication that perform local modifications to the circuit to meet the timing constraints.

Finally, physical synthesis is basically concerned about the layout of the design after the tech-

nology mapping phase. Placement assigns a position to each cell and routing calculates the route of the interconnections. Currently, the placement and routing phases tend to be internally divided into two phases. Initially, a global technique is applied to obtain an approximated result. Here, clustering techniques are used to consider blocks of cells as a unique cell and this clustering aids to run the optimization process with less complexity. The second phase amends the errors and refines the solution. In detailed placement, minor changes are done applying swapping or reordering techniques on individual cells. The final layout must be legal without any overlapping between the cells. In detailed routing, the paths of the wire are refined aiming at reducing the congestion.

1.1.2 DSM design flow

The traditional design flow has some drawbacks in the current *depth sub-micron* (DSM) technologies. In process technologies larger than 180*nm*, the problem to meet the timing constraints was focused on the logic domain. The interconnection delay was meaningless compared to the intrinsic delay of the gates. However, the wire delay becomes the predominant factor in the current 90*nm*-65*nm* (and future) process technologies, and new challenging problems have arisen that they can not be tackled with the current flow. Some of them are related to physical problems like coupling [90] or crosstalk [168]. The DSM design flow currently used is depicted in Fig. 1.1-(b). There are several approaches to deal with the new DSM design flow.

The first attempt is to apply minimization delay procedures in physical design that were previously applied only in the logic domain, like buffer insertion or duplication. However, this approach only performs small modifications locally in the placement and routing. Although, some improvements are obtained, the final solution is highly dependant on the design decisions taken in the logic domain phase.

The second popular attempt is to apply a *physical-aware logic synthesis*. Congested areas of the placement or critical paths are extracted from the circuit and they are pushed up to the logic domain to be processed again. Then, the improved portion is replaced into the original circuit. An incremental placement is performed to legalize it moving apart some nodes. This method uses the principle of the *Engineering Change Order* (ECO). ECO can be applied in any level of the design flow. If the specification constraints can not be met with local changes in the physical design, ECO must be applied on the other levels of abstraction, even in the RTL front-end level.

Finally, the third option is to use methods that combine information from multiple phases. This combination improves the final result at the expenses of increasing the complexity of the traditional techniques. Dividing the design flow into several stages and applying several approximate delay models produce final designs that can be far from the desirable one. To mitigate this gap between results from different phases, methods that combine information of multiple phases have been developed. For example, a method that combines divisor extraction and incremental placement is presented in [40], or a routing-aware technology mapper is developed in [144]. These techniques

are examples of approaches that unify in the same framework objective functions of several phases of the design.

1.2 Motivation and contributions

This thesis is focused in one of the main objectives in DSM design: *timing optimization*. A collection of several timing-driven techniques during technology independent optimization and physical synthesis will be described.

The presented contributions aim at solving optimization problems in circuit design. All the methods have been compared with industrial optimization processes and tested in academic and industrial benchmarks to check their effectiveness.

The motivation problems and the contributions of this thesis are described next:

Solver of Boolean relations

Flexibility in logic synthesis can be expressed in several abstract methods. Classical methods use conventional don't cares to represent the flexibility in logic functions. A Boolean relation captures a type of flexibility that cannot be expressed with don't cares.

The objective of a solver of Boolean relations is to find a compatible multi-output function with minimum cost. Note that, this is a complex problem. The current Boolean relation solvers are constrained to small- and medium- relations. Moreover, their are difficult to use on specific problems due to their limitation on the customization of the cost function. All of them aim at minimizing the size of the functions, without concerning about other objectives like delay. This heuristic approaches are based on a local search algorithm and they are often trapped in local minima, since they are not always capable of hill climbing.

Boolean relations can be applied to optimization problems like multi-level minimization [167] or pattern matching in technology mapping [26]. Moreover, there is an equivalence between solving a system of Boolean equations in Boolean algebra and solving the corresponding Boolean relation.

The first contribution is a **solver of Boolean relations**. A recursive paradigm is presented to explore the large space of possible solutions. Our solver explores a larger diversity of solutions compared to the local search approaches. Moreover, the cost function can be tuned for different parameters (area, delay) and the method can trade-off the quality of the solution and the runtime spent in the search.

Partitioning methods

Technology-independent optimization methods are restricted by its complexity. Many of them were standard minimization methods in the nineties. Nevertheless, the increment of the size of the

Boolean networks made them impractical for large instances.

Partitioning is often used when complexity problems arise. Current partition methods are mostly reduced to min-cut which is useful in problems that depend on the structure of a graph. Although a Boolean network is a directed acyclic graph, the min-cut objective is not always capable of capturing the Boolean information inherent in the graph and producing a partition well-suited for logic optimization.

A **timing-driven partitioning method** has been developed. Min-cut is useful when the number of interconnections between the different clusters is minimized. Nevertheless, when optimization algorithms are applied on these clusters, the min-cut could prevent the optimization on strategic regions of the circuits, i.e. the critical path, due to the performed cuts. We propose a new partitioning algorithm, based on the concept of vertex dominator, that aims at capturing fragments of critical paths that have small fanout to the rest of the circuit. Thus, the clusters tend to be deep (many levels) with internal nodes having little fanout to external nodes. This type of clusters offers more possibilities for restructuring towards delay minimization.

Timing-driven decomposition

Timing-aware decomposition [56, 120, 169, 171] techniques appeared to perform better optimizations based on bi-decomposition. These approaches build a new Boolean network from scratch based on a recursive method that decomposes the network using two-input gates.

A new decomposition method is presented. Decompositions based on gates with a larger number of inputs, called *n-way decomposition*, could help to find better decompositions that cannot be achieved with two input gates. A recursive **timing-driven n-way decomposition method** using the solver of Boolean relations is proposed. The type of flexibility exploited in n-way decomposition, that can not be captured with don't cares, is formulated as a *Boolean relation*. The n-way decomposition approach improves the results obtained by bi-decomposition.

However, a high computational cost is required to solve large Boolean relations and, therefore, its application is limited to small Boolean networks. The partitioning method previously defined makes feasible its application on larger networks based on a *window-based n-way decomposition*.

Interconnection optimization in physical design

The large fanout on gates can be optimized in the logic and physical step of the design flow. Previous work on this area reveals that three main approaches are mostly used: sizing, duplication and buffer insertion. These methods are used indistinctly on these stages.

The accuracy of the solution is improved when these methods are applied in the physical step due to the physical information of the circuit. However, a new problem is added to the fanout optimization: the wire delays. The interconnection optimization targets at reduce the long wires.

There are several benefits with this optimization: improve of the delay and the power consumption and reduction of the wire congestion.

The last contribution is a **layout-aware gate duplication and buffer insertion approach**. The objective is the optimization of the interconnection delays taking the physical information into account. Buffer insertion and gate duplication are complementary techniques that individually each technique contributes to improve the delay of the network. However the combination of both can lead to superior results. The improvement can still be more tangible if physical information is considered and, reciprocally, the changes produced by buffer insertion and gate duplication have a positive impact by incrementally changing the physical layout of the involved cells.

1.3 Organization of this document

The background is presented in Chapter 2. Moreover, the basic definitions and the problems we are dealing with are also introduced. Next, four main topics are addressed: Boolean relations, partitioning, n-way decomposition and interconnection optimization.

Chapter 3 describes a new paradigm to solve Boolean relations. Here, experimental results are presented to show its benefit with regard to previous solvers.

Our dominator-based partitioning method is presented in Chapter 4 that it reduces considerably the complexity of logic optimization method on large networks. The vertex dominator property helps to improve the results with regard to the generic min-cut partition methods.

The n-way decomposition approach is presented in Chapter 5 to illustrate the effectiveness of the Boolean relation solver. Moreover, it is also combined with the partitioning method to make feasible its application on larger networks.

Finally, the layout-aware interconnection optimization is presented in Chapter 6. Gate duplication and buffer insertion are combined with placement to improve significantly the delay of a netlist. Results in the current semiconductor process technology of 0.65nm are reported. Besides, future semiconductor process technologies are also reported to corroborate the increasing relevance of the interconnect optimization.

Chapter 2

State of the art

Logic synthesis is a fundamental phase on the process of synthesis of a circuit where logic optimization techniques and technology mapping are applied. Logic synthesis covers sequential and combinational minimization. Sequential minimization uses *state-based* techniques, like FSM encoding, to obtain an initial structural representation of the data path. Other more modern techniques, like retiming, seek equivalent circuits with the same behavior targeting at minimizing some objective function, i.e. cycle time, number of sequential elements,... In the combinational logic synthesis, two-level and multi-level optimization are the standard techniques applied to minimize the combinational blocks between the sequential components. In this chapter, we will only review *combinational* logic synthesis.

The evolution of logic synthesis has been closely related to the evolution of the manufacturing technology. New challenging problems usually arise and the optimization objectives also change with the new technologies.

In this chapter, an introduction to the state of art in logic synthesis is performed in Section 2.1. Moreover, a brief description is given to the problems we are dealing with in this thesis and how recent methods attempt to solve them. Decomposition is described in Section 2.2. Section 2.3 introduces several partitioning approaches and Section 2.4 reviews technology mapping and technology-dependent techniques.

2.1 Overview on Logic Synthesis

In this section, an overview in logic optimization is presented. First, two-level and multi-level minimization is defined. Finally, the basis of the flexibility in Boolean functions and BDD-based minimization is also introduced.

2.1.1 Terminology

In this section, some basic terminology used in this thesis is introduced.

Definition 2.1.1 Boolean Function. *A Boolean function f is a function $f(X) : \mathbb{B}^n \rightarrow \mathbb{B}$ where $\mathbb{B} = \{0,1\}$. The* support *of f is the set of input variables that f explicitly depends on.* □

A function can be specified using the operations of conjunction, disjunction, and complement of Boolean Algebra.

Definition 2.1.2 Literals, minterms, cubes and covers. *A* literal *is a variable or its complement. The conjunction (or product) of a set of literals is called a* cube. *A cube is called a* minterm *when the number of different literals of the cube corresponds to the number of variables of the function. A function can be represented by a* cover *that is defined as a disjunction (or sum) of cubes.* □

Example 2.1.1 *Suppose the function $F = abc + ab\overline{c} + \overline{a}bc$. The minterms of the function are $\{abc,\ ab\overline{c},\ \overline{a}bc\}$. The function F can also be represented by the cover $F = ab + bc$, where $\{ab,\ bc\}$ are cubes of the function.* □

Definition 2.1.3 Cofactor and existential abstraction. *The cofactors f_{x_i} and $f_{\overline{x_i}}$ of a Boolean function $f(x_1,\ldots,x_n)$ are defined as $f_{x_i} = f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$ and $f_{\overline{x_i}} = f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n)$. The existential abstraction $\exists_{x_i} f$ is defined as $\exists_{x_i} f = f_{x_i} + f_{\overline{x_i}}$. Cofactors and existential abstraction can be extended to multiple variables.* □

A combinational circuit can be modelled with a set of Boolean functions.

Definition 2.1.4 *A* Boolean network *is a directed acyclic graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes of the network and $\mathcal{E}$ the set of wires. A Boolean function is associated to each node of the network and the edges represent the interconnection between the functions. The inputs and outputs of a node are called* fanins *and* fanouts *respectively. The nodes with no fanin are the primary inputs, whereas the nodes with no fanout are the primary outputs.* □

Example 2.1.2 *Figure 2.1 shows the Boolean network associated to the next functions:*

$$\begin{aligned} F &= (c(a+b)+e)(g+f)+(a+b)d \\ G &= (a+b)d+ihc \end{aligned}$$

Note that, the figure depicts one possible representation. Other representations can be easily found applying the properties of the Boolean algebra, like the distributive or the associative laws. □

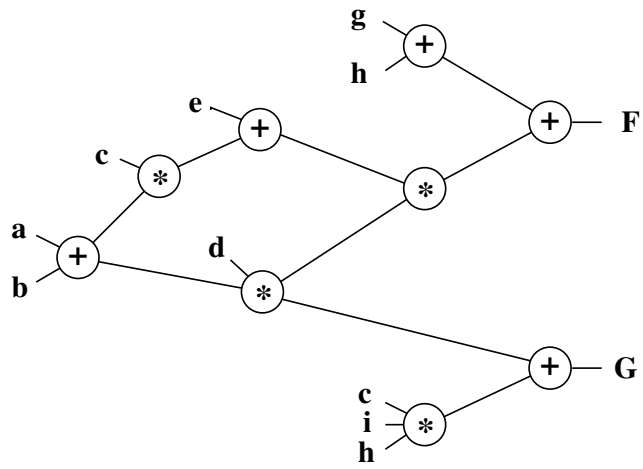


Figure 2.1: Example of Boolean network.

2.1.2 Two-level Minimization

Two-level minimization was the first attempt to develop automatic techniques to optimize circuits. Historically, two-level minimization became popular in the fifties. This approach was an effective and systematic approach to design circuits with low complexity.

In the seventies, the interest grew again. Two-level minimization was ideal for the implementations of Programmable Logic Array (PLA) [70]. However, this technique has some limitations: scalability and technology dependency, that will be further explained in the Section 2.1.3.

Basically, the objective of a two-level minimization is to find a cover with the minimum number of prime implicants.

Definition 2.1.5 Implicant and prime implicant. *An* implicant *is a cube included in the function. A* prime implicant *is an implicant that is not included in any other implicant.* □

Example 2.1.3 *In the function* $f = a\bar{b} + bc$*, the cubes* $a\bar{b}$ *and* abc *are implicants, but only* $a\bar{b}$ *is prime.* □

The usual two-level representation of a function is a sum-of-products (SOP) where the cost of the design, measured by the number of product terms, can be easily computed. This cost is ideal to estimate the complexity of a PLA since there is a one-to-one correspondence with the number of rows of the PLA.

Several exact and heuristic approaches have been proposed to solve this problem. The most relevant exact method is the Quine-McCluskey's, where the two-level minimization is reduced to

	$\overline{a}\,\overline{c}$	$\overline{a}\,b$	$b\,c$
$\overline{a}\,\overline{b}\,\overline{c}$	1		
$\overline{a}\,b\,\overline{c}$	1	1	
$\overline{a}\,b\,c$		1	1
$a\,b\,c$			1

Figure 2.2: Unate Covering Problem

the Unate Covering Problem (UCP), and ESPRESSO is the most remarkable heuristic approach. Let us review them.

Quine-McCluskey's method [74] reduces the problem to finding an optimal two-level representation to a UCP. However, first, all the prime implicants of the function must be computed. The covering matrix of the UCP is built by placing the prime implicants in the columns and the minterms in the rows. Only the minterms subsumed on each prime are marked in the matrix. The UCP will compute the smallest sum of primes that covers the function.

Example 2.1.4 *Consider the function* $F = \overline{a}\,\overline{b}\,\overline{c} + \overline{a}\,b\,\overline{c} + \overline{a}\,b\,c + a\,b\,c$ *with primes* $\overline{a}\,\overline{c}$, $\overline{a}\,b$ *and* $b\,c$. *The covering matrix is represented in Fig. 2.2 where the optimal cover is* $F = \overline{a}\,\overline{c} + b\,c$. □

Exact procedures always obtain the optimal solution of the problems, nevertheless, they are not scalable. The same happens in two-level minimization. UCP is an NP-complete problem where the size of the covering matrix is exponential with regard to the number of variables of the function. The number of minterms is 2^n and the number of primes is approximately $\frac{3^n}{n}$, where n is the number of variables. Therefore, heuristics approaches must be used on large functions.

There are several heuristic approaches [23, 32, 78, 134] in two-level minimization. All of them use a procedure similar to the minimization algorithm *reduce-expand-irredundant* shown in Fig. 2.3. They differ on which operations of the internal loop are selected for execution (some methods skip operations) and the order of the application of them. This optimization method explores the space of covers of a function performing a local search on its primes. This process tries to escape from local minima and search for other possible covers with less number of product terms. In the procedure shown in Fig 2.3, an initial cover is selected. In order to find other possible covers, this initial one is expanded using one literal to cover some minterms with more than one cube. The *irredundant* procedure will remove the redundant cubes that may appear. Finally, the function is reduced again to primes calling the *reduce* procedure.

Example 2.1.5 *Consider the function of Example 2.1.4. Figure 2.4 shows the application of ESPRESSO [32] based on the algorithm of Fig. 2.3. In this example, the function is represented by cubes where the symbol* $\{-\}$ *means that the variable is not in the cube. In the first* Expand *procedure, minterm* 000 *is expanded to the cube* $0-0$*, since minterm* 010 *is also included in the*

```
Heuristic_minimization (F)
{Input: A function F}
{Output: A minimized function}

   Expand(F);
   Irredundant(F);
   do
      Reduce(F);
      Expand(F);
      Irredundant(F);
   while (F is improved);
   return F;

end;
```

Figure 2.3: Two-level minimization heuristic.

$x\,y\,z$	f
$0\,0\,0$	1
$0\,1-$	1
$-1\,1$	1

$\xrightarrow{Expand}$

$x\,y\,z$	f
$0-0$	1
$0\,1-$	1
$-1\,1$	1

$\xrightarrow{Irredundant}$

$x\,y\,z$	f
$0-0$	1
$-1\,1$	1

Figure 2.4: Application of ESPRESSO

function. Cube $01-$ *becomes redundant because minterms that it covers are also covered by other cubes of the function. This cube* $01-$ *is removed using the process* Irrendundant. □

Although these heuristic methods, like ESPRESSO, are approximated techniques, in many cases they are able to find the optimal cover, as it is shown in the previous example.

2.1.3 Multi-Level Logic Synthesis

As we explained in the previous section, two-level minimization was initially used to simplify PLA arrays. However, circuits became more complex and PLA technology was impractical and it was replaced by technologies based on MOSFET transistors, like nMOS and CMOS. Although MOSFET transistors appeared in the sixties, they did not become predominant in the digital circuits until the eighties. This technology introduced some advantages with regard to PLA implementation. The circuits became smaller in area size, faster and less power was consumed. Circuits with this new

technology tended to be implemented with several levels. The abstraction of the circuits in Boolean networks was excellent to describe them in logic synthesis.

The advantage of multi-level logic optimization is the minimization of the Boolean network with a new complete methodology based on factorization, decomposition, don't care computation, etc. Moreover, multiple-objective minimization can be applied on this new representation in Boolean networks.

The Boolean functions of the nodes of a Boolean network can be represented in different ways. Each representation has advantages and drawbacks. Networks with nodes with complex functions are typically used on area minimization since better optimization can be done on large functions based on two-level minimization. The utilization of two-input function, e.g., NAND2 representation, is a good selection to measure the complexity of a network. The number of nodes and the maximum number of levels between PIs and POs based on this representation give a good estimation of the area and delay respectively of the network.

The structure of a network can be improved by applying local transformations. There are some standard transformations that can be applied in any order to modify the structure towards a desired objective minimization, for instance, the reduction of the size of the network. The most relevant transformations, extracted from [74], are reviewed next:

- **Substitution:** A node of a Boolean network can be simplified using as a new fanin another existing node of the network. This transformation requires the creation of an interconnection between these two nodes. For example, consider the nodes x and t of a network. The node x can be expressed in terms of t applying substitution.

$$\begin{array}{ll} x & = ya + yb + e \\ t & = a + b \end{array} \quad \xrightarrow{Substitution} \quad \begin{array}{ll} x & = yt + e \\ t & = a + b \end{array}$$

- **Extraction:** Multiple nodes can have a common factor[1]. This factor can be extracted and stored in a new node. The variable associated with the new node allows to simplify the two nodes by replacing the common factor. Consider the nodes x and y with the common factor $a+b$. This factor can be extracted as a new node z. Next, x and y can be expressed in terms of z.

$$\begin{array}{ll} x & = (a+b)e \\ y & = (a+b)(c+d)+e \end{array} \quad \xrightarrow{Extraction} \quad \begin{array}{ll} x & = ze \\ y & = z(c+d)+e \\ z & = a+b \end{array}$$

[1] A *factor* is defined as a product or a sum of single literals or other factors.

- **Elimination:** A node can be removed from a network by merging its function with its fanout nodes. This transformation is helpful to create larger nodes where other transformations can be applied. For example, consider the nodes x, y and z. The node x can be deleted and the nodes y and z can assimilate its functionality.

$$\begin{array}{ll} x & = a+b \\ y & = x+c \\ z & = xcd \end{array} \quad \overset{Elimination}{\longrightarrow} \quad \begin{array}{ll} y & = a+b+c \\ z & = (a+b)cd \end{array}$$

- **Decomposition:** A node can be decomposed in several nodes that will form a subnetwork equivalent to the original node. A unique node can be created by calling recursively to *elimination* and, then, decomposition can be applied depending on an objective function. This type of transformation will be reviewed in Section 2.2. Consider the node x. The factor $ac+bc+d$ can be extracted. Recursively, this factor can also be decomposed in $(a+b)c+d$.

$$x = ace+bce+de+g \quad \overset{Decomposition}{\longrightarrow} \quad \begin{array}{ll} x & = ye+g \\ y & = ac+bc+d \end{array} \quad \overset{Decomposition}{\longrightarrow} \quad \begin{array}{ll} x & = ye+g \\ y & = zc+d \\ z & = a+b \end{array}$$

All these transformations rely on the search of factors. There are two basic ways to find them: the algebraic and Boolean model. In order to obtain factors quickly, the algebraic model [34] was introduced in the beginning of the eighties. The functions are represented as polynomials, where some Boolean operations, like the identity, are ignored. This reduction helps to simplify the model at the expenses of the quality of the result. The *algebraic division* is the fundamental operation used in the algebraic model. Using division, all possible algebraic divisors can be found for a function.

A transformation like extraction of common factors requires the computation of all divisors. The concept of *Kernels* [34] was introduced to cut off the exploration of common factors. Basically, a kernel is a factor of a function that cannot be divided by any other factor that is a cube. The intersection between the kernels of different functions identifies good common factors among the functions. Using kernels, fast heuristic factoring algorithms have been also developed to find near-optimal factored forms.

Boolean division is more powerful. Boolean equivalence between logic functions can be used. The optimization quality improves but the runtime also increases because of the huge search space of possible factors. Nevertheless, a trade-off between speed and quality can be found. Besides, a combination method with algebraic and Boolean divisors [33, 121] can be also used.

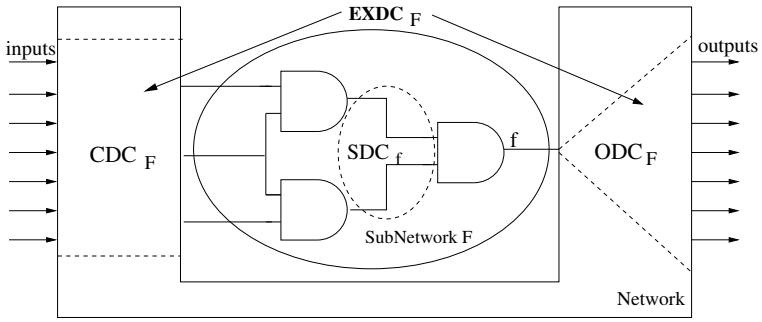


Figure 2.5: Don't care distribution.

2.1.4 Flexibility in Boolean functions

The flexibility of a function arises from the surrounding environment in the Boolean network. Based on the logic information of the transitive fanins and fanouts, a set of conditions can be computed related to the visibility of the targeted function in the global network. This flexibility generates an interval of Boolean functions that can be selected without changing the behavior of the network.

Flexibility computation was initially exploited in two-level minimization. However, flexibility is more relevant in multi-level minimization, since, similar to two-level, a subnetwork can be replaced by another one from an interval of possible subnetwork computed from the surrounding flexibility without affecting the functionality of the global Boolean network.

The flexibility can be expressed with *don't cares*. The set of minterms of the don't care function is called don't care set[2]. The don't care conditions related to a function implicitly generates an interval of feasible functions. In general, two-level minimization exploits this interval to find the optimal function in terms of number of cubes and/or literals,.... Likewise in multi-level minimization, the don't care conditions implicitly generate an interval of feasible subnetworks.

Several types of don't cares can be extracted from the structure of the circuit: controllability, satisfiability and observability don't cares.

- The *controllability don't care set* (CDC) is the set of input patterns that are never produced by the environment.
- The *satisfiability don't care set* (SDC) is the flexibility associated to each internal function of the network. The computation is straightforward using the equation $SDC = x \oplus f_x$.
- The *observability don't care set* (ODC) is the set of input patterns that produce situations where the output of the function is not observed at the primary outputs of the network.

[2]Hereafter, we will indistinctively talk about don't cares and don't care set.

The computation of the don't cares can be a complex process depending on the size of the network. The union of all don't care conditions are jointly denoted as *external don't care* (EXDC) conditions of the current network and they are extracted from the surrounding combinational logic or the reachable states of the sequential elements where the current network is enclosed. Figure 2.5, extracted from [116], depicts how the different types of don't care are distributed around a network.

Types of flexibility

The previous don't care conditions refer to the most basic types of flexibility commonly exploited. However, there are other types of flexibility that arise from the network. In this section, a brief overview to the different classes is given. Figure 2.6, extracted from [142], depicts the flowchart depending on the level of captured flexibility.

At the bottom of the flowchart, there are the *complete specified functions* where no flexibility can be used to optimize the function. Then, the *incomplete specified function* (ISFs) defines an interval of permissible functions using the don't care set. Here, the minimal function in number of cubes and literals can be obtained with two-level minimizers like ESPRESSO. Single-output functions can be easily generalized to multiple-output functions where several methods have been also proposed to optimize them [22, 62, 137].

Next, models that capture more flexibility are described. Intuitively, a Boolean relation (BR) captures several output "choices" for each minterm of a multiple-output function. Note that, conventional don't cares can not express this type of flexibility. The don't care set is limited to only express the flexibility for each minterm of a single-output function. Different problems can be solved with Boolean relations: decomposition, Boolean matching,... On Chapter 3, a more detailed description on Boolean relations is given and **a new methodology to solve a Boolean relation is presented as a contribution** of the thesis. Another type of flexibility is the *Compatible Set of Permissible Functions* (CSPFs). This set of functions arises from the don't care set of the network and they are described with ISFs.

Finally, the multiple Boolean relation (MBR) generalizes all types of flexibility. BRs and CSPFs are particular cases of a MBR. On [142] is presented a systematic approach to solve MBR. Problems like FPGA Rectification [100] and Synchronous Recurrent Equations [63] can be tackled with this type of flexibility.

Another type of flexibility, not detailed in the figure, is the *set of pairs of functions to be distinguished* (SPFDs). This set is a generalization of the ODC and a specialization of the MBR. A set of allowable functions can be determined from the ODCs of a target function on a Boolean network. Note that, this set of functions can be independent and infeasible to represent by an ISF function. The SPFD is represented as a set of pairs of input values that needs to be distinguished at the outputs. Basically, the objective of a SPFD is to find a function such that satisfies all the pairs of input values. Efficient methods have been developed to compute and solve SPFDs [170] for small-

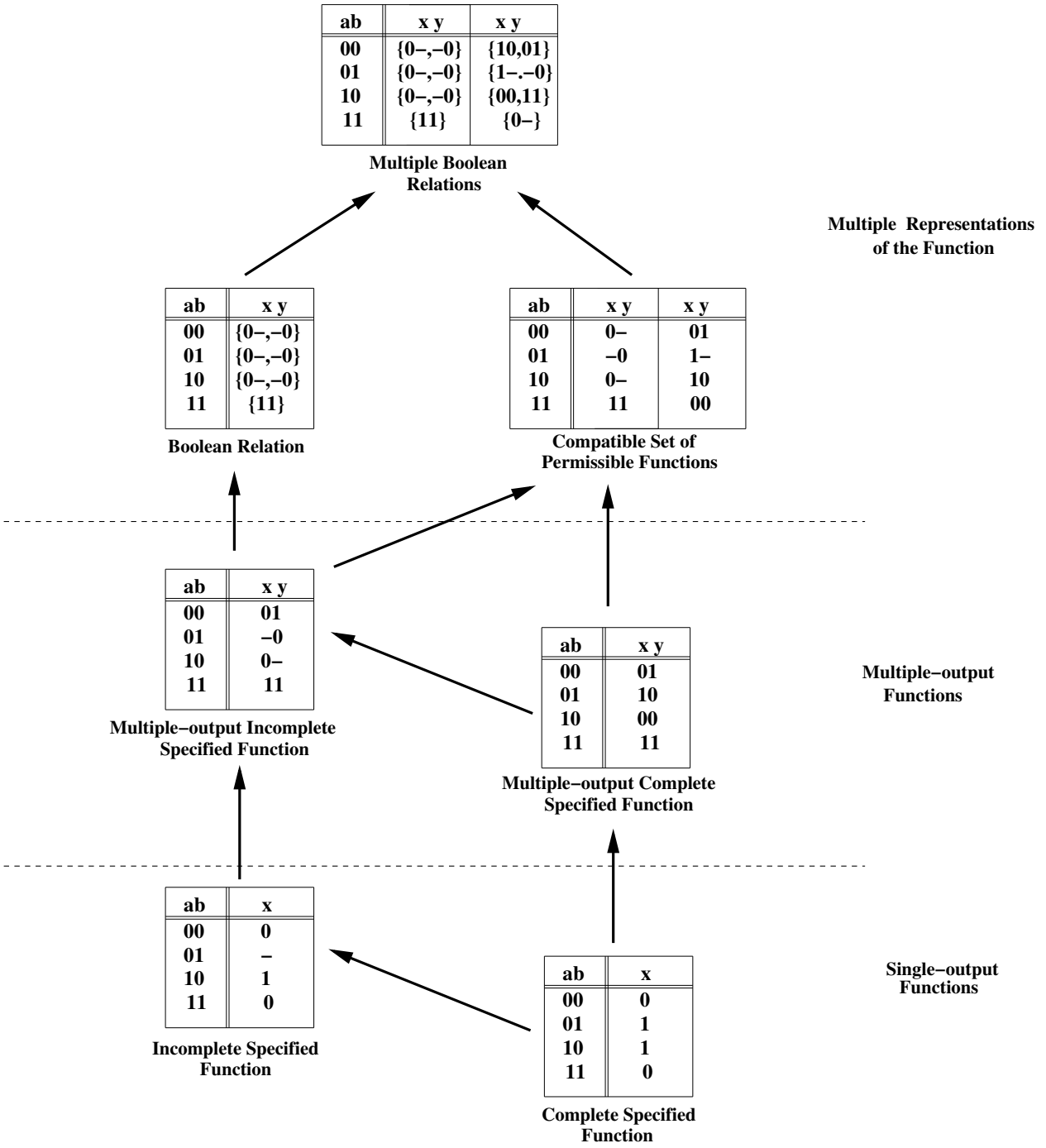


Figure 2.6: Types of flexibility hierarchy.

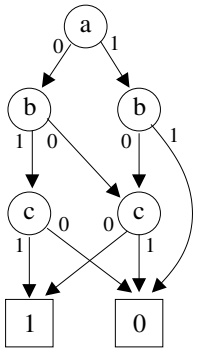


Figure 2.7: A BDD representation of $F = \overline{a}bc + \overline{b}\overline{c}$.

and medium-sized functions. In multi-level minimization, the flexibility of a node of a network can be computed by an SPFD. The best function that satisfies the SPFD can replace the original node. The implementation of the new function may require some changes in the surrounding environment. Other problems can be also tackled with SPFD, for example rewiring optimization on FPGAs [54, 95, 149].

2.1.5 BDD-based optimization

A Binary Decision Diagram (BDD) is a graph structure that is used to represent Boolean functions. The advantage of BDDs is that functions with a large number of variables can be described with a more compact structure compared to other representations, like truth tables. Although the size of a BDD may vary from linear to exponential with regard to the number of input variables in the worst case, many operations have a polynomial complexity.

Example 2.1.6 *Figure 2.7 illustrates an example of a BDD*[3] *for the function* $F = \overline{a}bc + \overline{b}\overline{c}$. *If the BDD is traversed from the root node to the node with constant value one exploring all the feasible paths, the three minterms* $\overline{a}bc$, $a\overline{b}\overline{c}$ *and* $\overline{a}\overline{b}\overline{c}$ *of* F *are found.* □

Although BDDs are relatively old [10, 102], the first applications in VLSI design appeared in the eighties. The new ideas brought by Bryant [38] helped to develop efficient BDDs packages [30, 150] based on Reduced Ordered BDD (ROBDD). With an efficient implementation, a BDD package offers a more efficient memory space representation and a lower cost operations compared to other approaches. Besides, a BDD provides a fast canonicity check. Two functions are equivalent if they have the same BDD representation.

[3]The figure shows the BDD in the Reduced Ordered BDD representation.

BDDs have been used in many problems in logic synthesis and verification, like circuit optimization, testing, combinational and FSM equivalence checking. BDDs are an important symbolic representation for the modelization of finite-state verification problems [82]. In many cases, BDDs have been able to handle state spaces that would be utterly hopeless if the states were stored individually. Currently, many variants of BDDs have been also proposed to improve the efficiency on particular problems: zero-suppressed BDDs (ZDD) [118], algebraic BDDs (ADD) [132], among others.

In this section, we will not survey how to build the BDD representation of a function. An extended explanation can be found in [38]. Several BDD-based logic optimization techniques will be reviewed.

Non-essential Variables [37]

Definition 2.1.6 Non-essential variable. *A variable $x_i \in \{x_1, x_2 \cdots, x_n\}$ of F is non-essential if and only if $F_{x_i} = F_{\overline{x_i}}$.* □

This property is particular useful in incomplete specified functions.

Theorem 2.1.1 [4] *Let be F a function in the interval $[G,H]$. A variable $x_i \in \{x_1, x_2 \cdots, x_n\}$ of F is non-essential if and only if*

$$\exists G_{x_i} \subseteq F_{x_i} \subseteq \forall H_{x_i}$$

□

In Section 2.1.4, an ISF was defined as an interval of feasible complete specified functions. The elimination of the non-essential variables contributes to reduce the size of the support on the ISF without loosing any feasible function. As a consequence, the complexity of the ISF minimization process is also decreased.

Note that, this property has a big impact as a preprocess in ISF minimization based on BDD representation. The elimination of non-essential variables reduces the size of the BDDs and, therefore, the operations that need to traverse the BDD are sped up.

ISOP minimization

The Minato-Morreale [117] method introduced a new way to minimize Boolean functions in BDD representation. Larger functions could be optimized compared to ESPRESSO. The technique is based on the recursive Morreale's algorithm [122] to find Irredundant sum-of-products (ISOP).

[4] The proof can be found in pp. 107 in [37].

This method recursively deletes redundant cubes and literals from a given SOP. Experimentally, it has been proved that this method produces solutions that sometimes are far from the optimal one. However, its implementation in BDDs is efficient in time and memory space.

Constrain-Restrict minimization

Couldert and De Madre [60] defined the *constrain* and *restrict* operations to simplify the size of the BDDs using don't cares. The inputs of these operations are two BDDs. The first BDD represents the function to be minimized and the second one defines the care set of the function.

There is only a basic difference between both operations. During the minimization, the *constrain* operation preserves all the variables of the BDD, meanwhile, the *restrict* operation minimizes the support of the BDD.

Safe minimization

The drawback of the constrain and restrict operations is that the size of the final BDD can increase. To avoid this problem, the implementation of these operations incorporates a final condition to check the size of the result. If there is such a case where the size is larger, the original BDD is returned.

Hong et al. [79] developed several heuristics to perform a safe BDD minimization. In Hong's approach, the size of the BDD is guaranteed. Two heuristics were developed to detect the nodes where a minimization process can be applied without increasing the size of the BDD: *basic compaction* and *leaf-identifying*. Then, the minimization is performed on the identified nodes.

2.2 Decomposition

Decomposition is a transformation method in multi-level minimization. In this section, a review of the *classical* algorithms is performed. These methods use factorization algorithms based on algebraic and Boolean division to obtain a good decomposition. The difference between factorization and decomposition is that decomposition creates a new node in the network for each division.

Intuitively, a good factorization algorithm has to find a good divisor to divide the Boolean function. The factorization algorithm can be recursively applied to the quotient, divisor and remainder of the division until no more divisions can be performed. Good factorization algorithms based on algebraic division are presented in [34] trading-off quality and runtime. Quick factorizations are found by reducing the level of accuracy during the search of good divisors.

Boolean factorization based on Boolean division has been also applied [33]. The recursive method differs on the selection of the divisor and the division algorithm. The divisor is found by selecting the best algebraic kernel and the Boolean division algorithm improves the algebraic one

by involving a two-level minimization step during the process of division. Although the results of the Boolean factorization are better, the runtime can be significantly increased.

In [33], a decomposition approach is also presented. Initially, the nodes are decomposed based on previous factorization methods. Then, area is saved by applying substitution on each node of the network, where common factors are identified, and by eliminating single literal functions.

The previous decomposition method targets at minimizing the size of the network. Timing-driven algorithms has been also presented [56, 120, 154, 169, 171]. Initially, the Boolean network is collapsed to create larger functions, where better factors can be found, and the Boolean functions are recursively decomposed targeting at reducing the number of levels of logic. These approaches, based on bi-decomposition, will be described in more detail in Chapter 5 with our **contribution in timing-driven n-way decomposition**.

2.3 Partitioning

Partitioning is a well-known technique that has been used in all the phases of the design flow and it is typically used to reduce the complexity of the optimization problems at the expense of the quality of the final solution. It has been extensively applied from behavioral synthesis, when the register transfer level (RTL) is partitioned, until physical design, on placement and routing.

Partitioning is often formulated as a min-cut problem. There are two main approaches extensively used to solve min-cut: Kerningham-Lin [93] and Fiduccia-Mattheyses [69]. Many modifications of these two approaches have appeared to solve different types of problems. The objective of these methods is to perform a two-way balanced min-cut bi-partition. Some extensions upgrade these algorithms to multi-way partition.

The Kernigham-Lin heuristic starts with a random balanced bi-partition. Pairs of vertices are swapped between clusters. The exchange with better improvement in the min-cut is selected. This process is iteratively done until no more improvement is found. The swapping heuristic helps to jump from local minimal solutions. However, the cost of this method is $O(n^3)$. Fiduccia-Mattheyses heuristic reduces the complexity of each iteration to linear time. This reduction is achieved by eliminating the swapping operation. The basic difference is that only one cell is moved at a time.

Currently, there is a generic tool that can be also used to perform min-cut partitioning: *hMetis* [89]. hMetis is a general purpose multi-level hypergraph partitioning method that recursively applies bisection to perform a multi-way partition. The improvement in comparison with the previous approaches is related to the fast coarsening method to obtain the partition and the final refinement step to improve the quality of the min-cut. The last publications of this approach [3, 140] showed the possibility to apply multi-objective functions.

Min-cut partitioning is a good option in many problems, like RTL partitioning [92, 165] or placement [4, 71], since the main factor is the graph representation of the circuit. Nevertheless, min-

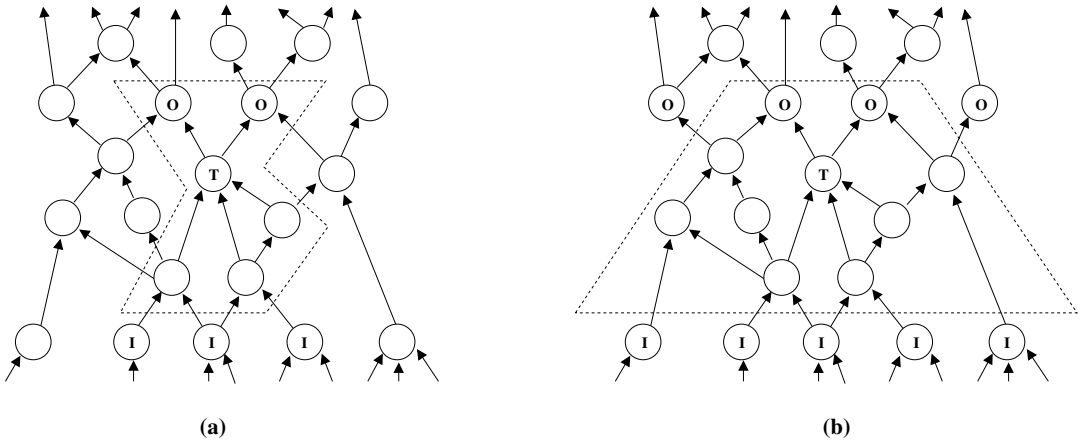


Figure 2.8: (a) Initial window of 1 x 1 from targeted node T. (b) Final window after applying reconverge paths heuristic.

cut is not always the best choice in the logic optimization phase since it does not take into account the internal *logic* behavior of the nodes.

In logic synthesis, partitioning has been also applied to different complex problems. An example is don't care computation. The set of don't cares must be approximated in large networks since it is not affordable the computation, even if a BDD-based approach is used. In [119], a partition method is presented to approximate this calculation. A small *window* of nodes around the targeted node is selected to limit the computation of the don't cares. In Fig. 2.8-(a), a window of 1 x 1 is depicted. Here, the window includes all the fanins and fanouts at distance 1 of the targeted node T. However, this window is too small to compute any useful don't care for logic optimization. In [119], the *reconverge paths* heuristic is used to converge the paths from the outputs to the inputs of the window. This window, illustrated in Fig. 2.8-(b), captures better the don't care information of the outputs of the window.

Another problem where partition can be applied is timing-driven minimization. Several approaches have been developed [9, 45, 46, 55, 125, 156, 172]. In Chapter 4, they will be reviewed together with our **contribution in timing-driven logic partitioning**. Our approach produces partitions based on the theory of vertex dominator [66].

2.4 Technology-dependent techniques

Technology mapping is the link between the logic and the physical domain. After technology mapping, the mapped circuit may not meet the delay constraints of the specification. An incremental

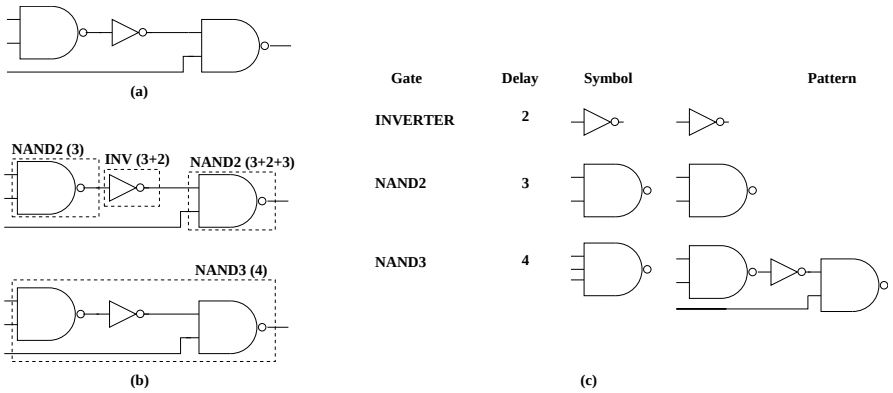


Figure 2.9: (a) Network before Technology mapping. (b) Two possible mappings of Network (a). (c) Library used for mapping.

logic optimization method on the critical sections of the circuit can be applied to improve the result. However, other post-process techniques can be also used. Here, an overview is presented in technology mapping and this post-process techniques like gate sizing, duplication and buffer insertion.

2.4.1 Technology mapping

Technology mapping is a complex problem that consists of mapping a Boolean network into a library of gates to obtain a circuit where each node represents a logic gate. Each gate stored in the library is represented by a logic function and it has associated a technology *cost* in area and delay. The mapped circuit must have a good cost in area and must satisfy the delay constraints (cycle time) imposed in the specifications. Formally, the representation of the network and the logic functions of the gates are defined as *subject graph* and *pattern graph* respectively.

Three general approaches have been used in technology mapping: rule-based, tree-covering, and graph-covering. Rule-based covering [84] was the first developed technique. A database of transformation rules is used to perform the covering. The algorithm searches patterns in the Boolean network and checks if there is some rule associated to this pattern. However, the database is technological dependant of the library of gates and the refinement of the rules depends on a large experimental process.

Tree covering [94] and graph covering [99, 103, 155] improves the results of the previous approach. The Boolean network is divided in a forest of trees in tree covering approach and each tree is independently mapped using a technique similar to the problem of code generation in compilers where the objective is to map a set of expressions onto a set of machine instructions. Graph covering improves the results by increasing the number of covering patterns found, mostly in the

multiple-fanout nodes where tree covering performs the cuts.

The covering between the subject graph and the patterns of the gates specified in the library is performed using a process called *pattern matching*. The most relevant matching approaches are structural matching [7] and Boolean matching [114]. In the structural matching, the patterns are compared using its DAG-representation. This technique is only feasible in small gates since all possible DAG representations on the pattern graph must be found. In the Boolean matching, this problem disappears. The pattern matching is performed based on a canonical representation of the pattern graphs.

Figure 2.9 shows a process of mapping. Picture (a) represents the subject graph and (c) represents the library of gates. The library describes for each gate its name, symbol, pattern and cost. For the sake of simplicity, only one cost is associated to the gate. Note that, an area and delay cost, i.e. resistance, capacitance of the pins, is usually defined for each gate. Initially, the process of matching identifies several patterns for each node as it is shown in (b). The possible covers are identified by an area-delay pair and they are stored in a curve of *Pareto points* where only the best solutions are kept. The selection of the best pattern is done depending on the objective function over the set of points in the curve. In this example, the primary output has two possible pareto points, NAND2 and NAND3 with cost 8 and 4 respectively. Gate NAND3 will be selected due to the best minimization of the global cost.

2.4.2 Gate Sizing

Gate sizing is a well-known approach to meet the timing and power constraints without changing the topology of the circuit. The input is a cell library and a mapped circuit. Gate sizing (or simply sizing) consists of changing cells with functionally-identical ones in the critical paths of the circuit targeting at minimizing the total delay cost.

Gate sizing is a post-technology mapping technique that can be used with or without layout information. Sizing is commonly applied after technology mapping [25] when large CMOS libraries are used. The runtime of a technology mapper mostly depends on the number of pattern graphs that are checked and the number of gates per pattern. Therefore, a representative cell can be defined for each family of cells and they can be used in technology mapping instead of the original library to reduce the runtime complexity. Then, gate sizing is applied with the standard library to refine the solution of the mapping towards the objective functions. Timing [57], power [52], crosstalk noise [75], and coupling noise [148] optimization are some examples of the state-of-art objectives.

2.4.3 Buffer insertion

Buffer insertion contributes to reduce the capacitance on gates with large number of fanouts by inserting multiple inverters in the interconnection. The total area of the circuit can be affected

since a large number of inverters may be needed. However, the improvement on delay and power consumption is considerable.

In load-based delay models, this problem is also known as *fanout optimization* [123, 124, 146] and sometimes these techniques have been integrated with the technology mapping tool as a post-process to improve the large fanout in some cells.

On routing phase, buffer insertion contributes to reduce considerably the delay and the congestion produced by the wires. Here, the problem is also known as *repeater insertion* since a buffer can be also seen as a "small amplifier" of the signal. The most referenced work in repeater insertion was published by Van Ginneken [163]. Buffer insertion is done after routing on top of the existing wires, and buffers are only explored on predefined fixed locations. Other approaches have been developed after this publication, and they will be reviewed on Chapter 6.

2.4.4 Gate duplication

A second technique to deal with high fanout is gate duplication. Gate duplication has been shown as an effective method in gates with large fanout. Basically, the targeted cell is duplicated and the fanouts are divided in two clusters. However, duplication is not always a good choice. Note that, the capacitance in the immediate fanins of the targeted cell increases. Therefore, there is a trade-off between the improved delay on the fanouts and the increased capacitance on the fanins.

An important advantage with regard to buffer insertion is the considerable reduction of the total area and levels of the circuit. Duplication produces a minor number of extra cells in the circuit. The area of application of gate duplication also covers from after technology mapping [151, 152] until placement [43, 96]. These techniques will be also reviewed in Chapter 6.

2.4.5 Physical-aware logic synthesis

In Chapter 6, our last contribution will be presented where **buffer insertion and gate duplication are combined taking into account layout information**. This technique uses the principle of ECO to incrementally improve the current placement design by performing small modifications. A reciprocal feedback between placement and gate duplication and buffer insertion has been implemented.

Chapter 3

A Recursive Paradigm To Solve Boolean Relations

3.1 Introduction

As we introduced in Section 2.1.4, flexibility in logic synthesis can be expressed using different abstract methods like don't care conditions (DCs), Boolean Relations (BRs), Multiple Boolean Relations (MBRs) [142], sets of pairs of functions to be distinguished (SPFDs) [116, 141].

Don't cares form the basis for minimization of incompletely specified functions (ISFs) and multi-level networks. Boolean Relations allow to capture more flexibility than ISFs. However, while minimization of ISFs is a unate covering problem, solving Boolean relations is a binate covering problem and hence is significantly more difficult [116].

Fig. 3.1(a) illustrates an example of a Boolean relation with two input and two output variables. It is a subset of $\mathbb{B}^2 \times \mathbb{B}^2$, where $\mathbb{B} = \{0,1\}$. The input vertex 10 is related to two different output vertices $\{00,11\}$, and 11 is related to another pair $\{10,11\}$. The flexibility for 10 and 11 is different. The latter can be captured by introducing a don't care into the range of output variables ($\{10,11\} \equiv \{1-\}$)[1]. The former, $\{00,11\}$, cannot be expressed with don't cares.

To solve a Boolean relation one needs to find a compatible multi-output function with minimum cost. Figures 3.1(b-c) depict two functions that are compatible with the original Boolean relation.

Many problems in logic design can be reduced to Boolean relations: Boolean matching techniques for library binding [26], FSM encoding [109], Boolean decomposition [64], etc. For example, given a cut in the network, the flexibility of the nodes at the cut can be specified with a Boolean relation. E.g. if the cut contains two nodes y_1, y_2 that reconverge to an *AND* gate, and for a given primary vector the output of the *AND* gate must be 0, then the flexibility at y_1, y_2 is $\{00,01,10\}$.

[1]The don't care value $\{-\}$ denotes that the output can take all the values from the codomain $\mathbb{B}$.

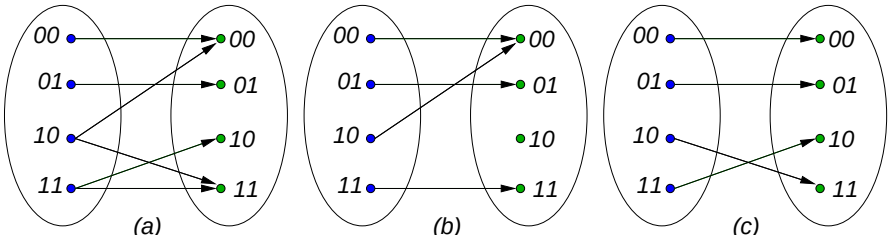


Figure 3.1: Example of Boolean relation (a) and two compatible functions (b,c).

This chapter, based on the results in [18, 21], presents a novel recursive algorithm for solving Boolean relations. The algorithm has an efficient strategy in exploring the large space of solutions and can be used in exact mode (for small relations) and in heuristic approximate mode for larger relations. The cost function can be tuned for different parameters relating to area or delay in computing a Boolean relation. Moreover, the algorithm can further be adjusted to solve Boolean equations. The experimental results show tangible improvements with regard to previous heuristic approaches. As an application of the solver, this chapter describes the use of Boolean relations for the problem of the multi-way decomposition of Boolean functions in logic circuits. The experiments show that significant delay and area improvements can be achieved by using our solver in logic circuit implementation.

The rest of the chapter is organized as follows. Section 3.2 gives an overview of the recursive paradigm. Section 3.3 introduces the previous work in solvers of Boolean relations. Section 3.4 and Section 3.5 present the basic definitions on the Boolean relation domain and the basis of recursive algorithm, respectively. Details of the solver are explained in Section 3.6. The major heuristics used to implement the recursive algorithm are presented in Section 3.7. Section 3.8 introduces how to solve a system of Boolean equations with a Boolean relation. Finally, Section 3.9 reports experimental results and Section 3.10 introduces an application where Boolean relations can be applied.

3.2 Overview

In this section, we will introduce the basis of the recursive paradigm. The formal details will be presented in Section 3.5. Consider the Boolean relation defined in Fig. 3.1(a). For the sake of simplicity, the Boolean relation will be represented with the same notation of sets of elements. The recursive paradigm illustrated in Fig. 3.2 is based on the following steps:
(a) Over-approximate the Boolean relation into a multiple-output incomplete specified function[2]

[2]See Section 3.4 for the formal definitions of Boolean relation and multiple-output incomplete specified function.

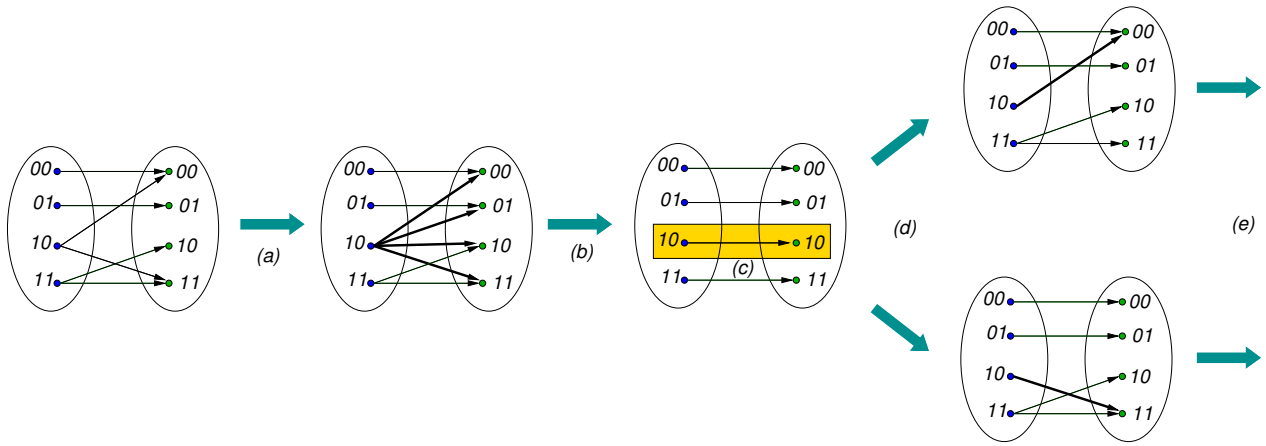


Figure 3.2: Steps of the recursive paradigm implemented in BREL. (a) Over-approximate the Boolean relation to a multiple-output incomplete specified function. (b) Multiple-output incomplete specified function minimization. (c) Selection of one input vertex where there is a conflict. (d) Decomposition in two new Boolean relations. (e) Recursively solve the subrelations.

(MISF): The input vertices such that their output vertices cannot be captured with conventional don't cares are expanded to cover more vertices of the output set. In the Boolean relation presented in Fig. 3.1(a), the output $\{00, 11\}$ of the input vertex $\{10\}$ cannot be covered with don't cares. Therefore, it is expanded to $\{--\}$.

(b) Use a standard MISF minimization method to obtain a multiple-output function covered by the MISF.

(c) If the resulting function has no conflicts with the original relation, then report the result. Otherwise, select one element of the input set where there is an incompatibility with the original Boolean relation. In the example, the incompatibility appears in the input vertex $\{10\}$, since in the resulting function it maps to the output vertex $\{10\}$ that was not in the original range ($\{00, 11\}$) for this input vertex.

(d) Decompose the original Boolean relation into two smaller relations by creating a partition in the range of the output vertices of the selected incompatible input vertex.

(e) Recursively solve the smaller Boolean relations and select the best compatible solution out of the explored solutions.

3.3 Previous work

Several exact and heuristic approaches have been proposed to solve Boolean relations. The exact methods reported in [35,36] tackle the problem of solving a Boolean Relation similarly to the Quine-McCluskey procedure [115]. The definitions of prime and prime implicant in Boolean functions are generalized to candidate prime (c-prime) and c-prime implicant in Boolean relations. Analogous to the Quine-McCluskey procedure, first all c-primes are generated and, then, the minimization is formulated as a binate covering problem (BCP). The covering problem is solved by Integer Linear Programming [139], where the objective is to find optimum sum-of-products representation of the Boolean relation. Other exact methods were presented in [85, 109] using a Branch-and-Bound algorithm based on the BCP formulation. The major contribution of these approaches was the representation of the constraints of the BCP with Binary Decision Diagrams [30]. This compact representation helped to solve larger relations consuming less memory. However, the exact methods are limited to solve small and medium instances due to the complexity.

Heuristic methods provide approximated solutions with a trade-off between the quality of the solutions and the computational complexity. Herb [72] was the first heuristic method for Boolean relations based on two-level minimization and test pattern generation techniques. The ESPRESSO [32] approach was taken as a reference in the sense that the loop reduce-expand-irredundant is repeatedly applied as long as the cost of the solution decreases. The drawback of this procedure is that the test pattern generation methodology limits the expand operation to one variable at a time. This reduction restricts the search space and increases the overall runtime. gyocro [166] was proposed as another heuristic approach also based on ESPRESSO where some of the Herb's weaknesses were amended. Basically, the difference appears in the expand procedure where multiple variables can be taken. The objective cost function in gyocro is slightly different compared with the previous approaches. The minimum sum-of-products with the smallest number of literals per product is searched.

Our experience demonstrates that the number of products is not necessarily a good metric for estimating the quality of the solutions. Sometimes one needs other objectives, e.g. to balance the functions for delay optimization or to balance the support of the functions for reducing layout congestion. The recursive approach presented in this chapter accepts a customizable cost function that allows to guide the search towards a user-defined goal. We also observed that heuristic methods, like gyocro, often cannot escape from local minima determined by the initial solution, since the reduce-expand-irredundant loop is not always capable of hill climbing. An example of this limitation is presented in Section 3.9.1.

3.4 Preliminaries

Definition 3.4.1 Boolean function. *A Boolean function f is a function $f : \mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} = \{0,1\}$. A Boolean function can also be interpreted as the set of vertices $x \in \mathbb{B}^n$ such that $f(x) = 1$.* □

Definition 3.4.2 Literals, minterms, cubes and covers. *A* literal *is a variable or its complement. The conjunction (or product) of a set of literals is called a* cube*. A cube is called a* minterm *when the number of literals of the cube corresponds to the number of variables of the function. A function can be represented by a* cover *that is defined as a disjunction (or sum) of cubes.* □

Definition 3.4.3 Multiple-output Boolean function. *A multiple-output Boolean function f is a function $f : \mathbb{B}^n \to \mathbb{B}^m$. It can be also specified as a vector of Boolean functions $f = (f_1, f_2, \ldots, f_m)$.* □

Hereafter, we will use $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$ to denote the set of inputs and outputs of a multiple-output Boolean function respectively.

Definition 3.4.4 Incompletely specified Boolean function. *An incompletely specified Boolean function (ISF) is a function $f : \mathbb{B}^n \to \mathbb{B} \cup \{-\}$, where $-$ is called the* don't care *value of the function. An ISF can be specified by three Boolean functions:* $\mathsf{OFF}(f)$*,* $\mathsf{ON}(f)$ *and* $\mathsf{DC}(f)$ *that characterize the vertices in $\mathbb{B}^n$ with image* 0*,* 1 *and $-$, respectively.* □

An ISF defines an interval of Boolean functions between $\mathsf{ON}(f)$ and $\mathsf{ON}(f) \cup \mathsf{DC}(f)$. An implementation of an ISF f is a Boolean function $\hat{f}$ such that

$$\mathsf{ON}(f) \subseteq \hat{f} \subseteq \mathsf{ON}(f) \cup \mathsf{DC}(f)$$

Definition 3.4.5 Multiple-output incompletely specified function. *A multiple-output* ISF *(*MISF*) is a function $f : \mathbb{B}^n \to (\mathbb{B} \cup \{-\})^m$. It can be also specified as a vector of* ISF*s $f = (f_1, f_2, \ldots, f_m)$.* □

An MISF also defines an interval of multiple-output functions. The objective of a two-level minimizer is to find a function with the minimum (or minimal) sum-of-products representation that covers the MISF. Efficient methods for computing minimal sum-of-products representations are well-known [6, 61, 134].

Definition 3.4.6 Boolean relation. *A Boolean relation (BR) R is a subset of $\mathbb{B}^n \times \mathbb{B}^m$, where $\mathbb{B}^n$ and $\mathbb{B}^m$ are called the input and output sets of R, respectively. A Boolean relation is* left-total *if for all $x \in \mathbb{B}^n$, there is $y \in \mathbb{B}^m$ such that $(x,y) \in R$. We will also refer to the left-total Boolean relations as* well-defined *following the nomenclature of [166]. A Boolean relation is* functional *if every input vertex is associated with a single output vertex.* □

Reusing the notation for the multiple-output functions, $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$ denote the set of inputs and outputs of a relation. Hereafter, we will indistinctively use the terms Boolean relation, BR and relation.

Definition 3.4.7 Natural join [50]. *The natural join over the input set X between two relations R and S is defined as*

$$R(X,Y) \bowtie_X S(X,Z) = \{(x,y,z) | (x,y) \in R \ \wedge \ (x,z) \in S\}$$

Note that, when the relations R and S have the same input and output set and the natural join is applied over all the variables, the natural join is equivalent to the intersection operator. □

The next two definitions describe how functions, ISFs and MISFs can be represented with the notation of Boolean relations.

Definition 3.4.8 Relationship between an incomplete function and a BR. *The* don't care *value* $\{-\}$ *assigned to the output of a minterm of an* ISF *denotes all the permissible values that the minterm can take from* $\mathbb{B}$. *Therefore, an* ISF f_y *can be also interpreted as a Boolean relation* $F_y \subseteq \mathbb{B}^n \times \mathbb{B}$ *such that*

- $(x,0) \in F_y$ *if and only if* $f_y(x) \in \{0,-\}$,
- $(x,1) \in F_y$ *if and only if* $f_y(x) \in \{1,-\}$.

where $f_y(x) = \{-\}$ *implies that both* $(x,0)$ *and* $(x,1)$ *belong to the relation. This definition implies a mutual relationship between left-total Boolean relations* $F_y \subseteq \mathbb{B}^n \times \mathbb{B}$ *and* ISFs.

An MISF f *can be also defined as a Boolean relation* $R \subseteq \mathbb{B}^n \times \mathbb{B}^m$ *such that*

$$R(X, y_1, y_2, \ldots, y_m) = \bigtriangledown_{i \in \{1,\ldots,m\}} F_{y_i}(X, y_i)$$

□

Example 3.4.1 *Consider the next two* ISF *functions in tabular representation:*

x_1x_2	y_1
00	0
01	0
10	–
11	1

x_1x_2	y_2
00	0
01	1
10	–
11	–

They can be represented as Boolean relations from Def. 3.4.8 where the don't care $\{-\}$ *takes all the permissible values from* $\mathbb{B}$. *Similarly, the defined* MISF *from the conjunction of these* ISFs *can*

be also described as a Boolean relation from Def. 3.4.8. The resulting Boolean relations of the ISF*s and the* MISF *are as follows:*

ISF*s*		
x_1x_2	y_1	y_2
00	$\{0\}$	$\{0\}$
01	$\{0\}$	$\{1\}$
10	$\{0,1\}$	$\{0,1\}$
11	$\{1\}$	$\{0,1\}$

MISF	
x_1x_2	y_1y_2
00	$\{00\}$
01	$\{01\}$
10	$\{00,01,10,11\}$
11	$\{10,11\}$

□

Definition 3.4.9 Compatible functions. *Given a Boolean relation R, the set of multiple-output functions compatible with R is defined as*

$$\mathbb{F}(R) = \{F \mid F \subseteq R \wedge F \text{ is a multiple-output function}\}$$

Note that $\mathbb{F}(R) = \emptyset$ *if R is not well defined.* □

Example 3.4.2 *This example below shows a tabular representation of the Boolean relation that corresponds to Fig. 3.1(a).*

x_1x_2	y_1y_2
00	$\{00\}$
01	$\{01\}$
10	$\{00,11\}$
11	$\{10,11\}$

The two Boolean functions below illustrate examples of a compatible and an incompatible function for the previously defined BR.

x_1x_2	y_1y_2
00	00
01	01
10	11
11	11

Compatible function

x_1x_2	y_1y_2
00	00
01	01
10	10
11	11

Incompatible function

□

We will next discuss the basic principles of solving BRs (Section 3.5), the details of the BR solver (Section 3.6) and the lower level implementation aspects (Section 3.7).

3.5 Basics of Solving a Boolean relation

3.5.1 Semi-lattice of well-defined Boolean relations

In this section, the boundaries of the search space are defined. As we will show, the search space of the well-defined Boolean relations is a semi-lattice. To demonstrate the existence of the semi-lattice, first, let us prove that there is a lattice over the set of Boolean relations in $\mathbb{B}^n \times \mathbb{B}^m$.

Property 3.5.1 Lattice of Boolean relations. *$(R, \subseteq)$ is a lattice with the top element $\mathbb{B}^n \times \mathbb{B}^m$ and the bottom element $\emptyset$. The join and meet operations are the intersection and the union of relations, respectively.* □

The proof of this property follows directly from the properties of the union and the intersection on sets of finite Boolean vectors.

Lemma 3.5.1 *If R is a functional Boolean relation and $R' \subset R$, then R' is not well defined.*
Proof: *By definition, there is only one output vertex for each input vertex in a functional Boolean relation. A relation R' such that $R' \subset R$ has at least one input vertex without image on $\mathbb{B}^m$. Thus, R' is not well defined.* □

Finally, the definition of the semi-lattice is straightforward from the previous lemmas.

Theorem 3.5.1 Semi-lattice of well-defined Boolean relations. *The set of well-defined Boolean relations with the partial order $\subseteq$ is a semi-lattice with one greatest element $\mathbb{B}^n \times \mathbb{B}^m$ and 2^{m2^n} least elements that correspond to the elements of $\mathbb{F}(\mathbb{B}^n \times \mathbb{B}^m)$.*
Proof: *Two statements have to be proved: the existence of the semi-lattice and the upper and lower bounds of this semi-lattice. First, the semi-lattice of well-defined Boolean relations can be easily derived from Property 3.5.1. A lattice implicitly defines two semi-lattices, one for each operator (union and intersection). Therefore, the semi-lattice over the operator union exists and, therefore, over the partial order $\subseteq$. Second, let us demonstrate the greatest and the least elements;*

- *The supremum element is $\mathbb{B}^n \times \mathbb{B}^m$. It is clearly well defined. There is no other relation that subsumes it.*
- *The least lower bound elements are all the compatible multiple-output functions with n inputs and m outputs from the set $\mathbb{F}(\mathbb{B}^n \times \mathbb{B}^m)$ defined by Def. 3.4.9. Lemma 3.5.1 defines that there is no relation R' such that $R' \subset \mathbb{F}(\mathbb{B}^n \times \mathbb{B}^m)$ and R' well defined. The number of elements in $\mathbb{F}(\mathbb{B}^n \times \mathbb{B}^m)$ is equal to 2^{m2^n} that is the product of m single output functions from a set of 2^{2^n} possible Boolean functions.*

□

3.5.2 Projection of a Boolean relation to a Multiple-output ISF

This section introduces a method to obtain an MISF from a Boolean relation. This approximation is useful to derive a fast solution for a Boolean relation.

Definition 3.5.1 Projection of a Boolean relation. *The projection of a relation $R(X,Y)$ onto the output y_i is another relation $(R \downarrow y_i)$ such that*

$$\begin{aligned}(R \downarrow y_i) \;=\; & \{(X,z) \mid \exists y_1,\ldots,y_{i-1},y_{i+1},\ldots,y_m \\ & \textit{such that } (X,y_1,\ldots,y_{i-1},z,y_{i+1},\ldots,y_m) \in R\}\end{aligned}$$

The projection of a well-defined relation R onto one output y_i implicitly defines an ISF for that output. Note that, the projection can be extended to multiple outputs. □

Example 3.5.1 *From the relation presented in Example 3.4.2 the following projections can be derived:*

x_1x_2	$R \downarrow y_1$
00	$\{0\}$
01	$\{0\}$
10	$\{0,1\}$
11	$\{1\}$

x_1x_2	$R \downarrow y_2$
00	$\{0\}$
01	$\{1\}$
10	$\{0,1\}$
11	$\{0,1\}$

□

Definition 3.5.2 MISF covering a Boolean Relation. *Given a Boolean relation R, an* MISF *covering R can be obtained as follows:*

$$\mathsf{MISF}_R(X,Y) = \bowtie_{X}\limits_{i\in\{1,\ldots,m\}} (R \downarrow y_i)$$

The relation $\mathsf{MISF}_R(X,Y)$ *is a vector of* ISF*s and, hence, it is an* MISF. □

Example 3.5.2 *From the projections of the relation presented in Example 3.4.2, the tabular representation of the original relation and the* MISF_R *is shown next*

x_1x_2	y_1y_2
00	$\{00\}$
01	$\{01\}$
10	$\{00,11\}$
11	$\{10,11\}$

Boolean relation R

x_1x_2	y_1y_2
00	$\{00\}$
01	$\{01\}$
10	$\{00,01,10,11\}$
11	$\{10,11\}$

MISF_R

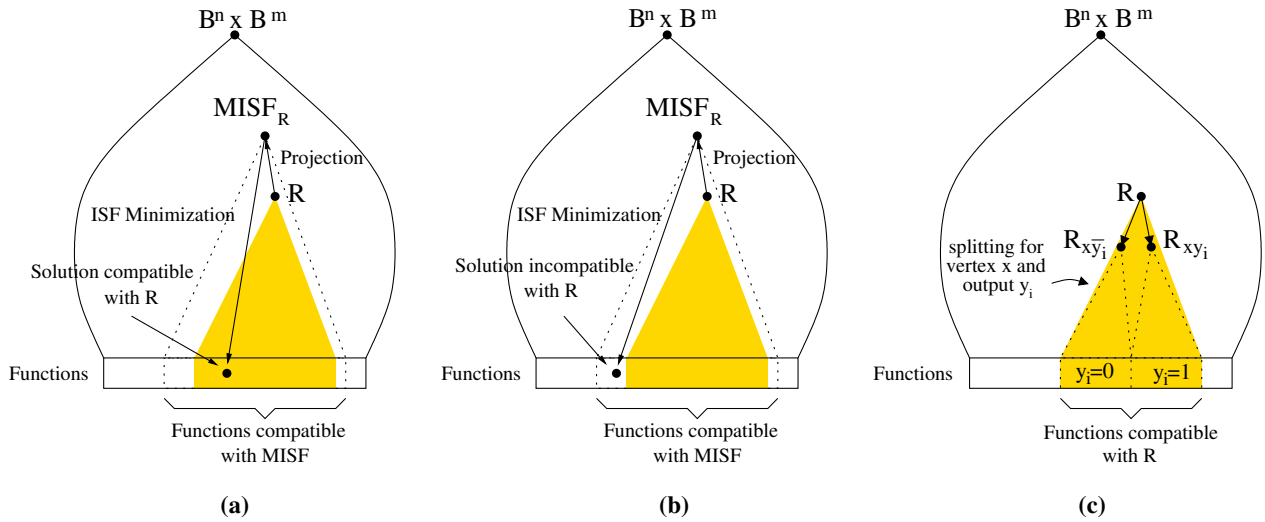


Figure 3.3: Solving a Boolean Relation R in the semi-lattice of well-defined Boolean relations. (a) The BR R is projected to MISF_R and MISF_R is solved with an MISF minimizer. In this case, the obtained solution is compatible with the original relation and the process is stopped. (b) The final solution is compatible with MISF_R but incompatible with the BR. (c) The split operation is applied on the BR R. The set of all the compatible functions of R is still contained in the set of compatible functions for $R_{xy_i} \cup R_{x\overline{y_i}}$

In this example, we can observe the image of the input vertex 10 *in* MISF_R *covers the output vertices* $\{01,10\}$ *that are not included in* $R(10)$. *This is because* MISF_R *effectively expands the output set* $\{00,11\}$ *to the smallest covering cube* $\{--\} = \{00,01,10,11\}$. □

Property 3.5.2 *The following property holds between a well-defined Boolean relation* R *and* MISF_R*:*

$$R(X,Y) \subseteq \mathsf{MISF}_R(X,Y)$$

Proof: *Let us assume that only two output variables are involved.*

$$R = (R \downarrow y_1 y_2) \subseteq (R \downarrow y_1) \bowtie_x (R \downarrow y_2)$$

This assumption can be proved as follows.

$$\begin{aligned} \forall (x,y_1,y_2) \in R \quad &\Longrightarrow \quad (x,y_1) \in (R \downarrow y_1) \ \wedge \ (x,y_2) \in (R \downarrow y_2) \\ &\Longrightarrow \quad (x,y_1,y_2) \in (R \downarrow y_1) \bowtie_x (R \downarrow y_2) \end{aligned}$$

The previous statement can be generalized to multiple outputs

$$(R \downarrow y_i \dots y_j) \subseteq (R \downarrow y_i) \bowtie_x (R \downarrow y_{i+1} \dots y_j)$$

Finally, the property $R(X,Y) \subseteq \mathsf{MISF}_R(X,Y)$ *can be proved based on the previous statement and the next formula*

$$\begin{aligned} R &\subseteq (R \downarrow y_1) \bowtie_x (R \downarrow y_2 \dots y_m) \subseteq \\ &\subseteq (R \downarrow y_1) \bowtie_x (R \downarrow y_2) \bowtie_x (R \downarrow y_3 \dots y_m) \subseteq \\ &\subseteq \dots \subseteq \\ &\subseteq (R \downarrow y_1) \bowtie_x \dots \bowtie_x (R \downarrow y_{m-2}) \bowtie_x (R \downarrow y_{m-1} y_m) \subseteq \\ &\subseteq \mathsf{MISF}_R \end{aligned}$$

□

The next property is important for the presented method, since the MISF_R obtained by projection to the outputs is the smallest MISF that still covers all the compatible functions of R.

Property 3.5.3 *Given a Boolean relation R and the* MISF_R *obtained from the projection onto the outputs, there is no other* MISF f' *such that* $R \subseteq f' \subset \mathsf{MISF}_R$
Proof: *By contradiction. Let us assume that* f' *exists. This statement implies that* $R \subseteq f' \subset \mathsf{MISF}_R$. *Taking into account that an* MISF *is a vector of* ISF*s:*

$$\bowtie_{\substack{X \\ i \in \{1,\dots,m\}}} (f' \downarrow y_i) \subset \bowtie_{\substack{X \\ i \in \{1,\dots,m\}}} (\mathsf{MISF}_R \downarrow y_i)$$

The previous statement implies that there is an output $y_i \in \{y_1, \dots, y_m\}$ *that generates an* ISF *such that* $(R \downarrow y_i) \subseteq (f' \downarrow y_i) \subset (\mathsf{MISF}_R \downarrow y_i)$. *However, this* ISF *does not exist since* $\forall y_i \in Y$, $(R \downarrow y_i) = (\mathsf{MISF}_R \downarrow y_i)$ *by Def. 3.5.2 and, hence,* $(f' \downarrow y_i) \subset (R \downarrow y_i)$. □

3.5.3 Solution of a Multiple-output ISF

This section describes the method to obtain a fast solution from the Boolean relation MISF_R that covers the relation R. Note that, we cannot guarantee the compatibility of the solution with R.

The MISF generated from the projection onto the single outputs is solved performing an individual minimization of the outputs with an ISF minimizer [22, 62, 137].

Example 3.5.3 *A possible solution for the* ISF*s* $(R \downarrow y_1)$ *and* $(R \downarrow y_2)$ *of Example 3.4.2 are*

x_1x_2	y_1
00	0
01	0
10	1
11	1

x_1x_2	y_2
00	0
01	1
10	0
11	1

The multiple-output function for the MISF_R *is shown next*

x_1x_2	y_1y_2
00	00
01	01
10	10
11	11

□

As it is shown in Fig. 3.3(a-b), a Boolean relation R can be projected to MISF_R where an MISF minimizer can be used. If the Boolean relation R is an MISF, then $R = \mathsf{MISF}_R$ and the solution is always compatible with the relation. However, the compatibility of the solution is not guaranteed when the Boolean relation R is not an MISF since $R \subset \mathsf{MISF}_R$.

Definition 3.5.3 Compatibility of a function with a Boolean relation. *Given a multiple-output function F and a relation $R \subseteq \mathbb{B}^n \times \mathbb{B}^m$, F is* compatible *with R if $F \subseteq R$. In general, we define the set of pairs of inputs and output vertices of F incompatible with R as:*

$$\mathsf{Incomp}(F,R) = F \setminus R$$

□

Example 3.5.4 *The multiple-output function presented in Example 3.5.3 is an incompatible solution of the relation of Example 3.4.2. The incompatible pair is* $\mathsf{Incomp}(F,R) = \{(10,10)\}$. □

3.5.4 Divide-and-conquer

Let us next discuss the basis of the divide-and-conquer approach presented in this chapter for dealing with relations in which the solution of the projected MISF_R is incompatible with the original relation.

Definition 3.5.4 Splitting of a Boolean relation. *Let $R \subseteq \mathbb{B}^n \times \mathbb{B}^m$ be a well-defined relation, $x \in \mathbb{B}^n$, and y_i one of the outputs of the relation. The following two relations can be defined:*

$$\begin{aligned} R_{xy_i}(X,Y) &= R - \{(x, y_1, \ldots, y_{i-1}, 0, y_{i+1}, \ldots, y_m)\} \\ R_{x\overline{y_i}}(X,Y) &= R - \{(x, y_1, \ldots, y_{i-1}, 1, y_{i+1}, \ldots, y_m)\} \end{aligned}$$

We denote the previous operation by

$$(R_{xy_i}, R_{x\overline{y_i}}) = \mathsf{Split}(R, x, y_i).$$

□

The Split operation is graphically illustrated in Fig. 3.3(c). Intuitively, given an input vertex x of the input set and one output y_i, the relation can be split into two relations such that one of them takes the value $y_i = 0$ and the other takes the value $y_i = 1$ for the vertex x. The two relations induce a partition over the functions compatible with R. The relations R_{xy_i} and $R_{x\overline{y_i}}$ still cover all the compatible solutions of R and no other functions.

Example 3.5.5 *Let us take the input vertex* $\{10\}$ *and the output* y_1 *from the relation in Example 3.4.2. Note that, the selected input vertex* $\{10\}$ *is included in* $\mathsf{Incomp}(F,R)$. *The objective of this selection is to remove this incompatibility in* R_{xy_1} *and* $R_{x\overline{y_1}}$. *Then,* R_{xy_1} *and* $R_{x\overline{y_1}}$ *are defined by the following tables:*

R_{xy_1}	
x_1x_2	y_1y_2
00	$\{00\}$
01	$\{01\}$
10	$\{11\}$
11	$\{10,11\}$

$R_{x\overline{y_1}}$	
x_1x_2	y_1y_2
00	$\{00\}$
01	$\{01\}$
10	$\{00\}$
11	$\{10,11\}$

R_{xy_1} *and* $R_{x\overline{y_1}}$ *are smaller relations. We can now recursively solve each one of them and choose the best solutions. After minimizing each one, the following multiple-output functions are obtained*

F_1	
x_1x_2	y_1y_2
00	00
01	01
10	11
11	11

F_2	
x_1x_2	y_1y_2
00	00
01	01
10	00
11	11

compatible with R_{xy_1} *and* $R_{x\overline{y_1}}$, *respectively and, therefore, compatible with R.* □

Property 3.5.4 *Given R,* R_{xy_i} *and* $R_{x\overline{y_i}}$ *as defined above, the sets of compatible functions* $\mathbb{F}(R_{xy_i})$ *and* $\mathbb{F}(R_{x\overline{y_i}})$ *are a partition of* $\mathbb{F}(R)$.

Proof: *A partition has the following properties:*

$$\mathbb{F}(R) = \mathbb{F}(R_{xy_i}) \cup \mathbb{F}(R_{x\overline{y_i}});\ \ \mathbb{F}(R_{xy_i}) \cap \mathbb{F}(R_{x\overline{y_i}}) = \emptyset.$$

Let us prove each one independently:

- $\mathbb{F}(R) = \mathbb{F}(R_{xy_i}) \cup \mathbb{F}(R_{x\overline{y_i}})$*: By the definition of the* Split *operation,* $R = R_{xy_i} \cup R_{x\overline{y_i}}$*. Therefore:*

$$\begin{aligned}
&(\mathbb{F}(R) = \{F \mid F \subseteq R\}) \ \wedge \ (R = R_{xy_i} \cup R_{x\overline{y_i}}) \implies \\
&\mathbb{F}(R) = \{F \mid F \subseteq R_{xy_i} \cup R_{x\overline{y_i}}\} \implies \\
&\mathbb{F}(R) = \{F \mid F \subseteq R_{xy_i}\} \cup \{F \mid F \subseteq R_{x\overline{y_i}}\} \\
&\qquad = \mathbb{F}(R_{xy_i}) \cup \mathbb{F}(R_{x\overline{y_i}})
\end{aligned}$$

- $\mathbb{F}(R_{xy_i}) \cap \mathbb{F}(R_{x\overline{y_i}}) = \emptyset$*: The Boolean relations* R_{xy_i} *and* $R_{x\overline{y_i}}$ *differ on the output vertices of the input vertex x, such that* $R_{xy_i}(x) \cap R_{x\overline{y_i}}(x) = \emptyset$*. Therefore, there is no Boolean function F such that* $F \subseteq R_{xy_i}$ *and* $F \subseteq R_{x\overline{y_i}}$*.* □

The next theorem defines the conditions for R_{xy_i} and $R_{x\overline{y_i}}$ to be well defined and strictly smaller than R.

Theorem 3.5.2 *Consider an input vertex x and an output* y_i *of the relation R.* R_{xy_i} *and* $R_{x\overline{y_i}}$ *obtained from* Split(R, x, y_i) *are both well defined and strict subsets of R (i.e.* $R_{xy_i} \subset R$ *and* $R_{x\overline{y_i}} \subset R$*) iff R is well defined and* $(R \downarrow y_i)(x) = \{0,1\}$*.*

Proof: *When the* Split *operation is performed on an input vertex x such that* $(R \downarrow y_i)(x) = \{0,1\}$*, it can be easily proven that* R_{xy_i} *and* $R_{x\overline{y_i}}$ *are well defined. By the Def. 3.5.4,* $\forall x' \neq x$*,* R_{xy_i} *and* $R_{x\overline{y_i}}$ *have the same output vertices, and for the input vertex x, the output vertices are split in such a way that* $(x, y_1, \ldots, y_{i-1}, 1, y_{i+1}, \ldots, y_m) \in R_{xy_i}$ *and* $(x, y_1, \ldots, y_{i-1}, 0, y_{i+1}, \ldots, y_m) \in R_{x\overline{y_i}}$*. Therefore, both* R_{xy_i} *and* $R_{x\overline{y_i}}$ *are still well defined and, moreover, both are strict subsets of R since at least one of the output vertices is dropped for both sub-relations. Let us assume for the contrary that* $(R \downarrow y_i)(x) \neq \{0,1\}$*, e.g.,* $(R \downarrow y_i)(x) = \{0\}$*. Then* $R_{x\overline{y_i}} = R$ *is well defined, but not a strict subset of R, while* R_{xy_i} *is not left-total and hence not well defined.* □

Example 3.5.6 *In the Example 3.5.5, the input vertex* $\{10\}$ *and the output* y_1 *are used in the* Split *operation. Note that if the vertex to split would be* $\{11\}$*, then* $R_{x\overline{y_1}}$ *would not be well defined, since* y_1 *cannot take the value* 0 *for this input vertex.*

□

3.6 Details of the Boolean relation solver

This section describes first a naive Boolean relation solver and then the recursive algorithm based on a branch-and-bound strategy. Let us start with introducing the representation of Boolean relations with characteristic functions that is used in our implementation.

```
QuickSolver (R)
{Input: A well-defined relation R(X,Y)}
{Output: A multi-output function compatible with R}
   S := R;
   for each output y_i do
      F_{y_i} := (y_i ⇔ Minimize(S ↓ y_i));
      S := S ∧ F_{y_i};
   return S;
end;
```

Figure 3.4: A naive algorithm to solve a Boolean relation.

3.6.1 Characteristic functions

A Boolean relation can be represented by its characteristic function.

Definition 3.6.1 Characteristic functions. *A Boolean relation R can be specified by a characteristic function* $\mathcal{R}$[3] $: \mathbb{B}^n \times \mathbb{B}^m \rightarrow \mathbb{B}$*, such that* $(x,y) \in R$ *if and only if* $\mathcal{R}(x,y) = 1$. □

Characteristic functions are convenient for the automation of solving Boolean relations since it enables reusability of algorithms and tools developed for Boolean functions.

Definition 3.6.2 Cofactor and existential abstraction. *The cofactors* f_{x_i} *and* $f_{\overline{x_i}}$ *of a Boolean function* $f(x_1,\ldots,x_n)$ *are defined as* $f_{x_i} = f(x_1,\ldots,x_{i-1},1,x_{i+1},\ldots,x_n)$ *and* $f_{\overline{x_i}} = f(x_1,\ldots,x_{i-1},0,x_{i+1},\ldots,x_n)$*. The existential abstraction* $\exists_{x_i} f$ *is defined as* $\exists_{x_i} f = f_{x_i} + f_{\overline{x_i}}$*. Cofactors and existential abstraction can be extended to multiple variables.* □

3.6.2 Quick solver

The algorithm presented in Fig. 3.4 allows to obtain a solution of the BR quickly. It was used in gyocro [166] to obtain the initial solution before applying the reduce-expand-irredundant iterations. The quick solver minimizes each output in order using the maximum flexibility provided by the relation. As long as the outputs are calculated, the constraints of the previous solutions are propagated to the rest of the outputs. The core of the algorithm is the function Minimize that performs the ISF minimization. Although this algorithm is fast, it has two drawbacks:

- The solution depends on the order in which the outputs are minimized.

[3] We will refer the characteristic function of a Boolean relation with the symbol $\mathcal{R}$.

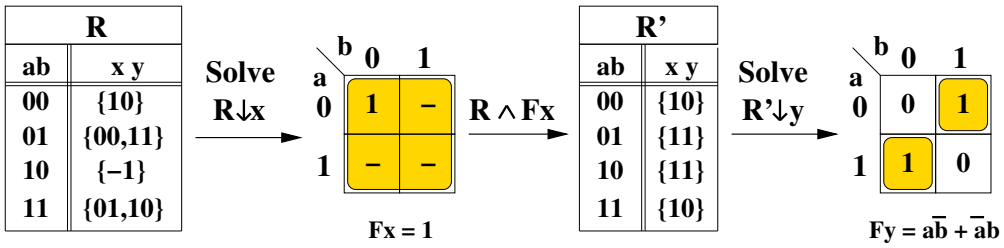


Figure 3.5: Example of solving a BR with QuickSolver.

- The first outputs tend to take advantage of the flexibility of the relation, whereas the last outputs inherit little flexibility. This leads to highly unbalanced and sub-optimal solutions.

Example 3.6.1 *Consider the example of Fig. 3.5. First, the flexibility of the output x is captured and the* ISF *is solved. Based on the solution F_x, the original BR R is constrained to the BR R' such that $(R' \downarrow x) = F_x$. The* ISF *for the output y is extracted from the relation R' and is solved. The solution is $f(a,b,x,y) = (x \Leftrightarrow 1)(y \Leftrightarrow a\overline{b} + \overline{a}b)$. Note that, the best function with the smallest number of product terms, $f(a,b,x,y) = (x \Leftrightarrow \overline{b})(y \Leftrightarrow a)$, can not be found with the quick solver.* □

The goal of this chapter is to propose a method that performs a better exploration of the space of solutions, while having an affordable computational complexity.

3.6.3 The recursive approach

The approach proposed in this chapter is based on the Split operation presented in Def. 3.5.4. The intuitive basis of this approach can be informally described as follows:

> Minimize each output independently with the maximum flexibility provided by the relation.
> If the solution is incompatible:
> Select a conflicting input vertex and an output.
> Generate two sub-relations.
> Branch-and-bound through the tree of BRs in which the leaves are the compatible functions.
> Select the best solution.

The recursive algorithm is shown in more detail in Fig. 3.6, where the cost of the best explored solution is used to prune the search space.

BREL is initially called with a null BR with infinite cost. Only the best solution is preserved in BestF. The algorithm checks if $\mathcal{R}$ is a function (the terminal case) in lines 1-3. In case $\mathcal{R}$ is not a function, the minimization of the $\mathsf{MISF}_{\mathcal{R}}$ is performed in lines 4-5 with BDD-based optimization methods (as further explained in Section 3.7.5). The solution, even if it is incompatible, is rejected if its cost is greater than the cost of the best previously obtained function (line 6). In case of an incompatible solution, constraining the relation further for solving the conflicts cannot improve the cost of a solution obtained for the problem with higher flexibility. If the new cost is smaller than the previous one, the compatibility of the solution is checked (line 7). In case the solution is incompatible (lines 9-10), an input vertex and an output are selected from the incompatible points to perform the Split operation based on Theorem 3.5.2. The largest input cube within the characteristic function of all input conflicting vertices and an output such that $(R \downarrow y_i)(x) = \{0,1\}$ are selected to apply the Split operation (further discussion in Section 3.7.4). Finally, the recursive calls are done (lines 11-12) for each of the smaller sub-relations $\mathcal{R}_1$ and $\mathcal{R}_2$.

This algorithm uses two additional parameters:

- The `cost` function can be customized by the user and is a parameter of the recursive algorithm. Previous algorithms, such as exact or heuristic solvers in [35, 166], aim at minimizing the number of cubes of the solutions.
- The algorithm can trade-off between the quality of the solution and the runtime spent in the search. As in any branch-and-bound algorithm, the search can be stopped as soon as some resources (e.g. the CPU time) have been exhausted.

Note that, incompatibilities may occur in this algorithm only at an input vertex x for which the output set cannot be precisely captured with don't cares. Consider the example in Fig. 3.1. BREL can potentially find an incompatibility for the input vertex 10, since its output set $\{00,11\}$ cannot be captured with don't cares, but it would not consider the input vertex 11 as a potential incompatible vertex, since its output set $\{10,11\}$ can be described as $1-$.

Example 3.6.2 *Figure 3.7 depicts how the relation $\mathcal{R}(a,b,c,x,y)$ is solved with BREL. In the first recursion, the same solution b is found for both outputs x and y. The minimization after the projection steps is represented using Karnaugh maps. After the individual minimization, a multiple-output function is composed from the individual solutions. Two conflicts are found between this function and the original BR on input vertices* 010 *and* 101. *In order to reduce the conflicts, the vertex* 010 *and the output y are selected to split the relation. The solver will find a compatible solution for each of the new sub-relations in the second recursive iteration:*

$$f(a,b,c,x,y) = \begin{cases} (x \Leftrightarrow ac)(y \Leftrightarrow b) & \textit{for y=1,} \\ (x \Leftrightarrow b)(y \Leftrightarrow a+c) & \textit{fot y=0.} \end{cases}$$

```
BREL (R, cost, BestF)
Input: A well-defined relation R(X,Y) and the best
       function to estimate the cost of the solution (cost)
       found compatible function (BestF)
Output: BestF returns the minimum-cost function
        compatible with R
   // Check for R to be a function
1: if R is a function then
2:    if cost(R) < cost(BestF) then BestF := R;
3:    return;

   // R is not a function
   // Compute MISF_R and minimize
4: MISF_R := Compute_MISF_R(R);
5: F := Minimize(MISF_R);

   // The solution cannot be better
6: if cost(F) ≥ cost(BestF) then return;

   // The solution is better, but it may not be compatible
7: I := Incomp(F, R);
8: if I = 0 then BestF := F; return;

   // Incompatible solution: split and call recursively
9: (x, y_i) := Pick (vertex, output signal) pair from I
               such that (R ↓ y_i)(x) = {0,1};
10:(R1, R2) := Split(R, x, y_i);
11:BREL(R1,cost,BestF);
12:BREL(R2,cost,BestF);

   return;
end;
```

Figure 3.6: A recursive algorithm for solving Boolean relations.

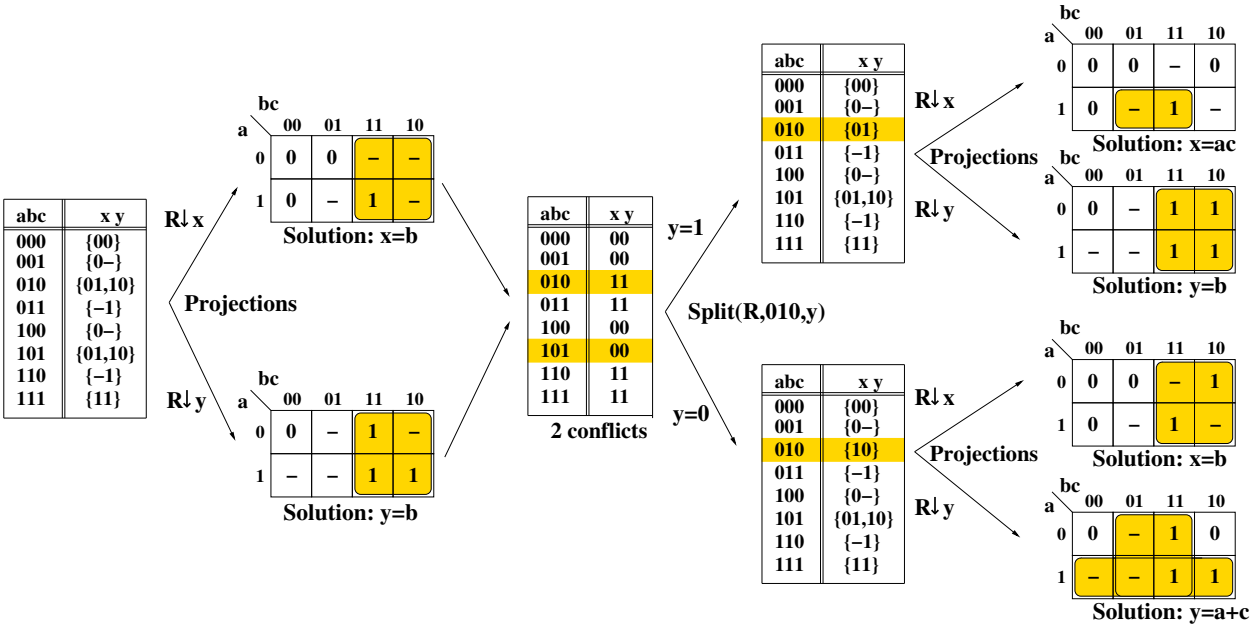


Figure 3.7: Example of solving a Boolean Relation.

The tabular representation of the solutions is described next:

for y = 1	
abc	*xy*
000	00
001	00
010	01
011	01
100	00
101	10
110	01
111	11

for y = 0	
abc	*xy*
000	00
001	01
010	10
011	11
100	01
101	01
110	11
111	11

For this example, the conflict for the input vertex 101 *is also solved in the second recursive call. However, in general, the number of required recursive calls to solve different conflicting vertices may differ.* □

3.7 Further implementation details

The general branch-and-bound approach presented in Fig. 3.6 can be implemented in different ways. There are multiple degrees of freedom in the implementation: selecting a data structure for representing relations, strategy to explore the branch-and-bound tree, particular cost functions, algorithms for ISF minimization, etc.

We will next present several implementation details of our solver, BREL, that lead to an efficient trade-off between the quality of the solutions and the computational complexity of the search. Many of the implementation decisions have been taken after experimenting with different strategies and choosing the most effective ones.

3.7.1 Representation of relations

Binary Decision Diagrams (BDDs) [30] are used to represent and manipulate the characteristic functions of the relations. All the transformations, evaluation of cost functions and the ISF minimization are implemented using BDD operations.

Since all the relations generated by the solver come from a single original relation, there is a lot of sharing in the BDD data representation. The solver invokes many similar low-level BDD operations that are cached and calculated only once. This has an important impact on the performance of the solver.

3.7.2 Exploration of solutions

The branch-and-bound tree of solutions is explored using a partial breadth-first-search (BFS). This requires a slight modification of the algorithm in Fig. 3.6. All the relations generated by splitting are stored in a bounded FIFO implemented as a list.

The size of the FIFO of solutions is a parameter of the solver. Due to the bound on the number of unresolved intermediate relations not all of the potentially generated relations are resolved into compatible functions and reach the functional leaves of the semi-lattice of relations. Therefore, the `QuickSolver` (Fig. 3.4) is used as the first step to guarantee that at least one compatible function is found during the exploration.

The BFS enables a larger diversity in the exploration of solutions and prevents the solver of spending all the resources in only one corner of the tree searching for a local optimum.

3.7.3 Cost function

A cost function is another parameter of the solver. For efficiency reasons, BDD-based cost functions are desirable since they are easy to compute. Even though the size of BDDs is not always the best

estimation of complexity for a Boolean function, typically there is a correlation between both. In the experiments we have used different cost functions depending on the minimization goal: sum of BDD sizes when targeting at area minimization and sum of the squares of BDD sizes when targeting at delay. The latter cost function biases the exploration towards solutions in which the complexity of the functions is balanced, and hence the delay is more evenly distributed along all paths. The former tends to minimize the overall size regardless of the relative complexity of the sub-functions.

The experimental results, demonstrating application of the BREL to logic decomposition (presented in Section 3.10.2), show that these cost functions lead to significant area and delay optimization.

3.7.4 Split strategy

When conflicts appear after the minimization of the $\mathsf{MISF}_{\mathcal{R}}$, an input vertex x and an output y_i must be selected for splitting (line 9 in Fig. 3.6). Intuitively, the solver selects the largest input cube within the characteristic function of all input conflicting vertices.

More precisely, given the characteristic function of the conflicts, Incomp, the outputs are existentially abstracted ($C = \exists_Y \mathsf{Incomp}$). Next, the shortest path in the BDD representing C is extracted. The shortest path represents the largest set of adjacent conflicting input vertices. Constraining the value of the relation in one of the vertices of this set forces many other adjacent vertices to acquire the same output value during the minimization.

The input vertex x is obtained from the incompatible input cube by assigning the value 1 to the variables with don't care value ($\{-\}$). The selection of the output y_i must fulfill Theorem 3.5.2. Therefore, an output such that $(R \downarrow y_i)(x) = \{0,1\}$ is selected following the variable order in the BDD manager.

3.7.5 Minimization of ISFs

We will next explain the details of the Minimize operation in the solver. Each ISF of the MISF_R is individually minimized with BDD-based optimization methods. A BDD-based approach contributes to the speed up the solver. ISFs are defined by a pair of functions that represent the interval of flexibility $[\mathsf{Min}, \mathsf{Max}]$ (or $[\mathsf{On}, \mathsf{On} \cup \mathsf{Dc}]$). There are different methods to minimize the ISF implementation using the flexibility within the specification interval. Versions of generalized cofactors, such as *constrain* and *restrict* [58, 59], have been often used to reduce the size of BDDs. A BDD operation to find irredundant SOPs is also possible by using Minato-Morreale's algorithm [117], even though the obtained solutions can be far from the optimum.

Another way to reduce the complexity is to reduce the support by eliminating non-essential variables. A variable z is called not essential if the interval $[\exists_z \mathsf{Min}, \forall_z \mathsf{Max}]$ is not empty (cf. [37], pp. 107–112).

	ISOP		Constrain		LICompact	
	LIT	CPU	LIT	CPU	LIT	CPU
Eliminate non-essential	1.00	1.00	1.09	1.03	1.02	1.02
Keep non-essential	1.16	1.59	1.16	1.57	1.17	1.57

Table 3.1: Normalized comparison between several ISF minimization based on BDDs.

Our solver first reduces the support of the ISF by greedily eliminating non-essential variables from the top to the bottom of the BDD representation. After that, an irredundant SOP is calculated using Minato-Morreale's algorithm. We found this combined approach more efficient, in performance and quality of the solutions, than other tested techniques. Three techniques have been tested: minimization of irredundant SOPs (ISOP) based on Minato-Morreale's approach [117], a constrain-restrict minimization (Constrain) [58, 59], and a BDD safe minimization (LICompact) [79]. Table 3.1 shows the normalized comparison of these ISF minimization approaches with regard to the selected ISOP minimization with the elimination of non-essential variables. The table reports the increment of the number of literals in SOP representation of the final solution (LIT) and the required CPU time (CPU) for the benchmarks used in Table 3.2. The elimination of the non-essential variables contributes to significantly reduce the runtime and improves the quality of the solutions of the ISF minimization. The table also demonstrates that the irredundant SOPs minimization on average provides slightly better solutions than other methods as measured by the literal count in the SOP form.

3.7.6 Symmetries in Boolean Relations

Symmetries in Boolean functions are often used for speeding-up equivalence checking between two functions by analyzing when functions (or their sub-functions) are structurally equivalent after permutation of some variables. Symmetries can be also exploited in the characteristic functions of Boolean relations.

Figure 3.8(b) depicts the first and second recursion of BREL solving a 2-input and 2-output BR shown in Fig. 3.8(a). Initially, BREL finds the solution $(x \Leftrightarrow 1)(y \Leftrightarrow 1)$ with three incompatible vertices $\{\overline{a}\overline{b}, \overline{a}b, a\overline{b}\}$. Let us assume that the input vertex $\overline{a}\overline{b}$ and the output x is selected to perform the split. In the second recursion, the solutions are

$$f(a,b,x,y) = \begin{cases} (x \Leftrightarrow a)(y \Leftrightarrow 1) & \text{for x=1,} \\ (x \Leftrightarrow 1)(y \Leftrightarrow a) & \text{fot x=0.} \end{cases}$$

Note that, these solutions are fully symmetric with respect to permutation of variables x and y. The two symmetric relations lead to solutions with equal cost as calculated by a BDD-based cost

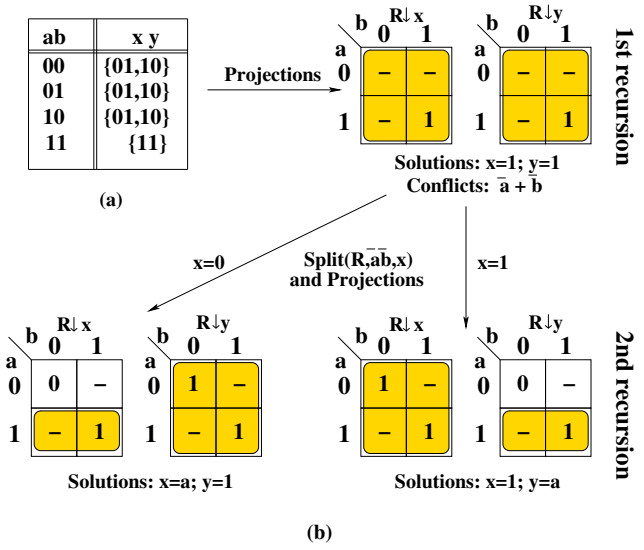


Figure 3.8: Example of a symmetry in BRs.

function. Therefore, the exploration for a relation can be stopped if a symmetric relation has already been processed by the solver.

BREL has a cache of processed relations. Symmetries can be checked for every new relation from the *Split* process to identify if a symmetric relation is stored in the cache. If it is found the exploration for this branch is stopped.

There are efficient methods for identifying the first-order [174, 175] and the second-order symmetries [49] in Boolean functions that can be applied to Boolean relations as well. However, the analysis of BR symmetries has a high complexity especially for large BRs. Therefore, the application of symmetry detection has to be limited in order to reduce the runtime impact. For BREL we have made a few implementation decisions regarding the use of symmetries:

- Symmetries are only supported for output variables. This implementation decision was based on experiments in the application domains (like logic decomposition) where BREL is currently used. In this domain, output variable symmetries appear very often: e.g. if the large stage of logic is a symmetric gate (such as AND, OR, NAND, NOR, etc.) permutation of two functions that feed this gate leads to a symmetric implementation of the same function.

- The solver supports all types of the first-order symmetries and the nonskew nonequivalence second-order symmetries [49]

- Symmetries are only explored during the initial recursions on the exploration tree for largest sub-relations that are close to the original relation. Here, significant cuts on the exploration

branches can be expected. Later, the symmetry check is turned off to avoid spending significant CPU time in the search for symmetries in the smaller relations.

An experiment has been done to identify the impact of the symmetry detection on the quality of results and runtime in logic decomposition problem (the experiments on logic decomposition will be described in details in Section 3.10.2). When the symmetry check is turned on the results are on average improved by 1.6% in delay and on 1.2% in area after technology mapping at the cost of runtime increase by 10.6%. The literal count in the SOP representation also decreases (on average by 1.30%). However, the improvement on some particular examples is significant. In the small netlist s208, there is an improvement of 16% on delay and 11% on area with similar runtime. In a larger netlist, s641, the delay is improved by 13% and the area is reduced by 17%. Nevertheless, the cost of the symmetry check increases the runtime by 15%.

The reason in improving the quality of results is as follows. For large BRs, BREL runs in a non-exact mode, since only a subset of solutions can be explored within the limited time and memory resources. Without symmetry detection the solver can explore more sub-relations within the given runtime. However, many of these sub-relations are symmetric and hence useless. With symmetry detection, the solver spends slightly less time in solving relations (due to the penalty of symmetry detection), but actually solves more relations from different equivalence classes and hence explores more different solutions.

3.8 Solving Boolean equations

Many problems in Boolean algebra with a finite number of elements can be reduced to solving a system of Boolean equations (cf. [37], pp. 153-154). In this section, we illustrate how to solve a system of Boolean equations by solving the corresponding BR. We use characteristic functions to represent BRs.

Definition 3.8.1 Boolean equation. *A Boolean equation is defined as*

$$P(\mathtt{X},\mathtt{Y}) \quad \odot \quad Q(\mathtt{X},\mathtt{Y})$$

where P and Q are multiple-output Boolean functions of independent variables X and dependent variables Y and $\odot$ *is the equivalence* $(=)$ *or the inclusion-relation operator* $(\leq)$*.*

□

Definition 3.8.2 *A* particular solution *(or, simply, a* solution*) of a Boolean equation is a multi-output function* $Y(X)$ *such that* $P(\mathtt{X},\mathtt{Y}(\mathtt{X})) \odot Q(\mathtt{X},\mathtt{Y}(\mathtt{X}))$ *is a tautology. A Boolean equation is* consistent *if it has at least one solution. A* general solution *of a Boolean equation is a representation of the set of*

all its particular solutions [37]. A parametric general solution can be formed from any particular solution using Löwenheim formula [37]. □

Here, we will focus of finding particular solutions for the system of Boolean equations.

Definition 3.8.3 Boolean system. *A Boolean system is a set of Boolean equations*

$$\begin{array}{rcl} P_1(\mathtt{X},\mathtt{Y}) & \odot & Q_1(\mathtt{X},\mathtt{Y}) \\ & \vdots & \\ P_k(\mathtt{X},\mathtt{Y}) & \odot & Q_k(\mathtt{Y},\mathtt{Y}) \end{array}$$

□

Property 3.8.1 *A Boolean equation $P(\mathtt{X},\mathtt{Y}) \odot Q(\mathtt{X},\mathtt{Y})$ can be transformed to the form $T(\mathtt{X},\mathtt{Y}) = 1$ using the following equivalence properties [37]:*

$$\begin{array}{rcl} P = Q & \Leftrightarrow & \overline{P \oplus Q} = 1 \\ P \leq Q & \Leftrightarrow & \overline{P} + Q = 1 \end{array}$$

□

Example 3.8.1 *The following system of Boolean equations with a set of independent variables $\{a,b\}$ and a set of dependent variables $\{x,y,z\}$*

$$\begin{array}{rcl} x + b\,\overline{y}\,\overline{z} + \overline{b}z & = & a \\ xy + xz + yz & = & 0 \end{array}$$

can be transformed using the previous equivalence properties to

$$\begin{array}{rcl} a\,\overline{b}\,z + a\,x + a\,b\,\overline{y}\,\overline{z} + \overline{a}\,\overline{b}\,\overline{x}\,\overline{z} + \overline{a}\,b\,\overline{x}\,y + \overline{a}\,b\,\overline{x}\,z & = & 1 \\ \overline{x}\,\overline{y} + \overline{x}\,\overline{z} + \overline{y}\,\overline{z} & = & 1 \end{array}$$

Note that, the characteristic function of a Boolean equation matches with the definition of the characteristic function of a Boolean relation. Fig. 3.9(a) depicts the two Boolean equations of the system as sets of vertices of Boolean relations with $\{a,b\}$ as input variables and $\{x,y,z\}$ as output variables of the BRs. □

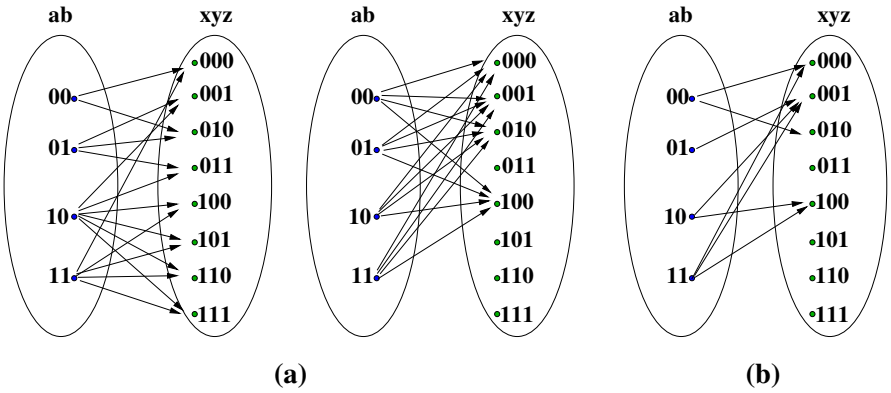


Figure 3.9: Representing a Boolean system of equations as BRs.

Theorem 3.8.1 Reduction. *A Boolean system*

$$\begin{aligned} T_1(\mathtt{X},\mathtt{Y}) &= 1 \\ &\vdots \\ T_k(\mathtt{X},\mathtt{Y}) &= 1 \end{aligned}$$

can be reduced to a single equation $\mathbb{E}(\mathtt{X},\mathtt{Y}) = 1$ *where* $\mathbb{E}$ *is the characteristic function*

$$\mathbb{E}(\mathtt{X},\mathtt{Y}) = \bigwedge_{i=1}^{k} T_i(\mathtt{X},\mathtt{Y})$$

□

The characteristic function $\mathbb{E}(\mathtt{X},\mathtt{Y})$ only contains the feasible solutions of the system. Note that, $\mathbb{E}(\mathtt{X},\mathtt{Y})$ can be also represented as a Boolean relation.

Example 3.8.2 *The Boolean system of the Example 3.8.1 is reduced to the following single equation:*

$$a\,b\,\overline{y}\,\overline{z} + a\,\overline{b}\,\overline{x}\,\overline{y}\,z + a\,x\,\overline{y}\,\overline{z} + \overline{a}b\overline{x}y\overline{z} + \overline{a}\,b\,\overline{x}\,\overline{y}\,z + \overline{a}\,\overline{b}\,\overline{x}\,\overline{z} = 1$$

Figure 3.9(b) shows the single BR associated with the characteristic function used in the above equation. It is easy to see from the figure that this BR only covers the solutions that are feasible in both BRs represented in Fig. 3.9(a). □

Property 3.8.2 Consistency of a Boolean system. *A Boolean system is consistent if for all* $x \in X$*, there exists a* $y \in Y$ *such that* $(x,y) \in \mathbb{E}(\mathtt{X},\mathtt{Y})$. □

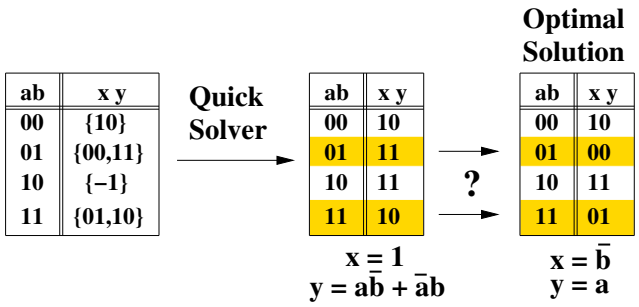


Figure 3.10: Example of the expand-reduce-irredundant approach.

Note that, a consistent Boolean system is equivalent to a well-defined BR. If the system is inconsistent, then it has no solutions and therefore there is no corresponding well-formed BR.

Example 3.8.3 *The set of functions $x = ab$, $y = \overline{a}\,\overline{b}$ and $z = \overline{a}b + a\overline{b}$ forms a particular solution of the Boolean system from Example 3.8.1. This can be checked by substitution into the equation from Example 3.8.2 and checking that the equations simplifies to a tautology $1 = 1$. Hence this system is consistent.* □

As shown in [37][4] Boolean equation $\mathbb{E}(\mathtt{X}) = 1$ is consistent if and only if existential quantification of all variables (also called smoothing of all variables) gives constant 1:

$$\exists_X \mathbb{E}(\mathtt{X}) = 1$$

Given a Boolean system, we convert it to a single Boolean equation, then check its consistency using quantification. If the system is inconsistent there are no solutions, otherwise we obtain an optimized particular solution by solving the corresponding BR using BREL solver.

3.9 Efficiency of the method

3.9.1 Comparison with the expand-reduce-irredundant paradigm

In this section, we illustrate the limitations of the expand-reduce-irredundant paradigm used in Herb [72] and gyocro [166] BR solvers. Let us consider the Boolean relation depicted in Fig. 3.10. The best compatible function with the smallest number of product terms is $f(a,b,x,y) = (x \Leftrightarrow \overline{b})(y \Leftrightarrow a)$. Although, this relation covers a small set of only eight compatible

[4]The consistency check presented in this section is a modification of Theorem 6.1.1 in [37] that gives conditions for a complemented form of a Boolean equation.

			gyocro					BREL				
	PI	PO	CB	LIT	ALG	AREA	CPU	CB	LIT	ALG	AREA	CPU
int1	4	3	5	8	8	9280	0.03	7	12	9	8352	0.01
int5	4	3	7	14	11	11136	0.02	7	14	11	11136	0.00
int10	6	4	25	88	32	44544	0.08	29	102	34	41296	0.02
c17b	5	3	7	12	12	10208	0.03	7	12	12	10208	0.00
c17i	5	3	15	37	34	34336	0.04	13	32	30	32016	0.01
she1	5	3	6	20	16	16240	0.04	9	26	15	17632	0.00
she2	5	5	10	33	31	30624	0.09	12	30	24	26448	0.01
she3	6	4	9	26	23	24592	0.08	9	27	21	21344	0.01
she4	5	6	20	91	56	62176	0.14	27	120	40	46864	0.03
gr	14	11	54	455	318	346608	3.43	86	590	313	322016	6.79
b9	16	5	270	2833	321	382336	4.68	137	1174	256	306240	0.19
int15	22	14	131	1083	506	525248	21.94	166	1062	459	472352	19.14
vtx	22	6	424	4460	117	151728	30.10	244	1809	101	94656	0.58
Normalized sum			**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.77**	**0.55**	**0.89**	**0.86**	**0.44**

Table 3.2: Comparison with gyocro [166].

functions, the expand-reduce-irredundant local search technique is not able to explore the whole search space and find the best one.

The initial solution $f(a,b,x,y) = (x \Leftrightarrow 1)(y \Leftrightarrow a\overline{b} + \overline{a}b)$ is obtained using the procedure QuickSolver. This solution is a local minimum and therefore gyocro gets trapped and cannot explore the complete set of compatible functions. From the initial solution, the *reduce* procedure can not simplify the function any further. The *expand* procedure can only be applied to the input vertex $\{10\}$ to reach another MISF compatible with the original relation with the output vertices $\{-1\} = \{01, 11\}$. This expansion results in two possible solutions: the initial compatible function and the function $f(a,b,x,y) = (x \Leftrightarrow \overline{a} + b)(y \Leftrightarrow a\overline{b} + \overline{a}b)$ that has higher cost. The expansion of the other input vertices $\{00, 01, 11\}$ produces MISFs incompatible with the original relation. At this point the exploration is stopped. Therefore, there is no feasible cube expansion that leads to the optimal solution. The reason for this limitation is that this local search exploration is not capable of exploring the range of output vertices since they cannot be covered with a set of cubes.

3.9.2 Experimental results

Table 3.2 presents comparative results with gyocro. In [166], a similar analysis is done between gyocro and the exact minimizer from [35] and Herb [72]. The cost function used by BREL in these runs is the sum of BDD sizes for each output, aiming at area minimization. The tree of solutions has been limited to the partial exploration of 10 Boolean relations. During this exploration, the QuickSolver procedure is applied on each smaller Boolean relation to obtain a solution, as it was

explained in Section 3.7.2. Exploring more solutions did not significantly contribute to improving the results. The table summarizes the number of input (PI) and output (PO) variables for all BR examples and reports the number of cubes (CB) and literals (LIT) in the sum-of-products representation obtained by each solver.

When comparing cubes and literals gyocro obtains better results in several examples, since its objective cost function targets to reduce these parameters.

For more practical comparisons, we also performed two more experiments to check the quality of solutions after multi-level logic synthesis and technology mapping in SIS [143]. First, the multi-output Boolean function in sum-of-products representation obtained from the BR solvers is transformed to a multi-level Boolean network in SIS by applying the *algebraic* script. This script reduces the size of the network by sharing common sub-expressions. BREL obtains 11% of improvement on average in literal count after *algebraic* (ALG column). gyocro obtains better results only in two cases: int1 and int10.

The improvement is also observed after technology mapping. For this comparison we used the technology mapper *map* [133, 162] and the library *lib2* of SIS for mapping a multi-level logic network into the library of logic gates. The area results obtained by BREL are better than gyocro in all cases except for she1 (see column labelled with AREA). BREL obtains on average a 14% area reduction. Although gyocro aims at minimizing the number of cubes, while BREL minimizes the number of BDD sizes, there are cases in which the solution obtained by BREL is significantly better (b9 and vtx) in the number of cubes as well. We attribute this phenomenon to the fact that gyocro could be trapped in a local minimum (e.g. after generating the initial solution), from which it cannot easily escape by simply reducing and expanding cubes. On the other hand, the BFS strategy used by BREL allows to perform hill climbing and explore a larger set of solutions.

Finally, the runtime (columns CPU) is usually better for BREL, with a tangible speed-up for two examples (b9 and vtx). The runtime of gyocro is significantly better than BREL's only for gr.

3.10 Application of Boolean relations

In this section we present an application to the problem of a multi-way logic decomposition and report experimental results.

3.10.1 Logic decomposition

The multi-way logic decomposition problem can be formulated as follows

Definition 3.10.1 *Let us assume a function $F(X)$ with the set of variables $X = \{x_1, x_2, \cdots, x_m\}$ and a gate $G(Y)$ with the set $Y = \{y_1, y_2, \cdots, y_n\}$. The decomposition of the function $F(X)$ with the*

gate $G(Y)$ is $F(X) = G(F_1(X), F_2(X), \ldots, F_n(X))$. The Boolean relation that subsumes all possible decompositions of the function $F(X)$ with the gate $G(Y)$ is defined as follows:

$$R(X,Y) = F(X) \Leftrightarrow G(Y)$$

□

We next present an example to clarify the decomposition problem. Consider the following Boolean function

$$f(x_1,x_2,x_3) = x_1(\overline{x_2} + \overline{x_3}) + \overline{x_1}x_2x_3$$

The goal is to decompose this function using a multiplexor and, therefore, to absorb part of the original function f within the multiplexor with the function $Q(A,B,C) = A \cdot C + B \cdot \overline{C}$. A BR will enclose all possible decompositions than can be performed using the multiplexor. The next tabular representations show the original function $f(x_1,x_2,x_3)$ and the corresponding BR for the multiplexor.

$x_1x_2x_3$	f
000	0
001	0
010	0
011	1
100	1
101	1
110	1
111	0

$x_1x_2x_3$	ABC
000	$\{-00, 0-1\}$
001	$\{-00, 0-1\}$
010	$\{-00, 0-1\}$
011	$\{1-1, -10\}$
100	$\{1-1, -10\}$
101	$\{1-1, -10\}$
110	$\{1-1, -10\}$
111	$\{-00, 0-1\}$

The construction of the BR can be done intuitively. The relation is built finding all the possible values of the inputs of the multiplexor that yield to the desired output value in the function. For instance, the multiplexor produces the output value $Q(A,B,C) = 0$ whether the value of (A,B,C) is -00 or $0-1$. Therefore, the output of the relation for the minterms where $f(x_1,x_2,x_3) = 0$ is $\{-00, 0-1\}$. The same reasoning can be followed to find the output set of the remaining minterms of the relation. Note that the solution of one output of the BR is conditioned to the values of the other outputs. For instance, the output A can only achieve the value 1 for the minterm $x_1x_2x_3$ if BC obtains the value 00.

Many decompositions can be found using the BR. Figure 3.11 depicts some of these solutions. A solver of BRs will explore the set of solutions and will return one of them based on the minimization objective.

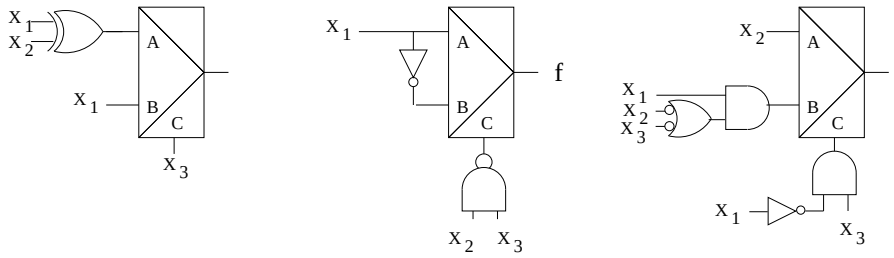


Figure 3.11: Some decompositions of $f(x_1, x_2, x_3)$ using a multiplexor.

3.10.2 Experimental results

Table 3.3 reports the results of an experiment designed to illustrate the applicability of BREL and the customization of its cost function. We consider the existence of a flip-flop with an embedded mux in the library, and the next-state equation $Q^+ = A \cdot C + B \cdot \overline{C}$. This three-input flip-flop (typically available in the industrial gate libraries) enables the implementation of the next-state function $F(X)$ as the composition of three functions: $A(X)$, $B(X)$ and $C(X)$. The Boolean relation specifying this flexibility is $F(X) \Leftrightarrow (A \cdot C + B \cdot \overline{C})$ where A, B and C are the output variables. The table summarizes the number of primary inputs (PI), primary outputs (PO) and number of flip-flops of the network (FF). The last row of the table summarizes the results and shows the global improvement obtained by the mux-based decomposition with Boolean relations. The table reports the results for two different cost functions. First, the cost function has been defined as the sum of the squares of the BDD sizes for the three functions. The squaring favors a tendency to balance the complexity of the function and, therefore, reduce the delay of the circuit. The second part of the table, the sum of BDD sizes has been used, aiming at minimizing the total area. In this table, BREL is limited to explore up to 200 BRs for each next-state function.

The table reports the area and delay of the combinational part of the circuit for each cost function. Only the area and the delay of the combinational logic are considered. We make an optimistic assumption considering that the mux is embedded in the flip-flop without any extra area and delay overhead. In the delay optimization, the results have been obtained by collapsing the next-state functions, running the algebraic script, speed_up[5] and technology mapping in SIS. For the mux-latch, the decomposition is done before running the algebraic script. In general, the results manifest several features of the approach: (1) the delay is usually reduced (sometimes significantly: e.g. s382, s641, s832), (2) in many cases area is also reduced due to the power of Boolean decomposition (e.g. s420, s526, s641), (3) in some cases the delay is reduced at the expense of increasing area due to the balancing tendency of the cost function (e.g. s953, sbc) and (4) the CPU time is affordable. In two cases (s349 and s1196), both area and delay became worse with the mux-based decomposition.

[5]This command in SIS produces more balanced solutions and contributes to improve the global delay of the network.

				Delay Minimization					Area Minimization				
				ORIGINAL		Decomp. mux-latch			ORIGINAL		Decomp. mux-latch		
	PI	PO	FF	Area	Delay	Area	Delay	CPU	Area	Delay	Area	Delay	CPU
daio	2	3	4	18096	3.47	11136	3.12	0.1	18560	4.39	12064	3.68	0.1
s27	4	1	3	15312	4.47	18096	4.01	0.2	13456	4.74	15312	3.81	0.2
s208	10	1	8	88160	7.58	77952	4.79	0.8	80272	9.91	35264	10.37	0.7
s298	3	6	14	140128	6.72	91408	4.18	1.4	125744	8.37	62176	4.72	1.4
s349	9	11	15	233856	9.45	333616	9.54	5.2	157760	10.81	199984	13.33	5.0
s382	3	6	21	223648	9.63	192560	6.68	4.0	154512	8.87	127136	6.94	3.9
s386	7	7	6	182352	7.84	155904	5.87	2.4	144304	7.96	115072	6.97	2.3
s420	18	1	16	207408	9.67	108112	6.74	41.3	160544	18.08	61248	19.77	41.3
s444	3	6	21	213904	8.54	182816	6.07	4.0	181424	10.23	75632	8.57	3.9
s510	19	7	6	300208	8.42	316448	7.91	297.0	287216	9.40	245456	9.17	303.6
s526	3	6	21	225504	8.75	166112	5.60	3.7	188848	9.02	97904	8.90	3.5
s641	35	23	19	522464	12.16	376304	8.18	9.5	191632	22.48	214480	10.69	9.2
s832	18	19	5	338256	10.28	327584	7.81	27.3	337328	11.15	270976	8.85	27.4
s953	17	24	29	503904	9.82	528032	8.31	32.3	423632	12.89	380944	9.66	27.3
s1196	14	14	18	1062560	11.91	1220784	12.62	5.7	558192	19.20	642384	13.77	5.6
s1488	8	19	6	741472	9.98	802720	9.73	7.6	723840	12.25	660272	11.85	7.4
s1494	8	19	6	729872	10.14	759104	10.00	7.7	688112	12.46	611088	13.03	7.4
sbc	40	56	28	920112	8.88	979504	8.73	21.0	775344	14.50	756320	11.18	21.2
Normalized sum				**1.00**	**1.00**	**0.98**	**0.82**		**1.00**	**1.00**	**0.87**	**0.85**	

Table 3.3: Logic decomposition for mux-latches.

For area optimization, the process of minimization is the same that the previous one without the speed_up command. The behavior of the results are similar: (1) the area is also reduced considerably (e.g. s298, s420, s444), (2) and in many cases the delay as well, (3) only in few cases the delay increases (e.g. s208, s420, s1494) (4) sometimes related to circuits where the area is also worse (e.g s349), and (5) the CPU time is similar to the delay decomposition. There are four cases (s27, s349, s641 and 1196) where the results were worse. Some of these circuits (s349 and 1196) are worse in both area and delay minimization. The heuristic methods applied in BREL and the limitation on the number of explored BRs sometimes lose some of the good solutions.

3.11 Conclusions

We have described a new algorithm for solving Boolean relations and Boolean equations. Experimental results demonstrate that this approach is capable of finding better solutions in shorter runtimes than the previously known techniques. The reason for this advantage is that our exploration technique is more immune to be trapped in local minima and better explores the solution

space. Depending on the complexity of the original BR our solver can work in the exact or in the approximate mode.

In this chapter we also demonstrated a successful application of our solver to the problem of decomposing a Boolean function. In the Chapter 5, this experiment is extended to a recursive n-way decomposition using the solver of Boolean relations.

Chapter 4

Dominator-based Partitioning for Logic Synthesis

4.1 Introduction

As we introduced in Section 2.3, graph partitioning is a well-known strategy to decrease the complexity of the synthesis problems. Mostly, the runtime is improved considerably by applying partitioning. However, there is also a degradation of the quality of the results. Many of the graph partitioning methods proposed in the synthesis domain are reduced to the *min-cut* problem [46, 47, 54, 89, 128]. The min-cut partitioning techniques are suitable in problems like placement [3] or FPGA partitioning [31, 156], where the minimization criteria is the number of wires among different parts of the circuit. Moreover, the low interaction among the different clusters contributes to reduce the gap between the cost of the solutions of the partitioning and the flat method[1]. However, min-cut is not always appropriate for delay and area logic optimization, since it only takes into account the structure the graph. However, the *Boolean* information is crucial on logic optimization to obtain good minimizations.

In this chapter, a new partitioning method for delay logic optimization is presented based on the concept of *vertex dominator*. A partition performed with vertex dominators generates clusters with acyclic connectivity that enables the correct propagation of the delay information. Tangible improvements can be obtained in delay with this strategy, thus better exploring the area-delay trade-off of Boolean networks. This chapter is based on the results presented in [19]

The rest of the chapter is organized as follows. A detailed description of the previous work on partition-based delay optimization is done in Section 4.2. An overview of the approach is presented in Section 4.3. Section 4.4 presents the required background on vertex dominators. The new parti-

[1] A *flat method* refers to an approach without partitioning.

tion method is described in Section 4.5. Finally, the overall delay minimization strategy is explained in Section 4.6, and experimental results are reported in Section 4.7.

4.2 Previous work

Different approaches have been developed for timing optimization [9, 45, 56, 147]. However, most of them cannot be used in large circuits because of their complexity.

On logic optimization, some of the work performed on partitioning has been proposed for FPGAs [31, 156, 172]. Here, the basic objective is to divide the circuit in a fixed number of clusters that is determined by the number of LUTs that are used to implement the FPGA. Note that, the min-cut is mostly applied as an objective function since the interconnections between clusters are constrained by the number of pins of the LUTs. A timing-driven method in FPGAs refers to a technique to reduce the cuts on the critical path to merge into the same clusters the maximum number of critical nodes. This contributes to decrease the number of components that the critical path has to pass through.

Our graph partitioning objective is different. The objective is to apply a good delay minimization algorithm on each cluster. The current state-of-art delay optimization processes are mostly restricted by the size of the netlist. Therefore, instead of performing delay minimization on the whole network, the optimization is performed on smaller netlists extracted by a partitioning approach. Here, there is a basic constraint to take into account: the size of the clusters. The size must be selected accurately depending on the complexity of the optimization technique. Note that, there is a trade-off between the size of the clusters and the result of the minimization, since few optimizations can be applied on small netlists compared to larger ones. Let us review some of the approaches that have been published to perform delay-driven partitions.

In [101], the Lawler's algorithm is presented. This work has been the foundation of timing-driven partition techniques. The authors defined an efficient labeling approach for Boolean networks in tree representation. Two main definitions were introduced on a directed acyclic graph (DAG): *rooted* and *non-rooted* trees. The main difference is graphically illustrated in Fig. 4.1.(a). Assuming that the DAG goes from inputs to outputs, a rooted tree has only a node with in-degree zero or a node with out-degree zero. A non-rooted tree is a tree where there are not for all pairs of primary inputs and primary outputs more than one path that connects them. By definition, a non-rooted tree can be decomposed into several rooted trees. Note that, there are multiple decompositions of a non-rooted tree into rooted trees.

The main algorithm is described in Fig. 4.2. Initially, the non-rooted tree is partitioned into rooted trees. Therefore, a particular decomposition is selected based on a labeling algorithm. Basically, all paths from an arbitrary primary input to the outputs are firstly traversed. The fanins of a traversed node such that they are not included in the explored path are defined as root nodes of new

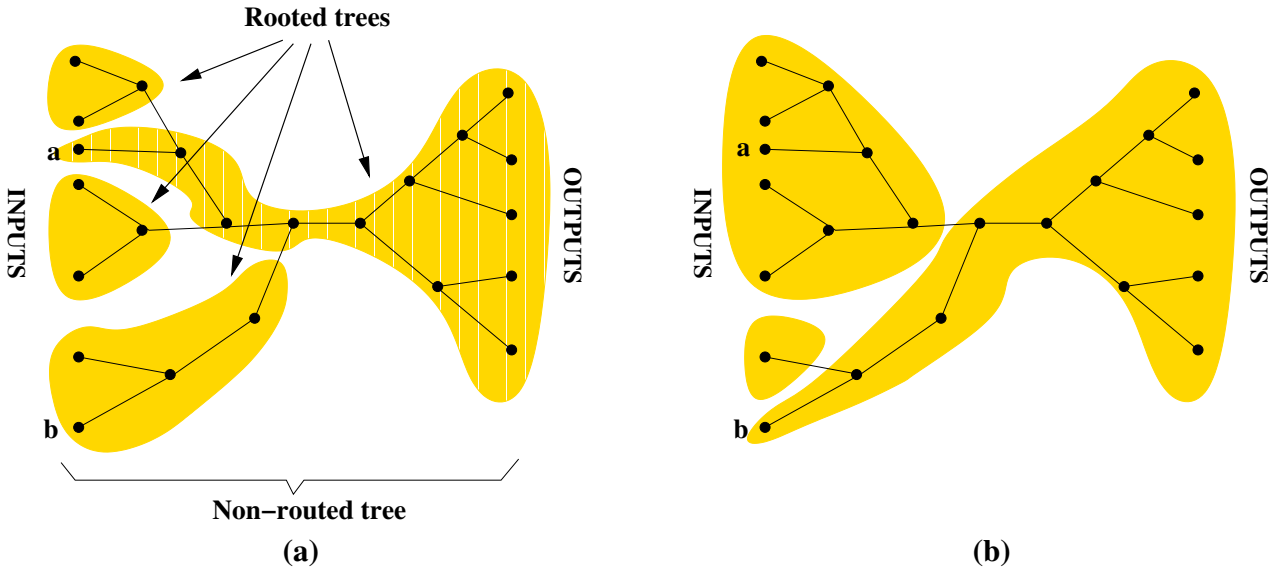


Figure 4.1: Rooted and non-rooted trees.

rooted trees. Figure 4.1.(a) shows the decomposition starting for the primary input *a*. Note that, the decomposition and the number of rooted trees is different depending on the starting node. If input *b* is selected the number of rooted trees is reduced to three (See Fig. 4.1.(b)).

After the division, a labeling process is done for each rooted tree. The labeling is performed in DFS order from the root node based on the depth of each node (See Fig. 4.3.(a)). Finally, a relabeling on the non-rooted graph is performed to create a better *delay*-oriented partition and taking into account the maximum capacity constraint of the clusters. The process begins from the primary outputs. Basically, the relabeling produces a partition onto the rooted trees. Many of the new clusters are included in the previous rooted trees. The minor changes only appear in the intersection edges between rooted trees. Note that, an estimation of the total length of the path can be computed in the intersection edges based on the depth of the adjacent nodes. The relabeling process checks the length of the paths (using the previous depths) and it relabels the fanins depending on the *critical* path. The most critical fanins will be merged in the same cluster of the current node even if they belong to different rooted trees (See Fig. 4.3.(b)). The drawback of the Lawler's labeling technique is that it is a limited delay-oriented approach. It only performs local modifications on the intersection edges depending on the criticality of the rooted trees.

The authors could not find a good heuristic for conventional graphs where there are multiple paths from inputs to outputs. Therefore, two methods are defined depending on whether duplication is allowed. Basically, the duplication approach creates a non-rooted tree for each path from a primary input to a node with multiple fanouts. Later on, a redundancy removal is performed to

```
Lawler_algorithm (G, S)
{Input: Network's graph G. Size limit for a cluster, S}
{Output: Graph G labeled into clusters }

   List_trees:=Partition non-rooted trees into rooted (G);

   foreach rooted tree T in List_trees do
      Labeling in DFS order from inputs (outputs) until
         out-degree (in-degree) zero root node (T,G,S);
   endfor

   Relabel nodes from primary outputs (G, S);
end;
```

Figure 4.2: Lawler's algorithm.

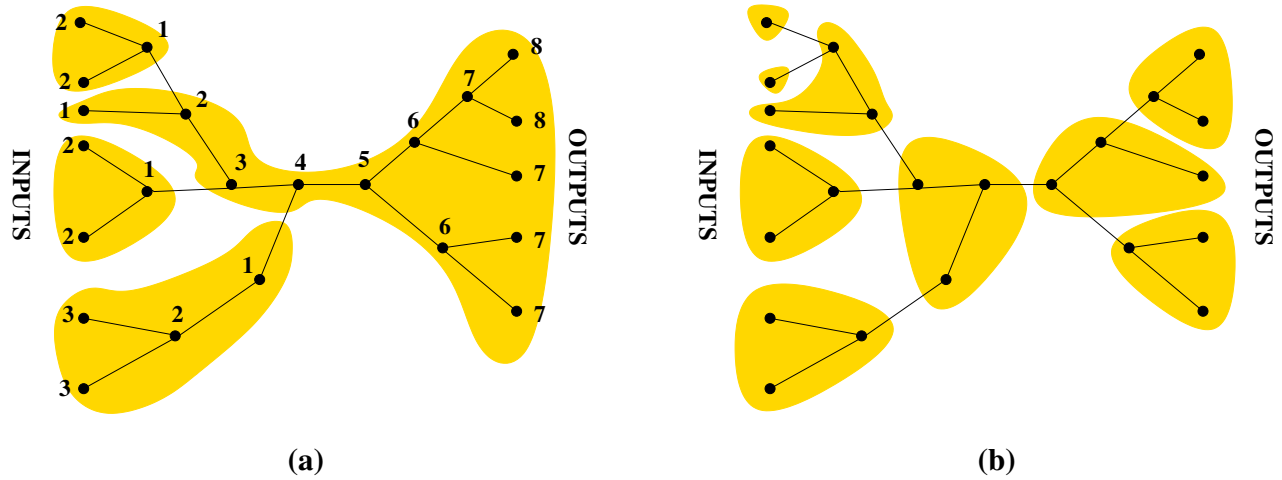


Figure 4.3: Labeling algorithm. (a) Computation of the depth of the rooted trees. (b) Labeling algorithm with maximum capacity constraint equal to three nodes.

```
Reduce_Depth_Script (G)
{Input: Boolean Network G }
{Output: Boolean Network G minimized }

   /* initial decomposition */
   sweep;                 /* Eliminate all the single-input nodes */
   decomp -q;             /* Decomposition using algebraic divisors */
   tech_decomp -o 2;      /* Decompose in OR2 gates */
   resub -a -d;           /* Resubstitute nodes into other nodes in the network */
   sweep;

   /* clustering */
   reduce_depth -b;       /* Lawler's algorithm with partial collapsing */

   /* logic minimization */
   eliminate -l 100 -1;   /* Eliminate nodes aiming at reducing the global literal count by 1*/
   simplify -l;           /* Simplify each node with local don't cares */
   full_simplify -l;      /* Simplify each node with global don't cares */
   decomp -q;

   /* Area recovery */
   fx -l;                 /* factor extraction */
   tech_decomp -o 2;

end;
```

Figure 4.4: reduce_depth script.

decrease the exponential grown of the network. The second approach assumes that no duplication can be done and the graph is divided in non-rooted trees. The second approach solves the excessive duplication, but the performance of the partition method drastically decreases since smaller non-rooted trees are extracted from the complex graph. The large number of nodes with multiple in-going and out-going edges prevents to create large non-rooted trees.

Several extensions of these algorithm were proposed. In [125], the labeling algorithm was extended to a more general delay model where nodes have an associated cost depending on the gate that represents and the number of fanouts. In [130], this approach is refined to a near-optimal delay-oriented partition. However, all these approaches do not report any result on logic minimization.

Finally, the Lawler's algorithm was introduced in a logic optimization procedure in [161]. The authors were concerned about the increment of area of Lawler's algorithm in general Boolean networks due to the excessive duplication. The relabeling approach was modified to reduce the number of duplicated nodes. This approach, called *reduce_depth*, is illustrated in Fig. 4.4. Note that, the

```
DEPART (G, S, DelayScript)
{Input: Boolean Network G;
Size limit for a cluster, S; Script for delay minimization DelayScript}
{Output: Boolean Network G minimized }

   Bins:=Create bins from transitive fanins of POs(G);
   Clusters:= Find Clusters with no-overlap(Bins,S);

   foreach cluster T in Clusters do
      Minimization Process(T,DelayScript)
   endfor

end;
```

Figure 4.5: DEPART algorithm.

figure gives the details of the optimization procedure in terms of commands in SIS [143]. We attach a small description for each command. Mainly, the minimization process consists to apply the clustering algorithm in a Boolean network in AND2/OR2 representation. The nodes have the same *complexity* in this representation and, thus, the unit delay model can be used to determine the complexity of the paths. After the Lawler's labeling, the clusters are collapsed into single nodes where don't care minimization is applied. Afterwards, area recovery methods, like common factor extraction, are applied to decrease the area on non-critical regions of the network. This method is relatively fast in small networks, but it runs out of time for large networks due to the computation of don't cares. Moreover, the results are far from the obtained ones with delay minimization methods applied without partitioning.

To our knowledge, DEPART [5] is the best method to perform a delay-driven partitioning for large Boolean networks suggested so far. The algorithm is presented in Fig. 4.5. This algorithm only has one constraint, the maximum capacity of the clusters, and it uses the unit delay model to calculate the criticality of the nodes. Initially, a partition based on the transitive fanins of the primary outputs is created. The clusters are called *bins*. Note that, the transitive fanins of two outputs may overlap in some nodes. The algorithm maximizes the sizes of the bins. Therefore, it merges multiples bins (multiple primary outputs) in a larger one if the size is less than the capacity constraint. The final partition is performed on top of the bins. Basically, the overlapping regions are pushed to the most critical bin and a primary input is created on the least critical bins. The drawback of this method emerges when the cone of logic of a critical output does not fit in one cluster. In this case, new clusters are greedily created removing the root nodes recursively until all sub-clusters are small enough to fit in the size constraint. Finally, the clusters are processed for minimization based

on the criticality of the root nodes (primary outputs) of the clusters. In this chapter, DEPART is taken as one of the references for comparison.

Another possible delay-oriented partitioning is the application of a generic partitioning method. *hMetis* [89] is a general purpose multi-level hypergraph partitioning method that recursively applies bisection to perform a k-way partition. This algorithm performs a fast coarsening step to obtain the partition and a final refinement step to improve the quality of the min-cut. Note that, hMetis accepts weights on nodes and edges of the hypergraph. Therefore, a delay partition approach can be performed by increasing the weight on the critical path. This method has been also selected for comparison.

4.3 Overview

The method DBP (Dominator-based partitioning) presented in this chapter aims at capturing fragments of critical paths that have small fanout to the rest of the circuit. Thus, the clusters tend to be deep (many levels) with internal nodes having little fanout to external nodes. This type of clusters offers more possibilities for restructuring towards delay minimization.

Clusters with little external fanout are sought by finding *dominators*. Intuitively, a dominator of a node *n cuts* all paths from *n* to the outputs [104]. This concept can be extended to multiple-vertex dominators [66] when the paths are cut by several nodes.

The moderate size of the clusters enables the use of conventional delay optimization techniques and to iterate over the network several times to gradually reduce delay with different cluster boundaries. This strategy produces results with better quality, still keeping the method scalable for large networks.

Figure 4.6 emphasizes the difference between hMetis [89] and DBP in a particular example[2]. It is an "artificial" circuit with 2-input AND gates that has been created only for this comparison. In both cases, the graph is partitioned into two clusters, denoted by the $\circ$ and $\bullet$ nodes, respectively. The cut-size generated by hMetis is 5 (the output edges of the same node are assumed to belong to the same hyperedge), whereas the one generated by DBP is 7. However, the clusters generated by hMetis cannot be topologically ordered, since there are edges $\circ \rightarrow \bullet$ and $\bullet \rightarrow \circ$. The clusters generated by DBP can be topologically ordered. Finding a topological order is crucial to propagate the arrival times of the outputs of one cluster to the inputs of the successor clusters. The partition obtained by DBP has been determined by the output node, that is a single dominator of the whole graph. The leftmost cluster has been obtained by taking the nodes closer to the dominator without exceeding the pre-defined cluster size. The rightmost cluster has been obtained by gathering the remaining nodes.

[2]For simplicity, the arrows of the edges are not shown and are implicitly assumed to go from the inputs to the outputs.

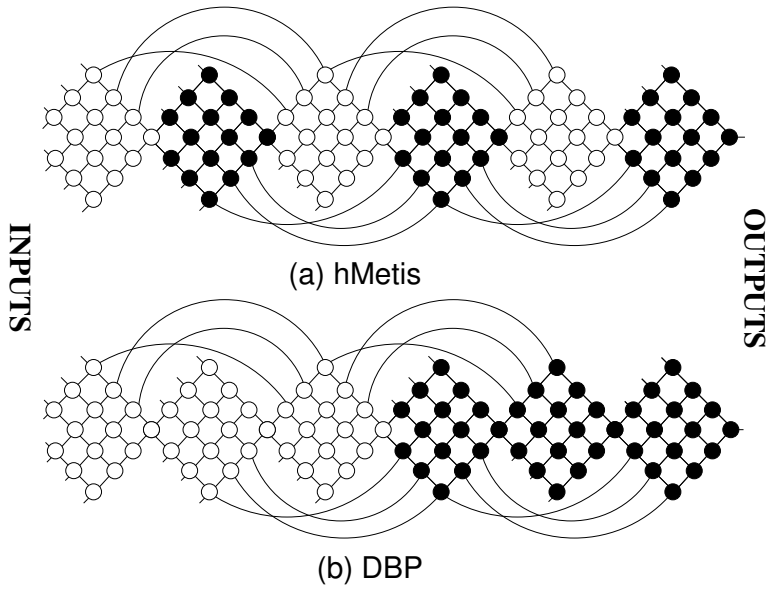
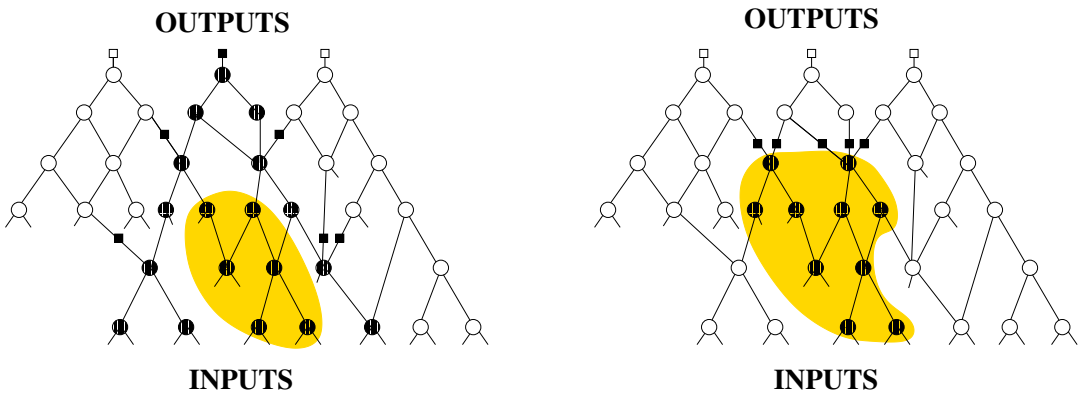


Figure 4.6: Comparison of hMetis and DBP.

Another consequence of the dominator-driven partition is that the longest path is divided into two parts using DBP. However hMetis splits it into six subpaths, thus preventing the optimization across the cluster boundaries. DBP can optimize every cluster in topological order, from inputs to outputs, propagating the obtained arrival times. This propagation is essential to achieve a global restructuring of the network. In this example, the circuit originally has 36 levels. The final circuit after restructuring each cluster, produces a result with 26 levels with hMetis and only 10 with DBP.

A second comparison between the partition performed by DEPART and DBP is done in Fig. 4.7. The white nodes and squares represent nodes and primary outputs that have not been selected in the cluster. Let us assume that only one cluster is performed. The black nodes correspond to the selected nodes and the black squares represent the new primary outputs of the subnetlist extracted from the cluster. DEPART (Fig. 4.7-(a)) clusters the nodes following the transitive fanins of the original primary outputs. The partition performed by DBP is shown in Fig. 4.7-(b). DEPART creates a cluster with larger number of outputs that implies a higher interaction with the rest of the network in comparison with the cluster obtained by DBP. The high interaction reduces the freedom in the subnetwork for restructuring. Only the shadowed region has total freedom. DBP increases the freedom to the whole the subnetwork, since the interaction with the rest of the network is limited by the dominators of the cluster.



(a) Depart (b) DBP

Figure 4.7: Comparison of DEPART and DBP.

4.4 Preliminaries

4.4.1 Vertex Dominator

The problem of finding single-vertex dominators in a graph was introduced in [111]. The authors proposed an $O(n^4)$ algorithm to compute all single-vertex dominators in a graph with n vertices. After several refinements [8, 127, 159], the Lengauer-Tarjan's algorithm was published in [104] with $O(\alpha(m,n))$ complexity, where m and n represent the number of edges and vertices respectively, and α is the functional inverse of the Ackermann's function. Basically, this algorithm performs a DFS-based approach with a good data structure to search the dominators in a graph. Linear time algorithms have been also presented in [14, 39, 76]. However, these approaches consume more computation time than Lengauer-Tarjan's algorithm on average. For this reason, we use Lengauer-Tarjan's approach in DBP.

The process to find multiple-vertex dominators is more complex. The first approach to compute all multiple-vertex dominators was presented in [66] with cost $O(n^k)$, where k is the size of the multiple-vertex dominator. However, as pointed out in the paper, the algorithm is only feasible for small k. In [160], a refinement for two-vertex dominators was presented. A new data structure (*dominator chain*) is introduced to reduce the memory space to compute the dominators. DBP uses these approaches to search multiple-vertex dominators.

The concept of *dominator* is a key in our approach. Dominators are widely used in several areas, such as code optimization in compilers [77] and test pattern generation techniques [15]. Recently, dominators have been used in logic synthesis for non-disjoint decomposition of Boolean functions [66].

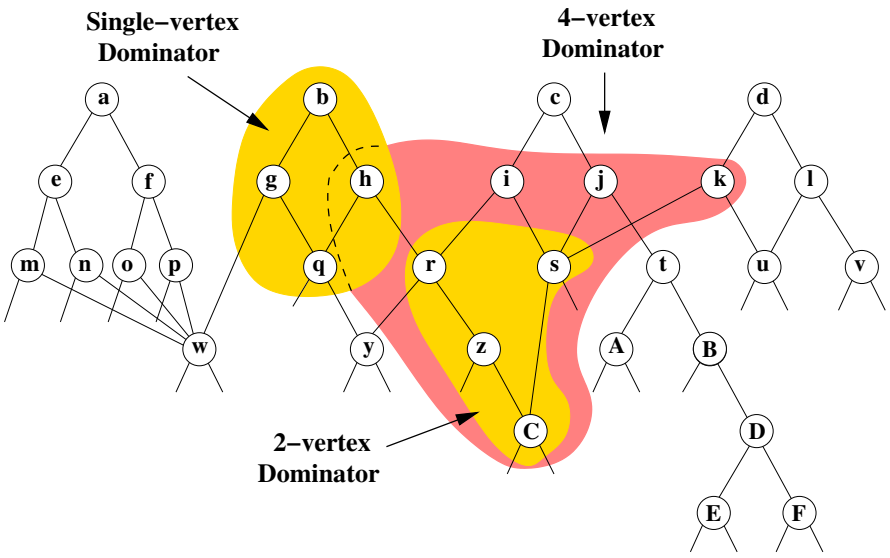


Figure 4.8: Example of several vertex dominators.

Definition 4.4.1 (Dominator) *Given a network $G = (\mathcal{V}, \mathcal{E})$, a subset of nodes $X = \{x_1, x_2, \cdots, x_k\} \subset \mathcal{V}$ is a dominator of a node $u \in \mathcal{V} \setminus X$, denoted by $\{x_1, x_2, \cdots, x_k\} \in Dom(u)$, if:*

- *Every path from u to a primary output contains some vertex $x_i \in X$, and*
- *no proper subset of X is a dominator of u.*

This definition can be naturally extended to sets of nodes, i.e. a subset of nodes being the dominator of another subset of nodes. □

Example 4.4.1 *In Fig. 4.8, $\{b\}$ is a single dominator of $\{g,h,q\}$, but does not dominate r since there is a path from r to c that does not cross b. $\{r,s\}$ is a double-vertex dominator of $\{z,C\}$. Dominators of larger size can also be found, e.g. the 4-vertex dominator $\{h,i,j,k\}$ of $\{r,s,z,C\}$. Note that every node (or set of nodes) can have different sets of multiple-vertex dominators. For example, $\{C\}$ is also dominated by $\{r,s\}$ and $\{h,i,s\}$.* □

4.4.2 Windows

In our work, we are interested in windows using vertex dominators. A subset of nodes induces a *window* (cluster) in a Boolean network. The window contains all edges between nodes of the window. A window can be *extracted* from the Boolean network, transformed and inserted back into

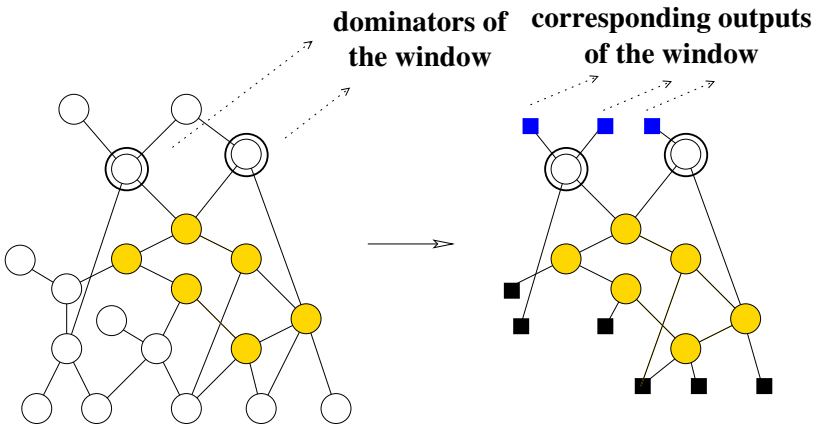


Figure 4.9: Window extraction in a Boolean network.

the network. For that, the set of inputs and outputs of the cluster must be identified to preserve the interface with the rest of network, as shown in Fig. 4.9. The dominators are important for partitioning since the produced windows have few primary outputs and, thus, low interaction in the rest of the network. In the example of Fig. 4.9, the extracted window has three primary outputs, derived from the three pins connected from the two-vertex dominator nodes of the window to their fanouts.

4.5 Partition Method

The method presented in this chapter aims at partitioning the nodes of the Boolean network into a set of disjoint windows. This process is initiated by considering every node as a window and iteratively clustering them to build larger windows without exceeding some capacity constraint. In this iterative clustering, the concept of *dominator* plays an essential role. We next describe the details of the partitioning algorithm, presented in Fig. 4.10. The execution of the algorithm is later illustrated with the example shown in Section. 4.5.2.

4.5.1 Core of the algorithm

Initially, several cost functions are computed in the graph. A weight is associated to each edge to denote the number of wires between a pair of windows. At the beginning, each node of the network is a window. Therefore, the associated cost to each edge is one. Moreover, the vertices have a delay cost based on the slack between the arrival and the required time of the network that represents the

```
DominatorPartition (G, S)
{Input: Network's graph G(V,E). Size limit for a window, S}
{Output: Graph G clustered into windows }
  repeat
    Doms := FindDominators (G)
    while Doms ≠ ∅ do
      bestDominator := SelectBestDominator (Doms, S)
      window := ClusterDominatedWindows (G, bestDominator)
      Doms := Doms - {Dom(v_i) ∈ Doms | v_i ∈ window} -
                  - {Dom(x) ∈ Doms | v_i ∈ window ∧ x ∈ V − {v_i} ∧ {v_1,...,v_i,...,v_k} ∈ Dom(x)}
    endwhile
  until No changes
  ClusterSmallWindows(G, S)
end;
```

Figure 4.10: Algorithm for partition a network into windows.

graph. In our experiments, the unit delay model has been used to calculate the delay information, after having decomposed the network into 2-input nodes.

The algorithm receives two parameters: the graph of the network G to be partitioned and the size S that defines the maximum capacity for a window. The function `FindDominators` calculates all the single and double-vertex dominators of the graph. Dominators of larger size are not computed due to the complexity of the process ($O(n^k)$). Moreover, it has been observed experimentally that calculating larger multiple-vertex dominators has a negligible impact on the results, while increasing the computational complexity significantly.

The innermost loop merges windows using the calculated dominators. The selection of the best dominator is performed according to the characteristics of the window of dominated nodes (including the dominator itself). The criteria to select the best dominator are the following in priority order:

1. Smallest number of output nodes in the window. The output nodes are those having fanout outside the window. Initially, the output nodes are the dominators themselves in the single-node windows. In a partially-clustered network, a window acting as a node in a dominator can have several clustered output nodes.

2. Smallest weight of the outgoing edges from the window.

3. Largest size of the window

The selection aims at capturing windows with low interaction with the rest of the network. Windows

with many output nodes are not desirable, since they favor large external fanout. As a consequence, this type of windows have fewer optimization opportunities.

When the size of the window induced by a dominator exceeds *S*, some of the nodes are excluded from the window. In the case of delay optimization, these nodes correspond to the least critical nodes in the window (largest slack).

The procedure `ClusterDominatedWindows` merges the dominated windows into a single one. The process of clustering updates the cost associated to the new node and the surrounding edges. The slack associated to the new node corresponds to the minimum slack among all clustered nodes, and every edge computes the number of subsumed interconnections on it.

The new window may contain other dominators of the network (a dominator can be dominated by another dominator). For this reason, all the nodes from the window are removed from the set of dominators. Moreover, some clustered nodes may be included on other dominators. Therefore, an efficient elimination process traverses the list of dominators in *Doms* and removes these nodes from the existing dominators. The clustering proceeds until no more dominators exist.

The outermost loop executes the clustering loop to build larger windows. After the first iteration, most of the windows are not single nodes any longer and the new windows are built by merging windows from the previous iteration. The process continues until no more clustering is possible.

It is important to realize that the partitioning algorithm never explicitly looks for dominators with more than two nodes. However, multiple-vertex dominators are implicitly used for clustering by iteratively applying the clustering with single- and double-vertex dominators. This property enables an efficient use of multiple-vertex dominators without an excessive computational complexity.

At the end of the main loop, some small windows or individual nodes might remain *orphan*, out from the large windows generated by clustering. The procedure `ClusterSmallWindows` still gives an opportunity for further clustering. Here is where *S* is taken as a soft constraint and a moderate growth of the windows is tolerated to incorporate neighbouring small windows. In the experiments done in this chapter, a 25% increase from *S* is tolerated in the final phase of partitioning. The criteria for selecting the small windows to be merged is similar to the used criteria to select the dominators:

1. Smallest number of output nodes in the small window.

2. Smallest weight of the outgoing edges from the small window.

3. Smallest size of the small window to be merged.

These criteria give priority to small windows with less weight on the edges. Windows with a large number of output nodes are not suitable for merging, since this decision would increase the number of outgoing edges of the new window and, as a consequence, the interaction with other

windows. It may occur that a small window has output connectivity with multiple windows. The next criteria is used to select the edge:

1. Largest weight on the edge.
2. Smallest slack on the new window.
3. Smallest size of the resulting window.

Therefore, orphan/small windows are merged with the windows with largest output connectivity rate and with smallest slack on the resultant window. This criteria will create windows that tend to be deep (many levels) with internal nodes having little fanout to external nodes.

4.5.2 Example of a partitioning

Figure 4.11 shows an example of applying the `DominatorPartition` algorithm to a small graph. The size limit for a window is defined as $S = 8$. The circles represent simple nodes, the boxes represent windows obtained after clustering. The numbers in boldface inside every window indicate the number of nodes contained in the window. The two other small numbers indicate the number of output and input nodes on the interface of the window. The connectivity cost is indicated by the weights in the edges (omitted if the cost is 1).

Using the original network graph (Figure 4.11.a) the single vertex dominators are first selected, since they produce windows with the least number of outputs. These dominators are used to cluster simple nodes into windows as indicated by the shadow clouds. If the size limit is exceeded, like in the case of the dominator c that dominates 9 nodes $\{c, i, j, t, A, B, D, E, F\}$, some nodes are excluded from the window using delay criticality information. Assuming, in this example, that all arrival times on the primary inputs are the same, node A is excluded from the window since it has a delay slack larger than other nodes dominated by c.

When the clustered object is a window with multiple nodes, the delay slack is calculated as the minimum slack among all the outputs of the window. Therefore the most critical path captured inside the window is the one that determines the criticality of the window.

After the first iteration, the partially-clustered graph shown in Fig. 4.11(b) is obtained. At this point, two windows (labelled with 4 and 2) are selected as a double-vertex dominator for node y. The clustering of the three nodes derives another window with 7 nodes and leads to the graph shown in Fig. 4.11(c).

The main loop of the algorithm completes and delivers the clustering in Fig. 4.11(c). We can observe that there are still some *orphan* nodes in the graph. Here is where the procedure `ClusterSmallWindows` does the rest of the work. First, node A is selected, since it is a single

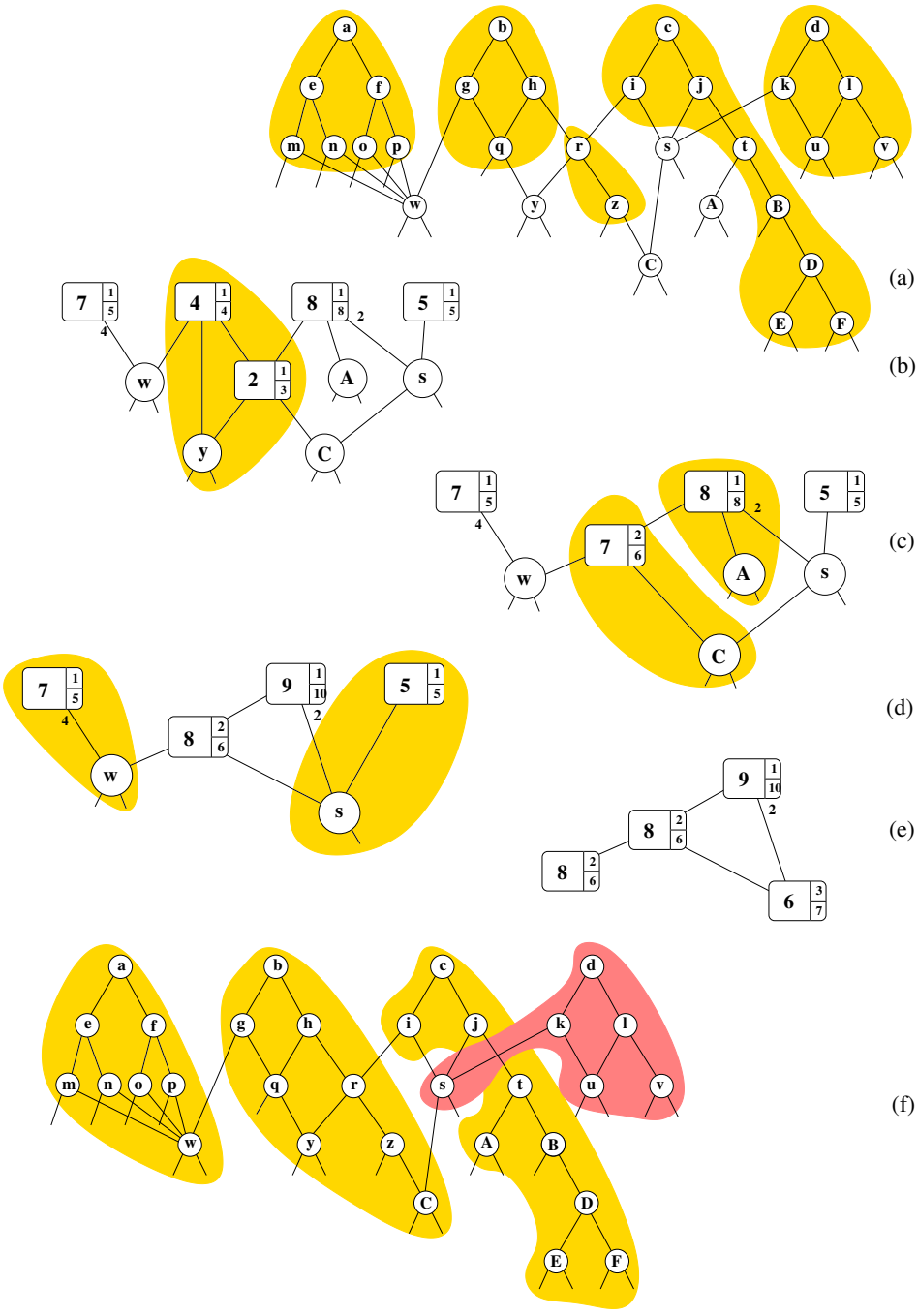


Figure 4.11: Example of a dominator-based partitioning.

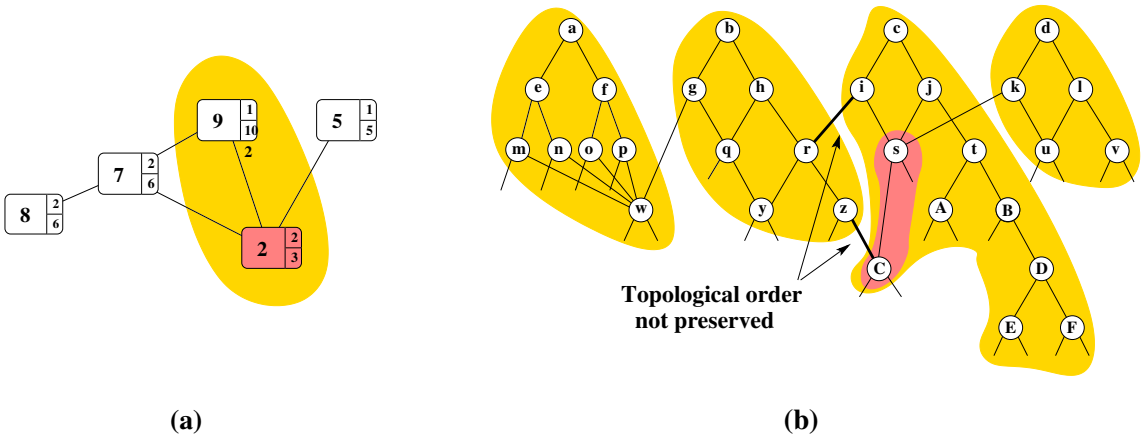


Figure 4.12: Example where the topological order is not preserved. (a) Clustering windows with 9 and 2 nodes. (b) Resulting partition after the clustering.

node with one outgoing edge. Its neightbour on top has reached the maximum capacity $S = 8$. However, an increment of the capacity (25%) is allowed in the procedure `ClusterSmallWindows` and node A is merged. Next, node C is merged with the node s since a smaller cluster is built. Figure 4.11(d) shows the clustering of the node w and the small window with 2 nodes. Node w with 5 outgoing edges in total is processed. Here, the merging criteria push the node to the window with higher interaction. Finally, the window with 2 nodes is merged with the window with 5 nodes. Note that, it should be combined with the window with 9 nodes, since the edge has largest size. However, node s is a special case that will be explained in the next section. Figures 4.11(e-f) depict the final result of the partitioning and the nodes on each cluster respectively.

4.5.3 Preserving topological order

A clustering algorithm using dominators preserves the topological order among the windows. However, the procedure `ClusterSmallWindows` does not guarantee this order. Consider the hypothetical partial partitioning of Figure 4.12. The partitioning algorithm should have clustered the window with 2 nodes with the window with label 9, since there are two wires that interconnect them. However, this clustering would create a cycle in the graph due to the three edges $2 \rightarrow 7$, $2 \rightarrow 9$ and $7 \rightarrow 9$ that form a cycle. Graphs with cycles cannot be topologically ordered, which makes the propagation of the delay information between windows impossible.

The DBP algorithm prevents cycles by imposing the following constraint:

> *A window cannot be created if one of its output nodes belongs to the transitive fanin of one of its input nodes in the graph before clustering.*

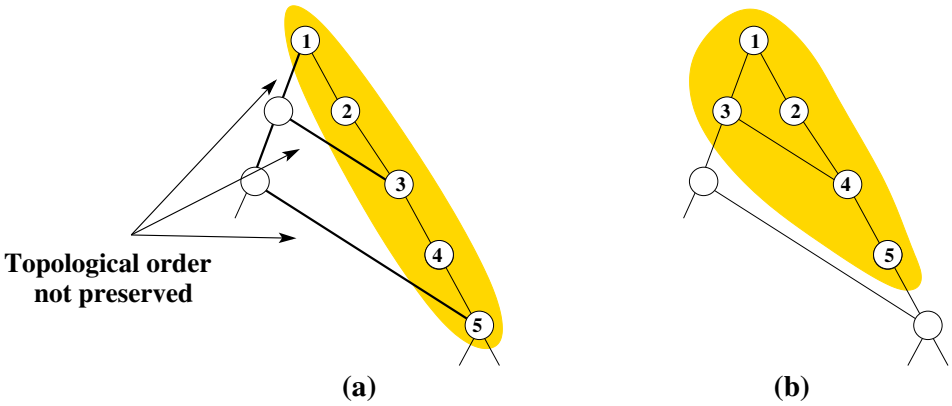


Figure 4.13: Order of clustering in a vertex dominator. (a) Topological order is not preserved. (b) Non-critical nodes are also selected to preserve the topological order.

Note that, this constraint is also applied when the number of dominated nodes exceeds the capacity constraint. As we explained, the most critical nodes are included in the window and the least critical are excluded. Figure 4.13(a) illustrates the order. Let us take the single dominator of the figure with a maximum capacity constraint of 5 nodes. The number assigned to each node refers to the order of selection. As we can observe, critical nodes may have non-critical outgoing branches included in the dominator. Therefore, these non-critical nodes in the transitive fanout must be previously selected to avoid potential cycles and fulfill the topological order between the nodes. Figure 4.13(b) depicts the current order of selection.

4.6 Timing-driven optimization

Figure 4.14 presents the algorithm for timing-driven optimization of large Boolean networks. Initially, a pre-processing step with low computational cost can be applied to the whole network. In our case, we implemented the algorithms in SIS [143] and used some of the typical scripts for logic synthesis. The network was finally decomposed into 2-input gates.

The algorithm has two main loops: one for delay optimization and another for area optimization. These loops use different optimization scripts (DelayScript or AreaScript in Fig. 4.14).

The structure of each loop is similar. It is a set of procedures that are repeated until no further improvement is observed. This iterative process is captured by the outermost `repeat-until` loops.

We next describe the details inside each optimization loop. Initially, the delay information is calculated: arrival times, required times, and slacks based on the unit delay model. Next, the critical

```
DelayOptimization (N, DelayScript, AreaScript, Slack, S_d, S_a)
Inputs:
   N: Boolean network;
   DelayScript: Script for delay optimization;
   AreaScript: Script for area optimization;
   Slack: slack to select the critical region;
   S_d: size limit for windows during delay optimization;
   S_a: size limit for windows during area optimization;
Output:An optimized network N

   Preprocess(N)
   {Delay optimization}
   repeat
      CalculateDelayInformation(N)
      Critical := SelectCriticalRegion(N, Slack)
      DominatorPartition(Critical, S_d)
      for each window W in Critical visited
                    in Topological Order from PIs do
         DelayScript(W)
         PropagateDelays (W)
      endfor
   until No delay improvement

   {Area optimization}
   repeat
      CalculateDelayInformation(N);
      NonCritical := SelectNoncriticalRegion(N, Slack)
      DominatorPartition(NonCritical, S_a)
      for each window W in Noncritical visited
                     in Topological Order from PIs do
         AreaScript(W)
         PropagateDelays (W)
      endfor
   until No area improvement

   return N
```

Figure 4.14: Algorithm for optimizing large networks.

	DEPART	hMetis		DBP
		GhM	hM	
Duplication	Non-overlap	Non-overlap		Non-overlap
Region of application	overall network	overall network	critical/non-critical region	critical/non-critical region
Partition method	Select transitive fanin of POs	min-cut + criticality weight on edges		Dominator-based
Order minimization	criticality of POs	Topological order from PIs + break cycles [67]		Topological order from PIs
Size constraint exceeded	remove root node	Never exceeded		remove the least critical nodes

Table 4.1: Comparison of DEPART, hMetis and DBP.

region of the network is extracted. The critical region is defined as the set of nodes with slack smaller than a pre-defined slack (parameter *Slack* in Fig. 4.14). In case of the area optimization loop, we select a non-critical region according to the same slack information.

The selected region of the network (either for delay or area optimization) is partitioned using the dominator-based algorithm described in Section 4.5. After this step, a set of disjoint windows that cover the region is defined. The size of the windows (S_d or S_a) is defined according to the computational complexity of the optimization script that is used in the innermost loop. For synthesis scripts with large run-time, the size of the windows must be smaller than the one used for simpler scripts.

The innermost loop visits the windows in topological order, from inputs to outputs. For each window, the synthesis script is applied to optimize either for delay or area. This script transforms the window W without changing its interface to other windows. Next, the delay information is propagated and re-calculated incrementally.

Once the innermost loop is completed, another iteration of the outermost loop is executed, until no improvements are produced.

4.7 Experimental results

To evaluate the efficiency of the optimization algorithms presented in this chapter we have conducted three types of experiments:

1. comparison with DEPART [5] on medium-size examples,

2. comparison with mincut partitioning (hMetis) on large examples, and

3. evaluation of the trade-off between area and delay depending on the size of the windows.

The next section summarizes the basic differences between DBP, DEPART and hMetis. Moreover, the configuration of the algorithms is also described.

4.7.1 Configuration of the algorithms

Table 4.1 shows the differences between the three tools. There are two different applications of the hMetis tool. The two variations differ on the region where they are applied. `GhM` performs the partition in the whole network similar to DEPART, meanwhile `hM` is applied on the critical region produced by the selected slack. Note that, DEPART and GhM do not have any area recovery step. Basically, hMetis has been applied using the algorithm of Fig. 4.14. The `DominatorPartition` procedure has been substituted by hMetis where `Ghm` has $Slack = \infty$ and `hm` has $Slack = 2$ (the same as in DBP). In both cases, the weights of the edges between nodes on the critical region were multiplied by a constant factor to bias hMetis towards avoiding cuts in the critical paths (the results of hMetis partitioning are much worse without weights). With these experiments, we estimated the contribution of running different optimization scripts for the critical and non-critical regions of the network.

As explained in Section 4.3, the main drawback in DEPART appears when the cone of logic of a primary output does not fit in the maximum capacity constraint. The root nodes are removed recursively until the capacity constraint is not exceeded. This technique deletes the top nodes of the transitive fanin of the primary output preventing to apply balancing techniques, commonly used on delay minimization, between the children of the primary output.

hMetis does not guarantee a topologically-ordered partition. Therefore, the delay information between windows is impossible to propagate. A minimum set of feedback edges is calculated and ignored to obtain an order close to topological order [67]. The delay information in the feedback edges is obtained from the one calculated in the previous iteration since it is not possible to propagate it in topological order in the current one.

All our experiments were run on a PC with a 3 Ghz Intel Pentium 4 CPU and 512 MB main memory. For the first two experiments, the size of the windows for DBP is $S_d = 50$ for delay optimization and $S_a = 100$ for area optimization with a *Slack* of 2 units. The delay script used for optimization was simply the *speed_up* [147] command in SIS, whereas the area script was the *algebraic*.

4.7.2 Comparison with DEPART and speed_up

The results for the first experiment are shown in Table 4.2. A technology-independent comparison is performed. The goal of this experiment is twofold: to compare our method with DEPART [5] and with *speed_up* applied to the flat netlist. The number of literals, the number of levels in 2-input gates and the runtime are reported for each netlist. The normalized average results are shown in the last

Bench.	Levels			Literals			CPU (sec)	
	sp_up	DEP.	DBP	sp_up	DEP.	DBP	sp_up	DBP
C880	21	21	20	834	913	873	8	8
alu4	26	26	25	1214	1917	1398	25	19
C2670	16	17	15	1450	1482	1442	50	46
apex5	14	14	14	1465	1580	1504	2	3
table3	40	44	40	1746	2750	1746	120	37
C3540	33	33	33	2275	2382	2359	33	14
apex3	13	12	13	2569	2810	2575	21	14
seq	14	15	15	2895	3003	2905	19	25
C5315	22	25	22	3081	3099	3152	29	23
pair	18	20	15	3105	3158	3522	16	6
C7552	23	22	21	4127	4577	4547	61	37
des	19	20	19	6083	6223	6385	52	51
C6288	69	75	66	6627	6367	7655	464	236
Norm	1.00	1.05	0.97	1.00	1.08	1.07	1.00	0.58

Table 4.2: Comparison of speed_up, DEPART and DBP.

row of the table. The experiment has been performed on the same MCNC benchmarks used in [5], using the same pre-processing script (twice *script.rugged* followed by *eliminate -1; speed_up -i*) and the same measurement units for delay and area.

The results for DEPART are the ones reported in [5] in which the windows were constrained to have 200 nodes at most. We observed that this size was excessive for *speed_up* in the optimization of some windows and, for this reason, we chose a smaller size ($S_d = 50$) for running DBP.

Even using smaller windows, DBP is on average superior to DEPART in delay (0.97 vs. 1.05), while similar in area. Since the networks have moderate size, *speed_up* was also executed on the whole network to compare with the same command applied within a window using the DBP-based partitioning. Surprisingly, the application of *speed_up* to the DBP-windows was superior in delay than applying it to the whole network (3% improvement in delay at the expense 7% increase in area). One of the reasons for that is that the restructuring obtained by *speed_up* depends on the order in which the transformations are performed (e.g. see Fig. 2 in [56]). The dominator-based clusters offer a better guidance for *speed_up* and prevent transformations that can later result in worse delays. Additionally, the runtime time between the flat method and DBP is compared, and a clear reduction for DBP is observed. The runtime for DEPART is not shown, since the CPU and the windows sizes used for the experiments were different.

This experiment gives us the confidence that our partitioning method does not incur in large penalties with regard to the flat method, even on relatively small examples.

Bench.	Literals				Levels				CPU (sec)		
	Orig.	GhM	hM	DBP	Orig.	GhM	hM	DBP	GhM	hM	DBP
b14	11122	15972	14443	14515	92	44	32	32	167	180	404
b14_1	9424	14701	11871	12393	70	40	30	30	191	117	329
b15	17829	22962	20359	20229	91	63	54	49	298	140	686
b15_1	17116	21422	18766	18829	71	38	31	30	590	157	528
b17	56405	77962	62990	63332	128	62	55	53	4266	617	1613
b17_1	53188	66455	57323	57889	71	37	33	32	2697	415	1274
b20	22727	34644	30969	32634	108	54	37	36	668	349	488
b20_1	19663	36471	27408	26008	105	46	36	36	1111	385	515
b21	23654	37815	30127	31364	109	51	38	36	893	538	523
b21_1	19652	33726	27465	28544	99	46	35	35	751	338	675
b22	34441	59030	39316	44592	101	43	51	38	2246	166	1078
b22_1	29799	54271	34535	40969	102	47	48	36	1711	229	892
s35932	16304	19376	19352	19500	22	10	9	9	267	620	104
s38417	22172	25785	23184	23694	31	25	24	23	546	69	84
s38584	20086	21470	20146	20222	28	22	21	20	71	184	327
Norm		1.00	0.80	0.82		1.00	0.85	0.78	1.00	0.27	0.57

Table 4.3: Technology independent comparison between hMetis and DBP for large networks.

4.7.3 Comparison with hMetis

The results of the second experiment are presented in Table 4.3 and Table 4.4. They have been executed on the largest ISCAS'99 benchmarks, selecting only those that were larger than 9000 literals in factored form after applying the pre-processing script. In this experiment, a lighter script was applied as pre-processing step (*algebraic* script and *speed_up -i*) instead of the *script.rugged*, since don't care calculation is infeasible on large examples.

The experiments were also run using *reduce_depth* [161]. This command was fast on small examples, but ran out of time for large networks. The obtained results were worse than the hMetis-based algorithm and, for this reason, are not shown in the table.

Table 4.3 reports the results of the technology independent optimization after the initialization script (Orig.), and for all three methods. Table 4.4 reports the results after technology mapping using the tree-mapping algorithm *map* [133, 162] in SIS with the *lib2* library. In the technology independent comparison, DBP improves delay by 22% and area by 18% when compared to global hMetis (GhM). After technology mapping the delay and area improvement are reduced to 16% and 18% respectively. Global hMetis produces networks with larger area, since non-critical regions are also optimized for delay. After technology mapping, DBP offers a 5% delay improvement with a cost of 4% in area comparing with hMetis focused on the critical path. DBP creates windows where the delay minimization script performs a better optimization.The runtime of DBP is approximately 2x higher than hMetis, since the delay script spends more time on the minimization of individual

Bench.	Area			Delay		
	GhM	hM	DBP	GhM	hM	DBP
b14	11017	9893	9809	44.4	37.7	36.7
b14_1	10058	8222	8706	40.5	34.3	33.6
b15	14734	12811	12814	61.9	56.4	53.4
b15_1	13636	11947	12127	40.5	35.4	35.1
b17	50048	38704	38669	68.6	60.3	58.0
b17_1	41832	35550	36630	43.1	38.0	35.2
b20	23123	20617	22072	57.3	41.5	40.1
b20_1	25101	18425	17512	46.3	40.7	40.5
b21	25859	20040	20988	50.0	41.6	40.6
b21_1	23288	18607	19531	48.2	40.0	40.2
b22	40543	24946	29228	45.2	51.3	42.0
b22_1	37291	22201	27111	48.9	47.6	40.1
s35932	13657	12742	12787	14.0	13.6	13.8
s38417	17349	15067	15462	26.6	26.7	24.5
s38584	13480	12787	12810	22.5	23.2	21.5
Norm	1.00	0.78	0.82	1.00	0.89	0.84

Table 4.4: Technology dependent comparison between hMetis and DBP for large networks.

windows (that have on average larger depth in case of DBP) and more iterations are executed to reach a network with no further improvements. Note that, the CPU time for partitioning is negligible compared to the time for the logic optimization within the windows.

4.7.4 Trade-off between area and delay

The objective of this experiment is to evaluate the impact of the window size on the results obtained by the previously described methods (Figure 4.15). The experiment is conducted using different sizes for S (12, 25, 37, 50, 62, and 75 nodes) and, as a reference point, the method *GhM* with $S = 50$.

The main conclusion is that DBP always obtains better results in delay regardless the size of the windows. Moreover, as the plot shows, the impact of the window size on the results of the DBP is much smaller than the impact on the results of the mincut-based methods.

The mincut-based methods (hMetis and global hMetis) perform better (delay-wise) results for windows with size of 20-40 nodes. For smaller sizes there is a significant degradation, which is easily explained by the restructuring limitations imposed by the size of the window. However, a similar degradation in delay is also observed when the window size grows. We studied this strange phenomenon in more detailed and we observed that the topological order of the clusters was violated more often when the window size grew, i.e. the larger the windows, the smaller the probability

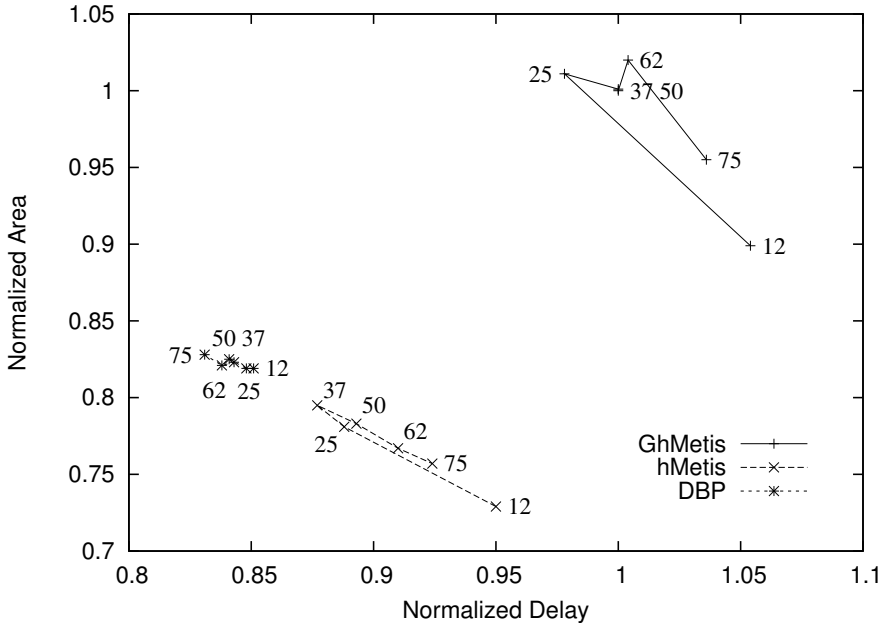


Figure 4.15: Trade-off area-delay depending on window sizes.

to obtain a topologically ordered partition. For this reason, the suboptimal propagation of delay information has a negative influence on the resulting delay.

The CPU time is also affected by the size of the windows. For DBP, the runtime increases with the size of the windows, whereas it decreases for hMetis. The dependency is the opposite with larger windows, hMetis stops on local minima much earlier and executes fewer iterations of the algorithm.

4.8 Conclusions

Scalability is a crucial aspect for the applicability of logic synthesis techniques on large networks. A partitioning technique based on the calculation of dominators has been proposed to tackle the complexity of delay optimization. The drawback of the conventional min-cut partitioning is that only the structure of the graph is considered. A dominator-based partition captures the Boolean information inherent in the graph. A second advantage in a dominator-based partition is that the topological order is preserved allowing the proper propagation of the delay information.

Related to the experimental results, we showed that DBP is superior to the other techniques in delay optimization. However, there is a trade-off between the results and the maximum capacity

constraint as we showed in Section 4.7.4. Depending on the delay optimization technique used on the windows, the maximum capacity constraint should be adjusted to not incur in large penalties in the runtime. However, the size has to be reasonable since a very small window could produce a non-effective minimization.

DBP is focused to improve the delay of a network. Internally, the selection of the windows is targeted at capturing blocks of logic where a major improvement on delay can be achieved. However, this selection can be modified. As future work, we propose to modify DBP to consider several objective functions. In DSM technologies, the combination of logic and physical synthesis seems to be essential to meet the demand of today's designers regarding delay and power optimization. We believe that the proposed partitioning strategy, enhanced with layout information, could be a valid approach for integrating and exploring logic and physical parameters of the design.

Chapter 5

Window-based timing-driven n-way decomposition

5.1 Introduction

Logic decomposition is a logic transformation that has been extensively used in multi-level logic minimization [28, 120, 169, 171]. Algebraic and Boolean divisors are computed for each node in the network and they are extracted as new nodes. Next, the existing nodes are re-expressed using the new introduced ones. Mostly, logic decomposition has been used aiming at reducing the area of the Boolean network, since common factors of different functions can be shared during the decomposition. Moreover, logic decomposition has been also applied targeting to other objective functions like timing [154] or layout [105].

Logic decomposition mostly depends on the initial structure of the network. Sharing common factors is more likely in networks with large functions on the nodes. Therefore, some decomposition approaches perform a partial collapse of the network, or even a complete collapse of the primary outputs, to obtain more factors and perform a better decomposition. In this chapter, we refer to timing-driven recursive decomposition methods applied to entirely or partially collapsed networks. In partially collapsed networks, the method is specifically applied in the collapsed regions.

This chapter presents a timing-driven n-way decomposition method as an application of the Boolean relation solver. The technique is an extension of the bi-decomposition method presented in [56]. The presented algorithm shows improvements on area and delay with regard to [56]. However, the range of application is limited to small- and medium-sized networks due to the complexity of the decomposition problem with Boolean relations. In order to process larger networks, we propose to apply the DBP partitioning method presented in Chapter 4.

This chapter is organized as follows. First, timing-driven logic decomposition methods are

```
Decomposition (F, Cost)
{Input: A function F,
        Objective cost Function Cost}
{Output: A decomposed function}

  var
    List_dec: list of decompositions;
    Best_dec: decomposition;
  end var

  if F is a primary input then return F;

  List_dec:=Calculate_Several_Decompositions(F);
  Best_dec:=Evaluate_Cost_Decompositions(List_dec,Cost);

  // Best_dec = α(f1, f2, ..., fn)
  // α refers to which function has been selected
  //   to perform the decomposition, e.g. AND,OR,XOR,...

  for each subfunction fi ∈ Best_dec do
    Deci:=Decomposition (fi,Cost)
  end for

  return α(Dec1, Dec2, ..., Decn);
end;
```

Figure 5.1: Standard decomposition algorithm

reviewed on Section 5.2. An overview of the n-way decomposition approach is presented on Section 5.3. Section 5.4 describes a brief background on logic decomposition. Section 5.5 presents the n-way decomposition method and the implementation aspects are described in Section 5.6. Finally, the experimental results are reported in Section 5.7.

5.2 Previous work

In this section, an introduction to several timing-driven logic decomposition methods is presented. Due to the extensive literature in this topic, the most relevant approaches are reviewed.

The basic structure of a recursive decomposition algorithm, showed in Fig. 5.1, can be divided in two steps. First, several decompositions are computed and evaluated based on the objective cost function. The best solution is selected and the algorithm is recursively applied on each subfunction

f_i. This process stops when the targeted function is a primary input. The second step rebuilds the Boolean function based on the selected decompositions on each recursive call.

The proposed techniques tend to use an algorithm similar to the presented one, but they differ on how the decompositions are searched and which cost function is used. The main differences between the decomposition techniques are described next:

- Disjoint and non-disjoint supports: The basic difference is in the intersection or not between the support of the subfunctions. Sometimes it is difficult to find disjoint decompositions, Mostly, the proposed methods tend to allow some intersection between the supports since a larger search space can be explored.

- BDD decomposition: BDDs contribute to improve considerably the runtime of the decompositions on large functions. However, the evaluation of a BDD decomposition may differ from the real cost[1], since cost functions based on BDD representation, like BDD size or length of the largest path to leaf 1, are highly dependant on the variable ordering.

- Sharing: The algorithm presented in Fig. 5.1 frequently produces equivalent functions during the recursive decomposition. The BDD representation enables to check the Boolean equivalence between functions in constant time. This property allows to share the decomposition and avoids the application of the recursive approach multiple times in the same function. A look-up table is commonly used to store the BDD of a function together with its decomposition to check if the current function has been already processed. However, if sharing is not applied, the same function can be decomposed using a different structure depending on the criticality of its fanouts in the network. Delay-driven decompositions with not sharing have notable improvements on delay at the expenses on increasing substantially the area.

- Technology-dependent or technology independent: Logic decomposition is usually performed in the technology independent phase. However, some technology mapping techniques for FPGA are combined with decomposition to improve the quality of the final mapped circuit.

There are several techniques combining logic decomposition with technology mapping [42,103] or layout-aware [105]. Logic decomposition has been also applied for restructuring critical regions of a Boolean network [138]. However, this section is focused on timing-driven recursive decomposition paradigms in the technology independent phase.

In timing-driven optimization, several methods have been proposed. A non-disjoint technique was presented in [169]. This approach is a bi-decomposition process that explores the AND and XOR decompositions. Basically, a branch-and-bound algorithm is used in other to explore all possible decompositions of a Boolean function. Figure 5.2 graphically illustrates the algorithm. First,

[1] *Real cost* refers to the cost in area and/or delay of the decomposition in 2-AND/OR DAG representation.

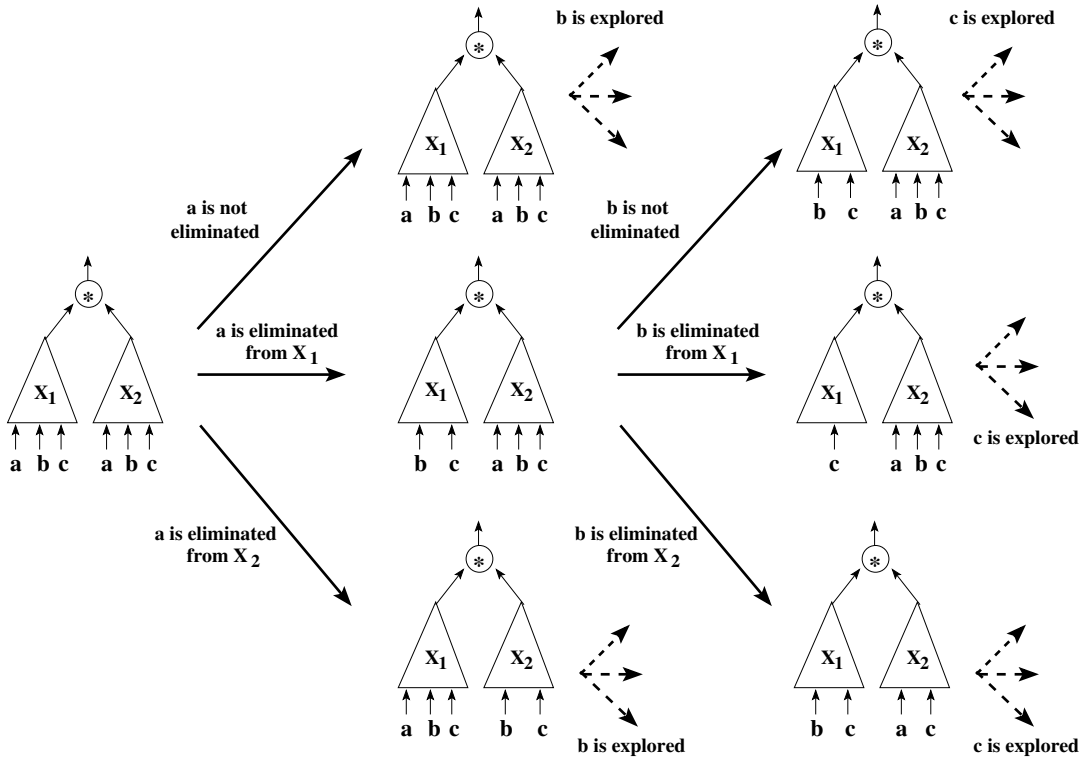


Figure 5.2: Branch-and-bound approach proposed in [169].

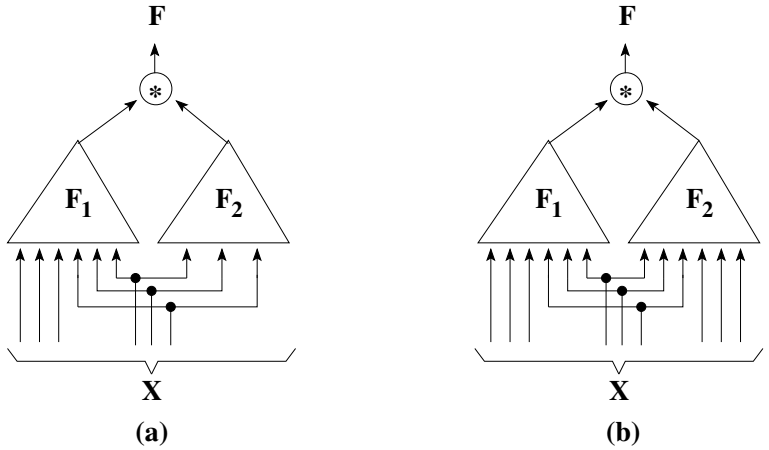


Figure 5.3: (a) weak bi-decomposition. (b) strong bi-decomposition.

an initial decomposition is selected. The branch-and-bound starts selecting one input variable and choosing whether the variable is removed or not from the children. Note that, the variables must be kept at least in one of the children. When an input variable is removed from one child, the other one must adopt the lost functionality to fulfill the original function. The search space is pruned by the cost of the best solution during the exploration. This process obtains high quality results. However, a high runtime is required on large functions.

In [171], this drawback is solved with the introduction of BDD decomposition and enabling sharing between equivalent functions. However, the results are slightly worse than the previous approach. Instead of using an exact branch-and-bound approach to seek the optimal decomposition, a good heuristic decomposition based on the BDD representation of the functions is obtained. Initially, a pre-process is performed to search a good order in the BDD variables since the evaluation of the solutions depends on it. Then, the recursive decomposition starts exploring algebraic and Boolean AND/OR/XNOR decompositions for each function.

The approach presented in [120] improves considerably the previous method. A BDD decomposition approach using EXOR-gates is performed allowing sharing and overlapping between the support of the children. Basically, this method is concerned about creating better balanced decompositions. The authors claim that their approach finds better balanced decomposition since *strong* decompositions are mostly selected. The concept of strong and weak decomposition is presented in [28]. Fig. 5.3 shows the difference. Let us assume the following decomposition $F(X) = F_1(X_1) * F_2(X_2)$, where $X_1 \subseteq X$ and $X_2 \subseteq X$. The next condition is never fulfilled in a strong decomposition: $X_1 \cup X_2 = X_1$ or $X_1 \cup X_2 = X_2$.

The technique proposed in [56] combines the best characteristic of each previous approach. Algebraic, function approximation and BDD decomposition with AND/OR/XOR gates are explored. Sharing is also applied to reduce the total area of the function and the runtime of the algorithm. Moreover, a tree-height reduction technique is applied after the decomposition to balance the correspondent functions of the children. Tree-height reduction [97] was originally proposed in the scope of compilers for the generation of optimized code for multiprocessor systems. Basically, a decomposition can be iteratively improved by applying simple transformations (associative and distributive). Figure. 5.4, extracted from [56], shows an example. Initially, the function is bi-decomposed. The result is far from a balanced solution. However, the total height is reduced from 8 to 4 levels by applying twice the distributive law. After that, the bi-decomposition continues on the children.

5.3 Overview

Let us assume the next Boolean function:

$$F = abcdfg + ab(\overline{c} + \overline{f} + \overline{g})e$$

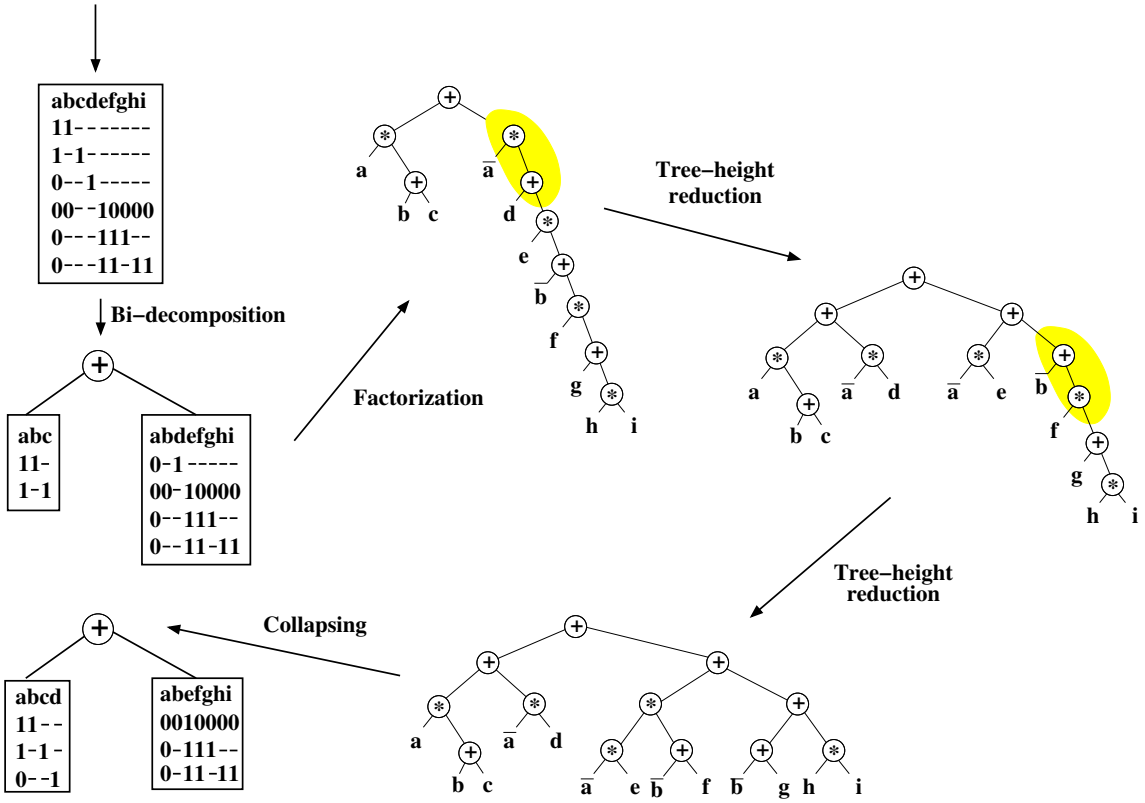


Figure 5.4: Logic decomposition proposed in [56].

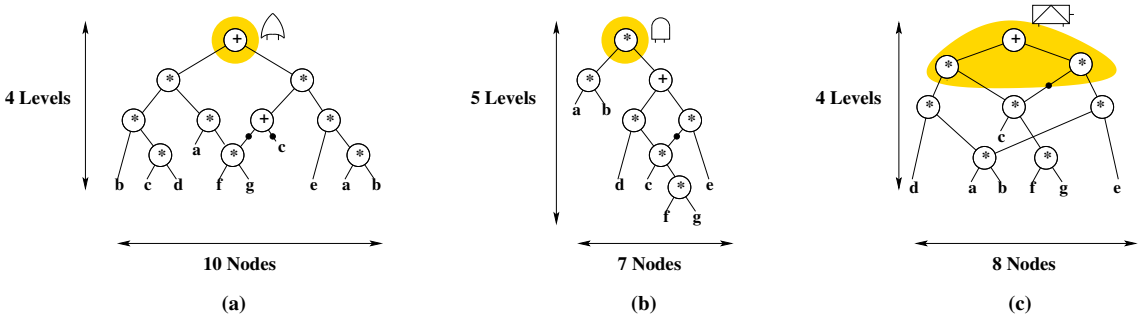


Figure 5.5: Bi-decomposition versus n-way decomposition.

Figure 5.5 illustrates three possible decompositions. The DAG representation has been used where the bubbles in the edges represent inverters. The shadowed nodes represent the functions used to decompose[2]. Figure 5.5(a-b) illustrates a decomposition based on OR and AND function respectively, and a MUX function is used in Fig. 5.5-(c). The AND/OR functions are able to find good solutions. The OR function requires one level of logic more, however, a solution with less number of nodes is produced. The MUX-decomposition finds an intermediate solution between them. A solution with the same number of levels than the AND decomposition and similar number of nodes than the OR decomposition is obtained. Therefore, n-way decomposition can produce better solutions since a large search space of decompositions is explored.

Bi-decomposition methods can also achieve the decomposition obtained by the multiplexor function using tree-height reduction post-processing approaches like speed_up [147] or acd_speed [56]. However, these post-processing methods are highly dependant on the original decomposition.

In this chapter, we propose an n-way decomposition approach to improve the bi-decomposition method presented in [56]. N-way decomposition is also merged with tree-height reduction techniques to improve the results.

5.4 Background

The n-way decomposition problem can be formulated as follows:

Definition 5.4.1 N-way decomposition. *Let us assume a function $F(X)$ with the set of variables $X = \{x_1, x_2, \cdots, x_m\}$ and a gate $G(Y)$ with the set $Y = \{y_1, y_2, \cdots, y_n\}$. The* n-way decomposition *of the function $F(X)$ with $G(Y)$ is $F(X) = G(F_1(X), F_2(X), \ldots, F_n(X))$.* □

Bi-decomposition is a particular case of n-way decomposition where $G(Y)$ is a two-input function. Hereafter, we will refer to bi-decomposition and n-way decomposition as BIDEC and NDEC respectively. The n-way decomposition problem can be naturally represented by a Boolean relation. A Boolean relation covers the potential set of decompositions.

Definition 5.4.2 Decomposition problem formulation. *Let us assume a function $F(X)$ with the set of variables $X = \{x_1, x_2, \cdots, x_m\}$ and a gate $G(Y)$ with the set $Y = \{y_1, y_2, \cdots, y_n\}$. The Boolean relation that represents all possible decompositions of the function $F(X)$ with the gate $G(Y)$ is defined as follows:*

$$BR(X,Y) = (F(X) \Leftrightarrow G(Y)) + DC(X)$$

where $DC(X)$ are the external don't cares of the function $F(X)$.

[2]We assume that the multiplexor function is internally represented by two levels of logic.

□

The function being decomposed usually belongs to a network where flexibility described with don't cares can be taken into account. The Boolean relation uses the don't care information to constraint the search to decompositions not covered in other regions of the network.

Finally, let us define the data structure to represent a decomposition.

Definition 5.4.3 Decomposition representation. *A bi-decomposition D can be represented as a triples*

$$D \equiv (op, left, right)$$

where op is the Boolean function used in the decomposition, and left and right are the resulting children. A decomposition can be represented as a node in a binary tree, in which the Boolean function op is the Boolean operator of the node that can be $*$, $+$ or input if the node is a literal. Recursively, left and right can be defined as other triples until the whole tree is constructed. □

A recursive decomposition can be represented by a binary tree where each node corresponds to a logic function 2-AND/OR. Note that, an n-way decomposition can be also represented as a binary tree if the gate is decomposed in two-input nodes (See the MUX function in Fig. 5.5-(c)).

A binary tree represented by these triples can be transformed to an And-Inverter Graph (AIG) by merging nodes that represent isomorphic functions.

Definition 5.4.4 And-Inverter Graph (AIG) [98]. *An And-Inverter graph is a directed acyclic graph, in which a node has either 0 or 2 incoming edges. A node with no incoming is a primary input. A node with 2 incoming edges is a two-input AND gate. An edge is either complemented or not. A complemented edge indicates the inversion of the signal.* □

An example is shown in Fig. 5.5-(c). The MUX-decomposition generates the functions *abe*, *abf* and *cd*. The recursive decomposition of the functions *abe* and *abf* generates the common factor *ab* that is shared in both functions.

5.5 Recursive n-way decomposition

In this section the recursive n-way decomposition is presented. Figure 5.6 illustrates the algorithm called BRCDEC(*Boolean Relation Combinational DEComposition*). The input of the procedure is the targeted function to be decomposed and the don't care information captured from the environment. A library with the set of logic functions that the procedure will explore to select the best decomposition is also provided. The last input specifies the desired required time that the result should have. This required time is measured in number of levels and it is decreased each time a new recursive call is invoked.

```
BRCDEC (F, DC, library, Req_time)
{Input: Function F to be decomposed, External don't cares DC,
        List of logic functions used in the decomposition, Required time.}
{Output: Function decomposed in 2-AND/OR representation}

   Collapse(F);

   {Obtain bi-decomposition}
   sol_tmp:=Bi-decomposition(F, DC, Req_time);
   List_sol:= List_sol∪ Tree-Height_Reduction(sol_tmp,Req_time);

   {Obtain decompositions using the logic functions of library}
   for each function G in library_gates do
      sol_tmp:=Decomposition_BR(F, DC, G, Req_time);
      List_sol:= List_sol∪ Tree-Height_Reduction(sol_tmp,Req_time);

   {All decompositions are in 2-AND/OR representation}
   sol:=Get_Best_Decomposition(List_sol);

   if Levels(sol.left)>Levels(sol.right) then swap(sol.left,sol.right);

   {First descompose fastest child}
   dec_left:=BRCDEC(sol.left,DC, library_gates,Req_time − 1);

   {Add observability don't cares}
   dec_rigth:=BRCDEC(sol.rigth,DC + ODC_left_child, library_gates,Req_time − 1);

   return Create_decomposition(sol.op,dec_left,dec_rigth);
end;
```

Figure 5.6: Recursive algorithm for logic n-way decomposition.

Initially, the function is collapsed into a single node. For each logic function in the library a decomposition is obtained. The decomposition problem is formulated as a Boolean relation as we defined in Def. 5.4.2 and the relation is solved with a BR solver[3]. Note that, a Boolean relation covers a huge space of decompositions and the calculation of the best solution may require a high runtime. In order to cut this runtime, a small set of decompositions is explored (e.g. 200 decompositions) and tree-height reduction is used to balance and, therefore, improve the decomposition. After this post-process the decomposition is represented by an AIG.

Therefore, our approach improves the technique used in [56] by providing n-way decomposi-

[3]The configuration of the solver will be described in the following section.

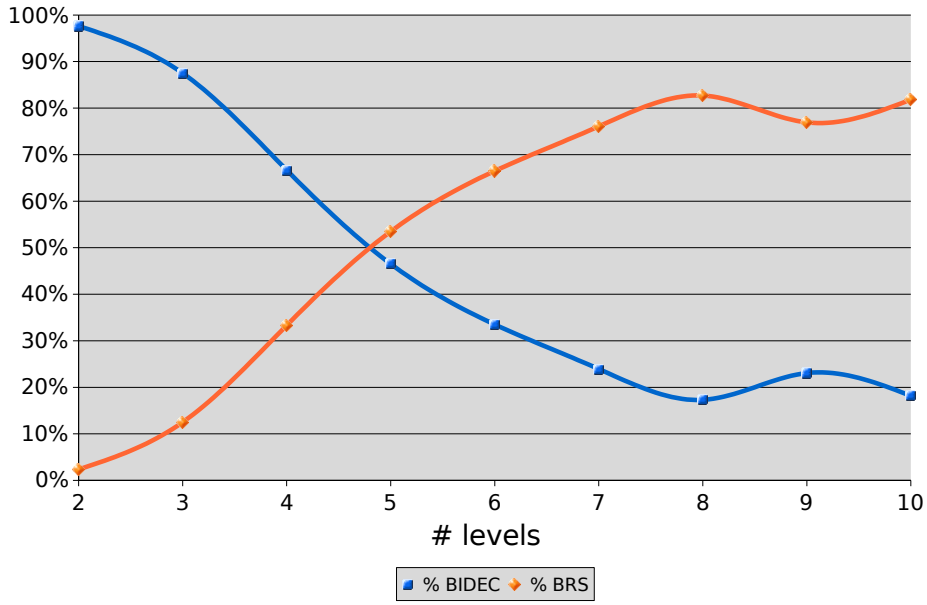


Figure 5.7: Selected decomposition by number of levels of logic of the decomposition.

tions to the tree-height reduction algorithm. As we defined previously, BIDEC is a particular case of NDEC. In addition to the decompositions of the logic functions of the library, the procedure also explores the bi-decompositions performed in [56]. This includes algebraic and BDD decomposition based on function approximation. Experimentally, we observed there is a relationship between the selected decomposition (BIDEC or NDEC) and the number of levels of logic of the function. When the function has few levels, there are few feasible decompositions, therefore, a near-optimal balanced solution is easily obtained using BIDEC. NDEC obtains better balanced decompositions when the complexity of the function grows. The accuracy of bi-decomposition is affected by the complexity of the function. Figure 5.7 shows a plot between the number of levels of the logic function and the selected decomposition. This plot is obtained by decomposing around 500 functions of several levels. Functions with larger number of levels are mostly decomposed with the n-way method since there is a larger search space of decompositions.

Next, the algorithm selects the best decomposition. The selection is performed evaluating their AIG representation using the cost function defined in the following section.

Finally, the algorithm recursively decompose the children. First, the smaller child is selected. The reason for this choice is due to the possibility to apply the observability don't cares from the

smaller child to posteriorly decompose the other one. Don't cares will help to avoid redundancy between the children. When the children are completely decomposed, the decomposition for the current function is constructed.

5.6 Implementation aspects

This section describes the heuristics applied in BRCDEC. Moreover, the configuration of the solver and the cost functions are also defined.

5.6.1 BREL solver

BREL solver is used to solve the n-way decomposition. The solver has been customized to support two cost functions The first function filters the huge space of solutions using the BDD representation of the decompositions and a small set of the best solutions is stored. The number of stored solutions can be customized by the user. The latter one selects the final decomposition using the AIG representation over the best solutions found by the former cost function. The solver has been limited to perform a partial exploration for each decomposition problem due to the large space of potential solutions.

5.6.2 BREL cost functions

The former cost function deals with Boolean functions represented with BDDs. Cost functions based on BDD representation are not accurate, since sometimes there is no correspondence between the complexity of the BDD and the function. Moreover, there is a high dependence on the variable ordering of the BDD manager. However, a BDD-based cost function provides a fast estimation.

Some cost functions, like number of levels of a function, are difficult to be computed in BDD representation. Although a BDD can be transformed to its AIG represenation to obtain an accurate estimation, it is not recommended since the execution can be slow down considerably.

A naive BDD-based cost function, presented in Section XX, to estimate the balance of a decomposition is the sum of squares of the BDD size. This approximation gives an intuition of the size of the function and, therefore, a perception of the quality of the balance of the decomposition.

Consider the bi-decomposition of the function $F = abc + \overline{f}gh + f\overline{g}h + fg\overline{h}$ with the OR logic operation with the same arrival time for all the inputs. Two possible decompositions are:

$$\begin{array}{lll} d_1 = (+, d_{11}, d_{12}) & d_{11} = abc, & d_{12} = \overline{f}gh + f\overline{g}h + fg\overline{h} \\ d_2 = (+, d_{21}, d_{22}) & d_{21} = abc + \overline{f}gh, & d_{22} = f\overline{g}h + fg\overline{h} \end{array}$$

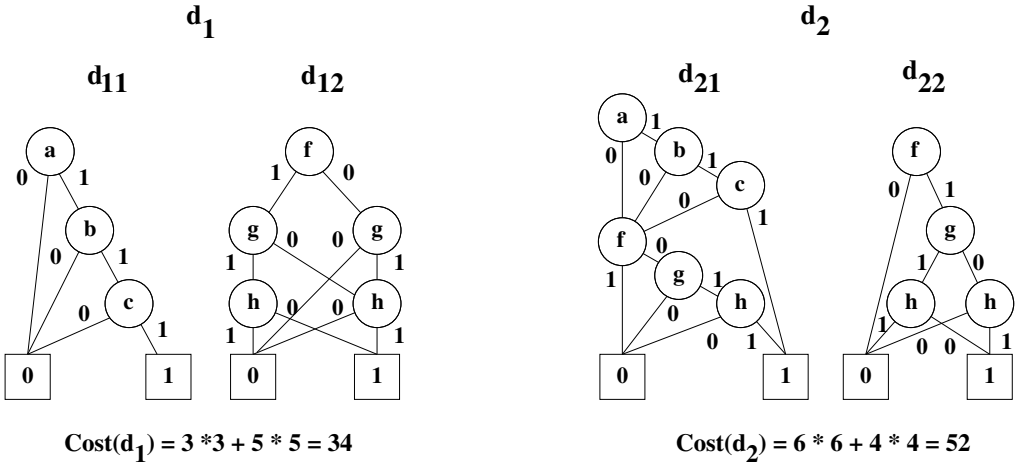


Figure 5.8: Sum of the squares of the BDD sizes cost function.

Figure 5.8 shows the BDD representation of these two decompositions. The BDD cost function "sum of squares" selects the first decomposition since the cost of d_1 is 34 compared to the cost of 52 of d_2.

The drawback of this cost function is that the arrival times of the inputs are not taken into account. Note that, the function being decomposed usually belongs to a larger network. The input variables also have timing information that has to be taken into consideration during the decomposition to obtain a better balanced solution.

The next BDD-based cost function is proposed to obtain a better estimation of the delay and the area. A balanced binary tree can be constructed from the disjunction of all prime implicants of the function where each prime implicant is a conjunction of input variables. The arrival time of the inputs is used to build the tree towards a balanced delay. Note that, this construction can be done by BDD operations without explicitly building the binary tree. The delay and area can be evaluated during the computation of the prime implicants. The delay is estimated by the depth of the tree and the area from the sum of the support of the primes.

Consider the same decompositions of Fig. 5.8. Figure 5.9 depicts both decompositions in binary tree representation where the cost function based on prime implicants is used. The selection of the best solution changes since the costs are $(d = 4, a = 12)$ and $(d = 3, a = 12)$ for d_1 and d_2 respectively.

The BDD-based cost function obtains a set of candidates. The latter cost function deals with this set of candidates and amends the inaccuracy using a cost function based on AIG representation. The complexity will be higher but the cost is affordable since a small set of the solutions will be processed. The cost is also defined as a pair $(delay, area)$. The delay and area cost functions can be

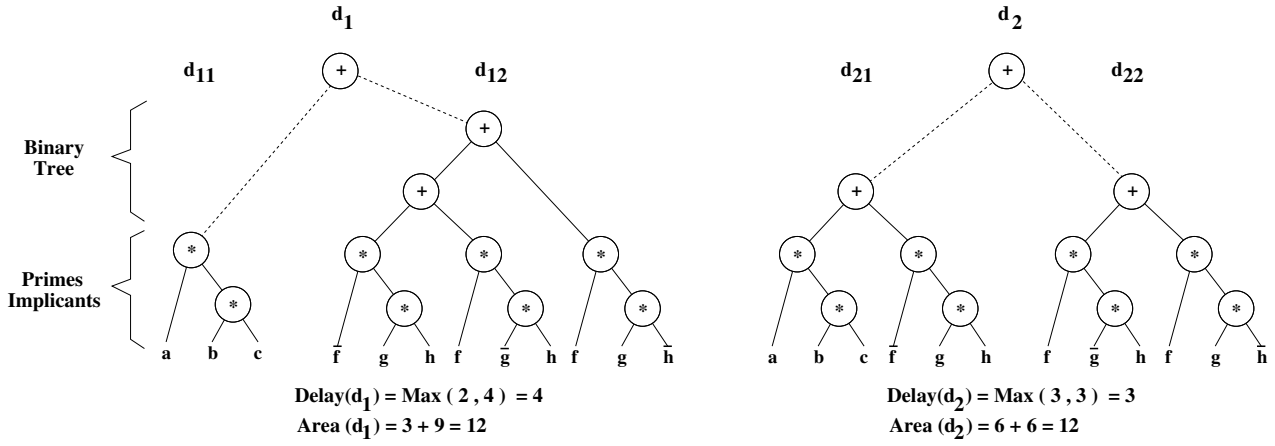


Figure 5.9: Prime implicants cost function.

computed based on the next formulas:

The delay $D(T)$ can be calculated by the depth of a tree T as follows:

$$D(T) = \begin{cases} AT(T) & \text{if } T.op = input \\ 1 + max(D(T.left), D(T.right)) & \text{otherwise} \end{cases}$$

where $AT(T)$ is the arrival time when the node is a primary input.

The number of nodes $N(T)$ of a tree T can be calculated as follows

$$N(T) = \begin{cases} 1 & \text{if } T.op = input \\ N(T.left) + N(T.right) & \text{otherwise} \end{cases}$$

These cost functions are more accurate since they work on top of the AIG representation of the decomposition. As we described, AIG representation allows to merge isomorphic functions that contributes to obtain a better estimation of the area of the function.

In the algorithm BRCDEC, the same cost function based on AIG representation is used to select the best decomposition over all logic functions of the library in Get_Best_Decomposition.

5.6.3 Look-up table

To prevent the decomposition of equivalent Boolean functions, a hash table stores all the previously decomposed Boolean functions. Note that, this look-up table reduces significantly the computation time of the decomposition process.

5.7 Experimental results

Two experiments have been performed to show the efficiency of the n-way decomposition approach presented on this chapter:

- Comparison with the bi-decomposition proposed in [56] on small and medium-sized examples.
- Comparison with bi-decomposition on large examples using a window-based approach presented in Chapter XX.

Our n-way decomposition approach BRCDEC has been implemented in SIS using the BREL solver. A library of multiple-input functions has been provided to BRCDEC to compute several decompositions. Specifically, the library consists in four functions: AO22, OA22, MUX, and XOR. Note that, no post-process is run after decomposition, like speed_up [147] or acd_speed [56]. The main objective of these experiments is to compare the performance of the decomposition processes. Any post-process may alter considerably the decomposition results.

The next sections summarize the results of these experiments. The tables of results report for each example the number of primary inputs and primary outputs, the number of levels of logic in the technology independent phase, the delay and area after technology mapping and the runtime. The circuits have been mapped using the tree-mapping map [133, 162] with the academic library lib2.genlib. The last row of the tables reports the normalized sum of the columns.

5.7.1 Comparison with bi-decomposition

In this section, a comparison with the bi-decomposition approach presented in [56] is performed. The experiment is run on a subset of small and medium-sized circuits of the MCNC benchmarks. The objective of this experiment is to show the performance of BRCDEC with regard to bi-decomposition.

Table 5.1 summarizes the results. The netlists on this table are a subset of the selected ones in [56]. The smallest circuits are not reported since the decomposition obtained with NDEC is identical to the solution provided by BIDEC.

The number of levels of logic is similar in both methods. NDEC reduces the depth of the circuits by 3%. However, the number of levels is actually reduced only on few examples (alu2, apex6 and b9). The improvement is closely related to the size of the network. For instance, the circuit alu2 the number of levels is reduced from 11 to 9. However, NDEC contributes to obtain better delay after technology mapping with similar results on area. In some examples the results are substantially better (alu2, apex6, apex6 and example). Only, the results are slightly worst in four examples (c8, i7, term1 and x2), since some bad decision is taken based on the cost functions. Nevertheless,

			bi-decomposition [56]				n-way decomposition			
	PI	PO	LEV	DEL	AREA	CPU	LEV	DEL	AREA	CPU
9symml	9	1	9	7.17	117392	6	9	6.73	131776	196
alu2	10	6	11	8.91	509008	13	9	8.15	475136	579
apex6	135	99	7	7.39	1529808	11	7	6.92	1517280	788
apex7	49	37	8	7.69	881136	7	7	7.57	919648	511
b9	41	21	6	5.29	212976	1	5	5.14	218080	64
c8	28	18	6	5.50	212048	1	6	5.55	207408	57
cht	47	36	4	4.94	272368	1	4	4.71	270048	29
count	35	16	7	6.46	497408	3	7	6.34	517360	142
cu	14	11	5	4.70	95120	1	5	4.73	100688	14
example2	85	66	6	8.54	1331680	7	6	7.49	1288528	379
f51m	8	8	7	6.43	219008	2	7	6.14	210192	85
frg1	28	3	7	5.40	72384	3	7	5.41	81200	121
i5	133	66	6	6.28	1013376	6	6	6.23	990176	363
i7	199	67	5	6.83	833808	4	5	7.10	942384	251
lal	26	19	6	5.30	220864	1	5	5.19	230144	59
pcle	19	9	6	5.48	139664	1	6	5.36	150336	43
pcler8	27	17	5	5.77	229216	1	5	5.73	237104	69
sct	19	15	5	4.90	167040	1	5	4.77	161008	43
term1	34	10	9	6.89	361456	4	9	6.93	353568	288
ttt2	24	21	6	6.40	362848	2	6	6.13	388368	139
x1	51	35	6	5.76	487200	4	6	5.62	498336	279
x2	10	7	5	4.46	62640	1	5	4.57	62176	16
x3	135	99	7	6.92	1543728	11	7	6.82	1525632	16
x4	94	71	5	6.74	1046784	6	5	6.63	1056064	424
z4ml	7	4	5	4.98	82128	1	5	4.90	84912	27
Norm.			1.00	1.00	1.00	1.00	0.97	0.98	1.00	50.6

Table 5.1: Comparison with the bi-decomposition method presented in [56] on small networks.

there is an improvement of 2% on average, since NDEC generates some decompositions that BIDEC is unable to find. These decompositions after tree-height reduction deliver AIGs with better delay and area characteristics. However, the cost on runtime to apply NDEC using Boolean relations is considerably higher with comparison to BIDEC.

These statements are confirmed in Table 5.2. This table reports the results on the largest MCNC netlists that can be run with BRCDEC without not incurring on large penalties on runtime. On larger ones, BRCDEC blows up due to the construction of too large BDDs. Here, the improvement on delay is more significant (5%) with a slightly reduction on the area. However, NDEC uses more CPU time than BIDEC. Performing an analysis on individual functions, we have observed that NDEC is able to find decompositions with similar number of levels but with less area on large functions (i8, i9 and vda). On a recursive decomposition, the solution provided by NDEC is commonly selected on

			bi-decomposition [56]				n-way decomposition			
	PI	PO	LEV	DEL	AREA	CPU	LEV	DEL	AREA	CPU
frg2	143	139	8	9.38	4139808	47	8	8.99	4360672	4284
i8	133	81	9	10.37	4084128	64	8	9.35	3849344	3691
i9	88	63	8	10.75	4447904	54	8	10.18	4382016	4146
table3	14	14	10	10.67	3216912	121	10	10.05	3258208	3580
vda	17	39	8	8.70	2219776	24	8	9.01	2093568	1307
Norm.			1.00	1.00	1.00	1.00	0.98	0.95	0.99	56.77

Table 5.2: Comparison with the bi-decomposition method presented in [56] on medium-sized networks.

the initial steps where larger functions are tackled. On inferior decompositions where the functions have a lower complexity, NDEC and BIDEC find similar results.

5.7.2 Window-based n-way decomposition

N-way decomposition is a powerful method since a large search space is explored using the Boolean relations. However, this exploration incurs on large runtimes that are more stressed on large functions. Therefore, NDEC is confined to be run on medium networks. In order to break this limitation, DBP is used. A window-based approach is applied to run the decomposition methods on the largest circuits of the ISCAS'99 benchmarks. DBP targets the decomposition methods on the critical regions of the circuits. The size of the windows for DBP is reduced to $S_d = 8$ for delay optimization and $S_a = 100$ for area optimization with a *Slack* of 2 units. We observed a reasonable trade-off between performance and runtime in windows of 8 nodes. Larger windows slow down substantially the execution of the window-based approach.

In this experiment, a pre-processing script is used to reduce the size of the netlist. algebraic script and speed_up -i are used to obtain a 2-input gates network. The area optimization script is the algebraic and the delay script consists on the decomposition method (BIDEC or NDEC) and acd_speed. The decomposition methods increase considerably the area of the windows. This drawback, that can be a problem on the largest netlist, is controlled by applying tree-height reduction on the window (acd_speed).

The results are summarized in the Table 5.3. Here, the results of speed_up with DBP reported in Section XX are also shown to see the comparison with the decomposition methods. The normalized sum takes as a reference the BIDEC method. The number of levels of logic is similar between speed_up and BIDEC. There is only 2% of difference on average. However, NDEC overcomes by 4% BIDEC. In some netlists, like b17, the improvement is notable. After technology mapping, this improvement is also observed. BIDEC by systematically decomposing windows on the critical regions reduces the delay by 7% at the expenses of increasing by 17% the area compared to

			speed_up			bi-decomposition [56]				n-way decomposition			
	PI	PO	L	DEL	AREA	L	DEL	AREA	CPU	L	DEL	AREA	CPU
b14	32	54	32	36.7	9809	32	33.0	12829	742	30	32.0	12151	10021
b14_1	32	54	30	33.6	8706	30	31.1	11075	521	28	30.5	14241	16086
b15	36	70	49	53.4	12814	46	50.4	18124	2026	44	49.8	15846	8697
b15_1	36	70	30	35.1	12127	29	31.8	14547	692	27	31.1	15706	10839
b17	37	97	53	58.0	38669	54	54.8	42118	3751	47	52.5	43730	32775
b17_1	37	97	32	35.2	36630	30	34.9	44169	13089	30	33.8	38150	23528
b20	32	22	36	40.1	22072	36	37.9	22650	1942	36	36.9	22727	12741
b20_1	32	22	36	40.5	17512	36	36.5	23312	2921	33	36.2	23510	12855
b21	32	22	36	40.6	20988	34	35.2	30308	4554	32	35.7	29926	22444
b21_1	32	22	35	40.2	19531	34	36.1	28010	3040	32	35.4	26168	13553
b22	32	22	38	42.0	29228	35	38.9	34914	12485	35	37.7	31555	15141
b22_1	32	22	36	40.1	27111	37	37.2	32363	4005	37	37.1	30947	24979
s35932	35	320	9	13.8	12787	8	13.0	12941	665	10	12.8	13235	5764
s38417	28	106	23	24.5	15462	22	25.4	16670	1151	21	24.6	16744	8580
s38584	38	304	20	21.5	12810	19	21.2	12677	468	21	20.7	12626	1699
Norm.			1.02	1.07	0.83	1.00	1.00	1.00	1.00	0.96	0.98	0.97	4.22

Table 5.3: Comparison between window-based bi-decomposition and window-based n-way decomposition.

speed_up. As in the previous section, NDEC slightly improves the results by 2% in delay and 3% in area by selecting better decompositions.

5.8 Conclusions

In this chapter, a new application of the Boolean relations has been shown. The experimental results confirm that the n-way decomposition can obtain better solutions than bi-decomposition, mostly, on large functions. However, there is a limitation on the size of the decomposed functions and a high runtime is required to run n-way decomposition.

A window-based approach has been proposed to apply this method on larger netlists. Although the runtime is enormous, the experimental results show some improvements in the largest ones. Note that, the runtime basically depends on the size of the windows. There is a trade-off between the quality of the solutions, the size of the windows and the runtime of the window-based approach.

Chapter 6

Layout-Aware Gate Duplication and Buffer Insertion

6.1 Introduction

As miniaturization evolves down to deep-submicron technologies, the impact of layout details acquire increasing relevance, since interconnect delays become dominant.

The work presented in this chapter, based on the results in [20], combines three different sub-problems in the same framework in such a way that the loss of information between logic and layout synthesis is reduced. The combination is performed by iteratively providing feedback from layout to logic synthesis and vice-versa. The three related sub-problems are: gate duplication, buffer insertion and placement.

The reason for the selection of these problems is because they are closely related, since they are at the boundaries between logic and layout synthesis. Second, they can be combined with an affordable computational complexity. Incorporating more sub-problems, e.g. technology mapping or routing, would prohibitively increase the complexity. Note that, gate sizing is also performed by our approach when considering different instances of the gates during duplication and buffer insertion. Moreover, gate sizing over all the circuit can be performed after placement with some existing tool to change some of the non-modified gates closed to the new inserted ones.

The method presented in this chapter, BufDup, is an approach that applies a technique similar to Engineering Change Orders (ECOs). The circuit is incrementally improved by performing small modifications on top of the current placement design. BufDup implements a reciprocal feedback between placement and gate duplication and buffer insertion.

As we previously pointed out, the combination of these methods could suffer a combinatorial explosion. There is a large potential set of possible gate duplication and buffer trees that can im-

plement a net with high fanout. To avoid an exponential search of candidates, the fanout points of each net are ordered according to the layout information. The trees are explored/generated using a dynamic programming approach that creates subtrees of adjacent points according to the calculated order. In this way, the set of gate/buffer trees are explored in a similar way as tree-based technology mapping algorithms are executed.

Another important feature of the presented approach is that there are no pre-defined insertion points for the new gates and buffers. In principle, there is total freedom to create any tree. The new gates and buffers are placed on top of the existing layout. Incremental detailed placement is used to legalize the new layout.

The experimental results show tangible benefits in delay that endorse the suitability of integrating the three sub-problems in the same framework.

The remainder of the chapter is organized as follows. In Sect. 6.2, the previous work in buffer insertion and duplication is presented. In Sect. 6.3, the contributions of the presented approach are introduced. Section 6.4 describes the interconnect optimization algorithm BufDup. The gate duplication and buffer insertion algorithms are introduced in Sect. 6.5 and Sect. 6.6 respectively. Finally, the experimental results are presented in Sect. 6.7.

6.2 Previous work

In this section, first the Elmore delay model is described. Later, we will review several approaches on buffer insertion and duplication. Due to the extensive literature on these topics, here we will survey the most relevant techniques.

6.2.1 Elmore delay model

In this section, we define the interconnect model used in BufDup. There are several ways to compute the delay of a circuit [24, 48, 135]. Instead of using continuous models, many methods tend to use discrete delay models since there are simpler to compute.

Here, the basic Elmore model [68] is used to estimate the delay of a RC circuit. In adddition to the intrinsic delay of the gates, the Elmore model takes into account the delay associated to the wires. First, the technological parameters (wire capacitance and wire resistance per unit length) are defined depending on the selected process technology. The associated delay of a wire is estimated from its length and these technological parameters. Figure 6.1-(a) depicts a network with its combinational representation and Fig. 6.1-(b) corresponds to the same network as the discrete RC circuit where the cells and wires have been replaced by their corresponding capacitance and resistance. The delay

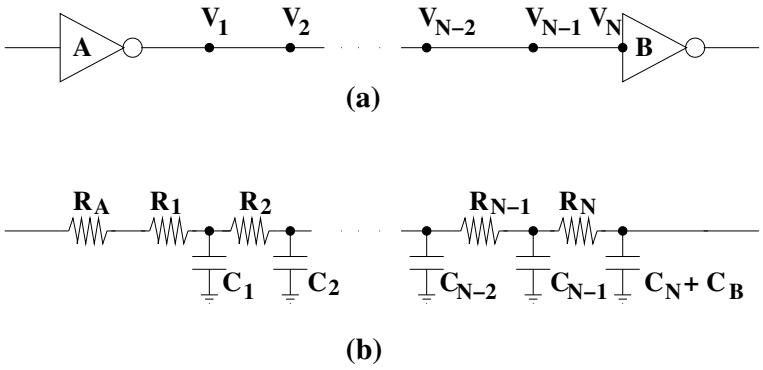


Figure 6.1: (a) Combinational representation and (b) RC representation of a circuit.

from the cell A to the cell B is defined by the following formula:

$$D_{AB} = \underbrace{R_A(C_B + \sum_{i=1}^{N} C_i)}_{cell\ A} + \underbrace{\sum_{i=1}^{N}(\sum_{j=1}^{i} R_j)C_i}_{wire} + \underbrace{C_B \sum_{j=1}^{N} R_j}_{cell\ B}$$

where the first term is the time to send the signal from the cell A, the second term is the time to send the signal through the wire and the last term is the time to receive the signal on cell B.

6.2.2 Buffer Insertion

Several approaches on buffer insertion [124, 146] have been proposed in the past using a load-based model. This problem, also called in this context *fanout optimization*, aims at reducing the high fanout of several gates. Buffer insertion improves considerably the delay on critical paths. Some approaches have also been integrated with technology mapping [110]. However, the insertion decisions are taken without considering physical information.

Buffer insertion has also been incorporated in the routing step of physical layout. Here, the goal of buffer insertion, also called *repeater insertion*, is to minimize the length and the congestion of the wires among the placed cells. Buffers tend to be inserted on free positions to preserve the legality of the placement. Some of these techniques are based on the dynamic programming approach proposed by Van Ginneken [163], that solves the problem in polynomial time with regard to the number of explored locations. Figure 6.2 illustrates an example of this approach. The buffer insertion is performed on a predefined routing tree where only legal positions for the buffers are explored. Initially, the tree is traversed from sinks to source. For each location, all the pairs $(Required\_Time, Capacitance)$ for the possible buffers are computed. The best pairs, commonly

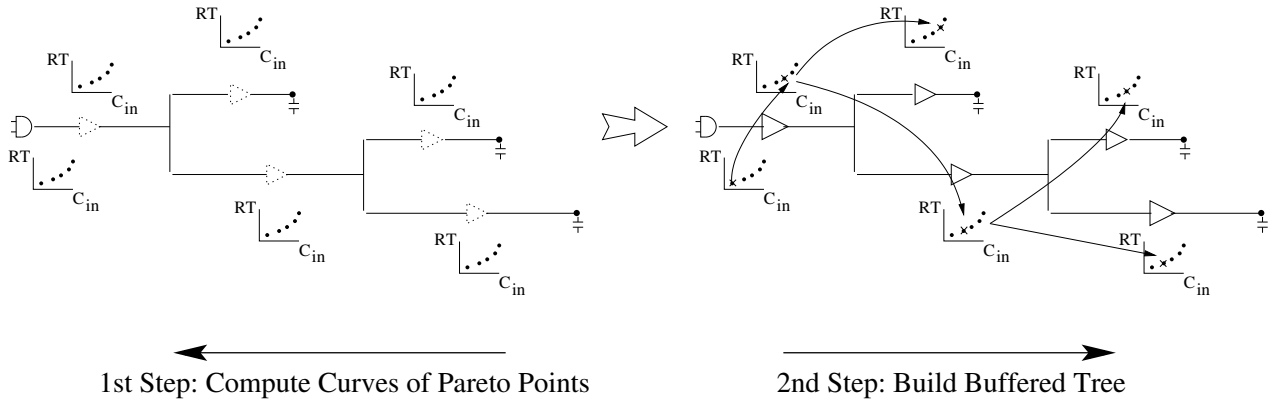


Figure 6.2: Van Ginneken's approach.

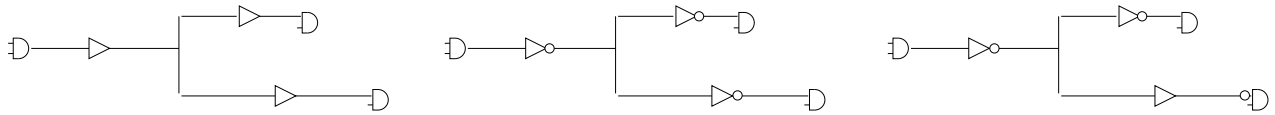


Figure 6.3: (a) Buffer insertion. (b) Inverter insertion. (c) Sink Polarity.

called *Pareto points* [86], are preserved. A post-process builds the buffered tree from source to sinks. This process identifies for each position the Pareto point and, therefore, the buffer that has led to the optimal solution in the source gate.

Several extensions to this algorithm have been proposed to improve the runtime, explore multiple candidates locations for the buffers [13], and generalize the algorithm for other objective functions, such us power consumption [106, 131]. A Fast buffer insertion technique (FBI) was proposed in [145]. It reduces the complexity of the conventional Van Ginneken's approach to $O(nlog^2n)$ with regard to the number of feasible locations. It uses several heuristics, such as predictive pruning and redundancy check, to reduce the number of Pareto points. These heuristics are based on the concept of *dominance*[1]. FBI also supports inverter insertion and sink polarity (See Fig. 6.3). Inverter insertion enables the possibility to handle sinks with negative polarity.

Another technique to achieve a buffered tree is a simultaneous construction of a routing tree and buffer insertion. This approach is more complex, since it has to deal with the routing tree construction. General methods combine buffer insertion with fast heuristics to construct them. Let

[1]A solution characterized by a cost function dominates another solution if the value of the cost is better. Note that, the concept of dominance can be extended to solutions with multiple cost functions.

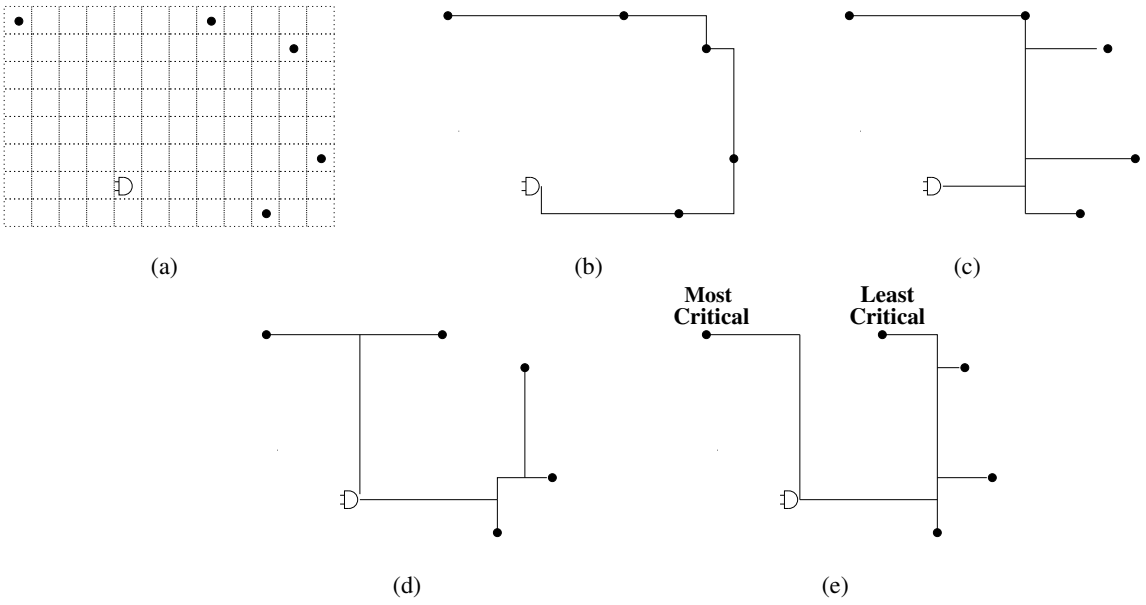


Figure 6.4: (a) Hanan grid. (b) Minimum Rectilinear Spanning Tree. (c) Minimum Rectilinear Steiner Tree. (d) A-Tree. (e) P-Tree.

us recall some of these heuristics. Figure 6.4 illustrates these types of trees.

- **Minimum Rectilinear Spanning Tree (MRST)**: The Rectilinear Spanning Tree (RST) was the first type of trees used in a routing algorithm. The objective of a MRST is to connect a set of vertices with a tree with minimum length. The complexity of the Prim's algorithm [129] is $O(n^2)$ with regard to the number of points. However, a MRST can be computed with $O(nlogn)$ using the approach in [73].

- **Minimum Rectilinear Steiner Tree**: Steiner trees allow to add new vertices in the tree in order to reduce the length of a spanning tree. However, the construction of a Minimum Rectilinear Steiner Tree is a NP-hard problem. Several heuristics have been proposed to construct Steiner Trees. We next describe the most relevant.

- **Batched Iterative 1-Steiner (BI1S)** [87]: This approach starts from a RST. The algorithm inserts one Steiner point based on the maximal reduction of the wirelength of the RST. The cost of this algorithm is $O(n^3)$ since the improvement of the wirelength is iteratively computed for all the points of the Hanan grid[2] with regard to all terminal nodes.

[2]The Hanan grid is defined as the embedded region in the bounding box created by the terminal nodes (See Fig. 6.4.(a)).

Figure 6.5: (a) LT-Tree example. (b) C-Tree example.

- **Borah-Owens-Irvin (BOI)** [29]: This approach reduces the complexity of the BI1S to $O(n^2)$. The process is similar, although only the intersection points between adjacent terminal nodes are candidates to become a Steiner point. Moreover, the cost function is only computed with regard to the nearest terminal nodes.

- **A-Tree** [53]: The previous heuristics only take into account the location of the nodes. Here, the technology dependent parameters, like resistance and capacitance of the gates and wires, are also used to create the Steiner tree. The goal of this heuristic is to create the shortest path from each sink to the source.

- **P-Tree** [108]: The A-Tree is refined including delay-oriented tree construction. Basically, the Steiner tree construction targets at minimizing the delay of the tree. The cost of this approach $O(n^5)$ limits its application on large instances. The complexity arises on the exploration of the Steiner points. Using an approach similar to 1-Steiner heuristic, all the points of the Hanan grid are explored for feasible intersection points. Due to the extremely large number of possible tree constructions depending on how the sinks are connected, an order on the sinks is initially predefined to reduce the complexity.

In buffer insertion techniques, A-Trees and P-Trees are preferred since wire delay is taken into account. The other heuristics have been used in some routing techniques [173]. In [126], the authors combine A-Trees with Van Ginneken's algorithm. The algorithm builds an A-Tree from sinks to source and performs the buffer insertion from source to sinks with several partial solutions stored in the tree. Another approach is presented in [136], where the buffered Steiner tree is constructed from sinks to source with a combination of LT-Trees [162] and P-Tree with a predefined order of the sinks. This algorithm is optimal depending on the order of the sinks but it has a high runtime complexity because of the explosion of the exploration of feasible locations for the buffers and the construction of P-Trees. Moreover, the LT-Tree structure (See Fig. 6.5.(a)) restricts the creation of cascaded buffered trees since a buffer can only drive at most one buffer.

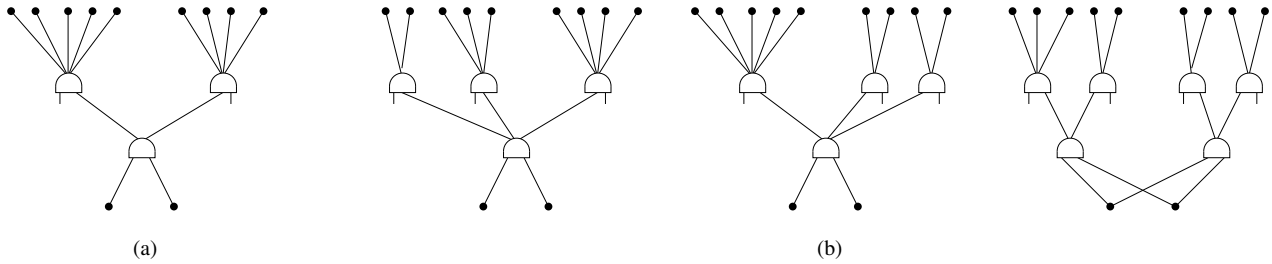


Figure 6.6: (a) Initial configuration. (b) Three different gate duplications.

A variant of the P-tree heuristic is presented in [80]. This approach, called S-Tree, combines the P-tree construction with buffer insertion. Initially, a predefined order of the sinks is given and a partition among the sinks is also provided. This partition divides the sinks depending on some criterion, i.e. polarity or criticality. During the construction from sinks to source, three trees are constructed at the same time. For each location of the Hanan grid three possibilities are analyzed: to merge two branches of the same clusters of sinks or to merge two branches from different clusters. The authors claim that the runtime of the algorithm is reduced with regard to the P-Tree construction. However, this is due to the support of obstacles and, therefore, the reduction of the search space. In [12], a hierarchical technique, called C-Tree, is presented. This approach is a generalization of the previous work where an n-way partition is performed depending on the polarity of the sinks. This technique produces Steiner trees with less number of buffers. Moreover, the runtime is reduced because the application of Dijkstra's algorithm to create the Steiner trees instead of any delay-oriented construction. Figure 6.5.(b) depicts an example of a C-Tree where each sink is depicted with its polarity. The wirelength of the Steiner Tree is larger compared with other types of buffered trees. However, only one inverter is required. Any other Steiner tree construction technique would use four inverters.

In [157], the search space of possible locations is reduced taking into account the obstacles of the layout. Although wire distances between locations are precomputed, this approach still has a high complexity due to the exploration. There were some refinements of this approach to improve the runtime. In [158], some of the partial solutions are pruned based on the concept of *dominance* between solutions. In [65], a similar technique based on precomputing some information is presented. However, a simulated annealing technique is used instead of a dynamic programming approach from sinks to source. This approach starts from a Steiner tree and iteratively performs perturbations on the tree using simulating annealing.

6.2.3 Gate duplication

Gate duplication has been extensively studied using a load-based model [41, 107]. The main work was performed by Srivastava [151, 153]. He proved that the global gate duplication problem is NP-Complete. Basically, the gate duplication algorithm has to take at least two decisions on which the final result will depend: decide the gates to be duplicated and the assignment of fanouts to each duplicated gate. Fig. 6.6 illustrates an example. There are several ways to perform the duplication. However, the local decisions influence future duplications in the immediate fanins and fanouts.

This section will refer to techniques oriented to physical design. The basis of layout-aware gate duplication was introduced in [27]. The nodes of the critical path are ordered depending on the criticality. The objective of this method is to duplicate gates where some delay improvement can be obtained. The new gates are placed on locations where the monotonicity of the critical path is fulfilled. Note that, this *ideal* location may overlap with existing gates. An incremental placement legalizes the placement. The drawback of this approach appears when the feasible region to place the duplicated gate is computed. This region is calculated based on the positions of the immediate fanins and fanouts of the targeted gate. Therefore, the monotonicity is only improved locally. An extension was presented in [43, 44]. An incremental timing-driven placement with duplication is also proposed. Here, the concept of feasible and super-feasible region to place the duplicated nodes is introduced. The objective is also to produce monotonic critical paths. However, a super-feasible region specifies the boundaries of a good region to place a node to improve the global monotonicity of the path.

Another extension of [27] was proposed in [81, 96]. Because of the good results of the previous approach, a more aggressive duplication is performed based on an *arborescence tree embedding*. Instead of duplicating particular gates, all the nodes in the critical path are replicated and placed in legal positions. Later on, a cell unification operation is done to save area on the global circuit.

6.3 Overview

Placement tools tend to place a cell with large fanout on a centered position among its immediate fanins and fanouts. This position is due to the minimization of the wirelength pursued by the placement tools. Even if the placement is timing-driven oriented, long wires may be implemented on the critical path because the low freedom on critical gates with large fanout.

The focus of BufDup is the optimization of the interconnection delays taking physical information into account. Buffer insertion and gate duplication are complementary techniques aiming at this goal. An example is shown in Fig. 6.7(a,b), for a net that connects the source cell S with the fanout cells $\{f_1, \cdots, f_7\}$. Individually, each technique contributes to improve the delay of the net, however the combination of both (Fig. 6.7(c)) can lead to superior results. The improvement

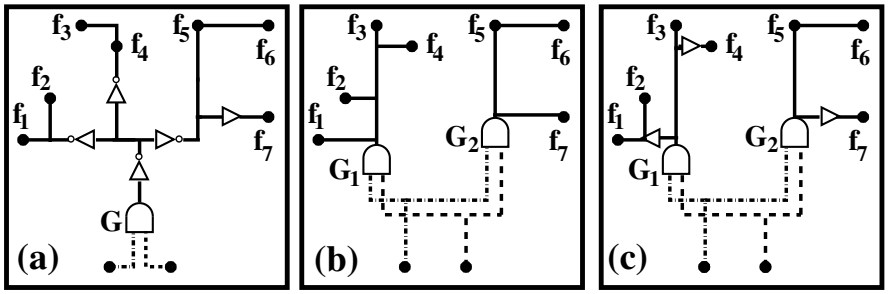


Figure 6.7: Example: (a) buffer insertion, (b) duplication, (c) combination of both techniques.

can still be more tangible if physical information is considered and, reciprocally, the changes produced by buffer insertion and gate duplication have a positive impact by incrementally changing the physical layout of the involved cells.

The main contributions of this method are the following:

- An interconnect optimization approach that combines the exploration of multiple Steiner trees for each net with the incremental placement of the intermediate solutions. In this way, the generated buffers are not restricted to be placed to free spaces. Note that, a legal placement is delivered after the incremental placement.

- The approach integrates gate duplication, buffer insertion and placement in the same framework.

- A gate duplication technique based on a modified layout-aware k-means algorithm for clustering [112].

- A dynamic-programming approach to incrementally build Steiner trees for buffer insertion. It is based on the approach proposed in [136], with several improvements aiming at (1) the construction of cascaded buffered trees, (2) the smart exploration of feasible locations for the buffers and, (3) the support of gate sizing, inverter insertion and polarity optimization.

To prove the effectiveness of our method, three experiments have been conducted on Section 6.7. First, a comparison of our buffer insertion algorithm with FBI [145], a public domain Van Ginneken's approach. Second, results on academic benchmarks are presented. Finally, results on future semiconductor process technologies corroborate the increasing relevance of the interconnect optimization.

```
BufDup (Net)
{Input: A mapped netlist Net}
{Output: A placed circuit C}
   C := Placement_and_Timing_Analysis (Net);
   do
      Critical_Gates := Calculate_Critical_Gates (C):
      while Critical_Gates ≠ ∅ ∧ cycle time not improved do
         G := Extract_Most_Critical_Gate (Critical_Gates);
         newGates1 := Duplication (G);
         newGates2 := Buffer_Insertion (G);
         newGates3 := Duplication_and_Buffer_Insertion (G);
         NewGates := Select_Best_Solution (newGates1, newGates2, newGates3);
         Insert_Solution_In_Circuit (C, newGates);
         Incremental_Placement_and_Timing_Analysis (C);
         if New_Worst_Slack > Previous_Worst_Slack then
            Undo_Insertion (C, newGates):
         end if
      end while
   while cycle time improved;
   return C;
end;
```

Figure 6.8: Algorithm for interconnect optimization.

6.4 Algorithm for interconnect optimization

We present a top-down description of the main algorithm (BufDup) for interconnect optimization. The algorithm is presented in Fig. 6.8. It receives a mapped netlist as input and produces a placed circuit as output. Initially, cell placement is performed to provide physical information during the interconnect optimization. Delay information is calculated using the Elmore delay model [68]. The timing analysis is performed by considering the physical location of the cells and the Borah-Owens-Irwin (BOI) heuristic for Steiner trees [29]. This heuristic is selected because it has a lower complexity $O(n^2)$ with regard to the 1-Steiner heuristic. Although this heuristic is applied individually for each net and does not take into account congestion, it provides a valuable fast lowerbound estimation of the routing cost.

The outermost loop of the algorithm iterates as long as the critical path is improved. At each iteration, the cells at the critical paths are ordered according to their criticality, calculated as a combination of their slack and their fanout. The worst negative slack is the priority factor for optimization, however cells with similar fanout are prioritized according to their higher fanout.

The innermost loop processes gates iteratively according to their criticality. Three different

solutions are calculated as shown in Fig. 6.7: (a) by inserting buffers, (b) by duplicating the gate and, (c) by duplicating the gate and inserting buffers after the duplication. The details on how duplication and buffer insertion solutions are computed will be described in the following sections. Each solution provides a list of new gates to the circuit and has an estimated delay that affects the critical paths of the circuit. The configuration with the best slack time is selected and *physically* inserted in the circuit. Experimentally, we have observed that duplication is mostly selected for gates with high fanout, whereas buffers contribute to reduce the delay on long wires.

The estimated slack time from the new inserted gates does not guarantee the final selection, since the physical location of the new inserted cells may overlap with the existing cells. For this reason, an incremental placement is done to perform slight modifications on the current placement and legalize the position of the new cells. Finally, an incremental timing analysis is performed to check if the selected solution, after legalization, improves the delay. If not improved, the last cell insertion is undone.

6.5 Algorithm for gate duplication

Given a gate G, gate duplication aims at creating a pair of gates, G_1 and G_2, such that the original fanout of G is distributed between them. As mentioned in Sect. 6.2, the techniques recently proposed for gate duplication [81, 96] are restricted to legal solutions that do not change the placement of the rest of the cells in the layout. In this section we present a layout-aware gate duplication approach that can be later legalized by incremental changes on the placement.

Clearly, gate duplication explores a trade-off between output and input capacitance. Gates G_1 and G_2, individually, have a smaller output capacitance than G, however the output capacitance of the gates at their fanin increases. The contribution of gate duplication to the performance of a circuit will depend on the particular instance of the problem and the proposed solution.

The algorithm for gate duplication is described in Fig. 6.9. It is based on the well-known k-means clustering algorithm [112]. This strategy is commonly used in data mining where efficient algorithms were proposed to process large quantity of data [88]. The complexity of this algorithm is $O(kni)$, where k is the number of clusters, n is the number of points to be clustered, and i the number of iterations to converge. In our case, $k = 2$ and n is the number of fanouts of the gate, which is typically small. Experimentally, the algorithm converges very fast when n is small, thus showing linear complexity on n.

The algorithm aims at clustering the fanout of G into two subsets, one for G_1 and another for G_2. Initially, two fanout points are arbitrarily chosen as the potential centers of the clusters and each fanout is assigned to the cluster with the closest center. Iteratively, the centers of each cluster are re-calculated at each iteration as the centers of gravity of the components of each cluster. The calculation stops when a fixpoint is reached. Note that, this algorithm is not optimal. The k-means

```
Duplication (G)
{Input: A gate G to be duplicated}
{Output: Gates {G1, G2}}
   C1, C2 := Coordinates of two fanouts of G;
   while changes in C1 or C2 do
      S1 := {Fanouts of G closer to C1};
      S2 := {Fanouts of G closer to C2};
      C1 := Center of gravity of S1;
      C2 := Center of gravity of S2;
   end while
   Cin := Center of gravity of the fanins of G;
   Place G1 at the mid-point between Cin and C1;
   Place G2 at the mid-point between Cin and C2;
   return {G1, G2};
end;
```

Figure 6.9: Gate duplication algorithm.

algorithm is a heuristic clustering approach that depends on the initially selected centers. Due to its fast convergence, multiple runs can be executed and return the best clustering found.

Figure 6.10 depicts the evolution of the algorithm. A net with four fanouts driven by gate G is depicted in Fig. 6.10(a). The gate only has one fanin X. Figures 6.10(b,c,d) show the locations of C_1 and C_2 (shadowed circles) and the sets S_1 and S_2 at each iteration[3]. The initial selected points are A and B (Fig. 6.10(b)), that classify the fanout in two subsets: $S_1 = \{A\}$ and $S_2 = \{B,C,D\}$. After re-clustering, point B is moved to the cluster S_1 and convergence is reached.

At the end of the loop, the fanouts are partitioned into the clusters $S_1 = \{A,B\}$ and $S_2 = \{C,D\}$. The location for G_1 and G_2 is now calculated as the mid-point between the center of gravity of their fanin (C_{in}) and the center of the clusters, respectively. In this particular case, C_{in} coincides with the coordinates of the single fanin X.

6.5.1 Delay-oriented duplication

The previous method for gate duplication does not take into account any timing information. To amend this unawareness, a postprocess can be performed to re-cluster some nodes before the final location of G_1 and G_2 is calculated. We next explain the strategy used in our work.

After the clustering algorithm, G_1 and G_2 may have different criticality according to their slack. Without loss of generality, let us assume that G_1 is less critical. Some of the least critical fanouts

[3]To be precise, the figure shows the state of the loop after the calculation of S_1 and S_2 and before the re-calculation of C_1 and C_2.

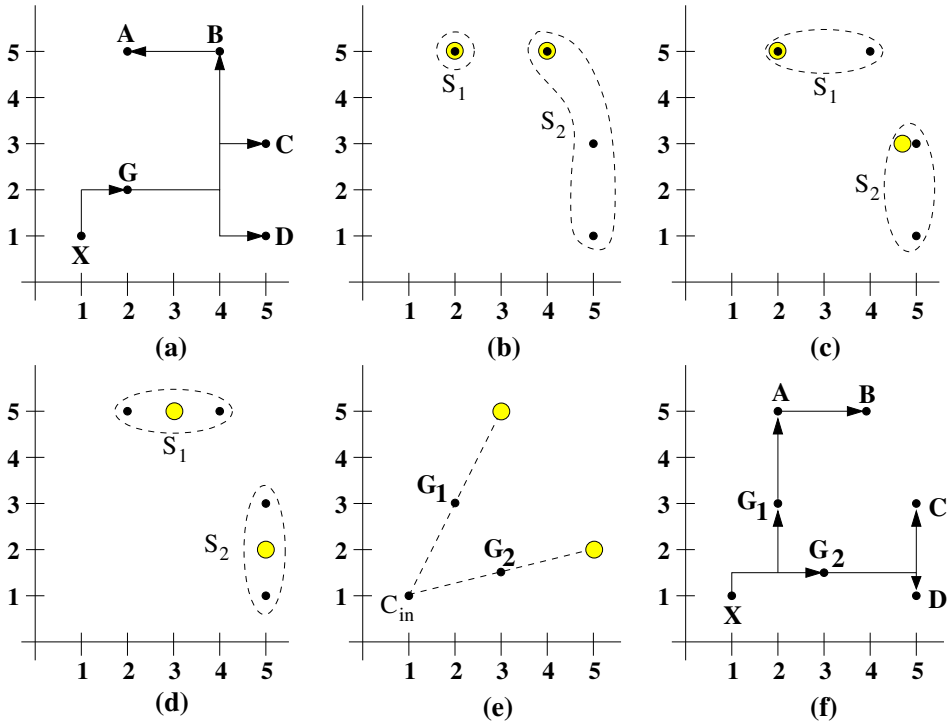


Figure 6.10: Gate duplication: (a) Initial net, (b,c,d) evolution of the k-means algorithm, (e) calculation of the locations for G_1 and G_2, (f) possible routing after duplication.

of G_2 that are physically closer to G_1 can be shifted to G_1. In this way, the total load for G_2 is reduced. This process can be iteratively done until the criticality of G_1 and G_2 is balanced. After the process is finished, the positions of the gates G_1 and G_2 are updated based on the new clusters. Note that, the new positions may change the required time of G_1 and G_2 and these gates may accept new shifts. This approach is applied iteratively and, experimentally, we observed that it converges on few iterations. In our approach, we have implemented a greedy postprocess along these lines.

6.5.2 Discussion

The current clustering approach is layout-oriented, with a postprocess that aims at improving timing by some local re-clustering. One might argue that this could be done the other way around: a timing-driven clustering and a layout-oriented postprocess. The initial experiments immediately showed that the chosen approach is superior, since placement has an impact on timing, congestion and routing, which results in better global results after layout.

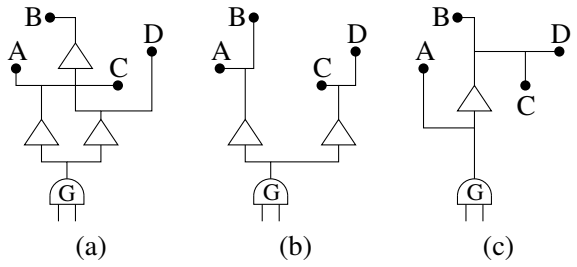


Figure 6.11: Buffer trees.

6.6 Algorithm for buffer insertion

Given a gate G with high fanout, the problem of buffer insertion consists of designing a tree of buffers[4] that drives the fanouts and minimizes the worst negative slack. This section describes a layout-oriented buffer insertion approach taking into consideration timing information. New buffers are allowed to overlap with other gates. As the gate duplication approach, the Steiner tree is later legalized with a post-incremental placement.

6.6.1 Mitigating the combinatorial explosion

Figure 6.11 depicts an example with three different solutions for a gate G with four fanouts. The number of buffer trees for n fanouts is enumerable but extremely large. To reduce the exploration, we use different strategies.

- **Binary trees.** Only binary trees are explored, in which each edge can hold a different number of buffers at different locations, according to their criticality. By only exploring binary trees we are not loosing the chance of building k-ary trees. This can be achieved by inserting no buffers at some intersection of the tree, as illustrated in the solution depicted in Fig. 6.11(c), where the buffer is driving three fanouts since one of the sub-trees has no buffers.

- **Ordered trees.** The number of possible binary trees with n leaf nodes is[5] $T(n) = n! \cdot C_{2n-1}$. We remove the factor $n!$ by imposing an order in the leaves. In this way, the search is reduced to binary trees whose traversal (pre- or post-order) gives the same order of the leaf nodes. For the examples in Fig. 6.11, the depicted trees can be represented by the following parenthesized expressions, respectively:

[4]We will indistinctively use the term *buffers* to refer to inverting and non-inverting buffers. The optimization of the polarity of the buffers will be briefly discussed at the end of the section 6.6.4.

$$((AC)(BD)) \qquad ((AB)(CD)) \qquad (A(B(CD)))$$

Only the last two expressions have the same order at the leaves.

- **Which order?** By imposing an order on the leaves, the search space is drastically reduced preventing the exploration of the large set of possible Steiner trees where many of them have similar structure. However, some optimal solutions may be lost. For this reason, it is important to choose a good order for the exploration. Several orders could be used: depending on the distances between the sinks, by delay criticality or, even, a combination of both criteria. The order chosen in the proposed approach aims at designing layout-aware trees as follows (see Fig. 6.12(a)):

  > *The polar coordinates (angle and distance) of each fanout with respect to the source node are calculated. The relative position of the nodes is defined by their angle. The distance is used only in the case that the angles are similar. The first and last point in the order is determined by the pair of adjacent fanouts with the largest angle between them.*

  The criterion to select this order aims at reducing the wirelength and congestion similar to the decision taken in gate duplication (See Sect. 6.5.2). The experimental results showed that creating an order depending on the polarity or the criticality of the fanouts affects to the wirelength and congestion (even it may appear intersections between different tree branches of the same Steiner tree like in Fig. 6.11.(a)). These orders have a negatively impact to the global results after the physical design.

6.6.2 Bottom-up construction of buffer trees

The exploration of binary trees for buffer insertion is performed bottom-up, from the leaves (fanouts) to the root (gate). This strategy poses a major problem in providing an optimal solution when two sub-trees converge in an intersection point: the criticality of each internal sub-tree is not known until the complete buffer tree has been constructed until the intersection point. For this reason, several solutions are calculated for each sub-tree, each one characterized by a pair (RT, C_{in}) that indicates the required time and the input capacitance at the root. The solutions for the left and right sub-trees are combined and provided as the solutions of the whole sub-tree.

[5] C_k is the k-th Catalan number, $C_k = \frac{1}{k+1}\binom{2k}{k}$, and represents the number of possible binary tree structures with k nodes (a tree with n leaves has $2n-1$ nodes). The factor $n!$ denotes all possible permutations of the leaves.

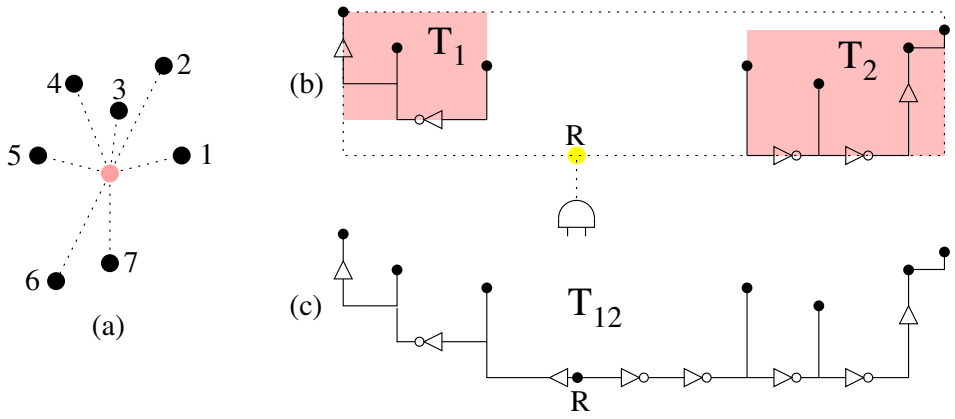


Figure 6.12: (a) Order of fanouts, (b) calculation of the root point for two sub-trees and, (c) connection of the sub-trees by repeater insertion.

In this section, the basic step to construct a tree from two sub-trees is described. This step is illustrated in Fig. 6.12(b,c), where a tree T_{12} is built from the sub-trees T_1 and T_2. The tree is built in two steps:

- Calculation of the coordinates of the root R. Given the bounding box of the leaves of both sub-trees, the root is the point of the box closest to the source node. In case the source node is inside the box, the root is the source node itself. Note that, this tree construction may overlap multiple buffers on the same intersection position. BufDup assumes that the incremental placement will legalize the buffered tree moving apart them to closer locations.

- Generation of the buffers between the root of the tree and the roots of the sub-trees. This is done by a repeater insertion algorithm that is next described.

6.6.3 Repeater insertion

The problem we want to solve is the following: given a library of buffers and inverters and two points (source and sink) with a required time for the sink, design a chain of buffers/inverters that maximize the required time of the source.

This problem is similar to technology mapping for delay and we use the approach presented in [162] for our problem. The approach is simplified and adapted to the design of buffer/inverter chains. The algorithm works in two steps:

1. The number of locations for repeater insertion is calculated. This number is estimated assuming that the same kind of buffer is used along the chain. With this assumption, the po-

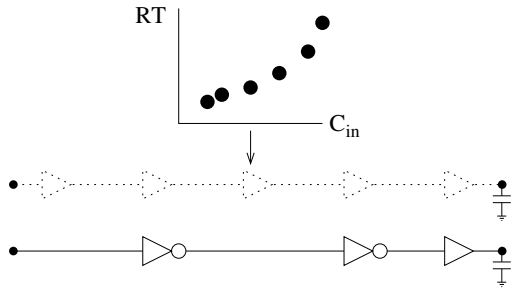


Figure 6.13: Repeater insertion.

tential locations are uniformly distributed along the wire using the optimum number given by Bakoglu's formula [17]:

$$N = \left\lfloor \sqrt{\frac{R_w C_w}{R_b C_b}} \right\rfloor$$

where the R_w, C_w, R_b and C_b are the resistance and capacitance of the wire and the buffer, respectively. The calculation is performed using R_b and C_b of the second smallest inverter in the library.

2. A dynamic programming approach for repeater insertion is executed. The algorithm works as a typical delay-oriented technology mapping algorithm [162], from sink to source. For each insertion point, a set of solutions is calculated. Besides the buffers and inverters in the library, the *wire* (no buffer) is also considered as a candidate for mapping. Note that, a subset of points is only preserved for each position to avoid an explosion of solutions. The pruning heuristic is explained in Sect. 6.6.6.

Figure 6.13 illustrates an example of repeater insertion. The dotted chain represents the set of potential points for insertion defined by Bakoglu's formula. At each point, a set of solutions characterized by the pair (RT, C_{in}) is stored. The chain at the bottom shows a possible solution, in which some of the locations have been simply substituted by wires.

When the no-buffer candidate is considered in an insertion point, its pair (RT, C_{in}) is stored in a different curve of Pareto points. The source gate of the current constructed wire is not know yet, therefore the resistance of the gate can not be used to compute the delay of the wire. The computation of this *partial solution* will be completed on the next insertion point when a buffer will be considered. Note that, the pruning heuristic is also applied to reduce the set of partial solutions.

When merging the solutions of the left and right sub-trees, chains with only one buffer at the nearest location of the source are typically selected when the other sub-tree is critical. This phenomenon is just an algorithmic approach equivalent to the *critical sink isolation* technique proposed in [126].

6.6.4 Polarity optimization

The exploration of inverters as candidates for buffer insertion gives some benefits. Mostly, inverters are smaller in size and they contribute to reduce the total area of the circuit. For instance, Fig. 6.13 shows a buffer insertion where three repeaters are inserted: one with positive polarity and two with negative polarity. In this example, a chain of two inverters reduces considerably the area and the delay compared to a chain of two buffers. The process of exploration is more complex since the solutions on each insertion point must be stored in the proper polarity. Therefore, two sets of solutions are maintained for each chain, one for each polarity. These sets of solutions are propagated towards the root of the tree to deliver the best solution for each possible polarity.

The exploration of both polarities enables the possibility to handle sinks with negative polarity and to apply *source polarity inversion*. For example, a NAND gate can be substituted by an AND gate (or vice versa) if the complemented polarity of the buffered tree is more convenient.

6.6.5 Exploration with dynamic programming

The main algorithm is described in Fig. 6.14. Initially, the order of the fanouts is calculated. The rest of the algorithm calculates the solutions for all possible ordered sub-trees, starting from the smallest trees ($f = 2$) and ending with the complete trees ($f = n$).

Each location of the matrix *Trees* stores several solutions for a sub-tree (only the elements at the upper triangle of the matrix are used). Thus, $Trees[i, j]$ stores all the solutions calculated for the sub-trees with the leaves $Fanout[i \ldots j]$. As an example, the sub-trees explored for $n = 5$ are the following:

$f = 2$ (12) (23) (34) (45)
$f = 3$ (1(23)) ((12)3) (2(34)) ((23)4) (3(45)) ((34)5)
$f = 4$ (1(234)) ((12)(34)) ((123)4) (2(345)) ((23)(45)) ((234)5)
$f = 5$ (1(2345)) ((12)(345)) ((123)(45)) ((1234)5)

Let us assume that the sinks of this example are distributed as shown in Fig. 6.15-(a). The sinks are ordered using the approach explained in Sect. 6.6.1 and the intersection points are obtained using the procedure defined in Sect. 6.6.2. The explored trees for the tree $(1(234))$ are shown in Fig. 6.15-(a,b). This tree is calculated when $f = 4$ and the solutions are obtained by combining the fanout 1 with the solutions of the sub-trees with fanouts $\{2,3,4\}$ calculated when $f = 3$, i.e. $(2(34))$ and $((23)4)$. Figure 6.15-(c) illustrates the distribution of the solutions of the buffered sub-trees in the matrix *Trees*.

For every combination of sub-trees, the procedure `Repeater_Insertion` inserts repeaters from each root of the left and right sub-trees to the root of the tree, respectively. The insertion is done using the approach described in the previous sections. The combination of both solutions (`Build_Tree`) also calculates the required time and the input capacitance at the root.

```
Buffer_Insertion (G)
{Input: The source gate G (assume the gate has n fanouts)}
{Output: A buffered Steiner Tree}
  var:
    Fanout[1...n]: array of fanouts;
    Trees[1...n, 1...n]: Matrix of buffered trees;

  Fanout := Sort_Fanouts(G);
  for f = 2 to n do
    {Explore sub-trees with f fanouts}
    for each pair (i, j) s.t. j − i + 1 = f, 1 ≤ i, j ≤ n do
      R := Root node for fanouts {i...j};
      for k = i to j − 1 do
        {Create tree from sub-trees with fanouts {i...k} and {k+1...j}}
        for each pair (T1, T2) ∈ Trees[i,k] × Trees[k+1, j] do
          B1:=Repeater_Insertion (R, Root(T1));
          B2:=Repeater_Insertion (R, Root(T2));
          T := Build_Tree ( R ↙ B1 → T1 , R ↘ B2 → T2 );

          Trees[i, j] := Trees[i, j] ∪ {T};
        Trees[i, j] := Select_Subset_of_Best_Solutions (Trees[i, j]);

  return Best_Solution (Trees[1, n]);
```

Figure 6.14: Algorithm of buffer insertion.

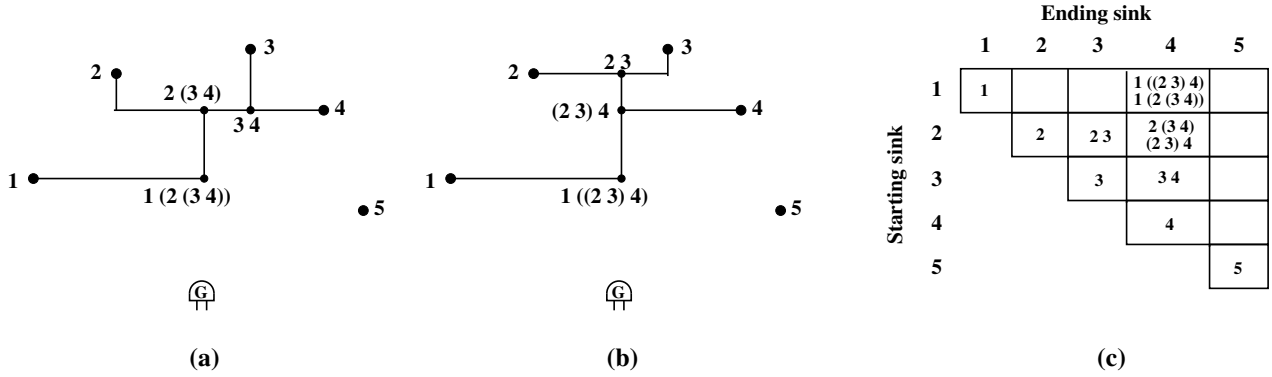


Figure 6.15: Example of the dynamic programming approach. (a) Distribution of the intersection points of the tree $(1(2(34)))$ and (b) the tree $(1(2(34))$. (c) Stored solutions for the trees $(1(2(34)))$ and $(1(2(34))$ in the matrix of buffered trees.

To avoid an explosion of solutions, only a subset of them are kept for each sub-tree. This is performed by the procedure `Select_Subset_Solutions`. The number of solutions has a direct impact on the runtime and the accuracy of the exploration. The strategy for this selection is discussed in the next section.

At the end of the algorithm, the solutions for the complete tree are stored in $Trees[1,n]$. Note that, the root node of $Trees[1,n]$ may not overlap with the source gate G. The source gate G can be outside the bounding box produced from its fanouts. Therefore, a repeater insertion must be done from the root node R to the source gate G. Finally, the best solution is returned.

6.6.6 Pruning solutions

During the exploration of solutions for sub-trees and repeaters, several solutions are calculated with different characteristics of required time and input capacitance. To reduce the complexity of the exploration, only a subset of points is selected for further exploration. We next describe the techniques that have been proven to be efficient and accurate.

- Only the Pareto points are represented. The worst solutions are removed based on the concept of *dominance*. A solution characterized by a cost function dominates another solution if the value of the cost is better. The dominance can be extended to solutions with multiple cost functions.

- If we have n points and we want to select $k < n$, a k-means clustering algorithm is executed (the same strategy used to find clusters on gate duplication [112]), starting with k distributed points along the curve as initial centers. After clustering, the points closest to the centers of the clusters are selected.

6.6.7 Area recovery

Although the non-critical solutions are also processed using the *critical sink isolation* heuristic, the algorithm still inserts some buffers in the non-critical wires. As we previously pointed out, the criticality of a sub-tree is not known until the intersection with other sub-trees. Therefore, extra buffers may be inserted on subtrees that are not critical at the source gate.

After the dynamic approach, a post-process removes these buffers from the non-critical paths that do not interfere with the critical delay of the buffered tree. This approach helps to save area removing useless buffers.

Figure 6.16 shows an example of this area recovery process. The critical wires are initially identified to block the deletion of their buffers. Next, the non-critical wires are traversed from the

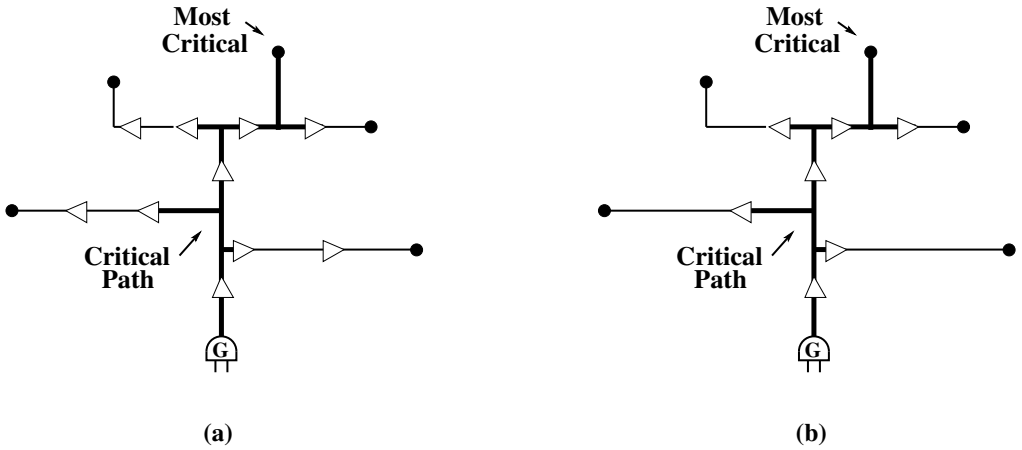


Figure 6.16: Area recovery post-process: (a) Initial buffer insertion. (b) After area recovery.

fanouts until a critical wire is reached. All the buffers that don't affect the delay of any critical wire are removed during the exploration.

6.6.8 Nets with high fanout

The computational complexity of the presented approach depends on the number of fanouts of the processed Steiner Tree. The performance decreases significantly on instances with high fanout.

A possible approach to reduce the complexity is to use a hierarchical approach [12]. The sinks can be partitioned in several clusters depending on their position and the buffer insertion can be applied to each cluster. The drawback of this techniques appears when it is combined with an incremental placement. The circuit may include useless buffers on non-critical paths after the optimization process since the topology of the sinks of the Steiner trees with high fanout can change significantly after each incremental placement step.

A different heuristic is used in BufDup aiming at avoiding the insertion of an excessive number of inverters on these instances. First, fanout optimization is applied after technology mapping to decrease considerably the number of fanouts. The nets of the nodes with more than a certain number of fanouts[6] are considered nets with very high fanout. On these instances, a pre-clustering strategy, using the k-means algorithm, is used to partition the fanouts into three clusters and connect each of them to the source gate with a buffer. The objective of this approach is to use a divide-and-conquer approach to reduce the size of the problem (See Fig. 6.17). This approach contributes to run the buffer insertion algorithm on smaller instances. Moreover, after the buffer insertion, the

[6]This number depends on a user parameter that has been assigned to 30 fanouts in the experiments in Section 6.7.

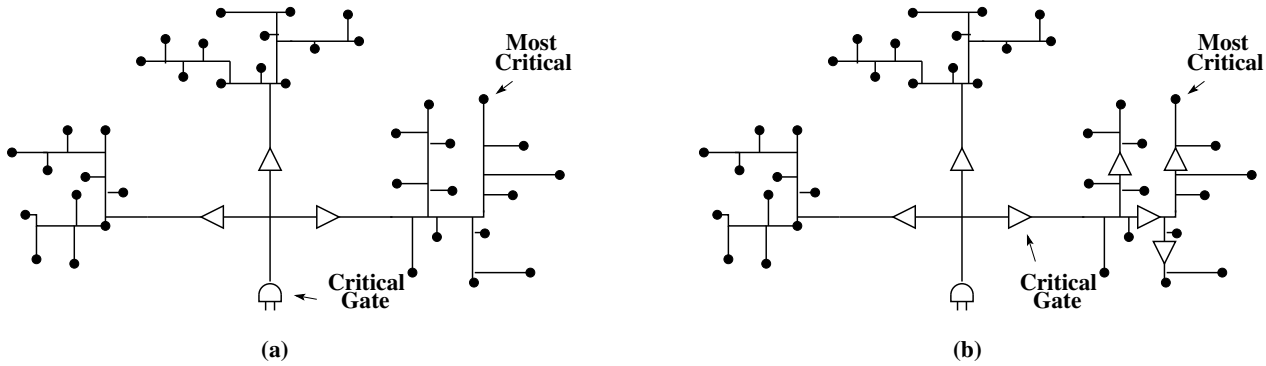


Figure 6.17: Buffer Insertion on large instances: (a) The k-means algorithm is used to create three clusters and the source gate is connected to them with buffers. (b) The buffer is selected as the critical gate in the next iteration of BufDup. The buffer gate has less fanouts and the dynamic programming approach can be applied.

incremental placement performs few modifications on these smaller instances leaving the inserted buffers closer to the original selected position.

6.7 Experimental results

To validate the presented approach, three experiments have been conducted: (1) comparison with FBI, (2) results on public benchmarks, and (3) results on future semiconductor technologies.

The $0.13\mu m$ *vxlib ALLIANCE* library [11] has been used for technology mapping. It includes three buffers and four inverters. The technological parameters have been scaled to different technologies, from 65nm to 22nm, using the *Predictive Technology Model* [83]. For 65nm, the wire capacitance and resistance are $2.71\Omega/\mu m$ and $0.19fF/\mu m$, respectively, that approximately correspond to M2/M3 metal layers of the 65nm technology described in [16].

The experiments have been run on the largest netlists from the ISCAS'99 suite. The initial netlists have been obtained by using the tree-mapping algorithm in SIS, including the fanout optimization step. A square layout with 25% whitespace has been created, with the terminals uniformly distributed around the bounding box.

Fastplace [164] has been used to calculate the initial placement. At each iteration, the detailed placer is used for incremental placement. For the final timing analysis, *labyrinth* [91] has been used to estimate the routing trees and calculate the delays using the Elmore model.

Iter.	fan.	Wire Length	Initial Delay	FBI			BufDup		
				buf	inv	$-\Delta$D	buf	inv	$-\Delta$D
1	67	6680	8531	3	0	901	3	0	1824
2	76	6168	8246	11	16	1130	0	3	1297
3	17	5332	7945	13	0	123	4	1	126
4	35	6024	7870	12	23	394	3	0	313
5	15	6313	7751	6	7	548	6	8	619
6	28	6176	7732	12	10	227	4	2	227
7	27	5508	7692	13	8	202	2	3	202
8	15	8871	7677	13	3	43	6	11	43
9	11	4550	7612	5	1	18	4	9	22
10	14	5040	7590	2	0	81	1	1	89
			Tot.	90	68	3668	30	38	4762

Table 6.1: Comparison of FBI and BufDup for individual trees.

6.7.1 Comparison of the Buffer Insertion algorithm

The first experiment compares the dynamic programming approach used in buffer insertion with FBI 1.0 with the cost package [145]. To the best of our knowledge, this is the only public domain tool based on Van Ginneken's approach that supports inverter insertion and sink polarity. Moreover, this approach improves Van Ginneken's using predictive pruning and redundancy check heuristics. This experiment has been designed to illustrate the impact of the features of BufDup. The Steiner trees used on this experiment correspond to the output wire of the selected critical gate during the first ten iterations of BufDup on the `b14` netlist. For FBI, the Steiner tree is computed using the *BOI* heuristic [29]. The potential locations for the buffers are the intersection points of the tree. Additional distributed locations have also been included for long wires using the number of buffers defined by Bakoglu's formula [17].

The results are presented in Table 6.1. For each tree, it reports the number of fanouts, the total wirelength of the estimated routing and the initial delay (in *ps*) of the tree using the estimated routing.

For a fair comparison, the BufDup has only been used for buffer insertion (no gate duplication). For each method, the number of buffers, inverters and improvement in delay ($-\Delta D$) are reported.

Only in iteration 4, FBI obtains a better delay reduction than BufDup. In the rest of trees, BufDup obtains a better result or similar. The improvements are significantly better in the first trees (the most critical), with high fanout. The improvements are also tangible in the number of buffers and inverters produced by each method.

The improvements are mainly due to two reasons:

- The wider exploration of trees in BufDup (binary trees with dynamic programming). The exploration of multiple sub-trees improves the initial wire distribution provided by BOI heuris-

Netlist	Gates					Critical-path delay (ps)				
	Initial	FBI	Buf	Dup	BufDup	Initial	FBI	Buf	Dup	BufDup
b14	4936	5319	5184	5056	5297	8505	7763	7627	7765	7429
b14_1	4490	4616	4634	4566	4570	6018	5389	5224	5252	5184
b15	7386	7906	7526	7410	7562	8102	7754	7743	7857	7657
b15_1	7189	7628	7363	7245	7448	6767	6292	5913	6026	5632
b17	23358	23515	23610	23434	23603	16286	14814	13831	14621	13860
b17_1	22345	23083	22612	22445	22821	6796	6156	5944	6052	5789
b20	10103	10769	10294	10195	10505	10406	9592	9406	9591	8628
b20_1	8890	9037	8940	8954	8974	8403	7911	7830	7921	7948
b21	10689	10971	10920	10753	10877	9513	9352	9449	9583	9156
b21_1	9005	9142	9122	9029	9110	9978	9039	8581	8869	8670
b22	15233	15684	15548	15361	15591	9666	8808	8540	8602	8426
b22_1	13325	13830	13640	13377	13699	9350	8531	8372	8400	8117
s35932	8432	8725	8835	8456	9010	5346	4373	1912	2206	1909
s38417	10606	11008	10954	10666	10868	5310	4419	3707	3738	3750
s38584	9723	9883	10137	9755	10226	5338	4804	2337	3354	2345
Norm.	1.00	1.03	1.02	1.01	1.03	1.00	0.91	0.85	0.87	0.83

Table 6.2: Results for different interconnect optimization methods (65nm technology).

tic. Moreover, the selected distribution of the sub-trees reduces drastically the number of buffers and inverters.

- The capability of flipping the polarity of the source gate (see Sect. 6.6.4) enables the possibility to deliver a better solution based on the complementary polarity of the source gate.

6.7.2 Academic benchmarks

Table 6.2 and Table 6.3 shows the results obtained by four methods: (1) FBI, (2) BufDup without gate duplication (label Buf), (3) BufDup without buffer insertion (label Dup), and (4) BufDup. The parameters of the netlist before buffer insertion are reported in the columns with label `Initial`. Table 6.2 reports the number of gates of the netlists and the delay of the one of the critical path in the netlist. Tablee 6.3 reports the routing wirelength and the runtime of the same experiment. The last row of the table shows a normalized average of the results.

The experiments have been run for a 65nm technology and tree-mapping *map* [133, 162] with fanout optimization has been run on these netlists before the results reported in the column with label `Initial`. Several conclusions can be drawn:

- The layout-aware interconnect optimization reduces delay by 17% with regard to the original netlist after technology mapping and fanout optimization.

Netlist	Global routing wirelength ($\lambda \cdot 10^{-3}$)					Runtime (sec)			
	Initial	FBI	Buf	Dup	BufDup	FBI	Buf	Dup	BufDup
b14	2828	2850	2970	2929	2996	94	105	67	195
b14_1	3654	2648	2731	2749	2922	50	216	29	34
b15	5560	5718	5915	5642	5747	48	154	24	394
b15_1	4983	5214	5248	5016	5015	79	114	32	227
b17	26588	19943	20515	20702	20522	161	487	149	391
b17_1	15167	15042	15080	15019	15022	111	347	157	518
b20	6645	6728	6654	6584	6606	368	311	93	357
b20_1	5530	5310	5443	5330	5504	51	63	51	72
b21	6934	6803	7352	6995	7233	98	210	59	189
b21_1	9833	6527	7419	7356	7240	47	107	20	119
b22	9447	9756	9776	9776	9816	127	265	117	356
b22_1	8094	8020	7962	7946	8089	88	157	48	306
s35932	5905	5944	6755	6410	6886	144	250	21	358
s38417	10693	7084	7743	7724	7916	99	180	64	198
s38584	5258	5112	6194	5461	6137	33	205	29	267
Norm.	1.00	0.89	0.93	0.91	0.93	1.00	1.99	0.60	2.49

Table 6.3: Results for different interconnect optimization methods (65nm technology).

- The wide exploration of BufDup (including incremental placement) has a tangible impact in the design of the Steiner trees (delay from 0.91 to 0.83 with regard to FBI). Even if only buffer insertion is applied in BufDup, the results are better that FBI.
- Buffer insertion techniques are superior with respect gate duplication on wirelength optimization at the expenses of increasing the total area of the netlist (1% of increment).
- The combination of gate duplication with buffer insertion contributes to improve delay in something more than 2% at the expense of 1% area increase.

On average, the wirelength after global routing is also reduced for FBI and BufDup. The reduction is more important for FBI. Although not reported in the table, *labyrinth* also showed a slight improvement in congestion for all examples. In one case (`b17`), the congestion was reduced by 79% when using BufDup. The runtime is significantly bigger in BufDup approach because of the extensively exploration of several sub-trees in buffer insertion. The runtime for gate duplication is meaningless compared to the runtime needed to apply buffer insertion.

6.7.3 Future semiconductor technologies

Table 6.4 summarizes the results of BufDup on several future technologies from 65nm to 22nm. The parameters for each technology have been scaled using the interconnection calculator in [83].

Tech.	Gates					Critical-path delay				
	Init.	FBI	Buf	Dup	BufDup	Init.	FBI	Buf	Dup	BufDup
65nm.	1.00	1.03	1.02	1.01	1.03	1.00	0.91	0.85	0.87	0.83
45nm.	1.00	1.04	1.03	1.01	1.02	1.00	0.87	0.82	0.84	0.81
32nm.	1.00	1.06	1.04	1.01	1.04	1.00	0.80	0.76	0.79	0.76
22nm.	1.00	1.08	1.06	1.01	1.07	1.00	0.69	0.64	0.66	0.63

Table 6.4: Impact of interconnect optimization on future generations.

The table shows the normalized sums of delay and area of the netlists used in the previous section. The table also compares the same methods: (1) BufDup, (2) BufDup without gate duplication and (3) without buffer insertion and (4) FBI.

The results corroborate the same conclusions extracted from the previous section. Buffer insertion contributes in larger ratio to the interconnection optimization at the cost of increasing the total area of the netlist. Moreover, the results of this table confirm that interconnect optimization will acquire an increasing relevance in future technologies due to the dominant role of wire delays. Efficient and accurate buffer insertion approaches will be crucial to reduce critical-path delays. As an example, BufDup was able to reduce the delay by 37% on average for a 22nm technology.

6.8 Conclusions

An integrated approach for layout-aware interconnect optimization has been presented. BufDup combines gate duplication and buffer insertion in the same framework with incremental placement. The combination with incremental placement reduces considerably the complexity of the interconnect optimization problem since it can be unaware to place the cells on legal positions.

Related to the designed buffer insertion algorithm, the wide exploration of buffered trees using an efficient dynamic programming approach and the incremental legalization of solutions has a tangible impact in the quality of the solutions. The drawback of our algorithm appears on gates with high fanout. However, a hierarchical approach has been presented to deal with these large instances. Moreover, this approach contributes to reduce significantly the number of inserted buffers. Finally, a fast heuristic to perform gate duplication based on the k-means algorithm has been also presented with a good performance even in gates with high fanout.

In the experimental results, a comparison between gate duplication and buffer insertion is also performed. There is a trade-off on the results of these techniques. Buffer insertion is superior to gate duplication with regard to delay optimization at the expenses of consuming more gates. However, the combination of both techniques improves significantly the results. The results have also shown the relevant role of interconnect optimization in future technologies.

BufDup is a layout-aware wire optimization procedure and, therefore, the congestion is also

affected after the minimization. As we pointed out in the experimental results, the congestion was reduced by 79% in one of the largest circuits.

Chapter 7

Conclusions

The complexity of digital circuits is continuously growing. The demand of high-performance circuits forces the designers to seek for innovative approaches in order to meet the specifications. The research in manufacturing is still progressing. New materials and manufacturing methodologies contribute to shrink the size of the transistors. This progress is an important factor on the reduction of the area of the circuits. However, the increment of the circuit density also has negative effects, mostly in the physical design. Congestion, power leakage, wire coupling, among others problems make *ideal* designs sometimes unfeasible to manufacture.

CAD tools help to attenuate these problems. Moreover, the design development cycle and design cost are dramatically reduced and large designs are sped up. However, the specialization of the digital circuits requires new advanced design methods well-suited for the challeging requirements. The new approaches are currently combining the logic and physical stages to lead to superior results in DSM design flow.

In this thesis, several contributions have been proposed aiming at delay minimization. Some experts on the area claim that *logic synthesis is dying*, e.g. [113], due to the integration with physical design. Our contributions show that there is still high-quality methologies in logic synthesis. Finally, our last contribution corroborates that the integration of logic and physical design is a good strategy.

Our contributions are summarized next. First, a new paradigm to solve Boolean relations has been presented in Chapter 3. The branch-and-bound method explores more efficiently the search space compared to previous heuristic approaches based on local search. Moreover, an exploration towards a user-defined objective makes possible to model other problems as Boolean relations.

The second contribution presented in Chapter 4 is focused to perform delay minimization on large Boolean networks. An iterative partition method is proposed. Compared with previous approaches, better clusters of nodes towards delay minimization are produced with the exploitation of vertex dominators. Delay minimization procedures obtain better results on these clusters, since they offer more possibilities for restructuring.

In Chapter 5, an application of the previous contributions is presented. A timing-driven approach is proposed to perform a n-way decomposition on large netlists using the partitioning approach. The objective is the exploration of several decompositions of a Boolean function using a particular gate targeting at reducing the depth of the decomposition tree. A Boolean relation is built to model the problem and the search is guided towards delay minimization.

The final contribution in Chapter 6 uses physical information on the fanout optimization problem. Physical information gives a better approximation of the critical regions of the circuits.

Delay minimization is one of the most relevant objectives in circuit design. However, there are other design objectives. As future work, the next natural step would be the integration of other minimization objectives, like power consumption, into the introduced methods to make them more general.

Our contributions are also opened to further improvements. For example, an option could be to adapt the recursive paradigm to solve Boolean relations to solve multiple Boolean relations (MBR). However, the set of compatible solutions of an MBR could be large and the method could be not scalable.

Chapter 5 shows that Boolean relations can be applied to the n-way decomposition problem. Boolean relations can also be applied to other problems, like pattern matching in technology mapping.

The paradigm developed for solving Boolean relations is a generic recursive approach that could be applied to other problems. The objective is to relax the problem to another one with lower computational cost. This relaxation could prevent to find a correct solution. However, the errors can be fixed by splitting the problem in smaller ones. An example of application could be to solve the binate covering problem using a unate covering problem solver.

Partitioning techniques are ideal to tackle with complex problem. Moreover, scalability is a crucial aspect for the applicability of logic synthesis techniques on large networks. In DSM technologies, the combination of logic and physical synthesis seems to be essential to meet the demand of today's designers regarding delay and power optimization. We believe that the proposed partitioning strategy, enhanced with layout information, could be a valid approach for integrating and exploring logic and physical parameters of the design.

Bibliography

[1] Microprocessor quick reference guide. http://www.intel.com/pressroom/kits/quickreffam.htm.

[2] Silicon innovation: Leaping from 90nm to 65nm. http://download.intel.com/research/silicon/silicon_paper_06.pdf.

[3] C. Ababei, N. Selvakkumaran, K. Bazargan, and G. Karypis. Multi-objective circuit partitioning for cutsize and path-based delay minimization. In *ICCAD*, pages 181–185, 2002.

[4] S. N. Adya, S. Chaturvedi, J. A. Roy, D. A. Papa, and I. L. Markov. Unification of partitioning, placement and floorplanning. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 550–557, 2004.

[5] R. Aggarwal, R. Murgai, and M. Fujita. Speeding up technology-independent timing optimization by network partitioning. *Proc. ACM/IEEE Design Automation Conference*, pages 83–90, November 1997.

[6] P. Agrawal, V. D. Agrawal, and N. N. Biswas. Multiple output minimization. In *Proc. ACM/IEEE Design Automation Conference*, pages 674–680, 1985.

[7] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 6:333–340, June 1975.

[8] A. V. Aho and J. D. Ullman. *Theory of Parsing, Translation and Compiling*. Prentice Hall Professional Technical Reference, 1973.

[9] H. Ajuha and P. R. Menon. Delay reduction by segment substitution. *Proc. European Conference on Design Automation (EDAC)*, pages 82–86, 1994.

[10] S. B. Akers. Binary decision diagrams. In *IEEE Transactions on Computers C-27*, pages 509–516, 1978.

[11] Alliance library.
www.vlsitechnology.org/html/vx_description.html.

[12] C. J. Alpert, M. Hrkic, J. Hu, A. B. Kahng, J. Lillis, B. Liu, S. T. Quay, S. S. Sapatnekar, A. J. Sullivan, and P. Villarrubia. Buffered steiner trees for difficult instances. In *Proc. International Symposium on Physical Design*, pages 4–9, 2001.

[13] C. J. Alpert, M. Hrkic, and S. T. Quay. A fast algorithm for identifying good buffer insertion candidate locations. In *Proc. International Symposium on Physical Design*, pages 47–52, 2004.

[14] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.

[15] M. E. Amyeen, W. K. Fuchs, I. Pomeranz, and V. Boppana. Fault equivalence identification using redundancy information and static and dynamic extraction. *Proc. of the Asian Test Symposium*, pages 124–130, April 2001.

[16] P. Bai, C. Auth, and et al. *A 65nm Logic Technology Featuring 35nm Gate Lengths, Enhanced Channel Strain, 8 Cu Interconnect Layers, Low-k ILD and 0.57μm^2 SRAM Cell*. Intel Developer Forum, August 2005.

[17] H. B. Bakoglu. *Circuits, Interconnections, and Packaging forl VLSI.* Addison-Wesley, 1990.

[18] D. Baneres, J. Cortadella, and M. Kishinevsky. A recursive paradigm to solve boolean relations. In *Proc. ACM/IEEE Design Automation Conference*, pages 416–421, June 2004.

[19] D. Baneres, J. Cortadella, and M. Kishinevsky. Dominator-based partitioning for delay optimization. In *ACM Great Lakes Symposium on VLSI*, pages 67–72, 2006.

[20] D. Baneres, J. Cortadella, and M. Kishinevsky. Layout-aware gate duplication and buffer insertion. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1367–1372, 2007.

[21] D. Baneres, J. Cortadella, and M. Kishinevsky. A recursive paradigm to solve boolean relations. In *IEEE Transactions on Computers*, 2007 (Submitted).

[22] T.C. Bartee. Computer design of multiple-output logical networks. *IRE Transactions on Electronic and Computers*, pages 21–30, 1961.

[23] M. Bartholomeus and H. De Man. PRESTOL-II: yet another logic minimiser for programmed logic arrays. *Proc. International Symposium on Circuits and Systems*, page 58, 1985.

[24] M. W. Beattie and L. T. Pileggi. Inductance 101: modeling and extraction. In *Proc. ACM/IEEE Design Automation Conference*, pages 323–328, 2001.

[25] F. Beeftink, P. Kudva, D. Kung, and L. Stok. Gate-size selection for standard cell libraries. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 545–550, 1998.

[26] L. Benini and G. De Micheli. A survey of boolean matching techniques for library binding. *ACM Transactions on Design Automation of Electronic Systems*, 2(3):193–226, July 1997.

[27] G. Beraudo and J. Lillis. Timing optimization of fpga placements by logic replication. In *Proc. ACM/IEEE Design Automation Conference*, pages 196–201, 2003.

[28] D. Bochmann, F. Dresig, and B. Steinbach. A new decomposition method for multilevel circuit design. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 374–377, 1991.

[29] M. Borah, R. M.Owens, and M.J. Irwin. An edge-based heuristic for steiner routing. *IEEE Transactions on Computer-Aided Design*, 13(12):1563–1568, December 1994.

[30] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. ACM/IEEE Design Automation Conference*, pages 40–45, 1990.

[31] D. Brasen and G. Saucier. FPGA partitioning for critical paths. In *Proc. European Conference on Design Automation (EDAC)*, pages 99–103, 1994.

[32] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

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

[34] R.K. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *Proc. of the Int. Symp. on Circuits and Systems*, pages 49–54, May 1982.

[35] R.K. Brayton and F. Somenzi. An exact minimizer for boolean relations. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 316–319, November 1989.

[36] R.K. Brayton and F. Somenzi. Minimization of boolean relations. In *Proc. International Symposium on Circuits and Systems*, pages 738–743, 1989.

[37] F.M. Brown. *Boolean Reasoning: The Logic of Boolean Equations*. Kluwer Academic Publishers, 1990.

[38] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[39] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. A new, simpler linear-time dominators algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(6):1265–1296, 1998.

[40] S. Chatterjee and R. Brayton. A new incremental placement algorithm and its application to congestion-aware divisor extraction. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 541–548, 2004.

[41] C.-H. Chen and C.-Y. Tsui. Timing optimization of logic network using gate duplication. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 233–236, January 1999.

[42] G. Chen and J. Cong. Simultaneous logic decomposition with technology mapping in FPGA designs. In *Proc. International Symposium on Field Programmable Gate Arrays*, pages 48–55, 2001.

[43] G. Chen and J. Cong. Simultaneous timing-driven placement and duplication. In *Proc. International Symposium on Field Programmable Gate Arrays*, pages 51–59, 2005.

[44] G. Chen and J. Cong. Simultaneous placement with clustering and duplication. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):740–772, July 2006.

[45] K. C. Chen and S. Muroga. Timing optimization for multi-level combinational networks. In *Proc. ACM/IEEE Design Automation Conference*, pages 361–364, 1990.

[46] D. I. Cheng, C. C. Lin, and M. Marek-Sadowska. Circuit partitioning with logic perturbation. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 650–655, 1995.

[47] Y. Cheon and D. F Wong. Design hierarchy-guided multilevel circuit partitioning. *IEEE Trans. on CAD of Int. Circuits and Systems*, 22(4):420–427, 2003.

[48] J.-H. Chern, J. Huang, L. Arledge, P.-C. Li, and P. Yang. Multilevel metal capacitance models for CAD design synthesis systems. *IEEE Electron Device Letters*, 13(1):32–34, 1992.

[49] M. Chrzanowska-Jeske. Generalized symmetric variables. In *IEEE International Conference on Electronics, Circuits and Systems*, pages 1147–1150, 2001.

[50] E. F. Codd. Further normalization of the data base relational model. In *Courant Computer Science Symposia 6, "Data Base Systems"*. Prentice-Hall, May 24-25 1971.

[51] T. Coe, T. Mathisen, C. Moler, and V. Pratt. Computational aspects of the Pentium affair. *IEEE Computational Science & Engineering*, 2(1):18–31, 1995.

[52] J. Cong and C. Koh. Simultaneous driver and wire sizing for performance and power optimization. *IEEE Transactions on VLSI Systems*, 2(4):408–425, 1994.

[53] J. Cong, K. Leung, and D. Zhou. Performance-driven interconnect design based on distributed RC delay model. In *Proc. ACM/IEEE Design Automation Conference*, pages 606–611, 1993.

[54] J. Cong, J.Y. Lin, and W. Long. A new enhanced SPFD rewiring algorithm [logic IC layout]. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 672–678, 2002.

[55] J. Cong and C. Wu. Global clustering-based performance-driven circuit partitioning. In *Proc. of the Int. Symp. on Physical Design*, pages 149–154, 2002.

[56] J. Cortadella. Timing-driven logic bi-decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):675–685, June 2003.

[57] O. Coudert. Gate sizing: A general purpose optimization approach. In *Proc. of the European conference on Design and Test*, page 214, 1996.

[58] O. Coudert, C. Berthet, and J.C. Madre. Verification of synchronous sequential machines using boolean functional vectors. In *Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, November 1989.

[59] O. Coudert and J.C. Madre. A unified framework for the formal verification of circuits. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 126–129, November 1990.

[60] O. Coudert and J.C Madre. A unified framework for the formal verification of sequential circuits. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 126–129, 1990.

[61] Olivier Coudert and Jean Chritophe Madre. New ideas for solving covering problems. In *Proc. ACM/IEEE Design Automation Conference*, pages 641–646, 1995.

[62] R. Cutler and S. Muroga. Useless prime implicants of incompletely specified multiple-output switching functions. *Int. Journal of Parallel Programming*, 9(4), 1980.

[63] M. Damiani and G. De Micheli. Recurrence equations and the optimization of synchronous logic circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 556–561, 1992.

[64] M. Damiani, J. Yang, and G. De Micheli. Optimization of combinational logic circuits based on compatible gates. *IEEE Transactions on Computer-Aided Design*, 14(11):1316–1327, November 1995.

[65] S. Dechu, Z. Cien Shen, and C. C. N. Chu. An efficient routing tree construction algorithm with buffer insertion, wire sizing and obstacle considerations. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 361–366, 2004.

[66] E. Dubrova, M. Teslenko, and A. Martinelli. On relation between non-disjoint decomposition and multiple-vertex dominators. *Proc. International Symposium on Circuits and Systems*, 4:493–496, May 2004.

[67] P. Eades, X. Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[68] W. C. Elmore. The transient response of damped linear networks. *Journal of Applied Physics*, 19:55–63, January 1948.

[69] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[70] H. Fleisher and L. Maissel. An introduction to array logic. *IBM Journal of Research and Development*, 19(2):98–109, 1975.

[71] T. Gao, C.L. Liu, and K. Chen. A performance driven hierarchical partitioning placement algorithm. In *Proceedings on European Design Automation Conference*, pages 33–38, 1993.

[72] A. Ghosh, S. Devadas, and A.R. Newton. Heuristic minimization of boolean relations using testing techniques. In *Proc. International Conf. Computer Design (ICCD)*, September 1990.

[73] L.J. Guibas and J. Stolfi. On computing all northeast nearest neighbors in the L1 metric. *Information Processing Letters*, 17:219–223, 1983.

[74] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, 2000.

[75] N. Hanchate and N. Ranganathan. Post-layout gate sizing for interconnect delay and crosstalk noise optimization. In *Proc.International Symposium on Quality Electronic Design (ISQED)*, pages 92–97, 2006.

[76] D. Harel. A linear algorithm for finding dominators in flow graphs and related problems. In *Proc. ACM Symposium on Theory of Computing (STOC)*, pages 185–194, 1985.

[77] W. A. Havanki, S. Banerjia, and T. M. Conte. Treegion scheduling for wide issue processors. In *Proc. Intl. Symp. on High-Performance Computer Architecture*, pages 266–276, 1998.

[78] S. J. Hong, R. G. Cain, and D. L. Ostapko. Mini: a heuristic approach for logic minimisation. *IBM Journal Reseach Develop*, pages 443–458, 1974.

[79] Y. Hong, P. A. Beerel, J. R. Burch, and K. L. McMillan. Safe BDD minimization using don't cares. In *Proc. ACM/IEEE Design Automation Conference*, pages 208–213, 1997.

[80] M. Hrkic and J. Lillis. S-Tree: a technique for buffered routing tree synthesis. In *Proc. ACM/IEEE Design Automation Conference*, pages 578–583, 2002.

[81] M. Hrkic, J. Lillis, and G. Beraudo. An approach to placement-coupled logic replication. In *Proc. ACM/IEEE Design Automation Conference*, pages 711–716, 2004.

[82] A. J. Hu, D. L. Dill, A. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *Proc. of the International Workshop on Computer Aided Verification*, pages 82–95, 1993.

[83] Interconnection calculator. www.eas.asu.edu/˜ptm/cgi-bin/interconnect/local.cgi.

[84] J. Ishikawa, H. Sato, M. Hiramine, K. Ishida, S. Oguri, Y. Kasuma, and S. Murai. A rule based reorganization system LORES/EX. In *Proc. International Conf. Computer Design (ICCD)*, pages 262–266, 1988.

[85] S. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of Boolean relations. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 417–420, November 1992.

[86] D.-J. Jongeneel, Y. Watanabe, R. K. Brayton, and R. Otten. Area and search space control for technology mapping. In *Proc. ACM/IEEE Design Automation Conference*, pages 86–91, 2000.

[87] A.B. Kahng and G. Robins. A new class of iterative steiner tree heuristics with good performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(7):893–902, July 1992.

[88] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(7):881–892, 2002.

[89] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: application in VLSI domain. In *Proc. ACM/IEEE Design Automation Conference*, pages 526–529, 1997.

[90] R. Kastner, E. Bozorgzadeh, and M. Sarrafzadeh. An exact algorithm for coupling-free routing. In *Proc. International Symposium on Physical Design*, pages 10–15, 2001.

[91] R. Kastner and M. Sarrafzadeh. Labyrinth: A global router and routing development tool. www.ece.ucsb.edu/˜kastner/labyrinth.

[92] K. Kawaguchi, C. Iwasaki, and M.Y. Muraoka. A RTL partitioning method with a fast min-cut improvement algorithm. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 57–60, 1997.

[93] B Kernighan and S Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 29, 1970.

[94] K. Keutzer. DAGON: Technology binding and local optimization by DAG matching. *Proc. ACM/IEEE Design Automation Conference*, pages 314–347, June 1987.

[95] S.P. Khatri, S. Sinha, and R.K. Brayton. SPFD-based wire removal in standard-cell and network-of-pla circuits. *IEEE Transactions on Computer-Aided Design*, 23(7):1020–1030, July 2004.

[96] H. Kim, J. Lillis, and M. Hrkic. Techniques for improved placement-coupled logic replication. In *Proc. of the Great Lakes Symposium on VLSI*, pages 211–216, 2006.

[97] D. Kuck. *The Structure of Computers and Computation*. John Wiley & Sons, 1978.

[98] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design*, 21(12):1377–1394, 2002.

[99] Y. Kukimoto, R.K. Brayton, and P. Sawkar. Delay-optimal technology mapping by DAG covering. In *Design Automation Conference*, pages 348–351, 1998.

[100] Y. Kukimoto and M. Fujita. Rectification method for lookup-table type FPGA's. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 54–61, 1992.

[101] E. L. Lawler, K. N. Levitt, and J. Turner. Module clustering to minimize delay in digital networks. *IEEE Transactions on Computers*, C-18(1):47–57, January 1969.

[102] C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38, 1959.

[103] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. *IEEE Transactions on Computer-Aided Design*, 16(8):813–834, August 1997.

[104] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions of Programing Languages and Systems*, 1(1):121–141, 1979.

[105] Yun-Yin Lian and Youn-Long Lin. Layout-based logic decomposition for timing optimization. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 229–232, 1999.

[106] J. Lillis, C.Cheng, and T. Y. Lin. Optimal wire sizing and buffer insertion for low power and a generalized delay model. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 138–143, 1995.

[107] J. Lillis, C. Cheng, and T. Y. Lin. Algorithms for optimal introduction of redundant logic for timing and area optimization. In *Proc. International Symposium on Circuits and Systems*, volume 4, pages 452–455, 1996.

[108] J. Lillis, C. Cheng, T. Y. Lin, and C. Ho. New performance driven routing techniques with explicit area/delay tradeoff and simultaneous wire sizing. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[109] B. Lin and F. Somenzi. Minimization of symbolic relations. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 88–91, November 1990.

[110] Q. Liu and M. Marek-Sadowska. Wire length prediction-based technology mapping and fanout optimization. In *Proc. International Symposium on Physical Design*, pages 145–151, 2005.

[111] E. S. Lowry and C. W. Medlock. Object code optimization. *Communications of the ACM*, 12(1):13–22, 1969.

[112] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proc. of 5th Berkeley Symp. on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, 1967. University of California Press.

[113] R. Madhavan. The death of logic synthesis. In *Proc. International Symposium on Physical Design*, pages 1–1, 2005.

[114] F. Mailhot and G. De Micheli. Technology mapping using boolean matching and don't care sets. *Proc. European Conference on Design Automation (EDAC)*, pages 212–216, March 1990.

[115] E. McCluskey. Minimization of boolean functions. *Bell Syst.Tech Journal*, 35:1417–1444, 1956.

[116] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[117] S. Minato. Fast generation of prime-irredundant covers from binary decision diagrams. *IEICE Transactions on Fundamentals of Electronics Communications and Computer Sciences*, E76-A(6):967–973, June 1993.

[118] S.-I. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

[119] A. Mishchenko and R. K. Brayton. SAT-based complete don't-care computation for network optimization. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 412–417, 2005.

[120] A. Mishchenko, B. Steinbach, and M. Perkowski. An algorithm for bi-decomposition of logic functions. In *Proc. ACM/IEEE Design Automation Conference*, pages 282–285, June 2001.

[121] N. Modi and J. Cortadella. Boolean decomposition using two-literal divisors. In *Proc. International Conference on VLSI Design*, pages 765– 768, January 2004.

[122] E. Morreale. Recursive operators for prime implicant and irredundant normal form determination. *IEEE Transactions on Computers*, 19(6):504–509, 1970.

[123] R. Murgai. On the global fanout optimization problem. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 511–515, 1999.

[124] R. Murgai. Efficient global fanout optimization algorithms. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 571–576, 2001.

[125] R. Murgai, R. K. Brayton, and A. Sangiovanni-Vincentelli. On clustering for minimum delay/area. *Proc. ACM/IEEE Design Automation Conference*, pages 6–9, 1991.

[126] T. Okamoto and J. Cong. Buffered steiner tree construction with wire sizing for interconnect layout optimization. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 44–49, 1996.

[127] Jr. P. W. Purdom and E. F. Moore. Immediate predominators in a directed graph [H]. *Communications of the ACM*, 15(8):777–778, 1972.

[128] C. I. Park and Y. B. Park. An efficient algorithm for VLSI network partitioning problem using a cost function with balancing factor. *IEEE Trans. on CAD of Int. Circuits and Systems*, 12(11):1686–1694, 1993.

[129] R.C. Prim. Shortest interconnection network and some generalizations. *The Bell System Technical Journal*, 36:1389–1401, 1967.

[130] R. Rajaraman and D. F Wong. Optimal clustering for delay minimization. *IEEE Trans. on CAD of Int. Circuits and Systems*, 14(12):1490–1405, 1995.

[131] R. R. Rao, D. Blaauw, D. Sylvester, C. J. Alpert, and S. Nassif. An efficient surface-based low-power buffer insertion algorithm. In *Proc. International Symposium on Physical Design*, pages 86–93, 2005.

[132] 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. ACM/IEEE Design Automation Conference*, pages 188–191, 1993.

[133] R. Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, UC Berkeley, April 1989.

[134] R. L. Rudell and A. Sangiovanni-Vincentelli. Multi-valued minimisation for pla optimisation. *IEEE Transactions on Computer-Aided Design*, pages 727–750, 1987.

[135] T. Sakurai and K. Tamaru. Simple formulas for two- and three-dimensional capacitances. *IEEE Transactions on Electron Devices*, 30(2):183–185, 1983.

[136] A. H. Salek, J. Lou, and M. Pedram. A simultaneous routing tree construction and fanout optimization algorithm. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 625–630, 1998.

[137] T. Sasao. An application of multiple-valued logic to a design of programmable logic arrays. In *Proc. IEEE International Symposium on Multiple-Valued Logic*, pages 65–72, 1978.

[138] H. Sawada, S. Yamashita, and A. Nagoya. Restructuring logic representations with easily detectable simple disjunctive decompositions. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 755–761, 1998.

[139] A. Schrijver. *Efficient parallel algorithms*. John Wiley and Sons, 1998.

[140] N. Selvakkumaran and G. Karypis. Multi-objective hypergraph partitioning algorithms for cut and maximum subdomain degree minimization. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 726–733, 2003.

[141] E. Sentovich and D. Brand. Flexibility in logic. In S. Hassoun and T. Sasao, editors, *Logic Synthesis and Verification*, chapter 3, pages 65–88. Kluwer Academic Publishers, 2002.

[142] E. Sentovich, V. Singhal, and R. Brayton. Multiple boolean relations. *Proc. International Workshop on Logic Synthesis*, May 1993.

[143] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.

[144] R. S. Shelar, P. Saxena, X. Wang, and S. S. Sapatnekar. An efficient technology mapping algorithm targeting routing congestion under delay constraints. In *Proc. International Symposium on Physical Design*, pages 137–144, 2005.

[145] W. Shi and Z. Li. A fast algorithm for optimal buffer insertion. *IEEE Transactions on Computer-Aided Design*, 24(6):879–891, 2005.

[146] K. J. Singh and A. Sangiovanni-Vincentelli. A heuristic algorithm for the fanout problem. In *Proc. ACM/IEEE Design Automation Conference*, pages 357–360, 1990.

[147] K. J. Singh, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 282–285, November 1988.

[148] D. Sinha and Hai Zhou. Gate-size optimization under timing constraints for coupling-noise reduction. *IEEE Transactions on Computer-Aided Design*, 25(6):1064–1074, 2006.

[149] S. Sinha, S.P. Khatri, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Binary and multi-valued SPFD-based wire removal in pla networks. In *Proc. International Conf. Computer Design (ICCD)*, pages 494–503, 2000.

[150] F. Somenzi. Cudd: Cu decision diagram package release, 1998.

[151] A. Srivastava, R. Kastner, C. Chen, and M. Sarrafzadeh. Timing driven gate duplication. *IEEE Transactions on VLSI Systems*, 12(1):42–51, 2004.

[152] A. Srivastava, R. Kastner, and M. Sarrafzadeh. Timing driven gate duplication: Complexity issues and algorithms. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 447–450, November 2000.

[153] A. Srivastava, R. Kastner, and M. Sarrafzadeh. On the complexity of gate duplication. *IEEE Transactions on Computer-Aided Design*, 20(9):1170–1176, September 2001.

[154] B. Steinback and A. Wereszczynski. Synthesis of multi-level circuits using exor-gates. In *Proc. of Reed-Muller'95*, pages 161–168, 1995.

[155] L. Stok, M. A. Iyer, and A. J. Sullivan. Wavefront technology mapping. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 108–113, 1999.

[156] C. N Sze, T. C. Yang, and L. C. Yang. Multilevel circuit clustering for delay minimization. *IEEE Trans. on CAD of Int. Circuits and Systems*, 23(7):1073–1085, 2004.

[157] X. Tang, R. Tian, H. Xiang, and D. F. Wong. A new algorithm for routing tree construction with buffer insertion and wire sizing under obstacle constraints. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 49–56, 2001.

[158] X. Tang and M. D. F. Wong. Tradeoff routing resource, runtime and quality in buffered routing. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 430–433, 2004.

[159] R. Tarjan. Finding dominators in directed graphs. *SIAM Journal on Computing*, 3(1):62–89, March 1974.

[160] M. Teslenko and E. Dubrova. An efficient algorithm for finding double-vertex dominators in circuit graphs. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 406–411, 2005.

[161] H. Touati, H. Savoj, and R.K. Brayton. Delay optimization of combinational circuits by clustering and partial collapsing. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 188–191, November 1991.

[162] H. J. Touati, C. W. Moon, R. K. Brayton, and A. Wang. Performance-oriented technology mapping. In *Proc. 6th M.I.T. Conference on Advanced Research in VLSI*, pages 79–97, April 1990.

[163] L.P.P.P van Ginneken. Buffer placement in distributed rc-tree networks for minimal elmore delay. In *Proc. International Symposium on Circuits and Systems*, pages 865–868, 1990.

[164] N. Viswanathan and C.C.-N. Chu. Fastplace: efficient analytical placement using cell shifting, iterative local refinement,and a hybrid net model. *IEEE Transactions on Computer-Aided Design*, 24(5):722–733, May 2005.

[165] M. Vootukuru, R. Vemuri, and N. Kumar. Resource constrained rtl partitioning for synthesis of multi-fpga designs. In *Proceedings of the Tenth International Conference on VLSI Design: VLSI in Multimedia Applications*, page 140, 1997.

[166] Y. Watanabe and R.K. Brayton. Heuristic minimization of multiple-valued relations. *IEEE Transactions on Computer-Aided Design*, 12(10):1458–1472, October 1993.

[167] B. Wurth and N. Wehn. Efficient calculation of boolean relations for multi-level logic optimization. *Proc. European Conference on Design Automation (EDAC)*, pages 630–634, 1994.

[168] J. Xu, X. Hong, T. Jing, L. Zhang, and J. Gu. A coupling and crosstalk considered timing-driven global routing algorithm for high performance circuit design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 677–682, 2004.

[169] S. Yamashita, H. Sawada, and A. Nagoya. New methods to find optimal non-disjoint bi-decompositions. In *Proc. ACM/IEEE Design Automation Conference*, pages 59–68, 1998.

[170] S. Yamashita, H. Sawada, and A. Nagoya. SPFD: A new method to express functional flexibility. *IEEE Transactions on Computer-Aided Design*, 19(8):840–849, August 2000.

[171] C. Yang, M. Ciesielski, and V. Singhal. BDS: A BDD-based logic optimization system. In *Proc. ACM/IEEE Design Automation Conference*, pages 92–97, June 2000.

[172] H.H. Yang and D.F. Wong. Circuit clustering for delay minimization under area and pin constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(9):976–986, 1997.

[173] M.C. Yildiz and P.H. Madden. Preferred direction steiner trees. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(11):1368–1372, November 2002.

[174] J. S. Zhang, M. Chrzanowska-Jeske, A. Mishchenko, and J. R. Burch. Generalized symmetries in boolean functions: Fast computation and application to boolean matching. In *Proc. International Workshop on Logic Synthesis*, pages 424–430, 2004.

[175] J. S. Zhang, A. Mishchenko, R. Brayton, and M. Chrzanowska-Jeske. Symmetry detection for large boolean functions using circuit representation, simulation, and satisfiability. In *Proc. ACM/IEEE Design Automation Conference*, pages 510–515, 2006.