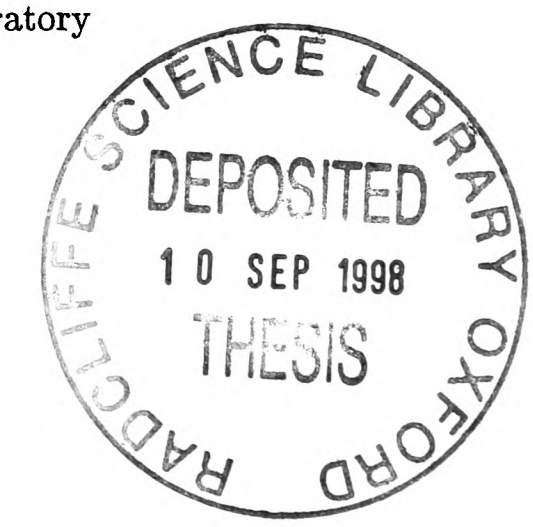
Technology Mapping of Heterogeneous Lookup Table Based Field Programmable Gate Arrays

Maurice Kilavuka Inuani

Thesis submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy at the University of Oxford

Hilary Term, 1998.

Programming Research Group,
Oxford University Computing Laboratory
and
Magdalen College, Oxford.



Acknowledgements

I wish to express my profound gratitude to the following.

Jonathan Saul for his excellent supervision. He provided me with the guidance, suggestions and comments that enabled me to complete the research. I am truly indebted to him for this. I also wish to thank my co-supervisor, Ian Page. I thank Shaori Guo and Dominic Camus for proof-reading the thesis and for their invaluable comments. I am grateful to Michael Zhu Ning for the discussions on bin-packing and "C" programming. Yonghong Bu provided wonderful moral support during the time we shared an office. The Rhodes Trust and the Selection Committee for Kenya sponsored my studies. I also acknowledge support from the Ministry of Education of Kenya. My extended family encouraged and supported me throughout in all possible ways. I give particular mention to Mr and Mrs Kyungu.

I dedicate this thesis to my parents.

Abstract

A lot of work has been done over the last decade on the logic synthesis and technology mapping of field programmable gate arrays (FPGAs) based on a single size of lookup table (LUT). A significant part of the FPGA market is occupied by devices based on more than one type of lookup tables. Examples of these heterogeneous LUT-based FPGAs are the Xilinx 4000 series devices. The technology mapping for this class of FPGAs has hardly been considered. This thesis covers work on the synthesis for heterogeneous LUT-based FPGAs.

The proposed scheme uses the typical steps of graph covering, decomposition, node elimination and Boolean graph simplification. The covering step is based on the concept of flow networks and cut-computation. A theory is devised that reduces the flow network sizes so that a dynamic programming approach can be used to compute the feasible cuts in the network. An iterative selection algorithm can then be used to compute the set cover of the network. For the decomposition, the conventional bin-packing (cube-packing) algorithm has been extended so that it produces two types of bins. It has also been enhanced to explore several packing possibilities and include cube division and cascading of nodes. The classical functional decomposition method is extended to heterogeneous graphs. In particular, variable partitioning is coupled with other decomposition methods and exploits the structure of the functions. Partial collapsing and re-decomposition are used to re-synthesise the graphs. A strategy for eliminating nodes within a heterogeneous graph is developed. A simplification strategy is also derived from logic optimisation techniques.

Comparisons of the mapping results on Xilinx devices show an improvement of over 11% over existing mapping tools for the same devices.

Contents

Chapter 1

Introduction

1.1 Programmable Devices

Integrated Circuits (ICs) have evolved tremendously since their introduction. A significant proportion of ICs are designed for customised use. The modern digital micro-electronics marketplace requires customised devices and short turn-around design times. An important aspect of customised ICs is that their functionality is determined by the designer. This gives them their versatility. It is therefore likely that customised ICs will continue to play a significant rôle in digital micro-electronics. An important segment of the IC market is occupied by ICs that are programmable by the user.

Among the early *programmable logic devices* (PLDs) were programmable logic arrays (PLAs) and programmable array logic (PALs). A PLA or PAL is an array of programmable AND gates whose outputs go to an array of OR gates. So, logic functions are implemented in *sum-of-products* form. The programmability of the AND array in PLAs and PALs means that almost any input combination can be decoded. PALs are limited by their relatively low density. PLAs are useful in state machine and sequencer applications, but they are slower than PALs.

Programmable read-only memories (PROMs) have a fixed AND array and a programmable OR array. The AND array consists of all combinations of input minterms. The OR gates are programmed to select the appropriate AND gate combination. PROMs can be thought of as storing the logic transfer function in look-up tables. One limitation that PROMs suffer from is in the number of inputs that are possible (due to chip area). PROMs and erasable PROMs (EPROMs) are mostly used to store fixed programs and microcode but they can also be used to implement logic. The input is the address and the output is the stored data. To program them a high current is passed through metal or poly-silicon links thus destroying the links. EPROMs are erased using ultra-violet light in contrast to electrically erasable PROMs (EEPROMs).

More advanced programmable logic devices include registered output and even registered

feedback. Macro-cells are used to: select the polarity of each output, register the outputs, select the combinatorial outputs by bypassing the registers, or feed the outputs back into the network. Such devices are ideal for implementing state machines as well as synchronised or pipe-lined combinatorial logic.

Mask programmable gate arrays (MPGAs) consist of thousands of gates with customised wiring. MPGAs can implement larger designs than most PLDs, but the customised wiring requires a manufacturing process. This can be both expensive and time-consuming.

Field programmable gate arrays (FPGAs) have evolved from MPGAs. The general architecture of an FPGA is a rectangular array of functional units interspersed with programmable interconnections. The functional units come in different forms, e.g., multiplexors, lookup tables or logic gates. A designer decides on the distribution of the logical functions among these units. Once the interconnections have been programmed, the FPGA will implement the desired circuit. It is this aspect of the user programming the FPGA "on site" that makes it very attractive to electronics designers.

Since their introduction in the 1980s, FPGAs have grown in importance in the system design field. When the size of FPGAs reached several thousands of usable gates, they became a feasible alternative to small size, low volume *application specific integrated circuits* (ASICs). FPGAs soon overtook mask-programmable gate arrays as prototyping devices [58].

There are several advantages of FPGAs, particularly compared to MPGAs [68]. If an IC design requires less than around a hundred thousand units, then FPGAs are cost effective. Beyond this figure, MPGAs are preferable in terms of cost. Because FPGAs are programmed by the user, they have much faster turn-around times. These two aspects make FPGAs attractive for rapid prototyping and for low-volume production systems. Other advantages are: lower design risk, better verification, lower testing costs, and better price/performance trade-off [68]. Naturally, FPGAs also have disadvantages. For one, the logic density of FPGAs is sacrificed to increase their programmability. The speed is also slower than MPGAs for the same reasons. In 1992, it was projected that with time these disadvantages would become less marked. As the prices of FPGAs fell, it was also projected that the FPGA share of the design market would increase [58].

FPGAs have been employed in a variety of applications. Today, FPGAs are used in system implementation, in "glue logic", and in re-configurable subsystems. Some applications are: network protocols, hardware co-processors, and reconfigurable computing units. In general, FPGAs improve the speed of software applications over more general processors, yet they retain greater flexibility over ASICs.

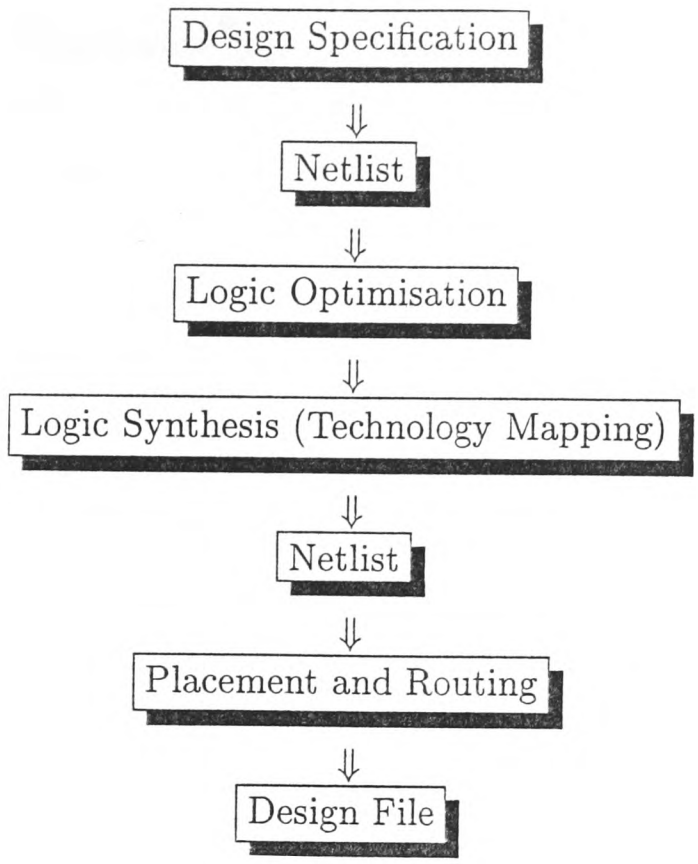


Figure 1.1: A design flow for an FPGA.

1.2 The Design Process

In a general design process (depicted in Figure 1.1), one begins with the circuit specification, perhaps in a hardware description language or through schematic capture tools. From the specification a netlist (a list of logic components) is obtained which describes the logic to be implemented. One may then optimise the logic description. *Technology mapping* binds an optimised representation of the circuit to a set of logic elements, taking into account a set of constraints. Finally, there follows the logic placement and routing of the signals in the FPGA. Clearly, all steps in the design process influence the final result. It could be said that each preceding step directly impinges on the level of difficulty experienced in achieving satisfactory results in the succeeding step.

This thesis addresses the technology mapping phase of the design process because it remains one of the key unsolved problems in the design automation process. Moreover, it provides a reasonable level of abstraction from the technology related issues encountered in placement and routing.

When an IC design is mapped onto an FPGA, several issues have to be addressed. Chief among these is whether the design can fit onto the FPGA. Other issues can be stated in the following questions: Will the logic elements implement the required functions? How many logic elements are required? Are the routing resources sufficient for a given distribution of the logic in the array? What is the maximum signal delay experienced in the circuit? How much power will the final

implementation consume? These issues impose constraints on the technology mapping solution. They can be broadly separated into four categories: *area* — the number of logic elements required; *delay* — the maximum signal delay; *routability* — the possibility of success of the placement and routing phase; and *power consumption.* These measures are often conflicting and thus cannot be optimised together.

Solving the technology mapping problem requires the application of Boolean algebra and graph-theoretic techniques. Any Boolean circuit can be represented by a *directed acyclic graph* (DAG). The nodes in the DAG correspond to the gates of the circuit. Therefore, the nodes have a Boolean function attached to them. The external inputs/outputs are called *primary input/output* nodes. An edge, (v, w), exists in the DAG if there is a directional wire connecting gate v to gate w with the signal travelling in that direction. Usually edges have no further attributes, but one can conceive of attributes representing the length of the wire, its impedance and so forth. Boolean DAGs are very versatile. For instance, the function attached to a node can be as complex as desired. The node could represent a simple gate or an entire sub-circuit. Therefore, the DAG can be used as a compact representation of the circuit. Given this, the DAG can be manipulated in two ways. Firstly, at a node level, each Boolean expression can be manipulated using Boolean algebra. Secondly, graph theory can be applied to the Boolean DAG as to any other graph. Technology mapping transforms the Boolean DAG into one representing a network of logic elements that implement the original DAG.

There are three principal techniques used in technology mapping. One involves simplifying a function by breaking it down into smaller (and simpler) sub-functions. This is called *decomposition.* It has the effect of making it possible to implement the functions in the logic elements and may yield a more economical layout by some measure. A second technique is concerned with assigning logic elements to the nodes. This is called *graph covering* since each node is "covered" by at least one logic element. The correspondence between logic elements and nodes is not one-to-one, and so the covering problem is non-trivial. Different optimisation criteria require different approaches. The third category, called *network simplification*, overlaps the other two. The general idea is to restructure the DAG to a more economical form. This usually involves eliminating some nodes, introducing new ones, and substituting for others. All these issues are addressed in this thesis.

Technology mapping cannot be viewed in total abstraction from the architectural styles of the FPGAs used in the final implementation. There are various architectural styles employed for FPGAs, based on the logic elements, routing resources, and interconnections. One type of logic element is based on the *lookup table* (LUT). A LUT is simply a table in which each entry corresponds to one of the possible input combinations. Therefore, the inputs are used to select one of these values as the output of the LUT. A LUT has the advantage that it can compute any function of maximum of k inputs, where k is the number of inputs to the LUT selection lines.

From a technology mapping point of view, LUT-based FPGAs are much simpler to deal with than FPGAs based on other functional units, e.g., multiplexors. There is no longer any constraint on the nature of the function each logic element covers. However, the number of inputs to each function is still bounded. For a DAG, this constraint corresponds to a limitation on the number of edges incident *to* any node. In the general Boolean DAG, nodes have a varying number of input and output edges. A node with more input edges than the input capacity, k, is said to be *k-infeasible.* Conversely, one that does not violate this constraint is *k-feasible.* The size of the node is defined as the number of edges incident *to* that node, i.e., the number of fanins to the node.

A large part of the FPGA market is occupied by devices based on LUTs. Xilinx pioneered the *logic cell array* (LCA) in which the functional building block is the *configurable logic block* (CLB). A CLB is equivalent to a macro-cell in PLDs and is used for logic function computation as well as for storage. Usually both combinatorial and registered outputs are available. The Xilinx FPGAs use LUTs as their logic elements.

Until recently, the technology mapping problem has involved LUTs of similar size, i.e., *homogeneous* FPGAs. Newer architectures are now *heterogeneous*, meaning that more than one type (in terms of size) of LUT exist within the functional blocks. Examples of such FPGAs in the market are the Xilinx 4000 series FPGAs. They occupy a large and growing portion of this market, and yet technology mapping for this class of heterogeneous architectures has hardly been considered. The Xilinx 4K FPGAs have two types of LUTs within a CLB. These LUTs are connected in a triangular fashion with a hard-wired link between them. An important aspect of this architecture is that the interconnections between logic blocks are programmable and therefore significantly slower than the hard-wired link. Because a substantial part of the delay suffered by FPGAs arises from the programmable links, it can be expected that the hard-wired connection will serve to alleviate this problem. This architecture is described in more detail in the next chapter.

1.3 This Study

The problem covered by this thesis is the production of mappings for circuits targeted to heterogeneous LUTs-based FPGAs. There are two challenges in technology mapping of such FPGAs: the first is to map a general Boolean DAG onto a minimal number of the two types of LUTs; the other is to limit the number of programmable links used in the mapping and thus minimise the delay. The primary objective is to minimise the area; the secondary objective is to minimise the circuit delay. This can be justified by the observation that fewer logic blocks can lead to a more compact layout. This in turn reduces the length of the interconnections and consequently, the circuit delay. Hence, one of the objectives is to maximise the utilisation of hard-wired links and minimise that of the programmable interconnections. Minimising area involves seeking a suitable

mix of the two types of LUTs.

Even though the delay-optimisation version of the homogeneous LUT mapping problem can be solved optimally in polynomial time [14], the area-optimisation version is known to be NP-complete [16, 25, 74, 49]. The heterogeneous LUT mapping problem is a variant of the original problem and can be expected to be as difficult. This means that the computation of an optimal solution in polynomial time cannot be guaranteed. Hence, heuristics will be needed to solve this problem.

The scope of this thesis is technology mapping of FPGAs with heterogeneous logic block architectures. The functional unit is the LUT. Only combinatorial circuits are considered. Issues relating to circuit specification, logic optimisation and, placement or routing are excluded.

The motivation of this work is based on the following observations.

- FPGAs are important devices in IC design. They are projected to increase in importance with time as their density and speed capabilities approach those of semi-custom devices.
- Computer Aided Design (CAD) is now indispensable in modern circuit design. Along with the hardware, the software must be developed sufficiently so as to harness the power of the hardware.
- Design tools for FPGAs play a critical rôle in the efficiency of such devices. In particular, the technology mapping phase is crucial.
- Rarely can a 100% utilisation of logic elements with full routability be achieved with FPGAs. Two ways of overcoming this barrier are, firstly, to improve placement and routing of the logic, and secondly, to generate smaller circuits for the equivalent logic.
- Not enough work has been done on technology mapping of heterogeneous FPGAs. There exists some scope of extensions to techniques used in technology mapping for homogeneous FPGAs as well as departures from these.
- The problem is especially hard for area minimisation. An analysis of the problem, as applied to heterogeneous FPGAs, provides new pointers for future research.

The contribution of this work is to develop an effective and coherent scheme for technology mapping of heterogeneous FPGAs. This is achieved by extending logic synthesis techniques to cover heterogeneous architectures. Improvements to algorithms for decomposition, graph covering and graph re-structuring are developed. The theoretical basis for the algorithms and their application are studied. Algorithms for solving various tasks are devised. These tasks are then harmonised to achieve good overall results. Thus, the thesis forms a basis on which the technology mapping problem for heterogeneous FPGAs can be studied further.

1.4 Overview

Most of the work presented here builds up on work previously done on Boolean DAGs whose node sizes are limited to one particular size. Such graphs are called *homogeneous graphs*. This thesis is concerned with graphs where nodes can be of two types with different size limits. Such graphs are referred to as *heterogeneous* graphs.

The general outline of the thesis is as follows.

1. First, there is an analysis of graph covering based on flow networks and minimum cuts.
2. Then decomposition is examined based on the bin-packing strategy.
3. Next, the partial collapsing of graphs coupled with functional decomposition is examined.
4. Finally, node elimination by selective collapsing as well as graph simplification based on standard logic optimisation techniques is examined.

To begin with, there is a preliminary chapter that covers some background material and a review of the literature. It covers all aspects that are relevant to this work and the most important work within the scope of the thesis. This includes the general synthesis of switching circuits, the logic synthesis for FPGAs, and technology mapping for LUT-based FPGAs.

1.4.1 Covering

Chapter 3 covers material on the mapping of a Boolean DAG onto a set of LUTs. First, the concept of flow networks and its application to the computation of cuts in DAGs is introduced. Flow networks can be used efficiently to discover clusters of nodes of particular sizes within a DAG. Then a theorem is presented that shows how the complexity of computation of the flows (and hence cuts) within the networks can be improved by harnessing some properties of these flow networks. Heterogeneous LUTs impose distinct requirements from those of homogeneous LUTs since the number of logic blocks required depends critically on how the utilisation of the different types of LUTs is balanced. The way in which the DAG, after feasible cuts (clusters) within the DAG have been identified, is mapped onto logic blocks is discussed. An iterative heuristic algorithm is developed based on a cost function for selecting a subset of the identified clusters which then become new nodes. Also shown is a way of limiting the circuit delay during the implementation of LUTs. In summary, the novel work is:

- A theorem is derived for the reduction in the sizes of flow network.
- Flow networks are used to compute heterogeneous cuts and hence clusters.
- An algorithm is presented for solving the covering problem for heterogeneous DAGs.

1.4.2 Bin-Packing Decomposition

Chapter 4 covers the decomposition of infeasible nodes in a DAG. In particular, the application of a bin-packing scheme to decomposition is discussed. Bin-packing involves the modelling of the decomposition process as a packing problem where the node's cubes are the items and the LUTs are bins. Hence, it is sometimes called *cube-packing* decomposition. When applied to homogeneous types of LUTs, cube-packing achieves good results and is one of the quickest decomposition methods, as well as being relatively simple to implement. A way in which the method can be modified to cater for heterogeneous LUTs is shown.

One of the main deficiencies of cube-packing arises when dealing with cubes which have several variables in common. For orthogonal cubes, cube-packing performs as well as any other decomposition method. A way in which these weakness can be remedied by expanding the solution space during packing is shown. By holding onto several options at each stage of the packing, a better solution will be found at the end.

Since bin-packing assumes that all items can fit into a bin individually, the decomposition of single cubes must be addressed. Within a Boolean function, as more variables are shared among the cubes, the greater the number of available packing options. Therefore, the decomposition of one of these cubes should aim to leave the variables which are shared most often in the residual cube. Secondly, common cubes that are a subset of some original cubes can be extracted so that both the original cubes and the node itself are decomposed simultaneously.

A cube-packing based decomposition scheme on a general node for heterogeneous LUTs is developed. The decomposition process can directly lead to a mapping of the logic blocks. Some LUTs are cascaded to reduce the number of logic blocks required. Then individual nodes are mapped onto logic blocks by carefully matching the two types of LUTs.

Cube-packing is coupled with a different decomposition method to improve its effectiveness. The combined methods achieve better results than one of them on its own. Thus, they form a powerful yet quick decomposition scheme that can be used either to decompose nodes or to compute a good upper-bound in the number of nodes needed for other decomposition methods.

The contribution is summarised as:

- Bin-packing is extended beyond the conventional best-fit decreasing (BFD) strategy so that the solution space is expanded.
- Cubes are pre-processed with the extraction of sub-cubes, thus making the parent cubes nearly orthogonal.
- Cascading is applied to nodes so as to reduce the size of the decomposition tree.
- Kernel extraction is applied to the functions concurrently with cube-packing.

1.4.3 Re-synthesis

Chapter 5 deals with techniques for restructuring a DAG in order to reduce the number of nodes needed to make it feasible for mapping. It extends the concept of the classical functional decomposition method to cover heterogeneous nodes. A key factor in functional decomposition is the partitioning of the variables of a functions into two sets. A heuristic method is developed which uses the information on the functionality of a node to derive suitable partitions.

A second aspect concerns the partial collapsing of a DAG. This gives extra freedom to manipulate the DAGs to our advantage by removing some nodes and introducing nodes more suited to the mapping sought. The techniques of Chapter 3 are applied to this end and an algorithm for selecting the nodes to collapse is introduced.

Partial collapsing and decomposition are combined to achieve the re-synthesis of a DAG. The methods for functional decomposition are generally very computationally expensive. Therefore, a scheme is developed in which other more economical decomposition methods are applied to limit the computational effort expended in functional decomposition.

In summary:

- The functional decomposition method is extended to heterogeneous DAGs. In particular, variable partitioning is coupled with other decomposition methods. It exploits the structure of the functions.
- Flow networks are used in the partial collapse of subgraphs.
- Partial collapsing and re-decomposition are used to re-synthesise the DAGs so that smaller DAGs result.

1.4.4 Elimination and Simplification

In Chapter 5, the removal of nodes from the DAG by collapsing them into their fanouts is considered. The nature of the DAGs dealt with here dictates that the choices of nodes to collapse has to take into account the various types of fanout nodes. A greedy strategy, coupled with a good cost function, is developed and used to select a maximal subset of the possible collapsing options. It is shown that this node reduction process can be performed before and after decomposition, and that the variation in node size restrictions gives better results than a fixed approach.

Next, a technique that is often used to simplify DAGs during technology-independent optimisation is discussed. It is shown that a modification of the criteria of simplicity can achieve better results in the final mapping.

The contribution is:

- A strategy for eliminating nodes within a heterogeneous DAG is developed.

- A simplification strategy for a heterogeneous DAG is derived from logic optimisation techniques.

1.4.5 Overall Algorithm and Results

Finally, in Chapter 6 the overall algorithm is presented. This ties together the work presented in the preceding chapters. The results of the unified scheme are also presented. Results for individual algorithms will be presented within their appropriate chapters.

Chapter 2

Background

This chapter covers background material required to understand the original work presented in subsequent chapters. Firstly, the terminology used throughout the thesis is given. Secondly, the theory of flow networks and their applicability to technology mapping is discussed. Then the theory of functional decomposition as applied to Boolean functions is discussed. Finally, a review of the relevant literature is presented.

2.1 Definitions

The following example is used to illustrate the concepts and terminology introduced in this section. For Boolean expressions, the "+" symbol represents disjunction (conjunction is implicit in the expression).

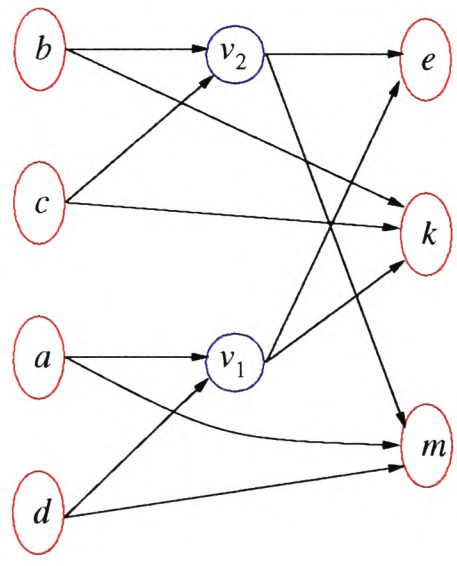


Figure 2.1: An example of a Boolean graph.

EXAMPLE 2.1.1 *Consider a Boolean graph defined by the following set of nodes.*

$$e = v_1 + v_2$$
$$k = v_1 + \overline{b} + \overline{c}$$
$$m = v_2 + a + \overline{d}$$
$$v_1 = a + d$$
$$v_2 = b \cdot c$$

The primary inputs are $\{a, b, c, d\}$ *and the primary outputs are* $\{e, k, m\}$. *There are two internal nodes,* v_1 *and* v_2. *The graphical representation (a directed acyclic graph) is shown in Figure 2.1.*

Boolean graphs are always directed. They are also acyclic, since the corresponding circuits should not have cyclic paths that are not registered. Hence, they are often called Boolean *directed acyclic graphs* (DAGs) or simply DAGs. Whereas many authors use the word "network" to mean DAG, in this thesis this word is used mainly in the context of *flow networks.*

Graphs and Flow Networks: In brief, a *flow network* is a directed graph whose edges have capacities. They also have two distinct vertices: a *source* and a *sink.* The source has no incoming edges while the sink has no outgoing edges. The edge capacities and these two vertices distinguish flow networks from other directed graphs. They are discussed more fully in Section 2.2. In discussing graphs, it is assumed that a graph $G = (V, E)$, where V is the set of vertices and E the set of edges in G. In graph theory, the words "node" and "vertex" are synonymous, as are "edges" and "arcs". The discussions that follow will generally use "nodes" for Boolean graphs and "vertices" for flow networks. The following definitions apply to Boolean graphs and flow networks.

- The fanin set, $\mathcal{F}_{\text{IN}}(g)$, for a node g is its set of input nodes. In the DAG, there is a direct connection from each node in $\mathcal{F}_{\text{IN}}(g)$ to g. Primary inputs have empty fanin sets. In Example 2.1.1, $\mathcal{F}_{\text{IN}}(e) = \{v_1, v_2\}$, $\mathcal{F}_{\text{IN}}(k) = \{v_1, b, c\}$, etc.

- The fanout set, $\mathcal{F}_{\text{OUT}}(g)$, for a node g is its set of output nodes. In the DAG, there is a direct connection from g to each node in $\mathcal{F}_{\text{OUT}}(g)$. In Example 2.1.1, $\mathcal{F}_{\text{OUT}}(b) = \{v_2, k\}$, $\mathcal{F}_{\text{OUT}}(v_2) = \{e, m\}$, etc.

- The *transitive fanin* of a node g is the union of $\mathcal{F}_{\text{IN}}(g)$ and the transitive fanins for each $f \in \mathcal{F}_{\text{IN}}(g)$. Thus, the transitive fanin of a primary input is the empty set. In Example 2.1.1, the transitive fanin of node k is $\{a, b, c, d, v_1\}$.

- The *fanin cone*, C_t, of t which is a subgraph rooted at t and including the transitive fanin of t in its vertex set. The edge set is composed of all edges in the original DAG that connect two vertices contained in the vertex set.

- The *transitive fanout* of a node g is the union of $\mathcal{F}_{\text{OUT}}(g)$ and the transitive fanouts for each $f \in \mathcal{F}_{\text{OUT}}(g)$. However, a subset of the primary outputs may have non-empty transitive fanout sets. In Example 2.1.1, the transitive fanout of node a is $\{v_1, m, k, e\}$.

- *Single fanout* or *fanout-free* nodes are those nodes with the property $|\mathcal{F}_{\text{OUT}}(g)| = 1$. A *tree* or *fanout-free* DAG is one in which each node is fanout-free. A *leaf-DAG* is one in which all nodes except some primary inputs are fanout-free.

- A *feasible* node (or subgraph) is one that has no more than a given bound of inputs (fanin nodes). More accurately, a node v is said to be k-feasible if $\mathcal{F}_{\text{IN}}(v) \leq k$. By extension, a graph is k-feasible if all its nodes are k-feasible.

- Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two flow networks with a source, s, and sink, t. $G_1 \subset G_2$ if:

  1. $V_1 \subset V_2$; and
  2. some paths $s \rightsquigarrow t$ in G_1 are shortened versions of the corresponding paths in G_2, obtained by skipping some edges.

  To shorten a path, some of the nodes are omitted. For instance, let there be a path $u_i \rightarrow v \rightarrow w_j$, where $u_i \in \mathcal{F}_{\text{IN}}(v)$ and $w_j \in \mathcal{F}_{\text{OUT}}(v)$. If the node v is deleted, then new edges (u_i, w_j) for all possible combinations of $u_i \in \mathcal{F}_{\text{IN}}(v)$ and $w_j \in \mathcal{F}_{\text{OUT}}(v)$ are introduced. At the same time, all edges (u_i, v) and (v, w_j) are deleted.

In flow networks, a *cut* is defined as a bisection of the graph. The source is in one half and the sink in the other. Below is a list of definitions which are explained fully in Section 2.2.

$\tilde{\mathcal{N}}_t$ — The flow network formed from $\mathcal{C}_t$; $\tilde{\mathcal{N}}_t = (\tilde{V}_t, \tilde{E}_t)$.

$\breve{\mathcal{N}}_t$ — A modified version of $\tilde{\mathcal{N}}_t$; $\breve{\mathcal{N}}_t = (\breve{V}_t, \breve{E}_t)$.

$\check{\mathcal{N}}_t$ — A partially reduced version of $\tilde{\mathcal{N}}_t$; $\check{\mathcal{N}}_t = (\check{V}_t, \check{E}_t)$.

$\hat{\mathcal{N}}_t$ — A reduced version of $\tilde{\mathcal{N}}_t$; $\hat{\mathcal{N}}_t = (\hat{V}_t, \hat{E}_t)$.

$(X, \overline{X})$ — The bisection or *cut* of a flow network so that the *source* is in X and the *sink* is in $\overline{X}$.

$Cut_A(v)$ — A cut in $\tilde{\mathcal{N}}_v$ of type α.

$Cut_B(v)$ — A cut in $\tilde{\mathcal{N}}_v$ of type β.

u' — The incoming vertex for a node u in a flow network.

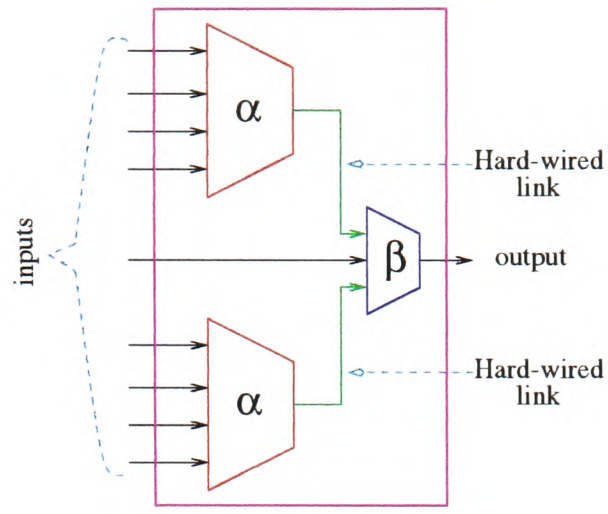


Figure 2.2: A heterogeneous, LUT-based logic block. (Registers have been omitted.)

u'' — The outgoing vertex for a node u in a flow network.

(u', u'') — A bridging arc in a flow network corresponding to a node u.

Heterogeneous FPGAs: The type of FPGA considered here is composed of a rectangular or square grid of identical *logic blocks*. The FPGA is assumed to be regular in structure. There are interconnection resources between these logic blocks, but their nature is not relevant for the purposes of this thesis. Each logic block contains function generators. These are simply LUTs of a fixed number of inputs. The structure of a logic block is shown in Figure 2.2. The inputs to the logic blocks are also inputs to the LUTs. Similarly, the outputs of the logic blocks are outputs of some of the LUTs. Because there are two types of LUTs, the FPGA is referred to as a *heterogeneous* FPGA. Usually, logic blocks contain some sequential circuitry, so that the outputs can be registered. These registers can be by-passed. A more detailed description of such an FPGA can be found in [72]. Xilinx refer to the logic block as a *configurable logic block (CLB)*.

For the purposes of this thesis, one type of the LUT will be called the α-type, which has a maximum of κ_α inputs. The second type will be called the β-type, which differs from the α-type in that two of its inputs are the outputs of two α-type nodes. In addition to which there are $(\kappa_\beta - 2)$ other inputs (from some α or β LUTs). Thus an α LUT can implement any function of 1 up to κ_α inputs, while a β LUT can implement any function of 1 up to κ_β inputs. It is assumed that $\kappa_\beta < \kappa_\alpha$. It follows that an α LUT can implement any function that a β LUT can, but not the converse. The following is a list of definitions that apply to a logic block and nodes in a DAG.

α — The first type of LUT (node) which appears first in the cascade of LUTs within a heterogeneous logic block. This corresponds to an F or G function generator in the Xilinx 4K FPGA.

β — The second type of LUT (node) which receives inputs from LUTs of type α within the

same logic block. This corresponds to an H function generator in the Xilinx 4K FPGA.

κ_α — The maximum input size of α-type LUTs or nodes. This will be assumed to be 4 unless otherwise stated.

κ_β — The maximum input size of β-type LUTs or nodes. This will be assumed to be 3 unless otherwise stated.

κ_{LB} — The input size of a logic block, $2\kappa_\alpha + \kappa_\beta - 2$; it is equal to 9 for the default value of κ_α and κ_β.

δ_{fast} — The delay in the hard-wired link within a logic block.

δ_{slow} — The delay in a connection between logic blocks.

δ_{L} — The delay through a LUT.

A logic block (or *configurable logic block (CLB)*) contains either one, two or three LUTs with a maximum of two α LUTs and one β LUT. Each β LUT has two of its inputs connected to the outputs of an α LUT via a hard-wired link. In addition, it may have more inputs from outside the logic block (since $\kappa_\beta > 2$). All the inputs of the α LUTs are external to the logic block. An important aspect of this architecture is that the interconnections between logic blocks are programmable and therefore significantly slower than the hard-wired link.

In this thesis, *logic elements* are referred to as nodes. Because the logic elements are LUTs, the words "LUT" and "node" are often used interchangeably.

Abbreviations: The following abbreviations apply throughout this thesis.

BDD ... Binary Decision Diagram.

BFD ... Best Fit Decreasing.

CLB ... Configurable Logic Block.

DAG ... Directed Acyclic Graph.

FFD ... First Fit Decreasing.

FPGA ... Field Programmable Gate Array.

LUT ... Look-up Table.

MSD ... Maximum Sharing Decreasing.

OBDD ... Ordered Binary Decision Diagram.

ROBDD ... Reduced Ordered Binary Decision Diagram.

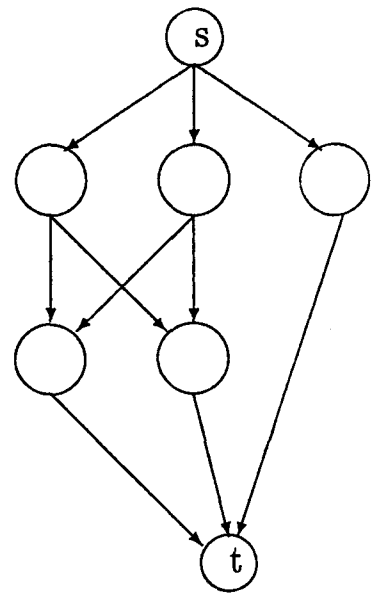


Figure 2.3: An example of a flow network

2.2 Background on Flow Networks

In this section, flow networks are introduced. Flow networks are used in standard graph-theoretic techniques to reason about certain problems. The maximum-flow problem is discussed as well as the algorithms used to solve it. This problem is then related to technology mapping.

A *flow network*, $G = (V, E)$, is a directed graph with two distinguished vertices, the *source* s and the *sink* t. Flow networks have special qualities that distinguish them from other directed graphs. Firstly, a flow network is associated with a *capacity function* $c : V \times V \rightarrow \mathcal{R}^+$. This is a function of the edges, i.e., each edge has a "capacity". One can visualise a flow network as a network of pipes connecting reservoirs. The name "flow networks" derives from the idea of fluid flowing through such "pipes". All the flow emanates from the source and eventually ends up in the sink. Both the source and sink are considered to be infinite. The capacities of the "pipes" impose some constraints on the overall flow through the network.

The capacity function is defined over all edges $(v, w) \in E$. If (v, w) is not an edge of G, then $c(v, w) = c(w, v) = 0$. The following five conditions apply to any legal flow, where $f(v, w)$ is the flow along the edge (v, w).

1. *Capacity Constraint:* $f(v, w) \leq c(v, w)$, i.e., no flow exceeds the capacity of the edge.
2. *Skew Symmetry:* $f(v, w) = -f(w, v)$.
3. Conservation of Flow: $\sum_{w \in (V - \{s,t\})} f(v, w) = 0$.
4. $f(s, w) \geq 0$.
5. $f(w, t) \geq 0$.

Descriptions of flow networks can be found in various books on algorithms, such as [2, 21, 50].

The flow f in the network is defined as

$$f = \sum_{w \in V - \{s,t\}} f(s,w).$$

For the purposes of this thesis, only integral flows are considered, so that $c : V \times V \rightarrow \mathcal{Z}^+$. The following property is assumed for G.

PROPERTY 2.2.1 *For any $w \in V - \{s,t\}$, there exists a path from the source to the sink passing through w, i.e., $s \rightsquigarrow w \rightsquigarrow t$.*

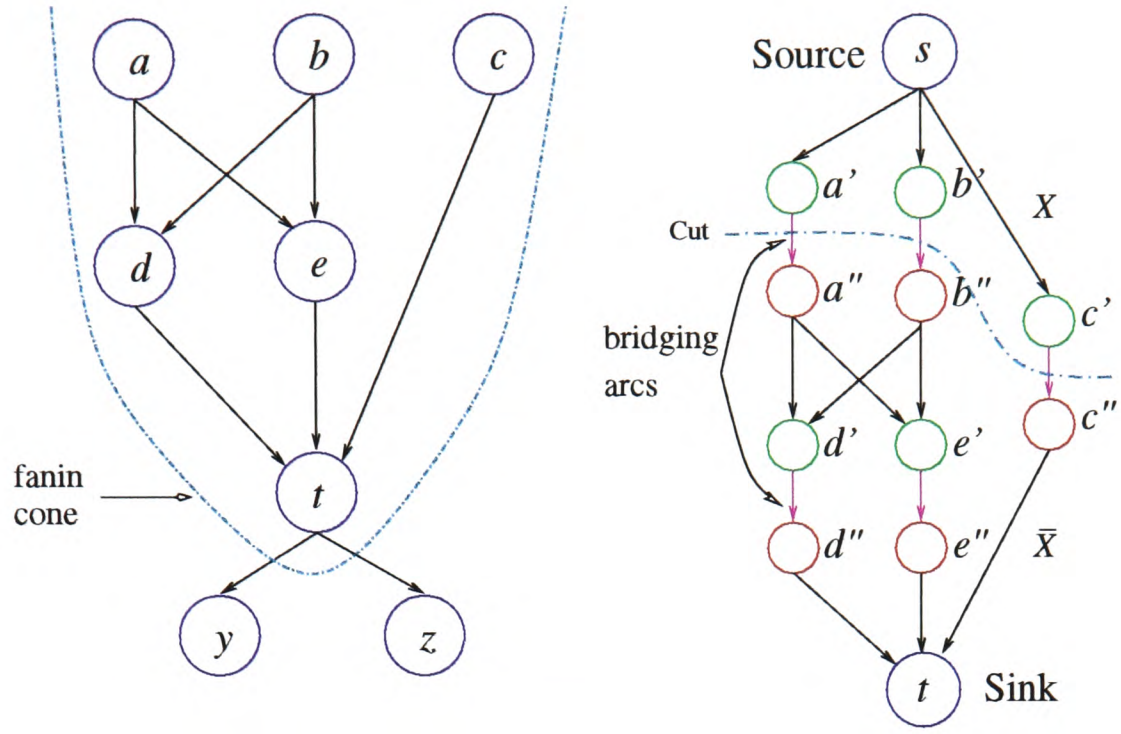
The import of this property is that $|E| \geq |V| - 1$. Figure 2.3 shows an example of such a network.

The flow network is useful to the technology mapping problem because with it a partition of the vertices in the network into two subsets can be identified. Such a partition, $(X, \overline{X})$, is known as a *cut*, where $s \in X$, $t \in \overline{X}$ and $\overline{X} = V - X$. The sum of the capacities of the arcs originating in X and ending in $\overline{X}$ defines the capacity of the cut, $C(X, \overline{X})$. In terms of fluid flows, this may be visualised as a physical cut across the network. Thus, $C(X, \overline{X})$ defines the *net* amount of fluid crossing from X to $\overline{X}$, if the pipes where filled to capacity.

This leads to the question: *what is the maximum rate at which the fluid may be pumped from the source without exceeding the capacities of any pipes*? The flow will have to reach an equilibrium point at the maximum rate. The answer can be found by measuring the flow *entering* the sink or *leaving* the source. The second question is then: *what is the relationship between the maximum flow and the capacity of any cut that can be made in the network*? The remarkable fact about such flows is encompassed in the *Max-flow Min-cut* theorem [2, 21, 50]. In essence, this theorem states that the maximum flow in the network is equal to the capacity of the minimum cut. It establishes the duality between flows and cuts. Consequently, flows can be used to determine the minimum cut in a graph.

Quite often in technology mapping one wishes to establish what the least number of inputs a node would have if a part of its transitive fanin were collapsed into it. Collapsing a node u into one of its fanouts v simply means subsuming the functionality and variables of u into v. Thus, the function of v would no longer depend on u and the edge (u, v) disappears from the DAG. Collapsing nodes in this way can reduce the number of edges incident to a given node. Since the nodes are implemented as LUTs, the number of edges is the only useful criterion. With a flow network, the question of the minimum input size can be answered. Moreover, the flow network can also be used to establish the fact that a node must have more than a given number of edges incident to it regardless of which transitive fanin nodes were collapsed into it.

Murgai et al. [52] suggested a scheme where flows are used to enumerate the set of *super-nodes* in the graph. A super-node is a subgraph of the DAG rooted at a given node. From these super-nodes, a few are selected to cover the entire graph. More recently, Cong and Ding [14] used a



(a) Boolean Subgraph (b) Flow Network

Figure 2.4: (a) The transitive fanin cone of t, C_t; (b) the transformed network $\tilde{\mathcal{N}}_t$ with bridging arcs of unit capacity and all others of infinite capacity.

flow network method to compute depth-optimal mappings for Boolean graphs with homogeneous nodes. The approach adopted here is very similar to both these works except that two types of super-nodes, or equivalently LUTs, are sought. The solution is also area-optimal rather than depth-optimal.

The following sections describe briefly the techniques used in the application of the network-flow method to technology mapping. More detail can be found in [2, 50, 14].

2.2.1 Transforming a Graph

In technology mapping, a flow network is generally used to determine if and where feasible subgraphs rooted at a particular node exist. Such a subgraph can be collapsed into a single node. The nodes within the subgraph are referred to as a *cluster*.

A Boolean DAG cannot in itself be a flow network. In order to model the technology mapping problem as a network flow problem, the DAG must be transformed into a flow network. This transformation applies to a given node t and its fanin cone, C_t. So, only one node may be examined at a time. Different nodes lead to different flow networks.

The fanin cone incorporates the transitive fanin of t and has the following properties:

- it is a directed graph, T, with a root t such that all the arcs are directed towards t; and
- if v is a vertex of T, then $\mathcal{F}_{\text{IN}}(v)$ is a subset of the vertices of T.

Clearly, the leaf nodes of T are the primary inputs in the transitive fanin of t.

A flow network can be constructed from T by first introducing a source s and connecting it to all the primary inputs in T. Then t is designated as the sink. If all the edges are assigned capacities, the result is a flow network. But as it stands, it cannot be used to compute the subgraphs required from T.

From a technology mapping viewpoint, it is useful to know the subgraph rooted at t and within T with the smallest fanin size since it may be mapped into a single LUT more efficiently. To find it, the flow network may be employed. However, the minimum cut in a flow network is defined as an *edge capacity* constraint. In the context of the fanin size, the edge capacities are not important. Instead, it is the number of vertices at the boundary of the subgraph that matters. The number and nature of edges incident to or from these "border" vertices is irrelevant. Therefore, the problem addressed here differs from the traditional edge-cut problem; it is in effect a node-cut problem. This leads to a further transformation of the original flow network.

A routine transformation of the vertices in the flow network is to split them into two. A vertex u (other than s and t) becomes two vertices u' and u''. Then a *bridging arc* (u', u'') is introduced. Thus each original arc (v, w) is relabelled as (v'', w'). All bridging arcs have unit capacity, and the others have any capacity greater then 1, say infinite capacity. This transformed network is denoted as $\tilde{\mathcal{N}}_t = (\tilde{V}_t, \tilde{E}_t)$ and is illustrated in Figure 2.4. The theory behind this transformation is discussed by Ahuja et al. [2] and was used in the FlowMap algorithm [14] and its variants.

With the transformed flow network $\tilde{\mathcal{N}}_t$, a flow will saturate the bridging arcs only. It is now possible to find the border vertices for a given maximum flow in the flow network as explained below.

2.2.2 Cuts in Flow Networks

A cut $(X, \overline{X})$ is essentially a bisection of a flow network such that X contains the source and $\overline{X}$ contains the sink. The cut crosses the edges that together constrain the maximum flow. These edges are therefore completely saturated. In order to compute the cut, it is necessary to define a *residual network* for the original flow network.

A residual network, $G_f = (V, E_f)$, is induced by a flow network, $G = (V, E)$, and a given flow f. It only has the edges that can admit more net flow [21]. The *residual capacity* of an edge (u, v) is given by

$$c_f(u, v) = c(u, v) - f(u, v).$$

Therefore, the edge set is

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}.$$

Due to the skew symmetry property of flow networks, $E_f \not\subset E$, since for any edge $(u,v) \in E$ with $f(u,v) > 0$, then $(v,u) \in E_f$ even if $(v,u) \notin E$. The capacity of a *reverse edge* $(v,u) \notin E$ is defined as zero. For example, let $f(u,v) = c(u,v) > 0$. Therefore, $c_f(u,v) = 0$ and $(u,v) \notin E_f$. On the other hand, since $f(v,u) = -f(u,v) < 0$, then $c_f(v,u) = 0 + f(u,v) > 0$. Hence, $(v,u) \in E_f$.

Hence, for a cut $(X, \overline{X})$, the vertices in X are defined as those vertices in the residual network that are reachable from the source. Figure 2.4 includes a cut $(X, \overline{X})$ across the bridging arcs for $\{a,b,c\}$. A breadth-first search starting from the source is used to identify X [21, 50, 2]. The running time for the breadth-first search (BFS) algorithm is $O(|V|+|E|)$. The algorithm is called FIND-CUT.

The *node cutset* is defined as all nodes, v, where $v' \in X$ and $v'' \in \overline{X}$ [14]. For instance, in Figure 2.4, the node cutset is $\{a,b,c\}$. An interesting aspect of the cut computation is that the search finds the minimum cut that maximises the size of $\overline{X}$. If $\tilde{\mathcal{N}}_t$ has more than one cut, then any one of them partitions $\tilde{\mathcal{N}}_t$ into two parts. A search from s thus finds the first cut and hence the size of $\overline{X}$ is maximised. The size of $\overline{X}$ is referred to as the *volume* of the cut $(X, \overline{X})$. A larger volume cut corresponds to a larger subgraph, rooted at t, with a minimum fanin size.

Before a cut can be computed, the flow network has to be saturated. The following section describes how to compute a flow in such a network.

2.2.3 Computing a Maximum Flow

Various methods exist for computing maximum flows [21]. They can be divided into two categories. In the first type, one iteratively looks for augmenting paths from the source to the sink. An *augmenting path* is a simple path from the source to the sink in the residual network. At each augmentation, the network is transformed into a residual network in which forward arcs of residual capacity zero are discarded and reverse arcs of capacity zero are introduced where necessary. These reverse arcs allow flows in the forward arcs to be cancelled in appropriate situations, thus effecting a back-tracking capability. The maximum flow occurs when no further augmenting path can be found for the residual network.

The second type of algorithms use a *gravity* or *pre-flow* concept. Here the vertices are processed one at a time. The local arcs are initially saturated so that the conservation of flow is violated. The flow is then pushed downwards to the sink until no more flow can be pushed further. At this stage the flows are cancelled to achieve a legal flow which corresponds to the maximum flow.

Of the existing algorithms, the gravity method ones are the fastest ones. However, the nature of the augmenting path algorithms recommends them in preference to the gravity ones in solving the technology mapping problem. The gravity algorithms must run to completion before it is known if the maximum flow, and hence the minimum cut, exceeds a given value k. With the augmenting path algorithms, if $(k+1)$ augmenting paths are found, it can be deduced that the

minimum cut is greater then k. Hence, the computation can be aborted well before the maximum flow is actually computed. In the next chapter, it will be shown how an augmenting path algorithm achieves a faster computation of a feasible, minimum cut in a flow network.

Dinic's Algorithm: Dinic's algorithm [50] is a simple and efficient augmenting path algorithm. Its total running time is $\Theta(|A| + \sum_{i=1}^{m} l_i)$, where $|A|$ is the number of arcs, l_i is the length of the i^{th} augmenting path, and m is the number of augmenting paths. In the worst case, it takes $\Theta(|V|^2 \cdot |A|)$, since l_i is bounded by $|V|$ and m by $|A|$. The modified form examines k paths only. Hence the running time for this new algorithm is $\Theta(k \cdot |A|)$. The function implementing Dinic's algorithm is denoted by MAX-FLOW(G, s, t, k). This function returns the amount of flow that G admits as long as it does not exceed k. Hence a value of $(k + 1)$ indicates that the computation of flows was abandoned after $(k + 1)$ augmenting paths were found in the residual network of G.

2.3 Classical Methods in Functional Decomposition

The section deals with the decomposition of a Boolean function using a well established functional technique.

2.3.1 Roth-Karp and Curtis Decomposition

The seminal work on the functional decomposition of Boolean graphs is due separately to Curtis [22] and Roth and Karp [57]. They derived the same conditions for the existence of decompositions. The difference lies in their methods for determining the decompositions.

Let a function f depend on the variables $X = x_1, \ldots, x_n$. Then f has the functional form

$$f(x_1, \ldots, x_n) = g(\omega_1(A), \ldots, \omega_t(A), B), \tag{2.1}$$

where $A \cup B = \{x_1, \ldots, x_n\}$ and ω_i are sub-functions of f for $i \in \{1, \ldots, t\}$. The function g is called the *composition function* and the sub-functions ω_i are called the *decomposition functions.* The variable set A is called the *bound set* (or *lambda set*) and B is called the *free set.*

Consider two vertices in the Boolean space a^1 and a^2 composed from the coordinates in A. Let b be a vertex in the Boolean space composed of the elements in B. If for all b such that $f(a^1, b)$ and $f(a^2, b)$ are defined and $f(a^1, b) = f(a^2, b)$, then a^1 is said to be *compatible* to a^2 with respect to the reference function f. If f is total, then compatibility is the *equivalence relation* and determines a partition of A into *equivalence classes* [57, 40]. All members of an equivalence class are mutually compatible, but they are *incompatible* with members of any other class. Such a decomposition is said to be *strict.* The decomposition is *disjunctive* if $A \cap B = \emptyset$; otherwise it is *non-disjunctive.* The lower bound of t is set by the number of mutually compatible elements

into which the vertices determined by A may be partitioned, with respect to f. A decomposition is *non-trivial* if g has fewer variables than f.

EXAMPLE 2.3.1 *Consider the following Boolean function, f, of five variables, $\{v_0, \ldots, v_4\}$.*

$$f = v_0v_1v_2v_3 + v_0v_1v_2v_4 + v_0v_1v_3v_4 + v_0v_2v_3v_4 + v_1v_2v_3v_4$$

Suppose that the bound set $A = \{v_0, \ldots, v_3\}$ and the free set $B = \{v_4\}$. A disjunctive decomposition in the form of Equation 2.1 is

$$\omega_1 = v_0v_1v_2\overline{v_3} + v_0v_1\overline{v_2}v_3 + v_0\overline{v_1}v_2v_3 + \overline{v_0}v_1v_2v_3$$

$$\omega_2 = v_0v_1v_2v_3$$

$$g = \omega_1\overline{\omega_2}v_4 + \overline{\omega_1}\omega_2$$

The above example shows that a function of five variables can be decomposed into sub-functions of four or less variables. For a non-trivial decomposition, the composition and decomposition functions must have fewer variables than the original function. This means (referring to Equation 2.1) that $t + |B| < n$ and $|A| < n$. One of the most difficult tasks in this decomposition is the selection of the free and bound sets. The configuration of these sets has a considerable influence on the final decomposition.

2.3.2 Other Methods

The Roth-Karp method for functional decomposition can be very expensive in terms of memory requirement. Generally, it is only used for functions with relatively few variables. Binary decision diagrams (BDDs) have been used to speed up the Roth-Karp algorithm [44]. The variables are ordered so that all variables in the free set come after those in the bound set. Hence, an examination of the cut will determine what type of decomposition the function admits for the given bound set. This method, using variants of BDDs, has been adopted by other researchers [43, 47, 71, 70, 67]. In the discussions in later sections, no distinctions are made between *BDDs, ordered BDDs (OBDDs)*, and *reduced OBDDs (ROBDDs)*. They are all referred to as BDDs in short.

An extension to this was suggested by Stanion and Sechen [67]. Their approach is a branch and bound which involves enumerating all minimal cuts in the BDD. Each cut has a cost associated with it. Therefore, sub-optimal partial cuts can be discarded before they are complete. Nevertheless, the procedure may sometimes have to be terminated before all cuts are accessed. An important difference with the Roth-Karp method is that the size of bound set is not restricted to a constant value k. The only restriction is the ordering of the variables. This method achieved an improvement of 46% over the Roth-Karp method used in SIS [67, 64].

Most researchers concern themselves with disjoint decomposition, since it is simpler and, in theory, more economical. However, many functions do not admit a disjoint decomposition regardless of the chosen variable partition. So a recourse to non-disjoint decomposition is necessary and sometimes even leads to a better overall results than a serial disjoint decomposition. Among the researchers that have used the non-disjoint decomposition approach are Rawski et al. [56] and Lai et al. [44, 43].

Murgai et al. showed that the selection of composition and decomposition functions can be formulated as an encoding problem [54]. In Equation 2.1 the decomposition maps vertices in $\mathbf{B}^{|A|}$ to vertices in $\mathbf{B}^t$. Murgai et al. reduce this mapping to an input encoding problem, where compatible vertices are assigned the same code. Unused codes are used as "don't cares" to simplify the composition function. The overall aim is to find the simplest sub-functions. Compared to "serial" encoding, this approach achieved an improvement of around 16% for the mapping of Xilinx 3K series FPGAs in terms of the number of CLBs used.

The problem of selecting a good bound set remains largely unsolved. Shen et al. use heuristics to pick this set based on the covers of the function [65]. They start with an empty set bound set and progressively add variables to it. A weight function is used select each variable in turn so that the number of decomposition sub-functions is minimised. This method achieved a 29% improvement over the Roth-Karp implementation in SIS, where the bound set is chosen almost randomly [64].

2.4 Technology Mapping

This section reviews the relevant work that has been done on the technology mapping of LUT-based FPGAs. The work has been categorised into groups with common methods or objectives.

The processes covered by technology mapping can be separated broadly into three areas:

1. *Node decomposition* — expressing a function in terms of some sub-functions so as to reduce the number of variables (inputs) it supports. Section 2.3 discusses this process for a particular method.

2. *Graph covering* — mapping the functional unit onto the nodes in the Boolean DAG with the aim of satisfying a set of constraints and optimising given measures. For instance, in Example 2.1.1 the two internal nodes and three primary output nodes can be covered by two- and three-input LUTs.

3. *Network simplification* — reducing the complexity of a DAG by collapsing or substituting some nodes into one another. This process is discussed in Chapter 5.

In this thesis, technology mapping is taken to encompass these three areas; other researchers

restrict it to graph covering. Two excellent reviews of the general area of logic synthesis for LUT-based FPGAs can be found in [17] and [59].

A good deal of work has been done by other researchers on the technology mapping of homogeneous FPGAs, but very little work has done on heterogeneous FPGAs. Most authors have concentrated on one of the objectives of technology mapping, that is: delay minimisation, area minimisation, or mapping for placement. Some have attempted to work out trade-offs between the competing objectives. In brief, these objectives can be described as follows.

1. *Area*: Use as few of the functional units as possible. This is important for large circuits that might not otherwise fit onto the devices.

2. *Delay*: Minimise the longest delay for a signal to propagate from a primary input to a primary output. The longest delay determines the speed on the circuit and hence the maximum frequency at which it will operate.

3. *Placement*: Make the placement of the final layout easier or more efficient. Sometimes circuits that satisfy the area constraint cannot be placed and routed due to the limitation on routing resources. If the circuit were restructured, it may be possible to do the placement and routing.

4. *Power*: Reduce the power consumption of the circuit. This is important for portable computing where the battery life is an important issue.

Labelling: Algorithms tailored for delay-optimisation often include a labelling phase. Each node is assigned a label corresponding to its level in the DAG. Primary inputs are at level 0. Any other node has a level which is at least as high as that of its highest level fanin. In most cases, the level of a node is equivalent to the maximum of number of edges in any path from the primary inputs to the node. Different tools use different labelling algorithms. The important criterion is the final highest label, after processing, among the primary output nodes.

2.4.1 Complexity Issues

Farrahi and Sarrafzadeh proved that the area minimisation problem for FPGA technology mapping is NP-complete, at least for an input bound greater than or equal to five [25]. Zhang et al. [74] and Levin and Pinter [49] show that the problem is also NP-complete for input bounds of three and four. This result means that there is unlikely to be an efficient polynomial algorithm to solve this problem optimally. However, by restricting the circuits to directed trees, the problem can be solved optimally.

2.4.2 Decomposition Methods

The following definitions are often used in logic synthesis and are taken from [4]. They will be used throughout the thesis.

Variable – A symbol representing a single coordinate of the Boolean space.

Literal – A variable or its negation, e.g., a or $\overline{a}$.

Cube – A set C of literals such that $x \in C$ implies that $\overline{x} \notin C$. Another definition of a cube is a tuple composed of an ordered sequence of 0's, 1's, and x's (for the "don't care" condition). A cube represents a conjunction of literals. Such a cube covers a set of product terms derived from the variables represented by the tuple. For instance, if the variables are (v_1, v_2, v_3), then $10x$ is a cube that covers the set $\{v_1\overline{v}_2v_3, v_1\overline{v}_2\overline{v}_3\}$.

Expression – A set of cubes. The expression represents the disjunction of its cubes. For example, $a + b\overline{c}$ is an expression over the cubes a and $b\overline{c}$. In a *Boolean expression* no cube properly contains another. For example, the cube a contains all cubes with the literal a, e.g., ab, $a\overline{b}$, abc, etc.

Cube-free – An expression is *cube-free* if no cube divides it evenly, e.g., $ab + c$ is cube-free but not $ab + bc$.

Primary divisors – If an expression is divided by a cube, the quotient is a *primary divisor.*

Kernels – The cube-free primary divisors.

Co-kernel – A cube used to obtain a kernel. For example,

$$x = (a + b + c)(d + e)f + g,$$

has a kernel $(a + b + c)$ corresponding to the co-kernels df and ef.

Co-factor Packing

Co-factoring decomposition involves expressing a Boolean function $f(X)$ as

$$f = x \cdot f_x + \overline{x} \cdot f_{\overline{x}} + f_{dc},$$

where $x \in X$, $f_x = f/x$, $f_{\overline{x}} = f/\overline{x}$ and f_{dc} is a don't-care co-factor of f with respect to x (the symbol "/" means substitution of the argument with "1" into the Boolean expression). Thus, each cube of f can be assigned to one of these sub-functions depending on the phase of x in the cube. Co-factoring can be viewed as the conversion of a decision diagram to a DAG [17]. In this case, x is the decision variable at a node.

This is the method was used by Park et al. [55]. They chose a co-factoring variable that led to sub-functions of minimal fanin sizes. Using a recursive application of this scheme, they generated a co-factor decomposition tree for the function. They then traversed the tree and packed the nodes into LUTs. When incorporated into the MIS mapping routines, the co-factoring packing method of Park et al. improves the LUT count by about 7%.

Kernel Decomposition

In kernel decomposition, one seeks good kernels to extract from the expression. The extracted kernel forms a new node in the DAG and the original function is expressed in terms of this new node.

EXAMPLE 2.4.1 *Consider the function x below.*

$$x = (a+b+c)(d+e)f + g.$$

Let p correspond to a kernel of x, $(a+b+c)$, and y be the result of substituting for p into x. Then,

$$y = (d+e)fp + g.$$

Since $|\mathcal{F}_{\text{IN}}(y)| < |\mathcal{F}_{\text{IN}}(x)|$, then the effect is to decompose x.

The work of Brayton and McMullen on factorisation and decomposition was targeted to PLAs and not FPGAs [3]. Nonetheless it is applicable to FPGA design. The essential idea is to seek common subexpressions (but not their negations) among a set of functions using algebraic techniques. Having pulled out the subexpressions (which consist of more than one cube), the residual expressions become relatively kernel-free (two expressions are relatively kernel-free if their only common divisors are cubes). Single common-cube divisors can then be sought.

The process of identifying common subexpressions is done as follows. Firstly, all kernels for each function are generated. Then a pairwise intersection of the kernels is computed and the pair that reduces the literals in the functions most is substituted into the functions. A similar pairwise intersection of cubes is also performed to identify common sub-cubes.

Roth-Karp Decomposition

The Roth-Karp decomposition is discussed in Section 2.3. This section describes enhancements to the original method.

Jiang et al. solve the variable partitioning problem using the a BDD [38]. As Lai et al. showed [44], the Roth-Karp decomposition can be effected by computing cuts in a BDD. The order of the variables in the BDD has a direct effect on the size of the cuts in the BDD. Therefore, it is

important to make the "bottle-neck" in the BDD as small as possible, since this determines the number of decomposition functions required. To determine a good variable ordering, and hence partition, Jiang et al. evaluate the number of branches that can be deleted by selecting an ordered pair of variables. They enumerate all possible pairs and then select the variable that gives the greatest benefit. This variable is placed at the head of the ordering. The process is repeated for the remaining variables based on the partial ordering at each stage.

The actual bound-set selection (λ-selection) is accomplished using a cost function that takes into account both the number of decomposition functions needed and the number of remaining input variables of the initial expression. Even though the target is five-input LUTs, λ-sets of size four and six are also considered with the aim of minimising the cost function. Exact selection algorithms are applied to nodes with ten or less inputs.

For five-input homogeneous DAGs, the results of Jiang et al. are 42% better than those obtained using the Roth-Karp decomposition of SIS. The heuristic and exact λ-selection algorithms have a demonstrable effect on the decompositions.

Huang et al. devised an encoding algorithm so that the decomposition sub-functions are independent of at least one of the bound set variables [35]. This enables two such sub-functions to be implemented in two LUTs of a Xilinx 3K CLB. The results of this algorithm compared to the one used in SIS are 41% better for two-level circuits. In multi-level circuits, the lambda selection algorithm [65] plays the most important rôle in the decomposition.

AND-OR Decomposition

This form of decomposition is applied to the *sum-of-products* form of a function. The node is broken up into a set of AND gates, OR gates and inverters. Usually the gates are limited to two-input gates. This decomposition method is rather simple but not as efficient as other methods. Its main use is in preparing a DAG for a packing or covering phase.

Bin-Packing

If the cubes in a node are considered to be items, then they can be assembled into a collection of nodes which, when connected together, implement the original function. Providing all the cubes have less than m literals, then they can be packed into m-input nodes. The resulting subgraph is then m-feasible. The process of packing the cubes or nodes can be modelled as the classical bin-packing problem. There are various algorithms for solving this problem. The aim of the solution is to minimise the number of "bins" (new nodes) used. This type of decomposition is addressed fully in Chapter 4.

Disjoint Decomposition

This decomposition is quite similar to *cube-packing*. Basically, the cubes are partitioned into sets with mutually disjoint variable sets. Each set of cubes forms a node. The final node is a disjunction over all the partition nodes. Not all nodes have disjoint cube partitions. Therefore, this decomposition does not always work. However, it can be very beneficial if applied as a precursor to some other decomposition method.

2.4.3 Packing Methods

The most common packing method is based on the bin-packing paradigm. If the cubes in a Boolean expression are modelled as the items, then the effect is to decompose node functions. New nodes are introduced for the original node. On the other hand, if the items are entire nodes, then the effect is to cover the DAG. Nodes are grouped together so that the number of nodes actually decreases. If it is assumed that a set of items is sorted in decreasing order of their weights, then there are three main packing strategies.

1. *First-fit decreasing* (FFD) - packs an item into the first bin in which it fits.
2. *Best-fit decreasing* (BFD) - packs an item into the "best" bin in which it fits.
3. *Maximal-sharing decreasing* (MSD) - packs items together if they maximally share a given attribute.

Area Minimisation

The *Chortle* algorithms were some of the first algorithms to be developed for LUT-based FPGAs. They use dynamic programming techniques and bin-packing heuristics. They also work on trees rather than DAGs. Any DAG can be converted into a forest of fanout-free trees. The idea in Chortle [29] was to compute mappings for trees independently. It does this by forming clusters in the tree so that the tree is completely covered. The overall technology mapping solution thus obtained is an amalgamation of the individual mappings of these trees. Chortle-crf [30] also uses a dynamic programming approach. The main difference with Chortle was that it used a bin-packing technique in node decomposition. The cost function employed minimises the number of LUTs in the first instance, and then maximises the number of unused inputs.

TreeMap by Farrahi and Sarrafzadeh is a (feasible) tree mapping tool [25]. TreeMap begins by computing the levels for each node. It then processes the nodes on a level-basis starting from the primary inputs. In essence, the algorithm maintains two types of LUTs: "closed" and "open". A closed LUT admits no further nodes and is rooted at a particular node. As the algorithm progresses, it tries to fill the open LUTs as much as is possible. To do this, TreeMap defines a

measure d_v (the dependency of v) which is the sum of the total number of inputs to the open LUTs and the number of closed LUTs rooted at the fanins of v. If $d_v \leq k$, then the LUT rooted at v subsumes all the open LUTs rooted at its fanins. Otherwise, the largest of the open LUTs is repeatedly closed until d_v satisfies the condition. At the end of TreeMap only the closed LUTs are implemented. The TreeMap algorithm is unlikely to be of much practical use, given the fact that most circuits are not trees but DAGs. Consequently, Farrahi and Sarrafzadeh also developed a heuristic algorithm for DAGs, called Level-Map. The essential difference between TreeMap and Level-Map is in the way the open LUTs are selected for closing. Level-Map employs a priority function based on the size as well as the number of fanouts of the open LUTs.

Levin and Pinter solve the covering problem for trees using a dynamic programming technique similar to the one used in Chortle-crf [49]. The tree is traversed starting from the leaves and each covering node is maximally utilised. Infeasible nodes are decomposed as the mapping is being done using a disjoint or co-factoring decomposition method. For general DAGs, they use heuristics that identify feasible clusters and select them based on their input sizes and volumes.

Delay Minimisation

Among the packing-based algorithms for depth-optimisation is Chortle-d which is a variant of Chortle and Chortle-crf [31]. In DOGMA, the DAG is first decomposed so that each internal node has only got two inputs [20]. The approach in DOGMA is similar to that of Chortle-d. The nodes are grouped in ascending order of their labels. The packing strategy is either the BFD, FFD or MSD strategy. In the MSD strategy nodes are packed together if they maximally share a cut. A fourth method partitions the nodes with the same label into subsets so that each subset has a feasible cut. The cuts are computed using the flow network technique. DOGMA is meant to be used together with other tools.

2.4.4 Mixed Techniques

Some tools use a variety of techniques, especially in the decomposition stage. The most prominent of these are the *MIS-pga* algorithms.

Area Minimisation

Murgai et al. developed Mis-pga and Mis-pga$_{new}$ for area-optimisation [51, 52]. As in Chortle-crf, Murgai et al. used a bin-packing approach. However, they considered the cubes of a function to be the boxes and the logic blocks to be the bins. A second decomposition technique used is *co-factoring*. The general idea of Mis-pga$_{new}$ is to collapse nodes into their fanouts and then apply some or all of the decomposition schemes and pick out the best result.

Hydra seeks to use two LUTs in each Xilinx 3K series CLB [27]. This means that the LUTs must have at most four rather than five inputs each. In the synthesis process, DAGs are first decomposed into an AND/OR form. Then a *shared-input graph* is constructed with the same vertex set as the Boolean DAG but whose edges have weights that correspond to the number of common fanins between the adjacent vertices. From it a decomposition is sought for the nodes with the greatest degree of shared fanins. The AND/OR decomposition is repeated to produce a feasible DAG and then some nodes are eliminated by collapsing them, providing the DAG remains feasible. Finally, a covering step merges LUTs so that a pair is implemented in one CLB.

TDD uses most of the common decomposition methods [24]. The authors show that the decomposition method is critical to the overall mapping results. They also couple the decomposition algorithm with their own tool Level-Map (see Section 2.4.3). Not surprisingly, their results are very good in comparison to Mis-pga$_{new}$ and others.

Delay Minimisation

The delay optimisation version of *MIS* programs, Mis-pga$_{delay}$, works on a DAG composed of two-input gates [53]. It performs a delay trace through the DAG so as to assign levels for each node and thus identify the critical nodes. Using this information, it then performs some logic synthesis and mapping. Finally, it does placement guided by the results of the mapping.

Huang et al. considered the tradeoff between area and delay optimisation in their tool ALTO [36]. Unlike FlowMap-r (see Section 2.4.8), they begin with an area-optimised DAG and work towards reducing its delay. The area optimisation is performed by a modified version of Chortle-crf (see Section 2.4.3). The principal modification being the use of a MSD strategy during bin-packing. Additionally, the tree is constructed with circuit depth considerations. These modifications give a 20% improvement in the number of levels required. Huang et al. also apply the Roth-Karp decomposition to DAGs with ten or less primary inputs after they have been collapsed. This mixed approach gives 17% fewer levels than Mis-pga. Another of the level reduction techniques is based on Chortle-d (see Section 2.4.3). It is followed by the partial collapse of the DAG while maintaining its feasibility.

2.4.5 Clustering Methods

The general idea of clustering is to find feasible subgraphs in a DAG and implement these as single LUTs. For a clustering algorithm to work, the DAG needs to be mappable (but not necessarily feasible). Most tools decompose the DAG into a feasible one beforehand to guarantee that it is mappable.

Area Minimisation

Xmap, by Karplus, uses *if-then-else* DAGs to perform the technology mapping of Boolean DAGs [41]. First, the *if-then-else* DAG is decomposed into a DAG with two-input nodes. Then, some nodes are identified as roots of logic blocks. The DAG is traversed from the inputs to the outputs. This marking phase groups together nodes which may sometimes become infeasible. When this occurs, high fanout descendants of the nodes are marked, thus hiding the nodes below the descendant. Alternatively, some of the direct input nodes are progressively marked with the selection determined by their respective fanin sizes. This eventually reduces the fanin size of the current node's cluster. The assignment of logic blocks and gates is then done. Karplus avoids using the min-cut algorithm to compute smaller cuts in the DAG. Instead, he simply traverses the DAG until he finds the marked nodes. Finally, some nodes are merged into a logic block as permitted in the Xilinx 3K series FPGA architecture. Xmap uses a greedy algorithm to select the matching pairs. The mergeable blocks are first identified and then sorted in descending order of their inputs. Then a match is sought for the first block with all other blocks. If a match is found, then both blocks are removed from the set; otherwise, only the first block is removed from the set.

There is a tool with a similar name by Groh [33]. For convenience, it will be called G-XMap here. This tool works on DAGs based on 2-input NAND gates. It finds feasible cuts within the transitive fanin of a node. These cuts are computed by traversing the subgraph and therefore, G-XMap is much simpler than other tools. The results of G-XMap are about 7% better than Mis-pga and 7% worse than both Chortle-crf and Hydra.

The *TechMap* algorithms, by Sawkar and Thomas, use an approach similar to clique partitioning [60]. In the first method, TechMap partitions each node into cliques, each of which can be realised by a single LUT. It then merges fanin nodes so as to reduce the number of LUTs and move nodes around. The second method applies various decompositions to the nodes and extracts common subexpressions which can be implemented by LUTs.

Delay Minimisation

TechMap-L is the delay optimisation version of TechMap [60]. It actually begins by applying TechMap and then identifies the critical nodes. These nodes are re-mapped using a clique partitioning method; the slacks are recomputed; and finally, the nodes are decomposed again using co-factoring to reduce the levels. TechMap-D is another delay optimisation algorithm by Sawkar and Thomas [61]. The basic idea of TechMap-D is to assign a cost on the merger of pairs of fanin LUTs. Using a greedy, iterative method, it selects pairs of LUTs for merger and thus creates a tree of LUTs implementing the node and its fanins.

The SWEEP tool of Shin et al. begins by calculating minimal levels for each node in a DAG and then minimising the number of LUTs in the second phase [66]. The DAGs are pre-optimised

so that all nodes have no more than two inputs. The computation of levels is done using an algorithm similar to the depth-first search algorithm, where nodes are processed from the primary inputs. Essentially, clusters of nodes are defined so that all nodes in a cluster have the same level, and the cluster is feasible. Hence, the level of a node is greater than that of all its fanins only if it cannot be grouped together with the fanins that have the highest levels. To reduce the utilisation of LUTs, the second phase alters the levels of some nodes without increasing the depth of the DAG. This change of levels corresponds to moving nodes from one cluster to another. An iterative procedure is used to minimise the number of clusters required. The post-processing step replaces the clusters with k-LUTs in decreasing order of gain as determined by a cost function.

Another approach by Chang et al. is to create clusters of nine or ten inputs and then decompose them using the BDDs [7]. They use "don't care" sets to enhance the BDD decomposition schemes. After decomposition, they reduce the depths of the DAGs using a rule-based approach. Critical paths are compressed guided by the Shannon expansion of Boolean functions. A critical signal can be moved closer to the output by Shannon decomposition. Another technique used is to collapse two nodes in a critical path together and decompose the super-node with respect to the sub-critical fanins. Related work by the same authors was reported in [6].

Chen's et al. work on two-input nodes on which they apply labels using an algorithm similar to that used in DagMap (see Section 2.4.8) [10]. They seek nodes within a cluster that are also roots of other clusters. If the slack of a node remains non-negative after it is removed from its encompassing cluster, then it is called a *releasable node*. Chen et al. use a greedy strategy to extract releasable nodes from clusters, using the slacks as the decision criterion. This process terminates when no further releasable nodes are found. The next phase involves merging LUTs so that a pair fits into a CLB. Chen et al. use a min-cut algorithm to partition the DAG into highly connected groups. They then construct a weighted compatible graph (similar to the shared-input graph in Hydra described in Section 2.4.4) and then use a greedy strategy to pair the LUTs.

2.4.6 Tools Based on Functional Decomposition

Area Minimisation

Lai et al. extended the Ashenhurst-Curtis/Roth-Karp method for computing decompositions by using BDDs [44]. Let a function $f(X, Y)$ have a BDD representation in which the variable ordering is such that

$$\forall x \in X, y \in Y \bullet x < y,$$

where X is the bound set and Y the free set. The *cutset* of a BDD with respect to a level b is the set of BDD nodes whose level is greater than b and which have a predecessor with a level less than or equal to b. This set is denoted by $cutset(f, b)$. The cardinality of $cutset(f, b)$ gives the

minimal number of functions in the disjunctive decomposition. By selecting a suitable encoding, the BDDs of these functions can be evaluated. Lai et al. extended this decomposition method to non-disjunctive decompositions and to incompletely specified functions. They also suggested the use of this method in the decomposition for heterogeneous FPGAs. Their results show a speed-up by a factor of 1.5 over the Roth-Karp algorithm in SIS.

The traditional approach to functional decomposition is to decompose each function individually. *Multiple-output functional decomposition* is the simultaneous decomposition of two or more functions, so that some or all of the decomposition functions are shared. Wurth et al. used this approach in IMODEC [71, 70]. They partition vertices into global classes and then seek sub-functions that are shared by two or more functions in the class. Such a function is called a *preferable decomposition function.* Their approach involves using positional-sets to represent the set of constructible functions. The sets of preferable functions are then represented by a characteristic function which is a Boolean formula. They used heuristics to group output functions and to partition the input variables. Compared to single-output decomposition, this method reduces the CLB count by 31 – 38% for Xilinx 3K FPGAs.

ISODEC-S, by Legl et al., is an extension of ISODEC where the variable sizes of the decomposition sub-functions are minimised [48]. The idea is to compute sub-functions that are independent of a single bound set variable and then maximise the number of such sub-functions.

BDDsyn, by Chang and Marek-Sadowska, seeks decomposition functions with small BDDs coupled with substitution [5]. Substitution involves introducing sub-functions for a set of variables determined by a partition node x. The transitive fanout of the node corresponding to x must have less than k distinct variables in the BDD. Repeated substitutions reduces the BDD to a k-feasible Boolean DAG.

Dresig et al. omitted the technology independent optimisations that are used in conventional synthesis strategies completely [23]. Instead they applied functional decomposition directly to the incompletely specified functions. Their tool ALOE-CLB still gives results comparable to Hydra and Chortle-crf.

Delay Minimisation

BoolMap-D by Legl et al. is another extension of ISODEC in which the variable-partitioning is geared towards, firstly, minimising the delay and then minimising the LUT count [47]. BoolMap-D uses an iterative heuristic to swap variables between the bound and free sets. The cost functions estimate the arrival time of the composition function g as well as the number of decomposition functions. BoolMap-D collapses the DAG first, decomposes it, and finally, covers it.

Hwang's et al. Factor-map is essentially a variant of functional decomposition [37]. One of the issues they address is that of variable partitioning. For this they apply two schemes. One

is a balanced partition of the variables; the other is a recursive, bounded-set partitioning of the variables. Their overall goal is to limit the number of interconnections among subgraphs. Factor-map first reduces the Boolean DAG into an EXOR/AND form. It identifies feasible clusters and decomposes infeasible nodes. Then it merges pairs of clusters so that they map onto a Xilinx 3K CLB. The results of Factor-map are particularly good for adder-type circuits, but it founders on other types of circuits.

2.4.7 Genetic and Integer Programming Algorithms

Kommu and Pomeranz used a genetic algorithm to solve the technology mapping problem in their tool GAFPGA [42]. Two types of genetic algorithm operators are used. The first, called a *crossover operator*, selects two individuals from the current population with the highest fitness and uses them to generate an offspring. Mutation is the second operator which randomly mutates an individual in the population. These operators may yield invalid offsprings. Therefore, GAFPGA has to transform invalid solutions into valid ones. This transformation is done randomly until a feasible solution is obtained. According to Kommu and Pomeranz, the randomness is beneficial to the search for a solution. GAFPGA works on k-feasible DAGs in which decomposition is accomplished by one of two methods: *complete binary tree decomposition* or *long decomposition.*

Chowdhary and Hayes used integer programming to perform technology mapping [12]. Mixed integer linear programming optimises a linear objective function of a set of variables under a system of linear constraints. The formulation of the technology mapping problem involves assigning variables to each node that is a base of a LUT as well as to the fanins of the LUT covers. The size constraint is that each LUT must be feasible. The problem can also be formulated for delay optimisation by considering the levels of the nodes. However, the area has to be fixed beforehand. In their method, Chowdhary and Hayes first partition the circuit by computing small cut-sizes in the DAG. They then map the sub-circuits independently. This approach requires some knowledge about the high level model of the circuit. When it is applied, the time needed to compute the mappings is reduced.

2.4.8 Network Flow Methods

The *FlowMap* and *DAGMap* algorithms were instrumental in showing how a delay-optimal (at least in terms of DAG depths) could be computed for a general Boolean DAG in polynomial time [14, 11]. They apply network flow techniques in computing cuts in the DAG and thus forming clusters. The method used in FlowMap generated sufficient interest leading to several variants [16, 18, 15, 13, 19], each of which uses the same basic approach but tries to remedy the weak points of FlowMap. The DagMap algorithm traverses the DAG determining the boundary of each LUT and filling the LUTs with nodes as far as possible. The idea, therefore, is to limit the maximum

label at each primary output.

FlowMap extended the ideas in DagMap, but used a network flow computation to determine k-feasible cuts within a general DAG and maximised the volume of these cuts. By finding the minimum height cuts, they were able to compute the optimal levels for each node. Secondly, by maximising the volume of each cut, they minimise the number of LUTs required to cover the DAG.

Cong and Ding also proposed using a nominal delay model in which they assumed that the net delay is proportional to the number of fanouts of a node [16]. An interesting result shown by Cong and Ding is that the delay-optimal mapping problem under the nominal delay model is NP-hard and remains so for duplication-free mapping (i.e., node duplication is not allowed) and $k \geq 5$, where k is the maximum number of inputs to a LUT. This is a surprising result since the area minimisation is known not to be NP-hard for duplication-free mapping. They devised a variation of the nominal delay model in which they used an arbitrary delay model so that the fanout net delays varied for each LUT [18]. In other words, the nominal delay model is simply an instance of this model.

FlowMap-r addressed the area/depth trade-off problem [15]. One characteristic of FlowMap is that it computes an optimal level for each node. Since this is unnecessary for non-critical nodes, there exists some scope for the levels of such nodes to be increased without affecting the optimality of the overall solution. FlowMap-r exploits this aspect of the solution computed by FlowMap. Cong and Ding also provided an invaluable insight into the problem of *duplication-free* mapping [15]. Their central thesis is that a general Boolean DAG can be decomposed into a set of disjoint subgraphs, called *maximal fanout-free cones (MFFCs)*. This implies that an optimal duplication-free mapping solution can be computed for each maximum fanout free cone independently, and that the whole solution is given by an amalgamation of these constituent solutions.

FlowSYN differs from FlowMap in its labelling phase since it incorporates some re-synthesis process that tries to limit the growth of the node levels as the DAG is traversed [13]. FlowSYN achieved very good results compared to all the algorithms in its category.

CutMap, by Cong and Hwang, combines area and depth minimisation [19]. CutMap tries to minimise the number of LUTs covering a depth-bound DAG by computing min-height cuts for critical nodes and min-cost cuts for the others. The idea here is very much the same as in FlowMap-r, except that the cost functions and area minimisation methods are different. The crucial part of CutMap is the prediction made during cost assignment as to whether a MFFC will be implemented in a LUT or not.

Edge-Map, by Yang and Wong, is a tool that maps DAGs by labelling the edges with delays [73]. It seeks an actual delay-minimal design using the same strategy as FlowMap. In contrast to FlowMap, both nodes and edges have delays assigned to them. Thus, a cut in the DAG has a maximum delay determined by the delays in the edges it bisects. A *d-cut* is defined as a cut in which the delays in all the edges are bounded by d. The problem Edge-Map solves is to find

a minimum edge-cut size. The construction of the flow networks differs from the approach in FlowMap in that a bridging arc (u', u'') is by-passed by an arc (u', w'') if the edge (u, w) has a delay greater than d. Hence, a min-cut cannot be induced across an edge with a delay greater than d. Edge-Map relies on delay estimations obtained after initial placement.

2.4.9 Mapping for Placement and Routing

RMap is one of the few tools that combine technology mapping with placement and routing [63]. It begins by performing an AND/OR decomposition to obtain a feasible DAG. Then some nodes are collapsed into their fanouts so as to form feasible clusters. All such clusters are considered. A unique feature of RMap is that it considers both single clusters and pairs of clusters that can be mapped onto a Xilinx 3K CLB. The next step is to select pairs of clusters iteratively until the DAG is covered by CLBs. The main criterion is to maximise the number of edges encompassed within the clusters. In terms of routability, RMap improves on the results of Chortle-crf (see Section 2.4.3).

M.map performs mapping and placement simultaneously [9]. It first decomposes the DAG into two-input gates and then labels the nodes according to their level in the DAG. During mapping, the primary outputs are first placed into CLBs. *Seed nodes* are defined as those nodes *without* unmapped fanouts. In each iteration a feasible cluster based at a seed node is placed into a CLB so as to maximise the cluster's size and minimise the circuit depth. The assignment of a CLB is done via a bipartite matching algorithm using a weighted bipartite graph. This graph is re-built each time a new seed set is constructed. In terms of delay, M.map produces circuits that are 30% faster than those of Chortle-d (see Section 2.4.3).

FlowMap (see Section 2.4.8) has also been combined with placement [32]. First the simple gates, with two inputs, are placed in the array of logic blocks. Hence, an estimate of the delays in the design are made. These are used with a modified FlowMap algorithm to map the circuit into its final design. For Xilinx 3K series devices, this approach yields a 7% decrease in actual circuit delays as compared to the Xilinx placement and routing tools. However, the number CLBs used increased, which limits the value of this approach.

2.4.10 Mapping for Power Minimisation

The minimisation of power is beyond the scope of this thesis. Nonetheless, it is an important part of design with FPGAs. The work by Chen et al. is an example of mapping for power minimisation [8]. They use the Roth-Karp decomposition method to transform the functionality of the CLBs so that the switching densities of the outputs of the CLBs are reduced. This transformation does not increase the number of CLBs already in use.

Wang and Kwan seek to reduce the number of interconnections between LUTs [69]. This

is because the charging or discharging that accompanies a state transition dissipates far more power than a read operation in a LUT. Wang and Kwan use a bin-packing approach coupled with heuristics to minimise the number of inter-LUT connections in the mapping.

Farrahi and Sarrafzadeh also try to map nodes so that the highly active signals are hidden inside the LUTs [26]. Their algorithm for power minimisation is related to their other algorithm Level-Map (see Section 2.4.3).

2.4.11 Heterogeneous Mapping

This section presents the most important work from the viewpoint of this thesis. The work reviewed here covers heterogeneous LUT-based FPGA.

Tempt

The *Tempt* algorithm, by Chung and Rose, was one of the first ones that dealt directly with heterogeneous FPGAs. Tempt uses some of the methods of the *Chortle* algorithms (see Section 2.4.3). It has routines for depth minimisation and area minimisation. The general idea is to find feasible clusters based at a node in the DAG and then pack these clusters using a bin-packing approach. In the architecture they considered, Chung and Rose assumed that each basic block's output in a hard-wired logic block (HLB) was accessible. This assumption does not hold true for the Xilinx 4K series FPGAs. On the basis of the assumption, they are able to pack unrelated functions into a single HLB.

He and Rose

The synthesis algorithms of He and Rose did not address the Xilinx 4K series FPGAs [34]. In the architecture they considered, there are two types of LUTs with completely independent routing structures, whereas the Xilinx devices have a hard-wired link between the two types of LUTs. In the FPGA, there are N_p p-input LUTs and N_s s-input LUTs such that $s < p$ and $r = \frac{N_s}{N_p}$. A *supertile* is a mixture of p-LUTs and s-LUTs in some ratio r. The number of supertiles in a mapped Boolean DAG is given by

$$N_{sup} = \max(N_p, \left\lceil \frac{N_s}{r} \right\rceil),$$

for $r \geq 1$. The mapping seeks to minimise this value.

Firstly, the Boolean DAG is broken into fanout-free trees which are then mapped separately. Each tree is mapped several times by constraining the value for $N_p = 0, 1, 2, \ldots$ and then minimising N_s for the given ratio r. A further multi-optimisation steps selects a mapping for each tree so as to minimise N_{sup}.

The mapping is done by an FFD bin-packing algorithm similar to that in Chortle-crf (see Section 2.4.3). A given node has n p-LUTs based at some of its fanin nodes. The rest of the fanins are packed into these n "bins" (LUTs); new p-sized LUTs are created as long as the total number does not exceed a given value m. If this happens, then s-sized LUTs are created instead. Thus, the packed tree is formed by cascading the LUTs in descending order.

Each tree T_i has a set of solutions $\mathcal{S}_i = \{C_j^i\}$, where j is the number of p-LUTs used. Taking two trees together yields a set of solutions $\{C_j^{1\cup 2}\}$ such that $C_j^{1\cup 2}$ minimises the number of s-LUTs in $C_x^1 \cup C_y^2$, where $j = x + y$, $C_x^1 \in \mathcal{S}_1$ and $C_y^2 \in \mathcal{S}_2$. The trees are combined in a pairwise fashion until all trees have been exhausted. From the final set $\mathcal{S}^\star$, a solution is selected to minimise N_{sup}.

ASYL

In the lexicographical ordering of an expression F, the variables in its *co-kernel* C_k precede those of its *kernel* K. Hence, a factorisation of the expression induces a partial order on the variables. Two kernels K_1 and K_2 and their corresponding co-kernels may be compatible to different total orders of the variables. However, if they agree on at least one order, then they are *lexicographically compatible*. To factorize the expression, a greedy strategy progressively selects lexicographically compatible kernels using the literal gain as the cost measure. Abouzeid et al. adopted this method in their tool called ASYL [1]. Given an input reference order, new nodes (called cut points) are inserted into the lexicographical factored tree so as to form new subtrees whose inputs belong to particular slices of the order. If the cardinality of these slices are restricted to a given value, say k, then the effect is to decompose the expression. The nodes are utilised more efficiently by clustering them together using a bin-packing approach.

In the case of Xilinx 3K series devices, the clusters are either of four or five variables. In the delay-optimising version of ASYL, a unit delay model is used and the packing of clusters picks the nodes with the least level first. For Xilinx 4K series devices, the clustering is done for four or three inputs. Nodes that are mapped onto the β-LUT (i.e., H-LUT) must have two corresponding α-LUTs (i.e., F or G) within the same logic block. In fact, this is too restrictive since it overlooks some beneficial patterns.

MOFL

Lee and Shragowitz's *MOFL* uses a fuzzy logic approach. Their system, *Multi-Criteria Optimization using Fuzzy Logic (MOFL)*, uses fuzzy logic techniques for technology mapping [45]. Given a set of criteria, fuzzy logic grades them in terms of their importance and then chooses a solution from the solution space that best optimises the criteria. Clearly, the decision function should achieve the correct balance of the often conflicting criteria. These criteria are essentially, delay minimisation and area minimisation as measured by the maximal path length, the node levels,

the fanin and fanout sizes of nodes, and the number of CLBs covering a node. They use weighting coefficients to rank their criteria which determine the computation of the solution.

FGSyn

The method used in FGSyn is really an extension of EVBDD (see Section 2.4.6) [43]. One difference is the use of ordered BDDs in FGSyn. A second is that in FGSyn both disjunctive and non-disjunctive decompositions are considered. An important extension is the extraction of common sub-functions in the circuits. One way of doing this is to examine the decomposition charts of a group of functions and find from these common sub-functions. Lai et al. use an encoding scheme to select a minimal number of decomposition functions for a set of functions under consideration. The composition of the groups (clusters) of nodes is done by grouping together nodes with several common fanins. The clusters are restricted in the size of their fanin sets and in the total number of nodes within the clusters. FGSyn is principally a decomposition program. It can be used both for homogeneous and heterogeneous mapping. For heterogeneous mapping, Lai et al. define a two-layer decomposition as follows. A function $f(X, Y, Z)$ can be decomposed as

$$f = g(\omega_1(X), \ldots, \omega_p(X), \mu_1(Y), \ldots, \mu_q(Y), Z).$$

Then, decomposition functions of the form $\gamma(\omega_i, \mu_j)$ are sought for $1 \leq i \leq p$ and $1 \leq j \leq q$.

2.4.12 Comparisons

Comparisons between various tools have been reported in the literature. An accurate assessment of all these tools collectively would require that experiments be carried out for the same set of benchmark circuits under identical conditions. This is not attempted here. Results are quoted from the relevant literature. It has been observed that even for the same tool, different researchers report varying figures for the mapping results on benchmark circuits. It is possible that the benchmarks are actually different or the pre-optimisation is different. Hence, no new comparisons are made unless the relevant results are reported by the authors. Only results relating to Xilinx 3K FPGAs are compared here; the Xilinx 4K comparisons are presented in Chapter 6.

A summary of the area-optimisation results for Xilinx 3K devices is tabulated in Table 2.1. Newer tools give better results, with FGSyn being the best overall. An interesting aspect is that tools based on functional decomposition perform very well.

Table 2.2 shows a summary of comparisons of delay-optimisation results for different tools. The tools are grouped according to the general methods used in the tools. FlowMap computes the optimal depth solution for a DAG without regard to the Boolean functions. If Boolean techniques are applied to the nodes, better results can be achieved. Hence, in practice FlowMap does not

Table 2.1: Comparisons of area results on Xilinx 3K devices. The entries refer to the percentage improvements of the column tool over the row one. The measurements are in number of CLBs except for the entries marked †.

	Xmap	Hydra	ALOE-CLB	GAFPGA	Level-Map	Mis-pga$_{new}$
Chortle	7†					
Mis-pga	14	26	15			28
Xmap						29
Chortle-crf		=	< 1	11	18†	20
Hydra			< 1			

	BDDsyn	IMODEC	IMODEC-S	ASYL	TDD	FGSyn
Xmap	22					
Chortle-crf	12			13		21
Mis-pga$_{new}$	9				13†	13
ASYL						17
FGMap		16	18			

Table 2.2: Comparisons of delay results on Xilinx 3K devices. The entries refer to the percentage improvements of the column tool over the row one. The measurements are in maximum circuit levels.

	Chortle-d	TechMap-L	SWEEP	[10]	TechMap-D	[7]	ALTO
Mis-pga$_{delay}$	3		6		12		19
Chortle-d		3	4	6	9	14	17
FlowMap				2		8	11

	DagMap	FlowMap	FlowMap-r	Factor-map	FlowSYN	ASYL	BoolMap-D
Mis-pga$_{delay}$	6	7	8		25		
Chortle-d	2		5	7	20	30	
FlowMap			=		15		
FlowMap-r			=				24
TechMap					10		
FlowSYN							12

perform as well as some methods. FlowSYN overcomes a lot of the deficiencies of FlowMap and is one of the best tools currently available. Surprisingly, the functional decomposition-based tool BoolMap-D gives the best results. □

Chapter 3

Graph Covering

This chapter is concerned with three issues, all relating to the cover of a Boolean DAG with LUTs. A common thread is the application of the flow networks technique described in Section 2.2. Firstly, some properties of flow networks when applied to Boolean DAGs are derived. These properties are then used to reduce the sizes of the flow networks. Smaller flow networks need less time to run the max-flow algorithm. Then the flow networks are used to determine cuts, and hence clusters, of various sizes in a Boolean DAG. Such clusters correspond to "super-nodes". Finally, an algorithm is developed for selecting a cover of clusters for a DAG. The notation used here is defined in Chapter 2.

3.1 Introduction

Along with decomposition, covering is a key step in the synthesis process for FPGAs. The aim of area-optimal covering is to cover a Boolean DAG with the least number of function generators, i.e., LUTs. The transformation of the initial DAG into a mapped one is achieved by clustering together a collection of nodes so that each node is covered by at least one cluster. The clusters may have any number of nodes providing that each cluster can be implemented by a LUT.

Other researchers have employed various covering techniques on optimised, feasible DAGs [59]. However, an infeasible DAG is not necessarily non-mappable due to the fact that the number of inputs to a subgraph is not monotone within a DAG. If G is a subgraph of H, it does not follow that $|\mathcal{F}_{\text{IN}}(G)| \leq |\mathcal{F}_{\text{IN}}(H)|$. The following example illustrates this.

EXAMPLE 3.1.1 *Consider the DAG shown in Figure 3.1. The node z has five inputs. But a cluster can be constructed around it with some of its fanin nodes that has four inputs. Examples of this are: $\{z, v_5\}$, $\{z, v_5, v_3, v_4\}$ and $\{z, v_5, v_3, v_4, v_1, v_2\}$. The number of edges crossing the cluster boundary are not important; it is the number of distinct inputs that counts. Any one of these clusters can be implemented by a 4-input* LUT.

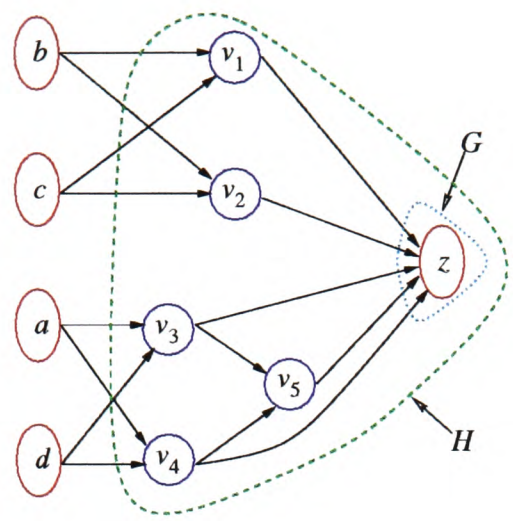


Figure 3.1: An example of a Boolean DAG showing that a subgraph G has more inputs than another subgraph H yet G is a subgraph of H.

The decomposition of an infeasible node introduces new nodes to the DAG and may affect the covering solution adversely. It is sometimes possible to map a node without prior decomposition. Where this is not possible, then the node should be decomposed before covering it. Chapter 4 deals with decomposition. The discussion in this chapter is based on the assumption that the DAG is mappable.

Whether a DAG is mappable or not can be determined by solving the *network flow* problem on the DAG. By transforming subgraphs of the DAG into suitable flow networks, mappable subgraphs can be identified given the input constraints. Various algorithms exist for solving the network flow problem efficiently. Their complexity is proportional to the sizes of the flow networks, as discussed in Section 2.2.

3.2 Preliminaries

In Section 2.2.1, a graph T was defined for a given node t of a Boolean DAG. The graph T is loosely based on the transitive fanin cone C_t of t. Its construction ensures that there is at least one edge incident *from* all vertices other than t. It is assumed that all internal vertices in T have at least two edges incident *to* them, since a vertex with fewer incoming edges is redundant. Any vertex with a single fanin can trivially have its fanin collapsed into itself. Hence, the following property for the flow network $\tilde{N}_t$ applies.

PROPERTY 3.2.1 *For any non-leaf vertex v of T, there are at least two paths from the source to v' (entry vertex for v) in $\tilde{N}_t$.*

Proof: By Property 2.2.1, there is a path to each internal vertex of $\tilde{N}_t$. Since v has at least two arcs incident to it, and v' has the same number of arcs incident to it, the proof follows. ■

A consequence of the transformation to a flow network is that the flow through any node u of T is limited to 1 in $\tilde{\mathcal{N}}_t$, due to the capacity of the arc (u', u''). Therefore, a minimum cut is induced across a set of bridging arcs. This fact leads to the definition of the *node cutset* as the set of vertices whose bridging arcs the cut bisects. A further property that bounds the maximum flow f^* in $\tilde{\mathcal{N}}_t$ can be deduced from this.

PROPERTY 3.2.2 *Let $G = (V, E)$ be a flow network with a source s and sink t. Let W be the set $\{w : (s, w'), (w', w'') \in E\}$ and U be the set $\{u : (u'', t), (u', u'') \in E\}$. The maximum flow in G is no more than $|W|$ nor $|U|$, i.e., $f^* \leq |W|$ and $f^* \leq |U|$.*

Proof: Only the case of U is considered since the proof for W is exactly the same. All paths to the sink go through some arc (u', u'') for some $u \in U$. Since each (u', u'') has unit capacity, the value for f^* cannot exceed $|U|$. ■

In terms of $\tilde{\mathcal{N}}_t$, $U = \mathcal{F}_{\text{IN}}(t)$, which means that $f^* \leq |\mathcal{F}_{\text{IN}}(t)|$.

A corollary to Property 3.2.2 restates the definition of the maximum flow in a different way. The definitions of G and U remain the same.

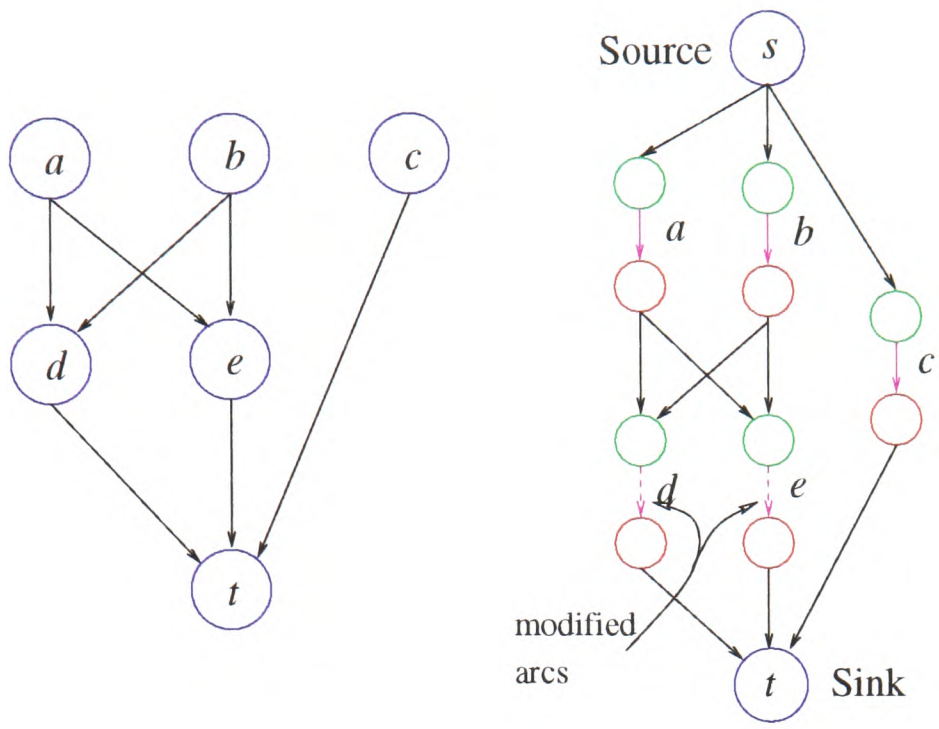
PROPERTY 3.2.3 *The maximum flow in G is given by $f^* = \sum_{u \in U} f(u', u'')$.*

Proof: Any path to t goes through some arc (u', u'') and $f(u', u'') = 1$. If the flow does not go along (u'', t), then it can be altered to do so, freeing some bridging arcs along its original path. This is permissible because $c(u'', t) = \infty$. Hence, any flow through an arc (u', u'') ends up in t. Since all flows originate from s, the proof follows. ■

3.3 Properties of Flow Networks

In Section 2.2, it was shown that the time complexity of the max-flow algorithm depends on the number of edges in the flow network. Therefore, to improve the speed of computing the cuts in the network, the size of this network must be reduced. Certain properties of the flow networks can be harnessed to achieve this end. A subset of the arcs and vertices in the graph are sought that may be omitted from the network without affecting the flow within it. In particular, the min-cut must be unaffected by the transformation.

The theorems presented in the following discussion consider a graph $\mathcal{C}_w$ and its subgraph $\mathcal{C}_v$. Therefore, v is in the transitive fanin of w. It follows that $\tilde{\mathcal{N}}_v$ is also a subgraph of $\tilde{\mathcal{N}}_w$. Assuming that the min-cut of v, $(X_v, \overline{X_v})$, is known, then $\tilde{\mathcal{N}}_w$ can be modified by omitting all edges and vertices in $\overline{X_v}$ and connecting the vertices in the cutset to v directly. This can be done for all nodes in $\mathcal{C}_w$ and thus reduces the size of $\tilde{\mathcal{N}}_w$. The following discussion covers the derivation of these properties.



(a) Boolean Graph (b) Modified Flow Network

Figure 3.2: An example of the construction of a modified flow network from its Boolean DAG.

3.3.1 Modifications to the Flow Network

The first property that will be derived affects the immediate fanin nodes of the node w. Let v be one such node. It is argued that if the edges and vertices in $\overline{X_v}$ obtained from $\tilde{\mathcal{N}}_v$ are omitted from $\tilde{\mathcal{N}}_w$, then the min-cut of $\tilde{\mathcal{N}}_w$ is unaffected. As an aid to this discussion, the following transformation is defined on $\tilde{\mathcal{N}}_w$.

- First, let F be a subset of the fanins of w such that $F = \{v : v \in \mathcal{F}_{\text{IN}}(w)$ and v is not a primary input$\}$.
- Set $c(v', v'') = \infty$.

The resulting network is called the *modified network* and is denoted by $\breve{\mathcal{N}}_w$. Restricting the membership of F to non-primary input nodes ensures that there is not a path of infinite capacity from the source to the sink. Notice that the maximum flow of $\breve{\mathcal{N}}_w$ could be greater than that in $\tilde{\mathcal{N}}_w$ due to the fact that the maximum flow in $\breve{\mathcal{N}}_w$ is no longer bounded by $|\mathcal{F}_{\text{IN}}(w)|$, as Property 3.2.2 suggests.

EXAMPLE 3.3.1 *As an illustration, Figure 3.2 shows a node t whose min-cut is across three nodes a, b and c. When $\breve{\mathcal{N}}_t$ is constructed, the vertices for d and e can be omitted, since they are contained in the cut for v. In this example, the saving in the flow network is of four vertices with two bridging arcs, plus four other arcs.*

The following properties are derived below:

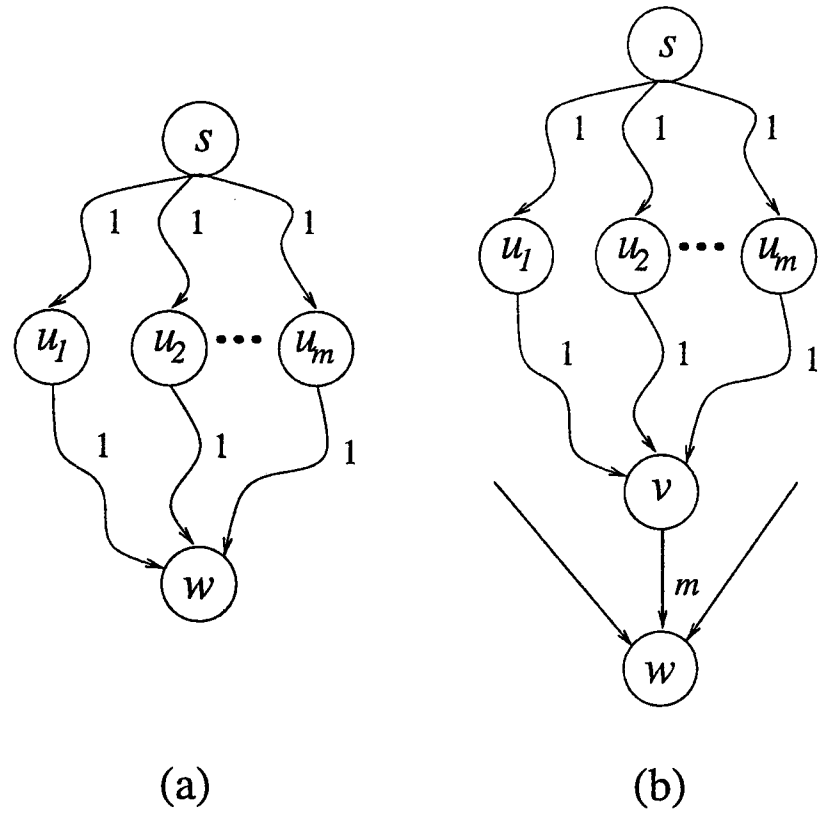


Figure 3.3: (a) The disjoint flows in a network. (b) The flow in a modified network. (Property 3.3.2)

1. If the maximum flow is m, then m disjoint paths exist from the source to the sink in the flow network. Each path saturates a bridging arc across the cut.

2. The maximum flow in the network is bounded on the upper side by the maximum flows through its constituent subnetworks.

3. If two paths leading to two different nodes, v_1 and v_2, intersect, then there is an intersection point at the minimum cutsets of subnetworks rooted at v_1 and v_2.

4. The omission of certain vertices and arcs in the $\overline{X}_1$ part of a minimum cut $(X_1, \overline{X_1})$ does not affect the maximum flow in the network.

The modified network is used to derive these properties, which will then be applied to the original network. Therefore, the modified network is only a tool for reasoning about flows.

Consider a network $\tilde{\mathcal{N}}_w$ with a maximum flow $f^* = m$. It is known that it has a minimum cutset $\{u_1, \ldots, u_m\}$. Therefore, there are m disjoint paths from the source to the sink, each of which goes through some vertex u_i (see Figure 3.3). The following property encompasses this situation.

PROPERTY 3.3.1 *If the minimum cutset in $\tilde{\mathcal{N}}_w$ is $\{u_1, \ldots, u_m\}$, then the m disjoint paths, $s \rightsquigarrow u_i \rightsquigarrow w$, each saturate a bridging arc of some vertex u_i and perhaps other bridging arcs along the paths. Moreover, w is only reachable from s via some vertex u_i.* ■

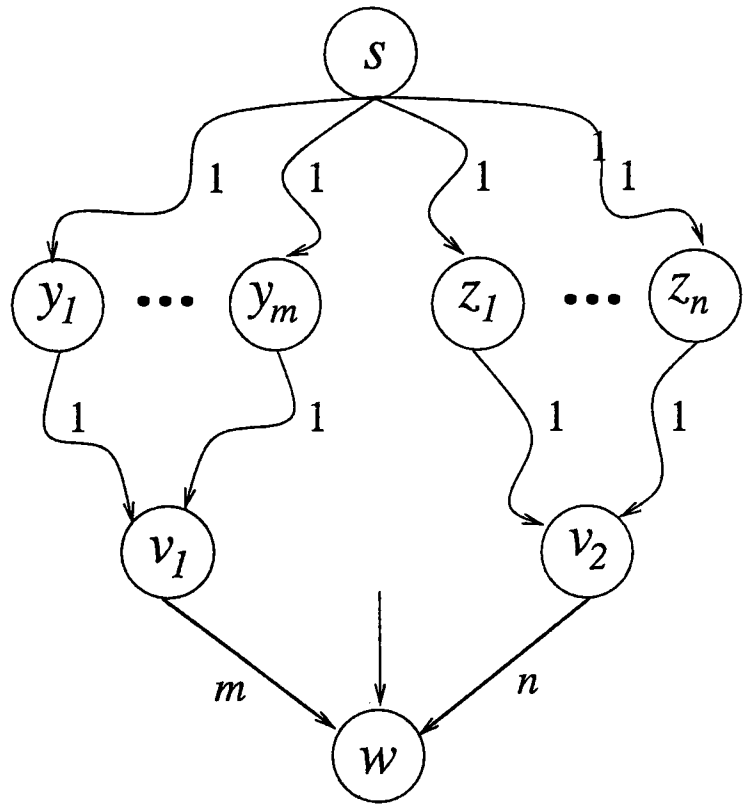


Figure 3.4: The disjoint flows in a modified network through two vertices. (Property 3.3.3)

Next, consider a node w with fanins $\{v_1, \ldots, v_q\}$. Let the minimum cutset of $\tilde{\mathcal{N}}_{v_i}$ be $\{u^i{}_1, \ldots, u^i{}_{m(i)}\}$ with a maximum flow of f_i^*, where $m(i)$ is the size of the i^{th} cutset. Then modify $\tilde{\mathcal{N}}_w$ with respect to just one $v \in \{v_1, \ldots, v_q\}$. This corresponds to setting $c(v', v'') = \infty$ as explained previously.

PROPERTY 3.3.2 *If the maximum flow in $\tilde{\mathcal{N}}_w$ is f^*, then $f^* \geq f_i^*$ for all $i \in \{1, \ldots, q\}$.*

Proof: As shown in Figure 3.3, all the flow through v enters w directly. Thus, the flow into w must be at least that of v. Since, v was chosen arbitrarily, the property is true for all $v \in \{v_1, \ldots, v_q\}$. ■

The next properties, extends Property 3.3.2 by considering the simultaneous modification of $\tilde{\mathcal{N}}_w$ with respect to two fanins of w. Two arbitrary vertices in the fanin set of w, say v_1 and v_2, are considered. Let the minimum cutsets in $\tilde{\mathcal{N}}_{v_1}$ and $\tilde{\mathcal{N}}_{v_2}$ be $\{y_1, \ldots, y_m\}$ and $\{z_1, \ldots, z_n\}$ respectively. First, consider the case where the paths from the source to v_1 and v_2 are disjoint (see Figure 3.4).

PROPERTY 3.3.3 *If all paths from the source through each y_i and z_j, for $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$, ending in w are disjoint, then $f^* \geq n + m$.*

Proof: By Property 3.3.2, f^* exceeds or equals the flow through both v_1 and v_2. Since the paths through v_1 and v_2 are disjoint, the property follows. (See Figure 3.4.) ■

Now consider flows through v_1 and v_2 that intersect at a vertex c (see Figure 3.5). Assume that there exist two paths $s \rightsquigarrow y \rightsquigarrow c \rightsquigarrow v_1$ and $s \rightsquigarrow z \rightsquigarrow c \rightsquigarrow v_2$, where y and z are in the minimum cutsets of v_1 and v_2, respectively. Hence, $\tilde{\mathcal{N}}_c \subset \tilde{\mathcal{N}}_{v_1}$ and $\tilde{\mathcal{N}}_c \subset \tilde{\mathcal{N}}_{v_2}$. The min-cuts of $\tilde{\mathcal{N}}_{v_1}$ and $\tilde{\mathcal{N}}_{v_2}$

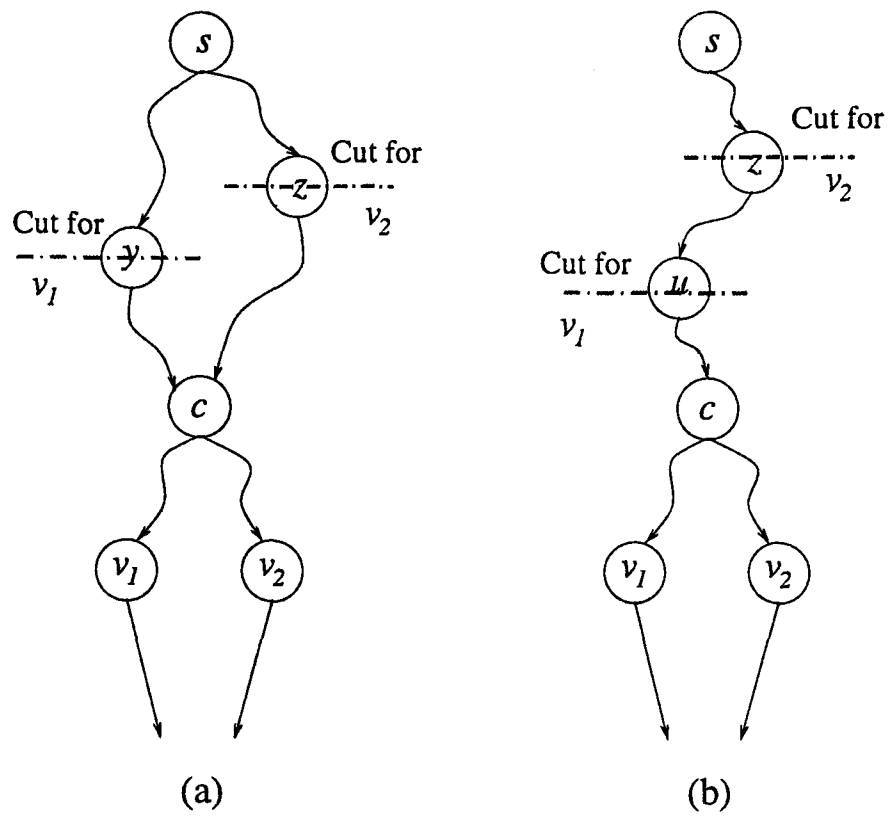


Figure 3.5: (a) The intersection of two flows leading to different vertices. This flow occurs after the minimum cutsets of both v_1 and v_2 (see Property 3.3.4). (b) Illustration of Property 3.3.5. If two flow paths intersect, then two nodes in the respective cutsets are affected simultaneously.

are $(X_1, \overline{X_1})$ and $(X_2, \overline{X_2})$, respectively. Since $(X_1, \overline{X_1})$ cuts off v_1 from s, it must also cut off c from s (since c comes after y). A similar argument can be made for v_2 (See Figure 3.5(a)). The aim is to show that by by-passing c and connecting y to v_1 and z to v_2 directly, an illegal flow does not result, i.e., a flow that could not have arisen in the original network.

PROPERTY 3.3.4 *Any flow through c saturates some bridging arcs that emanate from vertices in both X_1 and X_2.*

Proof: Any flow must saturate at least one bridging arc. Since all paths to c are bisected by the cuts $(X_1, \overline{X_1})$ and $(X_2, \overline{X_2})$, at least one arc in both X_1 and X_2 is saturated. (See Figure 3.5(a).) ■

In the modified network, c would be omitted. But the equivalent of the two flows shown in Figure 3.5(a) should not arise. In the original network, the bridging arc of c would ensure that only one of these flows is permitted, since it would admit only one unit of flow. It can be claimed that the modified network preserves this property.

PROPERTY 3.3.5 *Let $s \rightsquigarrow y \rightsquigarrow c \rightsquigarrow v_1$ and $s \rightsquigarrow z \rightsquigarrow c \rightsquigarrow v_2$ be two possible paths in a flow network $\tilde{\mathcal{N}}_w$, where y and z are in the minimum cutsets of v_1 and v_2, respectively. Then connecting y to v_1 and z to v_2 directly does not introduce an extra flow in $\breve{\mathcal{N}}_w$.*

Proof: The proof is trivial if y and z are the same node, since the bridging arc of y or z cannot admit two units of flow. Consider the case in which $y \neq z$ and $(X_2, \overline{X_2})$ is nearer to s than $(X_1, \overline{X_1})$ from the viewpoint of c. Any path to c crosses the cut induced by $(X_2, \overline{X_2})$ before that of $(X_1, \overline{X_1})$. Consequently, a vertex u can be found such that a path $s \rightsquigarrow z \rightsquigarrow u \rightsquigarrow c \rightsquigarrow v_2$ exists. Property 3.3.1 and Property 3.3.4 suggest that there is a vertex from the cutset of v_1 along this path. Let u be this vertex. Since u was selected arbitrarily, it may be assumed that u and y are the same node. A flow through $s \rightsquigarrow y \rightsquigarrow c \rightsquigarrow v_1$ blocks the potential flow through $s \rightsquigarrow z \rightsquigarrow u \rightsquigarrow c \rightsquigarrow v_2$. Consequently, no extra flow arises by by-passing c. (See Figure 3.5(b).) ■

The import of Property 3.3.5 is that the two intersecting flow paths each block a bridging arc in the cutset of the other destination node. So, if one path is used, the other may not be used. This means that two flows will not arise. Therefore, the common node can be bypassed safely by connecting both v_1 and v_2 to their minimum cutsets directly.

Since, both v_1 and v_2 were chosen arbitrarily, it follows by induction that the property holds for all pairs (v_i, v_j). In conclusion, the modified network $\check{\mathcal{N}}_w$ does not admit illegal flows.

3.3.2 Initial Reductions to the Flow Network

The principal aim is to find the min-cut of $\tilde{\mathcal{N}}_t$ using a smaller network. Using the modified network, a way of achieving this is deduced. This can be stated as follows:

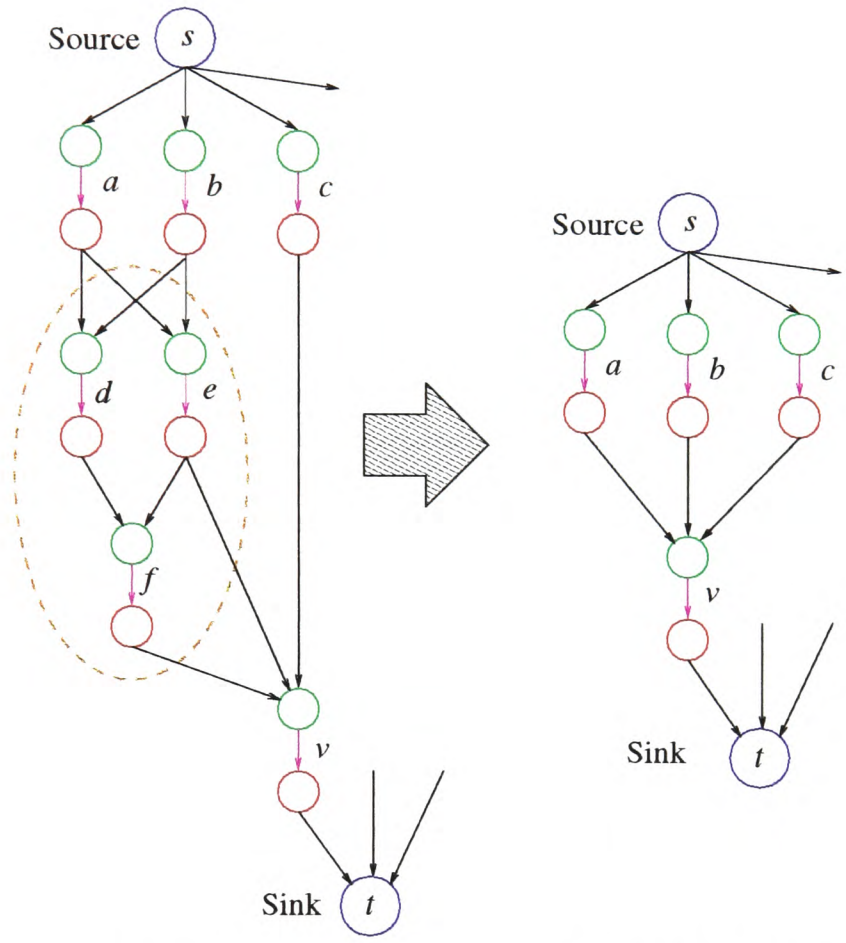
> If v is a non-primary input fanin of t, then connecting v to nodes in its minimum cutset, rather than its immediate fanins, in the flow network for t will result in the same max-flow as in $\tilde{\mathcal{N}}_t$.

This is the basis of the first reductions in the size of $\tilde{\mathcal{N}}_t$.

The following transformation of $\tilde{\mathcal{N}}_t$ to a new network $\check{\mathcal{N}}_t = (\check{V}_t, \check{E}_t)$ are defined.

- For each $v \in \mathcal{F}_{\text{IN}}(t)$, (v', v'') is a bridging arc and (v'', t) is a non-bridging arc in $\check{E}_t$.
- If v is a primary input, then (s, v') is a non-bridging arc in $\check{E}_t$.
- If U is the minimum cutset of v, then for each $u \in U$, (u'', v) is a non-bridging arc and (u', u'') is a bridging arc in $\check{E}_t$.
- Each $\tilde{\mathcal{N}}_{u'}$ is a subset of the network, i.e., u' is taken as the sink of sub-network equivalent to $\tilde{\mathcal{N}}_u$.

Such a network will be referred to as the *partially reduced network* of t and is denoted by $\check{\mathcal{N}}_t$. An example of this is shown in Figure 3.6. Here, the node v has a min-cut across $\{a, b, c\}$. The vertices of these nodes can be connected directly to the vertex of v, bypassing d, e and f.



(a) A Flow Network (b) Its Reduced Form

Figure 3.6: An example of the transformation of a flow network $\tilde{N}_t$ in (a) to its reduced form in (b). The min-cut of v, a fanin of t, is $\{a, b, c\}$.

An important property of $\check{\mathcal{N}}_t$ is that it has the same maximum flow as $\tilde{\mathcal{N}}_t$. The following discussion develops the proof for this fact by considering flows in the subnetworks of $\tilde{\mathcal{N}}_t$ and $\check{\mathcal{N}}_t$. For the purpose of the discussion, it is assumed that $\mathcal{F}_{\text{IN}}(t) = \{v_1, \ldots, v_n\}$. Also, let the maximum flow in $\tilde{\mathcal{N}}_t$ be f^*.

PROPERTY 3.3.6 *The value of $f(v', v'')$ is the same in both $\tilde{\mathcal{N}}_t$ and $\check{\mathcal{N}}_t$ for any $v \in \mathcal{F}_{\text{IN}}(t)$.*

Proof: Consider the flow in $\tilde{\mathcal{N}}_t$. Now, $f(v', v'')$ can only have one of two values, 0 or 1.

- *Case* $f(v', v'') = 0$: It can be inferred that there are one or more other nodes $v_i \in \mathcal{F}_{\text{IN}}(t)$ such that the paths $s \rightsquigarrow v'$ and $s \rightsquigarrow v_i'$ intersect at a bridging arc. This prevents any flow from reaching v'. The intersections are not eliminated in $\check{\mathcal{N}}_t$ so that the same blockage occurs. Hence, $f(v', v'')$ is still 0.

- *Case* $f(v', v'') = 1$: The effect of bypassing some vertices merely shortens the path $s \rightsquigarrow v'$ in $\check{\mathcal{N}}_t$. Hence, $f(v', v'')$ is still 1.

Thus, it can be concluded that $f(v', v'')$ is unchanged in the partially reduced network. ■

A flow property for the partially reduced network $\check{\mathcal{N}}_t$ is derived.

PROPERTY 3.3.7 *The maximum flows in $\check{\mathcal{N}}_t$ and $\tilde{\mathcal{N}}_t$ are the same.*

Proof: Any flow into the sink must pass through some arc (v', v''), where $v \in \mathcal{F}_{\text{IN}}(t)$. Since the flows in these arcs are unchanged in $\check{\mathcal{N}}_t$, according to Property 3.3.6, the flow into the sink must also be unchanged. Hence, $\tilde{\mathcal{N}}_t$ and $\check{\mathcal{N}}_t$ have the same max-flows. ■

3.3.3 The Minimum Cutset

In the previous section. it was shown how the network $\tilde{\mathcal{N}}_t$ could be reduced to a smaller one without altering the maximum flow in it. The first min-cut found in $\tilde{\mathcal{N}}_t$ by a breadth-first search corresponds to the maximum volume cut (see Section 2.2). In the following discussion it is argued that the min-cut found in $\check{\mathcal{N}}_t$ is the same as that of $\tilde{\mathcal{N}}_t$.

PROPERTY 3.3.8 *The minimum cutset of $\check{\mathcal{N}}_t$ is same as that of $\tilde{\mathcal{N}}_t$.*

Proof: The proof is by contradiction. Assume that $\check{\mathcal{N}}_t$ and $\tilde{\mathcal{N}}_t$ have different min-cuts, $(X_1, \overline{X_1})$ and $(X_0, \overline{X_0})$, respectively. If the minimum cutset of $\check{\mathcal{N}}_t$ is considered, it will be seen that it is composed of nodes represented in $\tilde{\mathcal{N}}_t$. So $(X_1, \overline{X_1})$ can also be found in $\tilde{\mathcal{N}}_t$. It must be the case that $\tilde{\mathcal{N}}_t$ has a min-cut across nodes that were eliminated in the reduction process. But $(X_0, \overline{X_0})$ must have a bigger volume than $(X_1, \overline{X_1})$, which is a contradiction. Hence, $(X_0, \overline{X_0}) = (X_1, \overline{X_1})$. ■

3.3.4 Further Reductions

In the previous sections, it was shown how the substitution of $\check{\mathcal{N}}_{v'}$ for $\tilde{\mathcal{N}}_{v'}$ in $\tilde{\mathcal{N}}_t$ does not affect either the maximum flow or minimum cutset, where v is a fanin node of t. Here it is shown that further reductions in the size of $\tilde{\mathcal{N}}_t$ are possible by applying similar substitutions for other subnetworks of $\tilde{\mathcal{N}}_t$.

It happens that these substitutions must be applied in a transitive manner from the fanins of t towards s. The proof of the admissibility of the substitution is an inductive one on the vertices. If it can be applied to v, then can it also be applied to a node u in the cutset of v and so on?

The investigation is based on u, a node in the cutset of some node $v \in \mathcal{F}_{\text{IN}}(t)$, and an arbitrary node r.

PROPERTY 3.3.9 *The intersection or otherwise of the paths $s \rightsquigarrow u'$ and $s \rightsquigarrow r'$ in $\tilde{\mathcal{N}}_t$ is unaffected if $\tilde{\mathcal{N}}_{u'}$ is substituted with $\check{\mathcal{N}}_{u'}$.*

Proof: If they intersect, then there is an intersection either at the min-cut of r or u. In either case, that bridging arc is part of $\check{\mathcal{N}}_{u'}$. The second case is that of disjoint paths. Nothing in the transformation will make them intersect. This proves the assertion. ■

PROPERTY 3.3.10 *For all nodes u in the minimum cutsets of all $v \in \mathcal{F}_{\text{IN}}(t)$, if $\tilde{\mathcal{N}}_{u'}$ is substituted with $\check{\mathcal{N}}_{u'}$ in $\tilde{\mathcal{N}}_t$, the max-flow is unaffected.*

Proof: According to Property 3.3.9, the paths $s \rightsquigarrow u'$ intersect or remain disjoint as before. So the flow along all these paths is the same after the substitutions. This means that the flow through each (v', v'') is unaffected, and hence the new network has the same flow as $\tilde{\mathcal{N}}_t$. ■

The above leads to the following definition of a *reduced* network $\hat{\mathcal{N}}_t = (\hat{V}_t, \hat{A}_t)$.

- $s, t \in \hat{V}_t$.
- For all $v \in \mathcal{F}_{\text{IN}}(t)$, $v', v'' \in \hat{V}_t$, (v'', t) is a non-bridging arc and (v', v'') is a bridging arc in $\hat{A}_t$.
- The network is defined recursively starting from the nodes $v \in \mathcal{F}_{\text{IN}}(t)$. If U is the minimum cutset of $\tilde{\mathcal{N}}_{v'}$, then for all nodes $u \in U$, (u'', v') is a non-bridging arc and (u', u'') is a bridging arc in $\hat{A}_t$. Hence, $u', u'' \in \hat{V}_t$, and the recursion is continues with each node u.
- For all primary inputs z, (s, z') is a non-bridging arc in $\hat{A}_t$.

The construction of $\hat{\mathcal{N}}_t$ implies that all the cutsets for each in the transitive fanin of t are known. In fact, this is simply the application of the partial reduction technique to each node in C_t.

PROPERTY 3.3.11 *The max-flows in $\tilde{\mathcal{N}}_t$ and $\hat{\mathcal{N}}_t$ are the same.*

Proof: By the application of Property 3.3.10 of the nodes in C_t in a topological order, starting at t, the deduction can be made. ■

To summarise, it was proved that the max-volume, min-cuts of $\tilde{\mathcal{N}}_t$ and $\hat{\mathcal{N}}_t$ are the same. Therefore, the use of $\hat{\mathcal{N}}_t$ does not lead to any degradation of the cut found but may improve the time in which it is found. An example of a reduced network is shown in Figure 3.6.

3.4 Computing Cuts and Clusters

This section is concerned with computing cuts within a flow network which are then translated into node clusters in the Boolean DAG. The properties derived in the previous sections are used in constructing the flow networks. The DAGs are assumed to be κ_α-mappable, otherwise feasible clusters would not exist. First, the cost measurements for the cuts are derived. Then, an approach of identifying different sizes of feasible cuts in the flow network is presented.

3.4.1 Cost Measurements

The approach adopted here is to traverse the Boolean DAG from primary inputs in a topological manner. An internal node v can have two types of feasible cuts within its fanin cone.

- $Cut_B(v)$ is a cutset whose size is no more than κ_β. At least one node within $Cut_B(v)$ has only a single fanout so that v can be covered by a β-LUT.
- $Cut_A(v)$ is any other cutset of size no more than κ_α.

Therefore, given v, its reduced flow network $\hat{\mathcal{N}}_v$ is constructed and the max-flow algorithm is run on it to find a maximum flow f^*. Depending on the nature of the cutset found, it will be assigned either the type $Cut_B(v)$ or $Cut_A(v)$. If $f^* > \kappa_\alpha$, then v has to be decomposed since by Property 3.2.2, v must be κ_α-infeasible.

Further feasible cuts, greater than the minimum cut, can be identifled in $\hat{\mathcal{N}}_v$. Consider the fanin cone of v in isolation. Then it can be shown that a dynamic programming approach yields the optimal solution for this subgraph if it is fanout free [15]. First, it needs to be shown that the optimal cover for v has an optimal-substructure in C_v. A cost function, $Mes(v)$, is introduced for the cover of C_v that counts the number of logic blocks (not LUTs) needed to cover the subgraph.

Figure 3.7 shows three different configurations of a logic block. If v is the root of the β-LUT, then there is at least one α-LUT associated with the LUT at v (see (a) and (b) in the figure). Let $A(v)$ be defined to be the maximal set of the nodes which are roots of such α-LUTs. It must be the case that $1 \leq |A(v)| \leq 2$, since at most two α-LUTs are permitted within a logic block. Let U be a cutset of v. If a LUT is based at v, then all nodes in U will have to provide inputs to the LUT. In a transitive manner, this defines a set of logic blocks that will cover the fanin cone C_v.

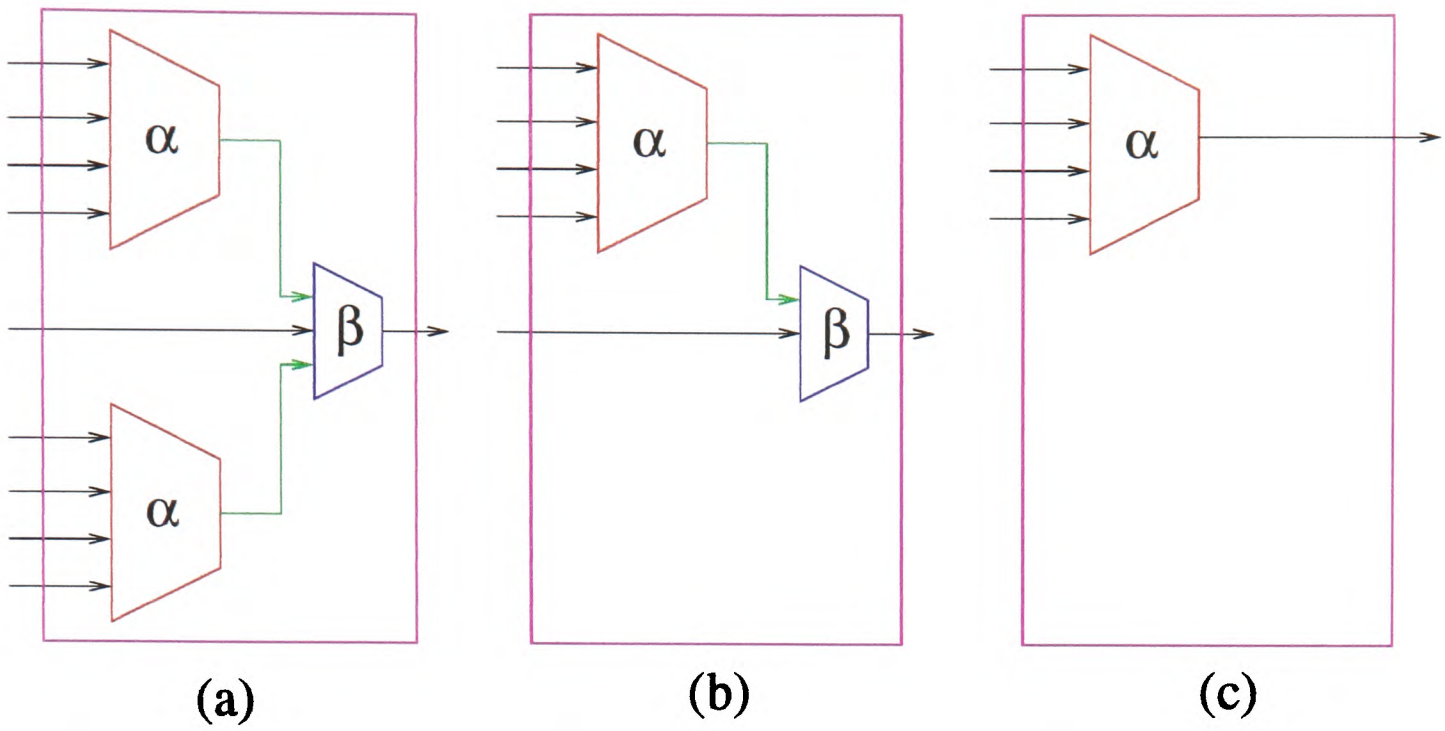


Figure 3.7: Three ways in which a logic block may be utilised

Let this set be $cover(v)$. Hence, the cost, in terms of logic blocks, of implementing the cut that defines U is defined as follows.

$$cost(v,U) = \begin{cases} 0 & \text{if } v \text{ is a primary input,} \\ 1 + |\bigcup_{u \in U} cover(u)| - |A(v)| & \text{if } U \text{ is of type } Cut_B(v), \\ 1 + |\bigcup_{u \in U} cover(u)| & \text{otherwise.} \end{cases} \tag{3.1}$$

Notice that $|A(v)|$ is subtracted if the cut is of type $Cut_B(v)$. This ensures that a logic block is only counted once for all of its constituent LUTs. Secondly, it is assumed that each α-LUT is implemented in its own logic block until a β-LUT is bound to it in latter stages. Thirdly, the union of all covers for logic blocks ensures the same logic block is not counted more than once.

Given the above, $Mes(v)$ is now defined as follows:

$$Mes(v) = \begin{cases} 0 & \text{if } v \text{ is a primary input,} \\ \min_{U}(cost(v,U)) & \text{otherwise,} \end{cases} \tag{3.2}$$

Equation 3.2 shows that the optimal value for $Mes(v)$ depends on firstly, a selection of an optimal cut, and secondly, the optimal cover of each subgraph. The recursive definition of $Mes(v)$ gives the dynamic programming formulation of the solution to the covering.

The following observations are made about the solution computed by Equation 3.2.

1. If a cutset consists entirely of primary inputs, then $Mes(v) = 1$. This follows from the fact that each primary input's contribution to the value of $Mes(v)$ is 0.

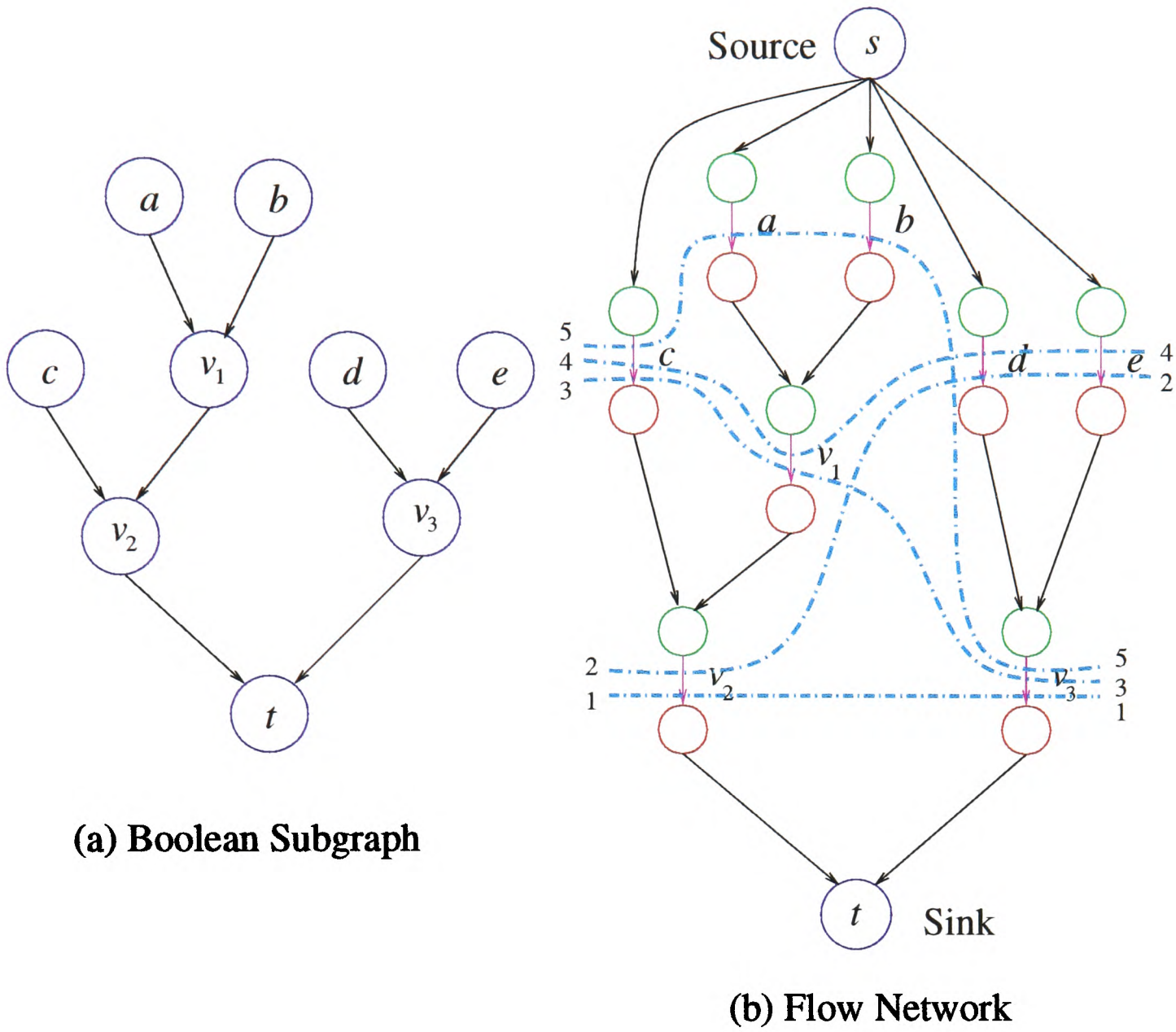


Figure 3.8: An example of a subgraph with several feasible cuts. In the flow network (b), there are five 4-feasible cuts: 1-1 to 5-5.

2. For any cutset of type $Cut_B(v)$, $A(v)$ must be maximal. In order to find $A(v)$, each node is labelled with the cut that gives a optimal solution for its cover.

Equation 3.2 requires that all feasible cuts for a node v are known. The next sections discuss the computation of these cuts.

3.4.2 Feasible Cuts

The issue of computing cuts of various sizes in a flow network is now addressed. The minimum cut is not always the best cut to implement for a given node. Generally, one expects to find more than one cut, the proviso being that the cuts are all feasible. As was shown in Section 3.4.1, the optimal cover depends on a selection of optimal cuts for certain nodes.

EXAMPLE 3.4.1 *Consider a subgraph C_t shown in Figure 3.8 (a). Its corresponding flow network $\tilde{N}_t$ is shown in Figure 3.8 (b). The min-cut is across the vertices $\{v_2, v_3\}$ (1 – 1). The cuts of size*

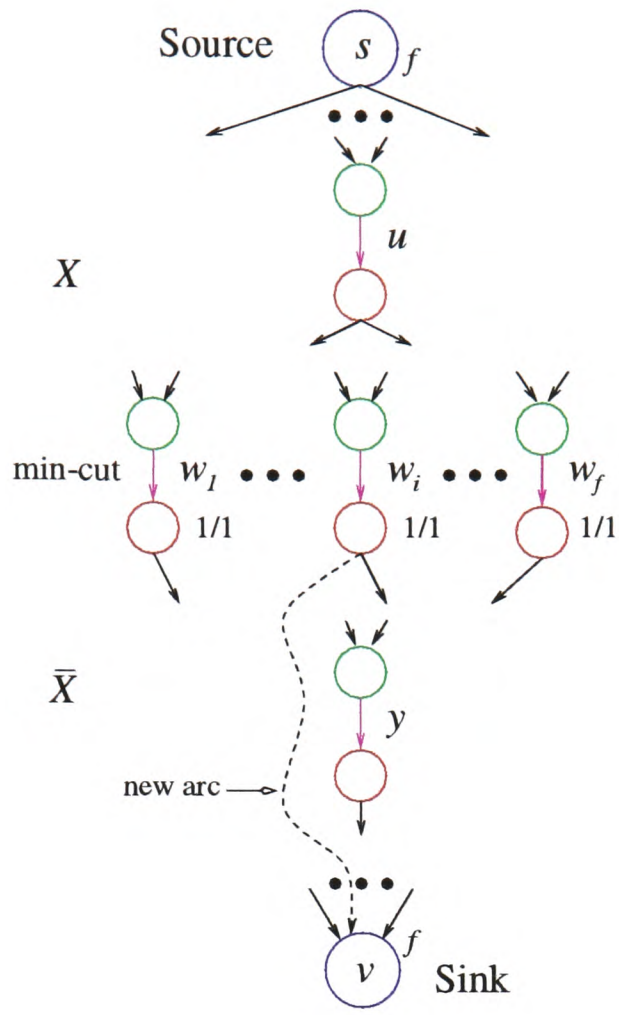


Figure 3.9: Introducing an arc from the min-cut to the sink in a flow network.

3 are $\{v_2, d, e\}$ *(2 – 2) and* $\{c, v_1, e\}$ *(3 – 3), Finally, the cuts of size 4 are* $\{c, v_1, d, e\}$ *(4 – 4) and* $\{c, a, b, v_3\}$ *(5 – 5).*

For a node v, such that $\hat{\mathcal{N}}_v$ has a maximum flow f, the cuts in the range f to κ_α whose volumes are greater than that of the min-cut are sought. Any cut that has a smaller volume than the min-cut produces an inferior cover for $\hat{\mathcal{N}}_v$.

PROPOSITION 1 *Let* $(X, \overline{X})$ *be the min-cut of* $\hat{\mathcal{N}}_v$. *Let there be another larger cut* $(X', \overline{X'})$ *which bisects* X. *Thus,* $(X', \overline{X'})$ *has a smaller volume than* $(X, \overline{X})$ *and hence,* $(X', \overline{X'})$ *gives an inferior cover for* $\hat{\mathcal{N}}_v$.

Proof: Let U and W, be the cutsets induced by $(X, \overline{X})$ and $(X', \overline{X'})$, respectively. By definition, $|U| < |W|$. Let M_U and M_W be the contributions of U and W to $Mes(v)$, respectively. The proof that $M_U < M_W$ is by contradiction. Note that no node contributes a negative value to the cost function. If it is assumed that $M_U > M_W$, then there is a node $w \in W$ such that for some node $u \in U$, $u \in \mathcal{C}_w$ and $Mes(w) < Mes(u)$. Since, $\mathcal{C}_u \subset \mathcal{C}_w$, the cut that gives $Mes(w)$ is also available to v. Hence, $Mes(w) \geq Mes(u)$. This is true for all $u \in U$ and their corresponding descendant nodes in W. Therefore, $M_U < M_W$. An important point to note is that any cut induced in $\hat{\mathcal{N}}_w$ can also be induced in $\hat{\mathcal{N}}_u$, so that $(X, \overline{X})$ always gives as good a cover as $(X', \overline{X'})$ for $\hat{\mathcal{N}}_v$. ■

It has been proved that the only cuts worth investigating are those that bisect X. Therefore, the following discussion is concerned with the generation of these cuts. Suppose a network $\hat{\mathcal{N}}_v$ has a max-flow f. It may be assumed that there are other cuts whose sizes are in the range f to κ_α. Now, if $W = \{w_1, \ldots, w_f\}$ is the cutset, then the flow into each arc (w_i', w_i'') is 1 unit as shown in Figure 3.9. Clearly, if a flow in excess of f is sought, then the capacity of at least one arc (w_i', w_i'') must increase. Yet it is required that the new cut be somewhere in X. This is ensured by introducing an infinite capacity arc (w_i'', v) if one does not exist already for the selected w_i. Now if the capacity of (w_i', w_i'') is increased, then an extra flow can be pumped along some path $s \rightsquigarrow w_i \rightsquigarrow v$. The amount of extra flow is limited by the capacity of (w_i', w_i'') and the number of unsaturated, disjoint paths $s \rightsquigarrow w_i$.

Let the new flow in $\hat{\mathcal{N}}_v$ be f'. It is important that a flow $f' > f$ does not merely induce the same maximum volume cut $(X, \overline{X})$. Essentially, the new cut $(X', \overline{X'})$ must move into X. (If the flow network is drawn with the source at the top and sink at the bottom, then X is the upper part of the bi-partition.) This can only be achieved if w_i is cut off from the source s. Consequently, the flow through (w_i', w_i'') must be such that the subnetwork $\hat{\mathcal{N}}_{w_i}$ is saturated. It is not difficult to see that $c(w_i', w_i'') \geq f_i$ is a condition for this, where f_i is the maximum flow in $\hat{\mathcal{N}}_{w_i}$. If $c(w_i', w_i'')$ is set to f_i, then $f' \leq (f + f_i - 1)$, since all the extra flow is through (w_i', w_i''). If all paths $s \rightsquigarrow w_i$ are disjoint from any other path $s \rightsquigarrow w_j$, where $w_i \neq w_j$, then $f' = (f + f_i - 1)$. This operation can be applied to a subset W' of W. Let the max-flow of a subnetwork $\hat{\mathcal{N}}_{w'}$ for some $w' \in W'$ be $g(w')$. Then,

$$f' \leq f + \sum_{w' \in W'} g(w') - |W'|.$$

The extreme cases are $W' = \emptyset$ and $W' = W$. The former suggests that there are no other feasible cuts with greater volume than $(X, \overline{X})$; the latter suggests that a feasible cut without any node in W exists. Yet, with the above scheme, it is not possible to generate all feasible cuts in X. On the basis of this scheme the feasible cuts can be computed.

To see this, the network is examined in a topological manner from the source. In the following discussion, assume that each node is assigned a label such that primary inputs are at level 0. Any other node v is at level m, where $(m-1)$ is the highest level of any of its fanin node. Dispensing with the primary inputs, consider a node r whose min-cut is composed solely of primary inputs (r is at level 1). No better cut exists for r. Now step up one level to a level 2 node, t. The min-cut of t either has primary inputs or level 1 nodes, such as r. As shown above, the various flows into t can be computed by altering the capacities $c(r', r'')$ of all or some of the nodes of type r. Finally, suppose that w is a level 3 node. Then the min-cut of w is composed of level 2 or less nodes, such as t and r. Now the flow through an arc $c(r', r'')$ can be varied in several ways, affecting the flow into w. An assortment of cuts in $\hat{\mathcal{N}}_w$ can be found by a combination of flows through the various arcs in the cutset.

Let f be the max-flow in $\hat{\mathcal{N}}_v$ and the cutset $U = \{u_1, \ldots, u_f\}$. Suppose the aim is to increase the flow into v by an amount m such that $f + m \leq \kappa_\alpha$. There are various ways in which this extra capacity can be distributed amongst the nodes in U. It is assumed that the capacities of bridging arcs for nodes in the transitive fanin of each u are known for a given flow $f(u', u'')$. Note that the quantity of the flow through (u', u'') is constrained by its viability. For instance, a primary input node can only have a unit flow. In general, the permissible distributions of flow satisfy the equation

$$\sum_{u \in U} c(u', u'') = f + m.$$

Let D be an f-tuple that represents the distribution of the extra flow among the nodes $u_1, \ldots, u_f$. Clearly, $0 \leq D[i] \leq m$ and

$$\sum_{i=1}^{f} D[i] = m.$$

It is assumed that a function ADMISSIBLE(u, k) checks to see whether (u', u'') can admit a flow of k units or not. To accomplish this, a variable $dis[k]$ records the distribution of k units of flow among the cutset of the given node, u. If $dis[k]$ = NIL, then the flow is not admissible for u. The following description shows how the capacities of bridging arcs in $\hat{\mathcal{N}}_v$ are altered given a distribution of flows D among the cutset U of v.

1. First find out if for all $u_i \in U$, the arc (u_i', u_i'') admits the extra flow $D[i]$. This is done using ADMISSIBLE. If any arc fails to admit the extra flow, then the alteration for the distribution fails.

2. Create a queue of tuples $(u_i, D[i] + 1)$ for all $u_i \in U$ and $D[i] > 0$. All vertices in the queue will have their corresponding arcs' capacities increased.

3. For each tuple (u, d) in the queue:

   (a) Set $c(u', u'')$ to the maximum of its current value and d. This ensures that the capacity never decreases. A situation in which d could be smaller than $c(u', u'')$ would arise if two or more subnetworks intersected at u.

   (b) Find the cutset W of u.

   (c) Let D' be the value of $dis[d]$ for u. For each $w_j \in W$ such that $D'[j] > 1$, enqueue $(w_j, D'[j])$. This means that only arcs whose capacities need to increase are enqueued.

This procedure is called ALTER-CAPS (see Section A.1.1 in the appendix). In general, the loop can execute up to $|E|$ steps, where $C_v = (V, E)$. However, for most cases, it executes $O(m)$

steps for an increase of m units of flow in $\hat{\mathcal{N}}_v$. Therefore, ALTER-CAPS takes linear time to alter the relevant capacities.

Once a distribution has been selected and implemented, the max-flow algorithm is run on the altered network. It is expected that m new blocking flows will be found. Indeed, for trees, this is guaranteed to be the case. However, non-trees may not admit the full extra flow due to the fact that some paths are shared among different nodes in the cutset. A network of this nature has the property that $c(u', u'') > f(u', u'')$ for at least one $u \in U$. If left in this state, the breadth-first search algorithm might compute the wrong cutset. Therefore, the network must be reformed whenever $f(v) < f + m$.

A recursive scan is applied to $\hat{\mathcal{N}}_v$ starting from v so as to identify bridging arcs with excess capacity. One relies on the fact that if w is a node with $c(w', w'') > f(w', w'') > 0$, then some bridging arcs of nodes in the cutset of w also share this property. The sum of the excess capacities in the bridging arcs of these nodes must equal $(c(w', w'') - f(w', w''))$. On the other hand, if $c(w', w'') = f(w', w'')$, then no arc belonging to a node in the cutset of w has excess capacity.

A procedure, called REFORM, works on a queue initially composed of v only. For each node w in the queue, it checks all the nodes u in its cutset U. All bridging arcs with excess capacity have their capacities reduced. The corresponding node is then enqueued. When the queue is empty, the reformation is complete and the network will be in its desired state. The listing of REFORM can be found in Section A.1.2 in the appendix.

As was argued for ALTER-CAPS, the procedure REFORM may also take time linear to the size of $\hat{\mathcal{N}}_v$. However, it normally performs relatively few adjustments in the order of m. The purpose of using REFORM is to ensure that any cut in an altered $\hat{\mathcal{N}}_v$ would cut across arcs of unit capacity only. Unfortunately, this does not always happen with non-trees. It is therefore necessary to discard any cuts in which the size of the cutset is less than the flow. A cut of this nature cuts across an arc of capacity greater than 1. However, trees satisfy this condition as the following property shows.

PROPERTY 3.4.1 *Assuming that $\mathcal{C}_v$ is a tree, let $\mathcal{C}_u \subset \mathcal{C}_v$ and W be the cutset of u. If $c(u', u'') > 1$, then u is not in the cutset of the given flow in $\hat{\mathcal{N}}_v$.*

Proof: When $c(u', u'')$ is increased, just enough bridging arcs of nodes in the fanin cone of u are also increased so as to allow all the extra flow to go through $c(u', u'')$. Hence, after running REFORM, all paths $s \rightsquigarrow w' \rightarrow w'' \rightarrow u'$ are blocked for any $w \in W$. The only way u' may be reached is via the arc (u'', u'). If this vertex u'' were ever reached, then u would not be in the cutset. ■

After applying REFORM, the FIND-CUT algorithm can be used to determine the new cutset W. Let $D(f)$ denote the distribution of capacities among the bridging arcs for the nodes in the cutset of u with a flow of f in $\hat{\mathcal{N}}_v$. Using $Mes(v)$ of Equation 3.2 taken over cutsets of size f, the

best distribution can be determined, i.e., one that minimises $Mes(v)$, for a given flow. The triple $(f(v), D(f(v)), Mes(v))$ should be computed and recorded for all viable flows $f(v)$.

It would be interesting to know the upper bound on the number of distributions that would be examined. It can be shown that the widest possible choice occurs in certain types of trees, where each flow $2 \leq f \leq \kappa_\alpha$ is permissible in any subtree. Let the cutset of $\hat{\mathcal{N}}_v$ be $U = \{u_1, \ldots, u_n\}$. The extra flow can be allocated to any subset of U. For instance, the capacity of u_1, $c(u_1', u_1'')$, can be increased by m. Alternatively, $c(u_1', u_1'')$ can be increased by 1 and the remaining $(m-1)$ extra capacity distributed among $u_2, \ldots, u_n$, and so forth. Let $P(m,n)$ denote the number of possible distributions. Therefore, the following recursion arises:

$$P(m,n) = \begin{cases} 1, & \text{if } n = 1 \\ 1 + \sum_{k=1}^{m} P(k, n-1), & \text{otherwise.} \end{cases}$$

It can be shown that $P(1,n) = n$, $P(2,n) = \frac{n(n+1)}{2}$, and $P(m,n) = P(m-1,n) + P(m,n-1)$. The table below shows the values for $P(m,n)$ for $1 \leq m \leq 5$ and $1 \leq n \leq 5$.

	n				
m	1	2	3	4	5
1	1	2	3	4	5
2	1	3	6	10	15
3	1	4	10	20	25
4	1	5	15	25	50
5	1	6	21	46	96

Since, $n + m \leq \kappa_\alpha$ and $n \geq 2$, $P(m,n)$ is expected to be between 1 and 3 for $\kappa_\alpha = 4$. This shows that only relatively few possibilities need to be examined for the sizes of cuts being sought.

Figure 3.10 illustrates the problem caused by non-disjoint paths. In (a) it is seen that the two paths, $a \to d$ and $a \to b \to c \to d$, are not disjoint. Equally, in (b) the paths $e \to f \to h$ and $e \to g \to i$ are not disjoint. Therefore, a flow through one path blocks the other path.

EXAMPLE 3.4.2 *Consider the flow in a network $\hat{\mathcal{N}}_t$ shown in Figures 3.11(a) and (b), both of which give a max-flow of 3. Assuming $\kappa_\alpha = 4$, then $m = 1$ and $P(1,3) = 3$. Let $D_j(x)$ denote the distribution of flow among the fanins of j given in alphabetical order. These distributions are tabulated below.*

$D_j(x)$	$D_f(2)$	$D_g(2)$	$D_h(2)$
Distribution	$[D_a(1), D_b(1)]$	$[D_c(1), D_d(1)]$	$[D_d(1), D_e(1)]$

The three possibilities for $D_t(4)$ are $[D_f(2), D_g(1), D_h(1)]$, $[D_f(1), D_g(2), D_h(1)]$, and $[D_f(1), D_g(1), D_h(2)]$. For the network in Figure 3.11(a), $[D_f(1), D_g(2), D_h(1)]$ does not increase the flow, since the paths

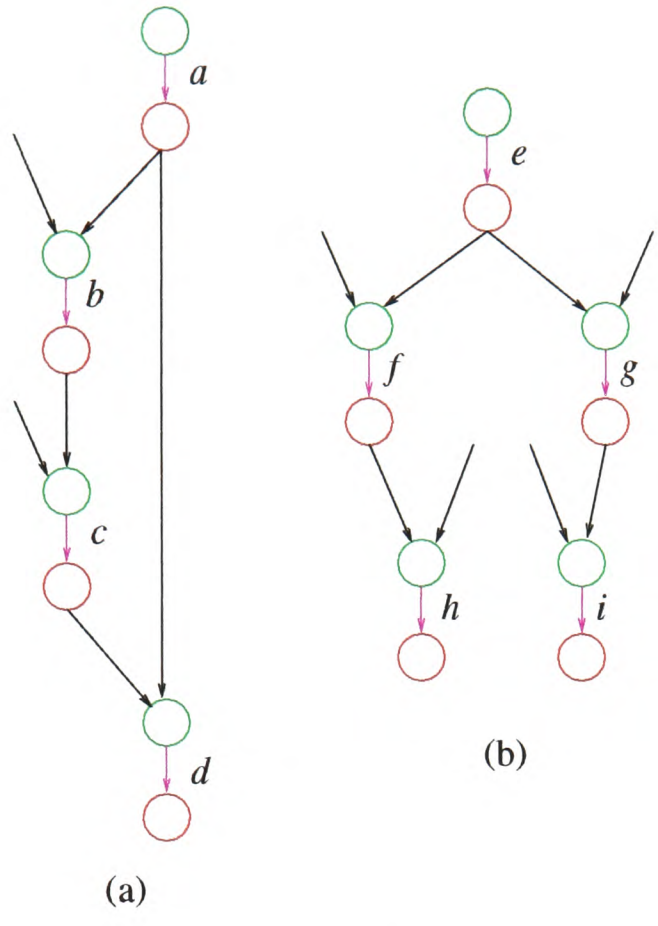


Figure 3.10: Examples of non-disjoint paths: (a) Only one unit of flow can go through (a', a'') yet two paths from a reach d. (b) h and i are both reachable from e whereas, only one unit of flow can go through (e', e'').

$s \rightarrow c \rightarrow g$ and $s \rightarrow d \rightarrow g$ are both saturated already. On the other hand, in Figure 3.11(b) this distribution permits an extra flow along $s \rightarrow d \rightarrow g \rightarrow t$. Figures 3.11(c) and 3.11(d) show the resulting cuts for a flow of 4 units. Notice that the cuts have different volumes.

EXAMPLE 3.4.3 *A further example is shown in Figure 3.12. Whereas $\hat{N}_t$ actually has a min-cut of size 2, there are other feasible cuts in it of size 3 and 4. Figure 3.13 shows these cuts. For all nodes $z \in \{a, b, c, d, e\}$, $Mes(z) = 1$. The tabulation below shows the values of $Mes(t)$ for various cuts.*

Cut	$\{a, b\}$	$\{c, h, b\}$	$\{a, i, j\}$	$\{f, g, h, b\}$	$\{c, h, i, j\}$
Flow	2	3	3	4	4
Type	β	β	β	α	α
# α-LUTs	2	2	1	2	2
# β-LUTs	1	1	1	0	0
$Mes(t)$	1	1	1	2	2

In this case, the best cover for C_t will be derived from the cut across $\{a, i, j\}$ (Figure 3.13 (b)) since it uses the fewest LUTs *and minimises $Mes(t)$ at the same time.*

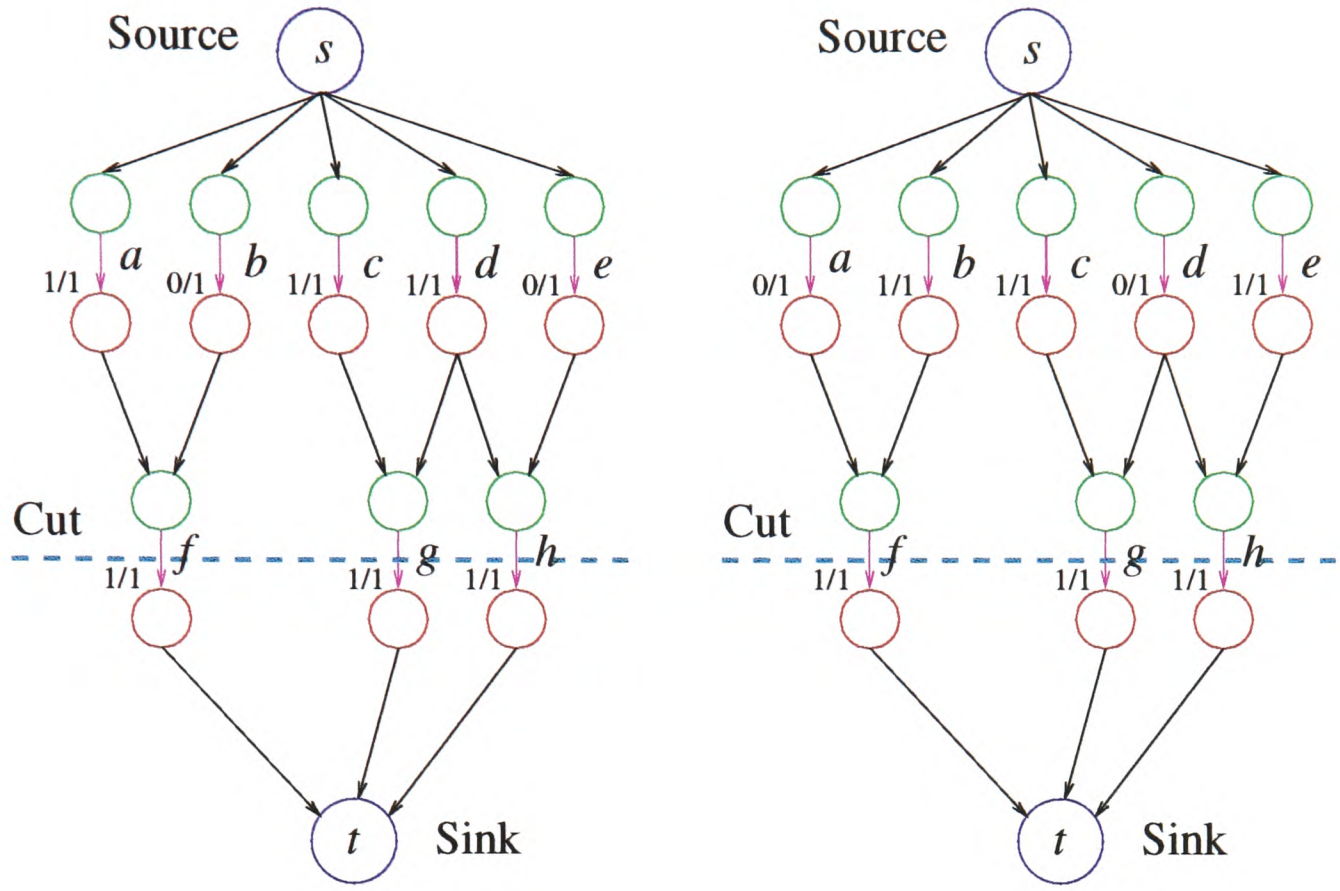


Figure 3.11: Examples of different flows in a network: (a) and (b) are two possible first-run flows of 3 units; (c) and (d) are two possible second-run flows of 4 units.

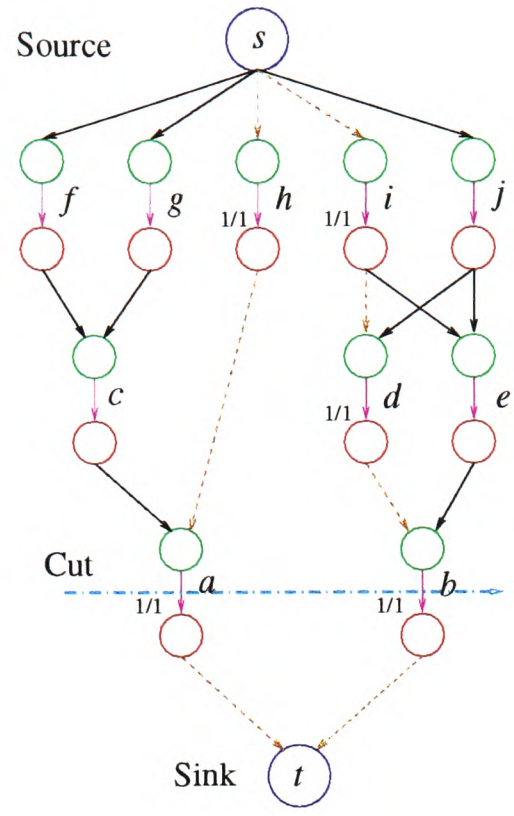


Figure 3.12: An example of a network with an initial max-flow of 2. Other feasible cuts can be found by altering the capacities of certain bridging arcs in the network. The dashed lines show the flow paths.

3.4.3 Algorithms for Computing Cuts

The algorithms that are required to compute cuts of various sizes in a network are now brought together. The following is assumed.

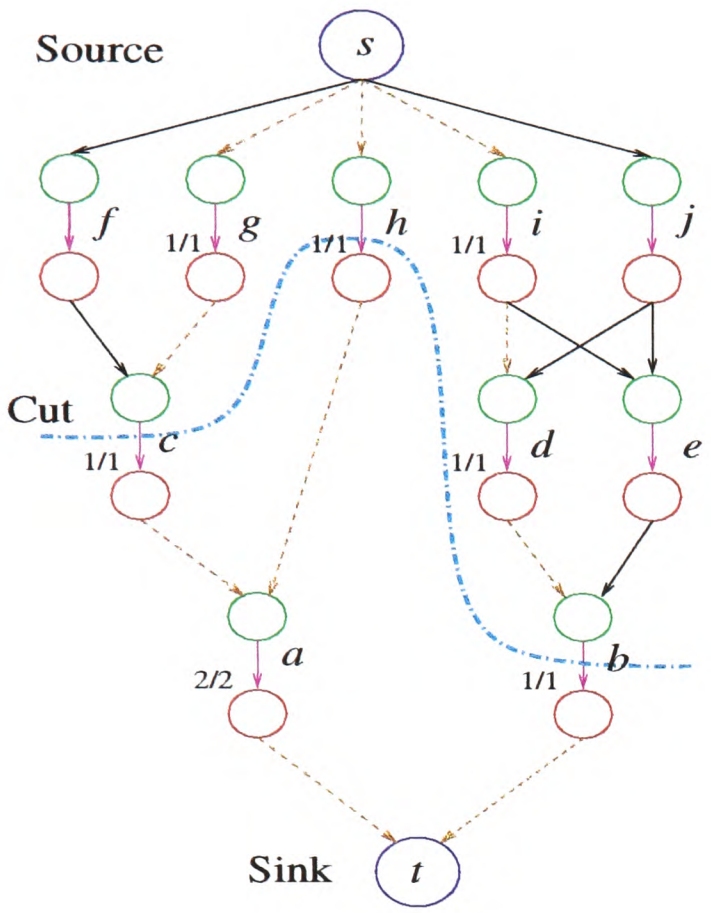
- There is a function, TABLE, that given two integers, m and n, generates a table of distributions. In other words, the i^{th} entry of the table contains a list of all distributions D_i among n subnetworks such that $\sum_{d \in D_i} d = i$. The table has m entries, where each entry is of varying length.

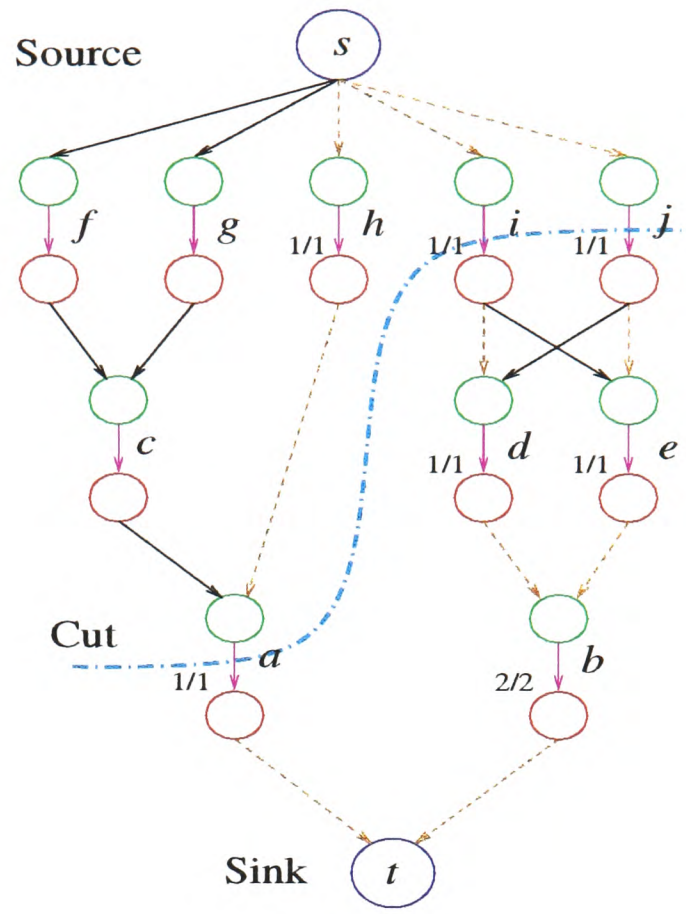
- A variable *DIS* stores the capacities of each bridging arc for nodes in the minimum cutset of $\hat{\mathcal{N}}_t$. Initially, *DIS* is an f-tuple of 1s since each bridging arc has unit capacity.

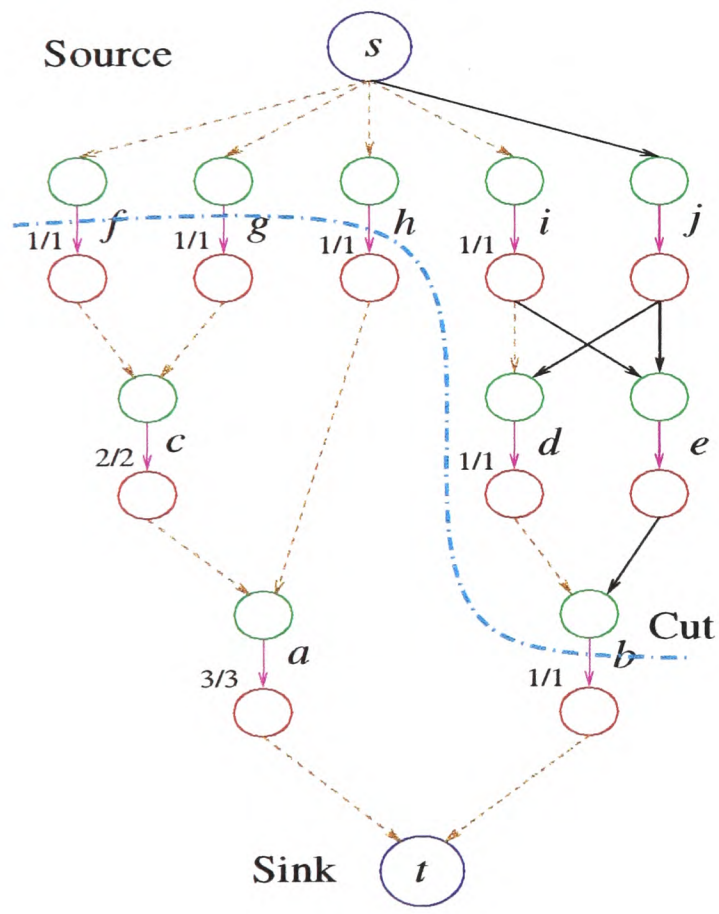
- An array R holds the pairs (DIS, z), where z is the cost associated with a distribution *DIS*. This cost is computed according to Equation 3.2. An auxiliary array M merely holds the z part of the pair.

- An array *CUT* contains the cutsets for various flows g such that $f \leq g \leq \kappa_\alpha$. Therefore, $CUT[f]$ is the minimum cutset.

There are two ways of finding a cut of size $g > f$. The first relies on altering the capacities of bridging arcs across in the minimum cutset U as well as the attendant minimum cutsets of their







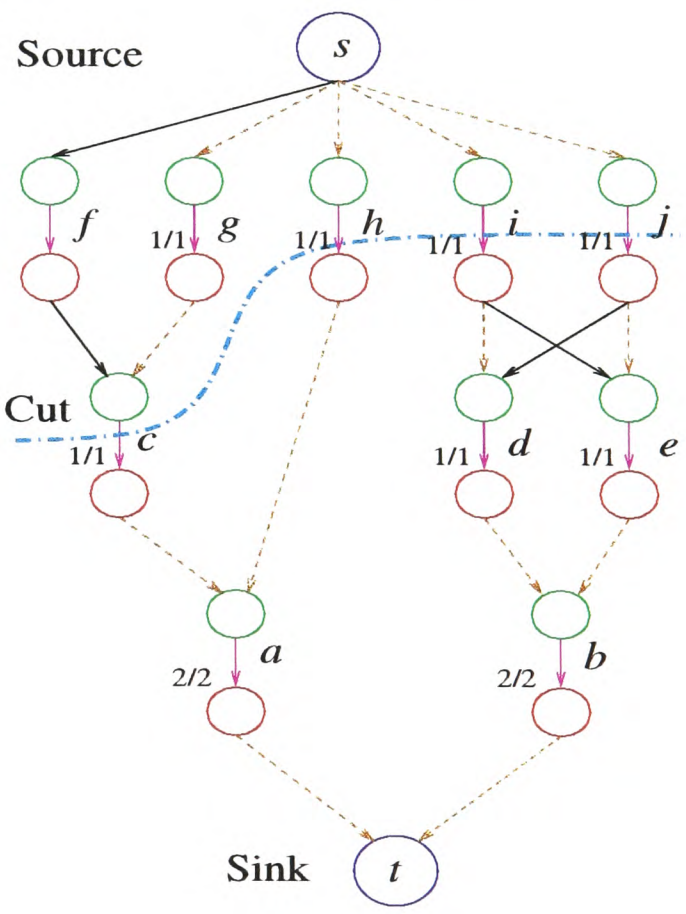


Figure 3.13: The various feasible cuts of Figure 3.12: (a) and (b) are for a flow of 3; (c) and (d) are for a flow of 4. The dashed lines show the flow paths.

subnetworks. The second approach uses a given cut as a base and only alters the bridging arcs across that cut.

Approach 1 It is assumed that $U = \{u_1, \ldots, u_f\}$ is the minimum cutset of $\hat{\mathcal{N}}_t = (\hat{V}_t, \hat{E}_t)$.

1. For all $u \in U$, create (u'', t) (if it is not already in $\hat{E}_t$).
2. For each extra flow j of 1 to $(\kappa_\alpha - f)$ units, and for each possible distribution D_j in f arcs do the following.
   (a) Alter the capacities of the capacities of bridging arcs of nodes in U and nodes in the subnetworks based at each u. ALTER-CAPS is used for this, checking that the alteration is permissible.
   (b) Run the max-flow algorithm on the new network. This gives an extra flow of g units.
   (c) If $0 < g \leq (\kappa_\alpha - f)$, then reform the network (using REFORM) and find the new cutset W.
   (d) If $|W| = (f + g)$, then this is a viable cut. Compute its cost z and compare it to the current value $M[g + f]$. The values of the array M are initialised to ∞. If z is smaller than the corresponding value in $M[g + f]$, then set $M[g + f]$ to z and $CUT[g + f]$ to W. The array R is updated accordingly.

The above algorithm is not an exhaustive search since it takes into account the cuts already computed in the subnetworks. For instance, let P be a cutset of $\hat{\mathcal{N}}_u$ for some $u \in U$. Assume that an extra flow of e units is required to flow through the nodes in P and that the distributions D'_e and D''_e are both possible with P. The algorithm does not attempt both distributions but selects the one that gave a lower cost when $\hat{\mathcal{N}}_u$ was processed. Since the cost measurements are not always entirely accurate from a global perspective, this filtering may lead to sub-optimal results. The procedure is called FEASIBLE-CUTS1 and is shown in Section A.1.3 the appendix.

Approach 2 Rather than gradually increasing the flows through the minimum cutset, a larger cut could be found and then an even larger one sought from there. This alternative approach uses a different cutset as its base each time an extra flow is sought; the previous approach always uses the minimum cutset as its base. Assume that C is a cutset of size k. Hence, k is initially equal to f and extra flows of $1 \leq j \leq (\kappa_\alpha - f)$ are sought.

1. Define $U = C$ as the base cutset. For all $u \in U$, create (u'', t) (if it is not already in $\hat{E}_t$).
2. For each possible distribution D_j in f arcs do the following.
   (a) Find out if all arcs (u'_i, u''_i) for $u_i \in U$ admit an extra flow $D_j[i]$. If not, take another distribution.

(b) Run the max-flow algorithm on the new network. This gives an extra flow of g units.

(c) If $0 < g \leq j$, then reform the network (using REFORM) and find the new cutset W.

(d) If $|W| = (f + g)$, then this is a viable cut. Compute its cost z and compare it to the current value $M[g + f]$. The values of the array M are initialised to ∞. If z is smaller than the corresponding value in $M[g + f]$, then $M[g + f]$ is set to z and $CUT[g + f]$ to W. Set k to the maximum of itself and $(f + g)$.

The procedure is called FEASIBLE-CUTS2 and is shown in Section A.1.3 the appendix. The effectiveness of the procedure depends on the selection of the base cutset C since it determines which cutsets will be found later on.

In terms of complexity, FEASIBLE-CUTS2 is similar to FEASIBLE-CUTS1. Its actual implementation allows it to examine more cuts and hence often gives better results. On the other hand it does not take into account the properties of previous cuts found in the subnetworks. The procedures are referred to by a generic name FEASIBLE-CUTS.

Finally, COMPUTE-CUTS is defined which works out the best feasible cuts in $\hat{\mathcal{N}}_t = (\hat{V}_t, \hat{E}_t)$. It works in two basic steps:

- Find the minimum cutset of $\hat{\mathcal{N}}_t$.
- Use this as a basis for computing other cuts through FEASIBLE-CUTS.

```
1  proc COMPUTE-CUTS (N̂_t, s, t, κ_α, κ_β)
2    comment: Assume that N̂_t = (V̂_t, Ê_t).
3    f ← MAX-FLOW(N̂_t, s, t, κ_α)
4    U ← FIND-CUT(N̂_t, s)
5    z ← cost(t, U)
6    Compute DIS
7    R[f] ← (DIS, z)
8    if (f < κ_α) then
9      FEASIBLE-CUTS(N̂_t, s, t, f, U, κ_α, κ_β)
10   endif
11 end.
```

Note that FEASIBLE-CUTS looks for cuts strictly larger than f in size. Line 3 is bounded by $O(\kappa_\alpha|\hat{E}_t|)$, and line 4 by $O(|\hat{V}_t| + |\hat{E}_t|)$. Based on the complexity of FEASIBLE-CUTS, it can be inferred that COMPUTE-CUTS has complexity $O(\nu(|\hat{V}_t| + |\hat{E}_t|))$, where ν is a constant function of κ_α.

3.5 Covering a Graph

This section is concerned with covering a DAG with LUTs. First, a cost function is defined for selecting cutsets of given nodes. Then a delay model is devised so that the covering optimises the circuit delay where possible. Finally, the algorithm for covering a Boolean DAG is presented.

In the previous sections, a dynamic programming approach to the area-minimisation of subgraphs (cones) rooted at primary outputs was devised. This approach is optimal for trees. For a general DAG, the covering problem is known to be NP-complete [25, 49, 74]. Some researchers have elected to divide DAGs into a forest and then cover the resulting trees optimally. This is tree-based covering. A second approach divides the DAG into maximum fanout free cones (MFFCs) and then applies a dynamic programming paradigm to solve the covering. DF-Map, by Cong and Ding, is an example of this approach [15]. In DF-Map, a feasible cut $(X, \overline{X})$ is computed for a node v. Then X is partitioned into disjoint fanout-free subgraphs whose cost will already be known. Therefore the cost of this cut can be found, and then compared to all other feasible cuts. The weakness with DF-Map is its restrictions on the duplication of LUTs. Level-Map, by Farrahi and Sarrafzadeh, defines a priority function for deciding which fanin node to include in a LUT [25]. By filling the LUTs to capacity, it generates the cover for the DAG.

The hypothesis presented here is that a better covering is obtained by taking into account the interaction of covers for different cones. The proposed covering scheme uses the information obtained by computing the cuts as described previously. It selects the cuts to implement based on the cover for the whole DAG rather than the single cone. Hence the covering tries to achieve a global optimal solution.

EXAMPLE 3.5.1 *As an illustration of the problem posed by intersecting cones, consider Figure 3.14. In (a) it is clear that C_a and C_f have C_x in common. Considering the covering of C_a alone, assume that a cover with* LUTs *A_1 and D in (b) is found to be superior to that of* LUTs *A_2 and X in (c). Secondly, considering C_f, suppose that the covering of* LUTs *F_1 and X in (d) is preferred over* LUTs *F_2 and H in (e). On the basis of this second finding, it is possible that the situation in C_a is reversed. A combination of the covers in (c) and (d) as shown in (f) may give the best result globally. This arises from the fact that once x is a root of a* LUT*, then any other* LUT *referring to it does not increase the number of* LUTs *required. The cover in (b), however, introduces a new* LUT *rooted at d and subsuming x. Situations in which several such conflicting dilemmas exist among various cones may be conceived.*

What is needed is a way of accessing the cuts to implement on the basis of two aspects:

1. the minimisation of the size of the cover within individual cones; and

2. concordant covers for all cones.

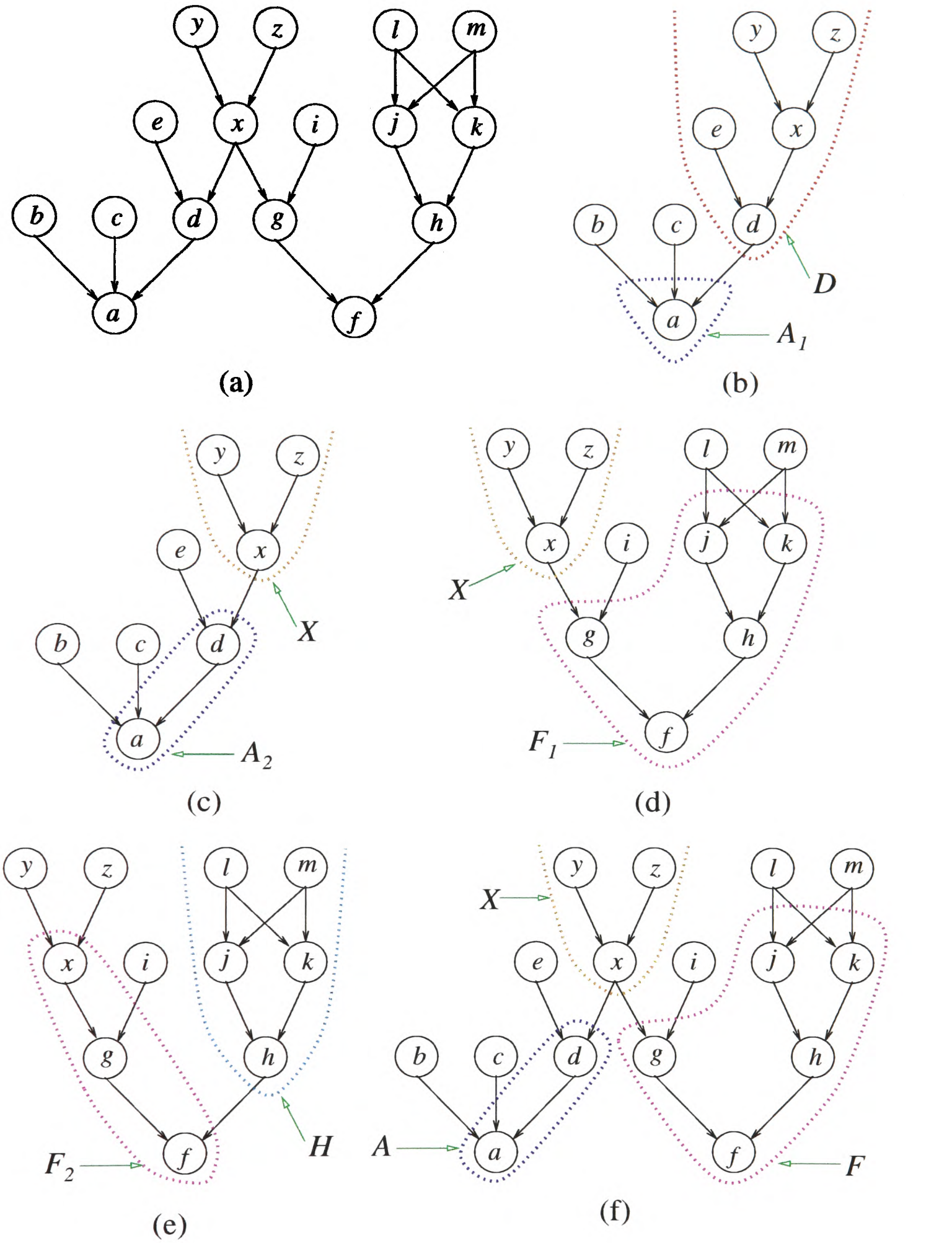


Figure 3.14: An example of covering a graph with intersecting cones: (a) $\mathcal{C}_x \subset \mathcal{C}_a$ and $\mathcal{C}_x \subset \mathcal{C}_f$; (b) and (c) are two possible covers of $\mathcal{C}_a$, while (d) and (e) are two possible covers of $\mathcal{C}_f$; (f) is a combination of (c) and (d).

In previous sections, a way of accomplishing the first aspect was developed. Unfortunately, the second aspect does not always coincide with the first. It is therefore necessary to define a method that balances the two in a smooth fashion.

3.5.1 A Cost Function

What is clear is that once a node has been selected as the root of a LUT, then it becomes more desirous to select other cuts that incorporate this node. However, this is not the duplication-free approach because the requirement that no cut is to subsume this node is not imposed. Instead, the use of a cost function is proposed to compare the competing cuts.

In Equation 3.2, each new α-LUT increments the measure by 1. This constant increment does not reflect the different natures of the LUTs in terms of their fanouts. Therefore, it needs to be replaced with a function of the number of fanouts, n, that the LUT has. Let this function be $(1 - \sigma(v))$, for a node v. There are various candidates for $\sigma(v)$. The most obvious ones are the following.

1. The step function — if the LUT already exists, then do not count it as a "full" LUT.
2. An exponential function $\varsigma(1 - \frac{1}{n})$, where ς is a constant — reduce the impact of a LUT on the count sharply as number of fanouts increase.
3. A linear function $\varsigma_1(1 - \varsigma_2 n)$, where ς_1 and ς_2 are two constants — reduce the impact of a LUT on the count steadily as number of fanouts increase.

The step function is suggested as the best choice based on the grounds that once a LUT has been established at a particular node, then the number of times it is referred to by other cuts does not alter this situation. The LUT exists to furnish an input to another LUT or act as a primary output regardless of the number of fanouts it has. Let n be 0 if no LUT is based at the node v yet, and equal to the number of fanouts otherwise. The step function for σ is then

$$\sigma(v) = \begin{cases} 0 & \text{if } n \leq 0, \\ \varsigma & \text{otherwise,} \end{cases} \tag{3.3}$$

where $0 \leq \varsigma < 1$. This constant provides a means of graduating the step.

Before employing $\sigma(v)$, an intersection of covers among different cones has to be defined. For this purpose, assume that $\mathcal{L}$ is the set of all logic blocks in the current covering. For convenience, these logic blocks are given the same names as the root nodes they cover. All primary output nodes are guaranteed to be members of $\mathcal{L}$, which develops as new logic blocks are inserted and others removed. When the entire DAG is covered, $\mathcal{L}$ defines the logic blocks needed to implement the circuit. Clearly, The aim is to limit the final size of $\mathcal{L}$.

Let $R(v)$ be the intersection of the logic blocks implementing $\mathcal{C}_v$ and the logic blocks in $\mathcal{L}$. Therefore,

$$R(v) = cover(v) \cap \mathcal{L}.$$

Using the functions $R(v)$ and $\sigma(v)$, Equation 3.2 is changed to

$$MesMap(v) = \begin{cases} 0 & \text{if } v \text{ is a primary input,} \\ \min\limits_{U}(cost(v,U) - \varsigma \times |R(v)|) & \text{otherwise,} \end{cases} \tag{3.4}$$

Therefore, if a logic block is already defined in $\mathcal{L}$, $MesMap(v)$ does not count it as 1 but reduces it by ς. Any new logic block defined by the cover of v counts as 1 and will be inserted into $\mathcal{L}$. Because $\mathcal{L}$ will change, it is necessary to iterate several times, evaluating $MesMap(v)$ in the light of the changing $\mathcal{L}$. This aspect will be addressed shortly.

3.5.2 A Delay Model

One important aspect of heterogeneous logic blocks is that they have a hard-wired link between α and β LUTs. This link is faster than the programmable links between logic blocks, even if they are adjacent to each other. The traditional approach is to count a LUT as representing one level in the path of the signal. Here, a delay model is proposed that takes into account the varying nature of the path a signal traverses as it is propagated from the primary inputs to the primary outputs.

The model is based on three simple ideas.

1. Firstly, each LUT, whether α or β type, has a constant delay δ_{L}. This is a simplification; in practice, the differences in the delays through these LUTs are likely to be insignificant.

2. The delay in a link from an α-LUT to the β-LUT within the same logic block is taken to be a constant δ_{fast}.

3. Finally, the delay between two logic blocks is a constant δ_{slow}.

Note that the last delay is a crude approximation. The actual delay is a function of several factors, the most important of which are placement and routing. These are not known at the time of covering and so a more accurate model cannot be attempted. Figure 3.15 shows examples of these delays.

Since all LUTs are assumed to have the same delay, δ_{L} is omitted from the definition of the level of a node/LUT v implementing a cutset U. First, the link-delay is defined as follows:

$$\delta_{\mathrm{link}}(u,v) = \begin{cases} \delta_{\mathrm{slow}} & \text{if } u \text{ is of type } \beta \text{ or } v \text{ is of type } \alpha, \\ \delta_{\mathrm{slow}} & \text{if } u \text{ and } v \text{ are in different logic blocks,} \\ \delta_{\mathrm{fast}} & \text{otherwise.} \end{cases} \tag{3.5}$$

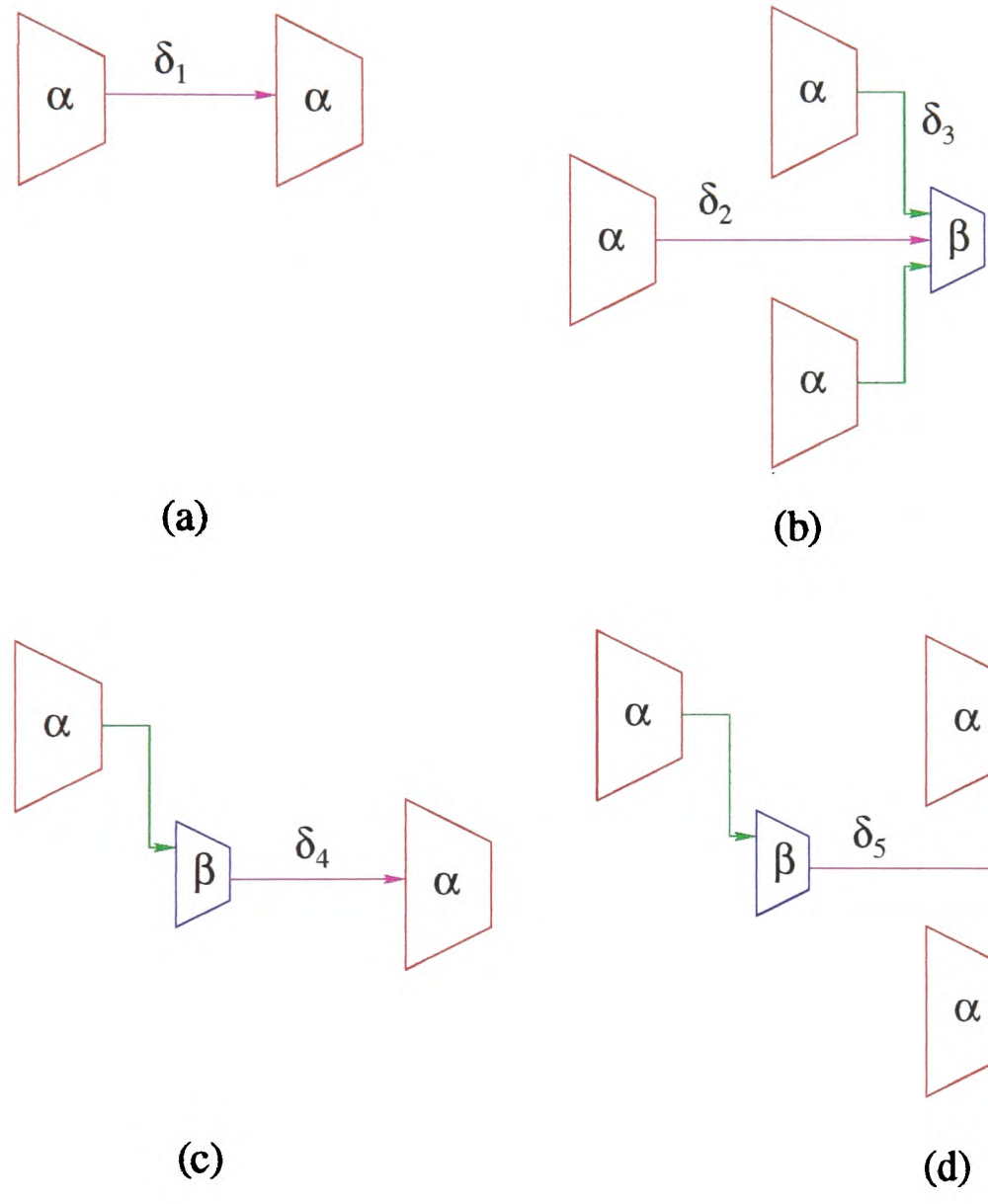


Figure 3.15: Various delay possibilities between two LUTs: $\delta_1, \delta_2, \delta_4, \delta_5 = \delta_{\text{slow}}$, whereas $\delta_3 = \delta_{\text{fast}}$.

Therefore, the level of v implementing a cutset U is

$$\Delta(v, U) = \max_{u \in U}(Level(u) + \delta_{\text{link}}(u, v)). \tag{3.6}$$

Primary inputs are at level 0. If a node v has various cutset options, then to minimise its level, a cutset U that minimises $\Delta(v, U)$ has to be selected.

3.5.3 A Covering Algorithm

Given the above, the covering algorithm can now be developed. The approach is an iterative one that determines a minimal value for $MesMap(v)$ for all nodes v that form roots of LUTs. At each iteration, the previous cover is used to determine whether any changes are desirable. When no further changes are desirable, the iteration ends. A way in which the number of iterations is limited to an upper bound N_{iter} is also provided.

The algorithm maintains a queue of nodes which are required as roots of LUTs. Such a queue is initially composed solely of primary outputs and may grow or shrink according to the requirements encountered during the traversal. Finally, it shrinks to the empty set at the end of each iteration. The attribute $Required(v)$, associated with node v, indicates the number of times the output of v is required by other LUTs. Initially, $Required(v) = 0$ for every node. At the start of each iteration, the queue is initialised with all the primary outputs. The values of these primary output node's $Required(v)$ is set to $|V|$, where V is the set of nodes in the DAG. It is important that $Required(v)$ remains greater than zero for these nodes at all times. It will be shown shortly that generally, the $Required(v)$ is incremented and decremented, but an initial value equal to or larger than $|V|$ guarantees that it is always greater than zero.

The main part of the iteration is to select the best cut on the basis of the cuts available and on the measure $MesMap(v)$ for a given node v. Essentially, the question asked is whether the current cover would still be the best one given other covering conditions. Notice that if u is an input of the LUT based at v, then $Required(u) \geq 1$. If $Required(u) > 1$, then it may be propitious to maintain u as an input of v and therefore retaining the cutset that includes u. Otherwise, another cutset may give better results.

In comparing the cutsets of v, the following criteria are applied in diminishing order of strength to select the best one, $U^\star$.

1. $U^\star$ must yield the minimal $MesMap(v)$.
2. $U^\star$ should minimise $\Delta(v, U^\star)$.
3. $U^\star$ should be of maximal size.

Each successive criterion is only applied where there is a tie in the preceding one.

The algorithm MAP-NODES does the following.

1. Initialise the set of LUTs that cover the DAG, $\mathcal{L}$. Also, initialise $Mark(v)$ and $Required(v)$ for each node v. The remaining commands are executed until no further change is possible or until N_{iter} iterations have been executed.

2. Set $Required(v)$ for each primary output v to the total number of nodes in the DAG.

3. A set Θ is used to keep track of the nodes that are processed during one covering iteration. Checking Θ prevents re-mapping the same node within the same iteration.

4. Define a queue Q to be composed of all primary output nodes. Because the first node processed may influence the final outcome, this queue is shuffled so that a new node starts the covering in each main iteration. Let $\Theta = Q$.

5. While Q is not empty, dequeue the first element v and do the following.

   (a) Let $U_0^\star$ be the best cutset of v at this stage.

   (b) Undo the current cover of v.

   (c) Compute $MesMap(v)$ again and select the best cutset $U_1^\star$. If $U_0^\star \neq U_1^\star$, then the cover has changed.

   (d) For each $u \in U_1^\star$, update $Required(u)$ and insert it into $\mathcal{L}$.

   - If $u \notin \Theta$, then enqueue it into Q.
   - Restore the cover of u if it was undone and if $u = (U_0^\star \cap U_1^\star)$.

The pseudo-code of the algorithm is listed below.

```
1  proc MAP-NODES (G, N_iter)
2    comment: G = (V, E).
3    comment: Required(v) = number of fanouts of v.
4    foreach node v ∈ V  do
5      Mark(v) ← UNMAPPED
6      Required(v) ← 0
7    endfor
8    L ← set of primary outputs of G.
9    iteration ← 0
10   changed ← true
11   Q ← ∅
12   while (changed = true ∧ iteration < N_iter)  do
13         changed ← false
14         Θ ← ∅
15         comment: Θ records the nodes that have been enqueued.
16         iteration ← iteration + 1
17         foreach primary output node v ∈ V  do
18           Required(v) ← |V|
```

```
19          ENQUEUE(Q, v)
20          Θ ← Θ ∪ {v}
21      endfor
22      Shuffle Q
23      while Q ≠ ∅ do
24          v ← DEQUEUE(Q)
25          Let U*_0 be the current best cutset of v.
26          if Mark(v) ≠ UNMAPPED then
27              UNDO-MAPPING(v, L)
28              restore ← true
29          else
30              restore ← false
31          endif
32          Mark(v) ← iteration
33          comment: Register that v has been mapped.
34          Compute MesMap(v)
35          Let U*_1 be the new best cutset of v.
36          changed ← changed ∨ U*_1 ≠ U*_0
37          U ← U*_1 ∩ U*_0
38          foreach internal node u ∈ U*_1 do
39              Required(u) ← Required(u) + 1
40              L ← L ∪ {u}
41              if restore = true ∧ Required(u) = 1 then
42                  if u ∈ U then
43                      RESTORE-MAPPING(u, iteration − 1, L)
44                  endif
45                  endif
46                  if u ∉ Θ then
47                      ENQUEUE(Q, u)
48                      Θ ← Θ ∪ {u}
49                  endif
50              endfor
51          endwhile
52      endwhile
53  end.
```

The complexity of MAP-NODES depends on the number of iterations executed. This is limited to an upper bound of N_{iter}. The complexity of the computation of $MesMap(v)$ is $O(\kappa_\alpha^2)$ since there are no more than $(\kappa_\alpha - 1)$ cutsets, the largest being of size κ_α. For the queue, Q, any node is enqueued at most once. Therefore the sub-loop processes no more than $|V|$ nodes. Since κ_α is a constant, it is deduced that the complexity of MAP-NODES is $O(N_{iter}C(\kappa_\alpha) \cdot |V|)$, where $C(\kappa_\alpha)$ is a polynomial function of κ_α.

Properties: Some properties can be derived about MAP-NODES.

PROPERTY 3.5.1 *For any node u, if $Required(u)$ is initially greater than or equal to $|V|$, then*

Required(u) *is always greater than 0.*

Proof: *Required*(u) is only decremented if it happens to be in the cutset of some node v. Once this happens, either *Required*(u) is restored to its value or otherwise, it will no longer be in the cutset of v. Therefore, the maximum number of times *Required*(u) is decremented continuously is $(|V|-1)$ and so *Required*$(u) - (|V|-1) \geq 1$. ■

PROPERTY 3.5.2 *If G is a tree, then Required$(v) \leq 1$ for all internal nodes $v \in V$ in the first iteration.*

Proof: All nodes in a tree have only got a single fanout. If a node u occurs in the cutsets of two different nodes v_1 and v_2, then either $C_{v_1} \subset C_{v_2}$ or $C_{v_2} \subset C_{v_1}$. Without loss of generality assume that $C_{v_2} \subset C_{v_1}$. Therefore, if v_1 causes u to be enqueued, then v_2 is never enqueued and *Required*(u) becomes 1. All nodes that are not enqueued have *Required*$(u) = 0$. ■

PROPERTY 3.5.3 *For a tree, only one iteration is executed to compute the cover.*

Proof: For all nodes v in the tree, *Mes*(v) = *MesMap*(v) so that $U^\star$ is always the same. Hence, the variable *changed* remains equal to *false* throughout the inner loop. At the end of the inner loop, the conditional forces the termination of the main loop. ■

This is a useful property since the optimal cover of trees can be guaranteed. In the general case, a value for ς must be established that yields the best cover for DAGs. A value for N_{iter} that gives good results for the average circuit is also sought.

The overall algorithm executes the following steps.

1. Traverse the DAG in topological fashion from the primary inputs. For each internal node v:
   (a) Construct $\hat{\mathcal{N}}_v$.
   (b) Compute the cuts of $\hat{\mathcal{N}}_v$ and the corresponding cutsets using COMPUTE-CUTS.
2. Use MAP-NODES to cover the DAG.
3. Implement the LUTs and assign them to logic blocks.

3.6 Experiments

The following experiments relate to the algorithms presented in this chapter. The overall results are presented in Chapter 6.

3.6.1 Reduction of Flow Networks

The first experiment tests the gain in using reduced networks in the computation of covers for a set of benchmark circuits. To do this, the circuits were decomposed, using the mixed bin-packing/kernel decomposition method, and then mapped, using MAP-NODES and its attendant algorithms. One experiment uses the reduced networks, $\hat{\mathcal{N}}_t$, whereas the other uses the full networks, $\tilde{\mathcal{N}}_t$. The number of vertices and edges required by these networks are counted and then compared. The percentage reductions that the reduced networks achieve are evaluated. The circuits are then grouped according to the extent of the gain. The results are tabulated below both in terms of vertices and edges, quoting the total number of circuits in each defined group.

% red.	Same	0 – 5	5 – 10	10 – 15	15 – 20	20 – 25	25 – 30	30 – 35	35 – 40
Vert.	35	90	16	7	1	1	0	2	0
Edges	35	79	24	5	4	1	1	0	3

From the 152 circuits tested, most register a reduction of between 1 and 5% in the number of both vertices and edges used. For those 35 that do not change, the reduced networks are the same as the full networks. Overall, the average reduction is 5.2% for edges and 4.0% for vertices.

3.6.2 Parameters

The following table shows the number of circuits that require various numbers of iterations for MAP-NODES to converge.

Iterations	1	2	3	4	5	> 5
No. of Circuits	51	10	43	7	1	1

In the experiment, $\varsigma = 1$ and $\delta_{\text{slow}} = \delta_{\text{fast}} = 1$. One circuit does not converge even for $N_{iter} = 10$. This is due to some covers being undone in one iteration only to be re-instated in the next. For the remaining experiments, N_{iter} is set to 6 since this allows almost all the covers to converge.

The second experiment tests various values for ς. Again, δ_{slow} and δ_{fast} are set to 1. The following table shows the comparison of the percentage *decrease* in numbers of LUTs and logic blocks used for different values of ς against $\varsigma = 0$.

ς	0.1	0.2 & 0.3	0.4	0.5	0.6 & 0.7	0.8 & 0.9	1.0
$\sum$ % LUTs	83.11	83.42	85.25	87.38	114.39	108.74	134.27
$\sum$ % *logic blocks*	96.58	96.92	99.30	102.46	144.81	140.29	129.77

A total of 150 circuits were tested. In terms of logic blocks, the optimal value for ς is 0.6 or 0.7. The value of $\varsigma = 1.0$ yields fewer LUTs than $\varsigma = 0.7$, but with $\varsigma = 0.7$, there are more β LUTs and fewer α LUTs. Hence the reversal in the number of logic blocks needed. Because the number of logic blocks is of greater importance, it will be assumed that $\varsigma = 0.7$ unless otherwise stated.

The delay between LUTs and logic blocks depends on the technology. In the next experiment, various models for the delay parameters were considered. The ratio $\delta_{\text{slow}} : \delta_{\text{fast}}$ was varied from 1:1 to 4:1 and the numbers of LUTs and logic blocks needed were counted after the covering step. The results obtained show that out of 150 circuits, 3 register an improved cover and 3 register the opposite for ratios greater than 1:1. The total percentage improvement in number of logic blocks used is over 12 for the 6 circuits that change. A surprising observation is that the change in ratios up to 4:1 has the same effect throughout, e.g., 7:2 yields the same results as 5:3 or 2:1. It can be concluded that the covering is generally insensitive to the delay model except when the ratio 1:1 is used. The ratio 2:1 is chosen as the default.

3.6.3 Summary

The experiments lead to the conclusion that the reduced flow networks have a significant effect on the MAX-FLOW algorithm's speed and memory requirements. Secondly, the step function gives the best results if each repeated LUT is counted as $\frac{1}{3}$ ($\varsigma = 0.7$) rather than 1. Thirdly, only six iterations are sufficient to achieve convergence in the global covering algorithm. Finally, a delay ratio of 2:1 for δ_{slow}:δ_{fast} will suffice for optimising the area utilisation. □

Chapter 4

Bin-Packing Decomposition

Decomposition is an indispensable part of technology mapping. Since the general Boolean DAG can contain infeasible nodes, it may be impossible to map all its nodes onto LUTs. Once the nodes have been decomposed into smaller nodes, then they can be mapped onto the LUTs. The covering step was addressed in Chapter 3. This chapter deals with the decomposition of a node based on the bin-packing problem.

4.1 Introduction

Various methods have been employed for node decomposition. They rely on functional decomposition techniques, which are non-trivial tasks leading to computationally expensive algorithms. *Cube-packing* is a decomposition method that models this problem as a *bin-packing* problem.

The *bin-packing* problem can be stated as follows: *given a set L of items with specific weights between 0 and 1, place these items into a minimum number "bins" of unit capacity.* The total weights of the items within a single bin must not exceed 1. Some of the practical problems that can be modelled as the bin-packing problem are: table formatting, pre-paging and file allocation [39]. In itself, the bin-packing problem is NP-complete and thus heuristics must be employed to solve it. Two commonly used heuristics are *first-fit (FF)* and *best-fit (BF).* Both are iterative algorithms that process the set L in the order of its elements. The FF heuristic places the i^{th} item in the first bin into which it fits. In contrast, the BF heuristic places an item into the bin with the most weight that will accept it. Modifications to these algorithms require that the set L be sorted in decreasing order. Hence the corresponding *first-fit decreasing (FFD)* and *best-fit decreasing (BFD)* algorithms.

A general node in a Boolean DAG implements a Boolean function of a given number of variables defined by the edges incident to the node. These functions may be represented in the *sum-of-products* form; each product term is a *cube.* The important attribute of a function is not the number of cubes it has but rather the number of its distinct variables. The number of literals

in any cube of the function must, by definition, be no more than the number of distinct variables. The objective, therefore, is to reduce the function to a set of cubes (including single literal cubes) such that the node is feasible. The result of this decomposition is then a tree of logic blocks that together implement the original function.

It is possible to model the problem of finding a decomposition for a node's function as a bin-packing problem. To do this one visualises the cubes in the the node's function as the items to be packed. The capacity of each bin is the maximum number of inputs permitted for any node and the "weights" of the individual cubes is the number of literals in the cubes. This has been the approach adopted by Francis et al. [29] and Murgai et al. [52]. Francis et al. use the FFD strategy, whereas Murgai et al. use the BFD strategy. When cubes are packed together, the bin weight is the total number of distinct inputs that have literals in the cubes. This differs from the conventional bin-packing since the bin sizes are determined more by the affinity of the cubes to one another rather than their sizes. It has been shown that if the cubes are disjoint, and for $k < 5$ these heuristics compute the optimal node decomposition solution [52, 29].

Very good results can be achieved with cube-packing under certain conditions. Additionally, cube-packing is much simpler to implement and much quicker than all other schemes. To overcome some of the deficiencies of cube-packing, a modified bin-packing scheme is needed. Furthermore, forming disjoint cubes and cascading derived nodes improves the performance of the cube-packing scheme.

The cube-packing algorithm presented here closely follows the one implemented by Murgai et al. [52]. The criterion applied in the approach minimises the *increase* in the weight of the bin at each step, i.e., if an item can fit into more than one bin, the packing that minimises the increase in bin weight is chosen. The difference between the problem solved here and the one solved by Murgai et al. is that there are two types of "bins" with different capacities.

In the following sections, a *cube-packing* algorithm that handles two types of "bins" is presented. The algorithm accepts a general Boolean DAG and decomposes any infeasible nodes that exist within the DAG. Such a node would be implemented by a tree of feasible nodes of both types. The root node of the decomposition tree supplies the original node's output.

Firstly, in Section 4.2 the decomposition of a single cube is considered. This is a much simpler problem than decomposing multi-cube functions and can be solved optimally. In Section 4.3, a modified bin-packing algorithm, which computes a multiplicity of solutions, is presented for use in cube-packing. Section 4.4.1 deals with pre-processing nodes before packing the cubes so that the cube-packing can achieve better results. Similarly, Section 4.4.2 covers the issue of cascading nodes and shows that the decomposition tree can be minimised in this way. Section 4.5 deals with the decomposition of a multi-cube function into a tree of heterogeneous nodes. It brings together the concepts of the preceding sections and introduces a method for building a decomposition tree. A more powerful approach that mixes cube-packing decomposition and kernel-decomposition is also

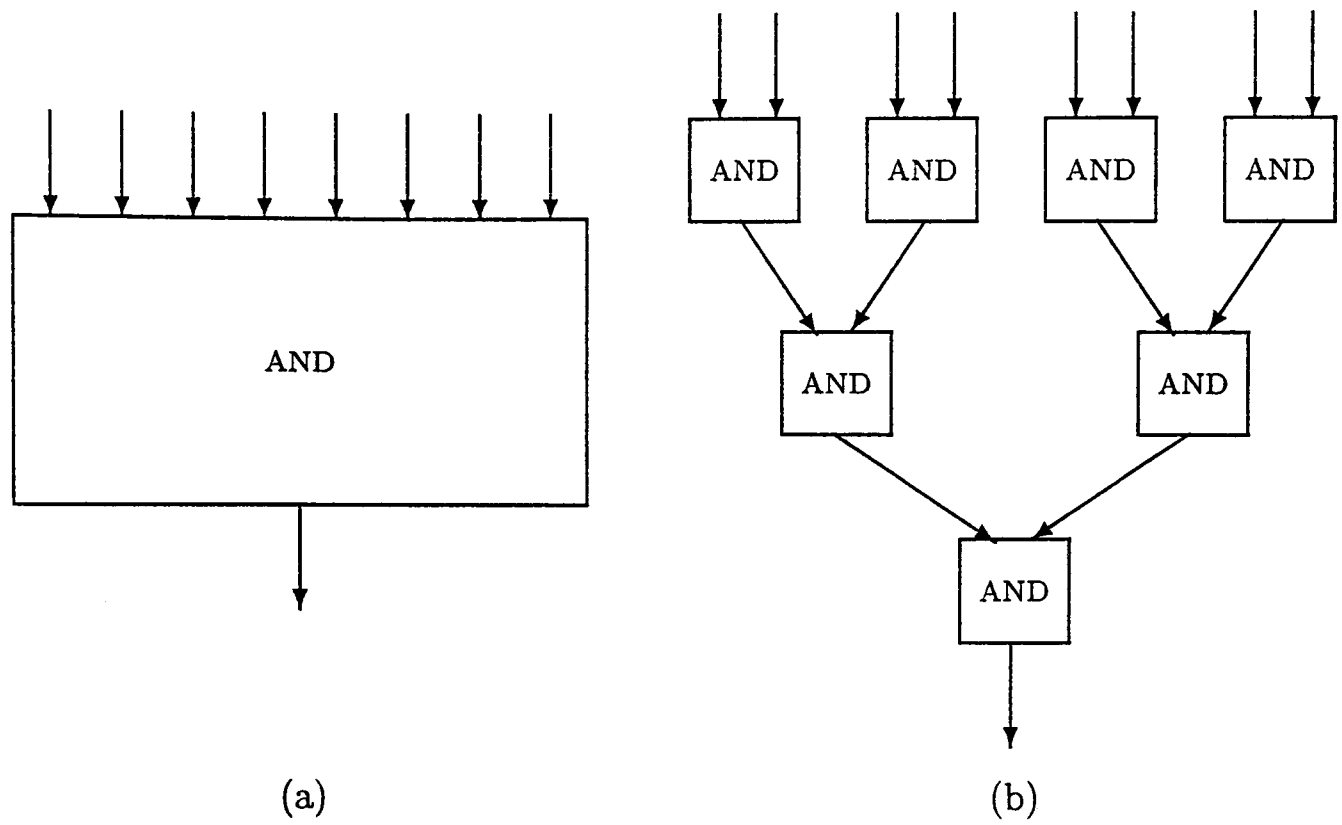


Figure 4.1: Decomposing an AND gate. In this example the result in (b) is a tree of 2-input gates.

presented in Section 4.6. Finally, results from various experiments on cube-packing are presented.

4.2 Decomposing Cubes

This section is concerned with the decomposition of a cube. It is possible for a node to have a single-cube function. Some of these may be infeasible and therefore need to be decomposed.

A cube is simply a product term. Its size is the number of literals present in the term. Therefore, its decomposed form is essentially a tree of AND gates with the root gate providing the value of the entire cube. In the case considered here, two types of AND gates are considered: κ_α-input and κ_β-input gates. Two κ_α-input gates and a κ_β-input gate can be packed into a single logic block. The decomposition should seek to use a minimum number of logic blocks. A second aspect addressed in this section concerns minimising the depth of the decomposition tree. This is important if the depth of the node (and hence the depth of the DAG) is to be minimised.

4.2.1 An Optimal Scheme

Consider the problem of forming a k-ary tree for a cube. For example, Figure 4.1 shows how an 8-input AND gate can be decomposed into a tree of 2-input gates. By equating k to κ_{LB} the solution can be translated into a tree of logic blocks. This is easy to see if the cluster of two α nodes and their descendent β node are regarded as single output "hyper-nodes" with κ_{LB} inputs, it then follows that the optimal tree formulation of the k-ary decomposition is equal to that of

the κ_α,κ_β-input gate problem.

The following decomposition scheme is implemented on a cube with s literals.

1. Divide the literals into $\frac{s}{k}$ groups each of size k.
2. Each group can be covered by a k-input gate giving a single output. Collect the resulting $\frac{s}{k}$ into a new set.
3. Solve recursively for this new set.

4.2.2 Minimising the Depth of a Cube's Decomposition Tree

The notion of levels is extended to include literals. If the output of a node n has a literal z, then $Level(z) = Level(n)$. Let p be the highest level of all the fanins of n. Therefore, $Level(n) = p+1$. It follows that to minimise $Level(n)$, p must be minimised. A set G is defined as the subset of $\mathcal{F}_{\text{IN}}(n)$ such that all nodes in G have level p. The minimisation of p then reduces to the minimisation of the levels of all nodes in G. Carrying on this argument leads to the conclusion that the leaf nodes in the decomposition tree must have a minimal level.

The literals are first sorted in decreasing order of the levels. Let L be the set of literals in the cube. The following algorithm generates a k-feasible tree for the cube.

```
1  proc DECOMPOSE-CUBE0 (v, C, k)
2     comment: C is a cube composed of the literals L = {l_1, ..., l_n}.
3     Sort L in descending order of the levels of each literal.
4     while (|L| > k) do
5           L* ← {l_i ∈ L | 1 ≤ i ≤ k}
6           g ← ⋀_{l∈L*} l
7           Substitute for g in the function of v
8           L ← L − L*
9           z ← Literal(g)
10          L ← L ∪ {z}
11    endwhile
12 end.
```

In DECOMPOSE-CUBE0, it is assumed that line 8 leaves L sorted and line 10 inserts z at the correct position within L.

PROPOSITION 2 *The procedure* DECOMPOSE-CUBE0 *computes a minimum level tree decomposition for a given cube.*

Proof: The proof is an inductive one. At the first iteration, the node g certainly has a minimum level of any k-input node that could be devised from the elements in L. This condition is maintained at all point in the iteration. Therefore all nodes in the tree have minimal levels. Finally, the root node's level is minimal if all its input nodes have minimal levels. Since this is true, therefore the root node also has a minimal level. ■

4.2.3 Heterogeneous Cubes

DECOMPOSE-CUBE0 above assumes a homogeneous size of AND gates. It needs to be modified to produce the heterogeneous logic block structure. Indeed, if κ_{LB} literals are grouped together, then a logic block of two κ_α-input and one κ_β-input AND gates connected such that the first two feed the third can be formed. Therefore, the only problem with DECOMPOSE-CUBE0 is that it might terminate with $|L| \geq \kappa_\alpha$. This would leave the cube as κ_α-infeasible. To correct this, the terminal conditions of DECOMPOSE-CUBE0 need to be examined. They are divided into three categories:

1. $|L| \leq \kappa_\alpha$;
2. $\kappa_\alpha < |L| < 2\kappa_\alpha$; and
3. $2\kappa_\alpha \leq |L| < \kappa_{\text{LB}}$.

The first condition requires no further action since the cube is feasible. The second condition is easily remedied by creating a κ_α-input gate. This removes κ_α literals from L but introduces a new one. Let n be the number of literals remaining in L after this operation. Then

$$\begin{aligned} n &= |L| - \kappa_\alpha + 1 \\ &< 2\kappa_\alpha - \kappa_\alpha + 1 \\ &= \kappa_\alpha + 1 \end{aligned}$$

Hence, $n \leq \kappa_\alpha$ as required.

The final condition dictates that at least two κ_α-input gates be created. This time two new literals are added to L giving,

$$\begin{aligned} n &= |L| - 2\kappa_\alpha + 2 \\ &< 2\kappa_\alpha + \kappa_\beta - 2 - 2\kappa_\alpha + 2 \\ &= \kappa_\beta \end{aligned}$$

Hence, $n < \kappa_\beta < \kappa_\alpha$. It is possible to use a κ_β-input gate here and generate a single, final literal. Since this gate would be under-utilised, this is not done.

Adding the above steps to DECOMPOSE-CUBE0 gives a new procedure called DECOMPOSE-CUBE1. The initial sorting of DECOMPOSE-CUBE0 is bounded by $O(n\log(n))$. In the i^{th} iteration it takes $O(\log(n_i))$ comparisons to locate where to insert z in L^*, where $n_i = |L^*|$. In general, it takes linear time to form the new L. There are $(n-1)/(k-1)$ iterations all together, which gives a time complexity of $O(n^2/k + n\log(n)) \equiv O(n^2)$.

4.2.4 Cubes in Multi-cube Functions

Generally, a function is expected to comprise of more than one cube, say $f = c_1 + \cdots + c_i + \cdots + c_m$. If c_i is decomposed into c_i^*, then $f = c_1 + \cdots + c_i^* + \cdots + c_m$. The question that relates to multi-cube

functions is the following: *from the set of literals in c_i, which ones should be retained in $c_i^\star$ in order that the cube-packing finds a minimal number of bins for the cubes of f?* An answer to this question may be deduced from the following example.

EXAMPLE 4.2.1 *Let $f = x_1x_2x_3 + x_2x_4x_5x_6x_7 + x_5x_8x_9$. For a bin capacity of 4, the second cube has one literal more than permitted. Four literals must be extracted from it and represented by a single literal. There are 5 possible ways of selecting four literals from a set of 5. Consider two such groups: $y_1 = x_4x_5x_6x_7$ and $y_2 = x_2x_4x_5x_6$.*

- *$f_1 = x_1x_2x_3 + y_1x_2 + x_5x_8x_9$ can be packed into two bins: $[(x_1, x_2, x_3, y_1), (x_5, x_8, x_9)]$.*
- *$f_2 = x_1x_2x_3 + y_2x_7 + x_5x_8x_9$ can only packed into three bins: $[(x_1, x_2, x_3), (x_5, x_8, x_9), (y_2, x_7)]$.*

It is seen that f_1 is better than f_2 because of the choice of x_2 in f_1 which is shared with the first cube. On the other hand, x_7 is unique to the second cube, making it more difficult to pack it with other cubes.

From the above example, it may be inferred that whenever a cube is decomposed, the remainder ought to contain those variables that are shared among several cubes. Hence, an attribute *Count*(x) is introduced that indicates the number of cubes in which the variable x is present. Clearly, *Count*(x) is bounded by 1 and the total number of cubes within the function. Of the two sorting keys, *Level*(x) and *Count*(x), *Count*(x) is made the stronger since the size of the decomposition is more important than its depth. With this modification, DECOMPOSE-CUBE0 and DECOMPOSE-CUBE1 are revised so that the sorting is based on these criteria. The procedures remain the same otherwise.

This modification no longer guarantees that the resulting decomposition tree is depth-optimal. Reversing the sorting keys priorities would yield such a tree. Also, for disjoint cubes, *Count*(x) = 1 for all x, which means that the same decomposition tree is computed. The modified version of the procedures is referred to as DECOMPOSE-CUBE (assuming that it means DECOMPOSE-CUBE0 in homogeneous cases and DECOMPOSE-CUBE1 otherwise).

4.3 Extensions to Bin-Packing

This section presents a new cube-packing algorithm, based on the BFD bin-packing algorithm. Recall that the BFD algorithm processes a list L of items sorted in decreasing order of their sizes. When an item is being packed, a bin B_j is sought for it amongst the existing ones such that:

1. B_j will accommodate the item without exceeding its capacity; and
2. B_j is filled to the highest level amongst the candidate bins.

The following notation is used in this discussion.

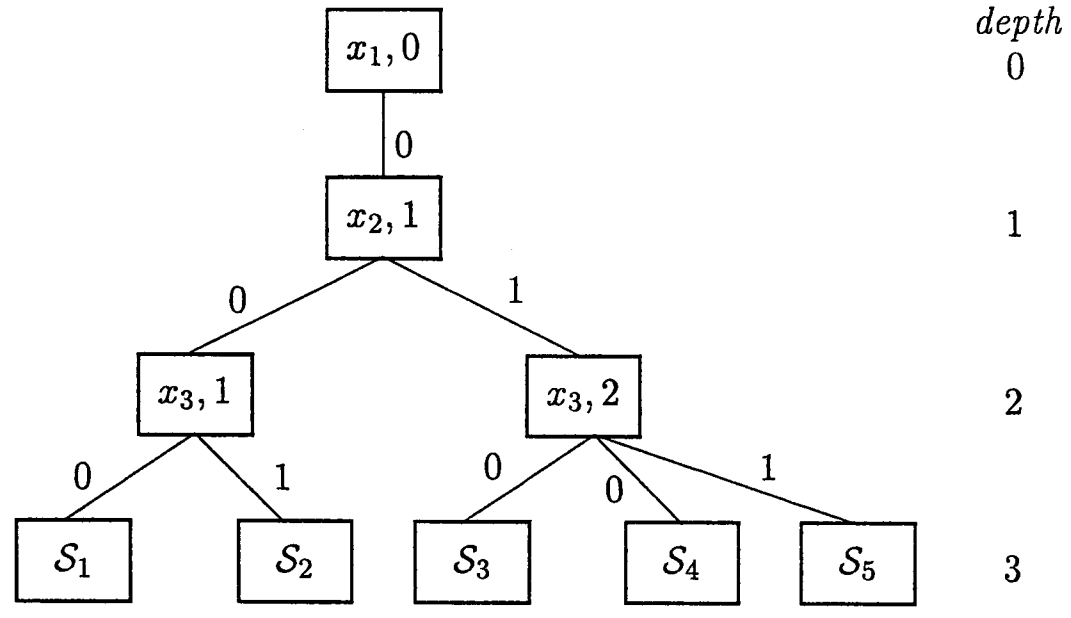
- C is the fixed capacity of all bins. Assume that $C = 4$ unless otherwise stated.
- $W(B)$ is the *weight* of the contents of a bin B. By definition, $0 \leq W(B) \leq C$.
- $I(y)$ is the input variable set of a cube or bin y, e.g., $I(p \cdot q \cdot r) = \{p, q, r\}$.

In terms of cubes, the packing of bins and the weights of these bins assume special meanings. Two or more cubes packed in the same bin implies a disjunction over these cubes. The weight of the bin is the number of distinct inputs to the resulting disjunction. For instance, placing the cubes $\{pqr,\ qu,\ pt\}$ into a single bin yields the disjunction $(pqr + qu + pt)$ and a weight of 5, which is the number of distinct variables. It will be assumed that L is non-empty and that it is composed of the items $\{x_1, \ldots, x_m\}$.

The crucial departure of the approach adopted here from previous methods is that up to τ solutions, $S_1, \ldots, S_\tau$, are maintained at any stage of the packing process. As τ tends to infinity, the algorithm implements an exhaustive search over the bin-packing solution space. To aid this discussion the solution space is modelled as a rooted tree of height m. The tree has the following properties.

1. The root vertex r represents the start of the packing, i.e., when x_1 is just about to be packed. Each *internal* vertex at depth $(i-1)$ represent a similar situation for item x_i. Therefore, the leaf vertices indicate the termination of the packing.
2. An edge leading from a vertex to its child represents a unique packing option for the current item given a particular set of occupied bins. It has a cost associated with it. If the packing involves using a new bin then the cost is 1; otherwise the cost is 0.
3. An internal vertex, x, has an attribute $Bins(x)$ which is equal to the number of occupied bins along the path from the root to x. Thus, if u is the parent vertex of v, then $Bins(u) \leq Bins(v) \leq Bins(u) + 1$.
4. If $Bins(x) = p$, then the *degree* (number of edges) of x is no more than $(p+1)$. This arises from the fact that x can be packed into no more than p different occupied bins as well as into a new bin.
5. The sum of the costs of edges along any path from the root to a leaf vertex gives the number of bins used in the packing.

From the above, it will be seen that the number of *complete* paths (from the root to a leaf vertex) is equal to the number of possible packing solutions.



$\mathcal{S}_1 = \{[x_1, x_2, x_3]\}$; $\mathcal{S}_2 = \{[x_1, x_2], [x_3]\}$; $\mathcal{S}_3 = \{[x_1, x_3], [x_2]\}$; $\mathcal{S}_4 = \{[x_1], [x_2, x_3]\}$; $\mathcal{S}_5 = \{[x_1], [x_2], [x_3]\}$.

Figure 4.2: An example of a solution tree for three items. Each internal vertex shows the item and the number of occupied bins. The leaf vertices show the final packing.

EXAMPLE 4.3.1 *As an illustration, consider the example of three items* $\{x_1, x_2, x_3\}$ *that can be packed into a single bin. The* solution *tree for this example is given in Figure 4.2, where* $\mathcal{S}_1$ *is the optimal solution.*

Only the least costly complete path is of interest. There is also a limit to constructing a maximum of τ paths at each depth. In pruning the branches in the tree, one observation prevents the exploration of paths that would certainly lead to sub-optimal solutions. This follows from the following property.

PROPERTY 4.3.1 *If at a depth* i *there is an edge leading from the vertex to a child at depth* $(i+1)$ *such that it has a cost 0 and the selected bin does not increase in weight, then all other edges leading from the vertex cannot give a better solution.*

Proof: If the cost of an edge is 0 and the selected bin does not increase in weight, then the packing remains the same from one depth to the next. So whatever packing possibilities are available by using a different option are also available with this option. Hence, the optimal packing is still available after this. ■

Therefore, more than one path from a vertex are generated only if none of its edges to any child has cost 0. In Figure 4.2, only the leftmost path leading to $\mathcal{S}_1$ would be produced.

A key aspect to the approach is the selection of the τ best paths at any depth. A greedy strategy is used which selects τ of the least costly paths found so far at the current depth. To distinguish between the vertices at the same depth, a label is attached to each vertex in the form

$x(1), x(2), \ldots$, so that the cost of the path from r to $x(j)$ is $c(j)$. Thus, $c(j) \leq c(j+1)$ for all j. Furthermore, let S_j be the set of occupied bins within this path. The expansion of the solution tree by one level from a depth $(i-1)$ at vertex $x_i(j)$ is performed as follows:

1. Find p bins into which $x_i(j)$ can be packed within the bins of S_j.
2. If $p < \tau$, then consider placing $x_i(j)$ into a new bin. Otherwise, if $p > \tau$, then select τ possibilities.

This approach could be applied equally to an FFD-based or an BFD-based algorithm. One aspect of cube packing must be borne in mind, however. If the input set of a cube x is a subset of that of a bin B, $I(x) \subseteq I(B)$, then placing x in B does not increase $W(B)$. The following instance shows how a FFD approach may be inferior to a modified BFD one.

EXAMPLE 4.3.2 *Let $C = 5$, $I(B_1) = \{a, b, c\}$ and $I(B_2) = \{c, d, e\}$, and B_1 appears before B_2 in the bin list. Suppose that one wants to pack a cube $x = d \cdot e$. Here is how the two approaches pack x.*

- FFD: *x fits in B_1. So $I(B_1) = \{a, b, c, d, e\}$ and $I(B_2) = \{c, d, e\}$.*
- BFD: *x fits in both B_1 and B_2, but B_2 would have a lower increase in weight (none). So $I(B_1) = \{a, b, c\}$ and $I(B_2) = \{c, d, e\}$.*

Clearly, the BFD gives a better solution since neither bin increases in weight. The above observation suggests that a modified BFD approach is preferable to the the FFD one. Hence, the algorithm is based on the BFD approach as did Murgai et al. [52]. It should be noted, however, that if the cubes were mutually disjoint, the two approaches would give almost the same results [39].

4.3.1 The Construction of a Solution Tree

Let v be vertex of a solution tree, T. Each vertex, v, has the following attributes.

1. *Parent*(v) is the parent of v in the tree. The root r is the only vertex without a parent.
2. Each v is associated with an item x. This item has one or more cubes with the variable set given by $I(x)$.
3. The vertex v represents the packing of an item x into a bin given by an index *Bindex*(v).
4. After x is packed into the appropriate bin, it may introduce new variables into the bin, and thus increase its weight. These new variables, if any, are given by the set $I(v) \subseteq I(x)$. Note that $I(v)$ may be empty but not $I(x)$.

It is assumed that a procedure NEW-VERTEX(x, p, i, A) creates v such that: $Parent(v) = p$, $Bindex(v) = i$, and $I(v) = A$. When comparing two options v_1 and v_2, three attributes are examined. In order of their importance, they are as follows:

1. if one leads to fewer bins being used then it is preferred;
2. v_1 is better than v_2 if it causes a smaller increase in the weight of the bin it occupies; and
3. the option that results in a bin among the currently occupied ones with the most free space is preferred.

Extending a Path

A procedure EXPLORE-PATH is defined that, given a leaf vertex v in T, works out the possible new leaf vertices for T if an item x were packed into the bins. Assume that there are n bins currently in use in a solution given by the leaf vertex v. This set of bins is denoted by $\mathcal{B} = \{b_1, \ldots, b_n\}$. Also, let the set of items already packed be $\mathcal{X} = \{x_1, \ldots, x_m\}$, and x be the next item to be packed.

A trace is made of the path $v \rightsquigarrow r$, where r is the root of T. At each vertex u, the bin $b_k \in \mathcal{B}$ corresponding to u is found. Then a check is made on whether x can be packed into b_k. Once r has been reached, the feasible options are selected from which new leaves, with the parent vertex v, are constructed in T. Recall that $Bins(v) \leq m$, since each vertex can introduce at most one new bin. The leaves must be pruned so that they are no more than a specified number. Moreover, if x fits into a bin without increasing that bin's weight then only one leaf is created. This is simply an application of Property 4.3.1. If the quota has not been exceeded, then a new leaf where x is placed into a new bin is also introduced. This ensures that x is packed into at least one bin. Otherwise, the best q are selected, where q is the maximum number of options sought.

Packing Items

The packing of a set of items, $\mathcal{X}$ is now described. The number of possibilities explored at each depth of the solution tree, T, is τ. A packing option is defined to be better than another if it uses fewer bins. In many instances, however, two paths may have the same number of occupied bins. To resolve this tie, the option with more total free space in its bins is defined as the better one. The space in a bin is simply the difference between its capacity and its current level. In using the total free space as a criterion to break ties, it is projected that the more space there is available, the likelier it will be to fit the remaining (and smaller) items into the currently occupied bins. However, this is not always the case since the space may be fragmented over many bins, so that a single item cannot fit into one bin. To counter this, options where the latest item is placed into

its own bin are preferred since this gives some contiguous space within a single bin. In summary, the preference of paths $v \rightsquigarrow r$ is determined by the following criteria in order of strength.

1. The least possible number of bins used.
2. The largest amount of total space remaining in all the occupied bins.
3. The greatest free space in the bin with the smallest weight.

A procedure PACK-BINS takes a set of items, $\mathcal{X}$ and returns a set of bins, $\mathcal{B}$, into which these items are packed. The procedure constructs a solution tree one level at a time, so that all the leaf vertices correspond to the same item. There are two special cases for $\mathcal{X}$. The first is when $\mathcal{X}$ has only one item. In this case a single bin is the only solution. Secondly, when $\mathcal{X}$ has two items, then it can easily be checked if one or two bins are required.

In the more general case, the root r of the solution tree is initialised to correspond to x_1. Hence, the set of leaf nodes $\mathcal{V}$ is initially $\{r\}$. Then each item x_i for $2 \leq i \leq m$ is packed in turn. Using EXPLORE-PATH on each vertex $v_j \in \mathcal{V}$, a new set $\mathcal{V}^\star$ of leaf vertices is computed corresponding to x_i. The number of solutions computed at each expansion of the tree is counted. This is then used to limit the number of solutions sought in EXPLORE-PATH so that fewer options are explored for subsequent vertices within the set $\mathcal{V}$. Initially, up to τ are sought. Once enough expansions have been undertaken, then the new leaves are pruned to the best τ. Hence, at all times, $|\mathcal{V}| \leq \tau$. The pseudo-code is shown below.

```
1  function PACK-BINS (𝒳, capacity, τ)
2    comment: Returns a set of bins. Each bin is set of items.
3    comment: 𝒳 is the set of items to be packed.
4    if 𝒳 = ∅ then
5      return ∅
6    endif
7    comment: Special cases.
8    if |𝒳| = 1 then
9      return {{x_1}}
10   endif
11   if |𝒳| = 2 then
12     if |I(x_1) ∪ I(x_2)| > capacity then
13       comment: These items cannot fit in one bin.
14       return ({{x_1}, {x_2}})
15     else
16         return ({{x_1, x_2}})
17     endif
18   endif
19   comment: 𝒱 is the set of leaf vertices in the solution tree.
20   r ← NEW-VERTEX(x_1, NIL, 1, I(x_1))
```

```
21  comment: r is the root of the tree.
22  V ← {r}
23  comment: Loop for all the remaining items.
24  for i ← 2 to |X| do
25      q ← τ
26      V* ← ∅
27      for j ← 1 to |V| do
28          comment: Expand the solution path: v_j ∈ V.
29          P ← EXPLORE-PATH(v_j, x, capacity, q)
30          V* ← V* ∪ P
31      endfor
32      comment: Select the best solutions for next iteration.
33      Sort V* in descending order of preference.
34      V ← {V*[i] | 1 ≤ i ≤ τ}
35  endfor
36  Let B be the set of bins formed by traversing the path v_1 ⇝ r.
37  return (B)
38 end.
```

The working of the procedure will be illustrated by the following example.

EXAMPLE 4.3.3 *Suppose that the list of cubes is*

$$L = \{(a_1b_1c_3), (a_2b_1c_3), (a_2a_3c_3), (a_1b_2)\}.$$

Assume that the bin capacity is 4, with $\tau = 2$. Let x_i denote the i^{th} element in L. The tabulation below shows how S_1 and S_2 are evaluated at each step.

Step	x_i	*Options*	*Selection*
1	$a_1b_1c_3$	$\{[a_1, b_1, c_3]\}$	S_1
2	$a_2b_1c_3$	$\{[a_1, a_2, b_1, c_3]\}$	S_1
		$\{[a_1, b_1, c_3], [a_2, b_1, c_3]\}$	S_2
3	$a_2a_3c_3$	$\{[a_1, a_2, b_1, c_3], [a_2, a_3, c_3]\}$	S_1
		$\{[a_1, b_1, c_3], [a_2, a_3, b_1, c_3]\}$	S_2
		$\{[a_1, b_1, c_3], [a_2, b_1, c_3], [a_2, a_3, c_3]\}$	
4	a_1b_2	$\{[a_1, a_2, b_1, c_3], [a_2, a_3, c_3], [a_1, b_2]\}$	S_2
		$\{[a_1, b_1, b_2, c_3], [a_2, a_3, b_1, c_3]\}$	S_1
		$\{[a_1, b_1, c_3], [a_2, a_3, b_1, c_3], [a_1, b_2]\}$	

The final step shows that the best solution arises from what was initially a sub-optimal solution at step 2. A conventional BFD strategy would have found three bins instead of the two computed here.

Compared to a conventional BFD approach, this scheme explores τ times as many options at each stage. Since τ is a constant, the complexity of this approach in time and space requirements

is bounded by the same constant, i.e., it has τ times the space requirement of the conventional BFD algorithm. Consequently, it is possible to vary τ depending on the size of the problem being solved. In general, one expects that a larger τ would give better results.

4.3.2 BFD Packing

The key feature of cube-packing (and bin-packing) is how bins are chosen for each item as it is packed. Indeed, this is where the FFD and BFD strategies differ. The following is a brief explanation of the selection of the best bin in the BFD strategy.

Suppose that the set of bins is $\mathcal{B}$, and the node that should be packed is n. The BFD strategy defines the *best* bin as one whose inputs differ least from those of n. Where more than one bin's inputs differ equally, then the larger one is selected. With these criteria, a procedure FIND-BEST-BIN is defined that selects the best bin for n from the set $\mathcal{B}$.

The procedure FIND-BEST-BIN is used to fill a set of bins to their maximum capacity. Given a set of nodes, S, sorted in the appropriate order, it tries to fit each element in S into an existing bin in $\mathcal{B}$. If no bin accepts an element s_i, then s_i is inserted into a *remainder* set R. This set is returned by the procedure so that its elements can be packed into new bins. The procedure FILL-BINS does not introduce any new bins into the set $\mathcal{B}$.

4.4 Extra Features

This section addresses two features that make cube-packing more efficient. The first is a pre-processing operation that makes the cubes as disjoint as possible. The second applies when the decomposition tree is being constructed. It seeks to utilise the nodes efficiently by cascading them.

4.4.1 Common Cube Division

According to Francis, the FFD algorithm finds the optimal packing of the bin-packing for a bin size of $2 \leq k \leq 5$ [28]. The proof of this is an exhaustive analysis of the cases, and it would apply equally to the BFD algorithm. This result shows that in order to achieve optimal cube-packing, the cubes must be disjoint. Generally, cubes of an arbitrary function are not disjoint. Therefore, the first step ought to be to find a disjoint decomposition of the function, if a simple one exists, before embarking on the cube-packing.

The following section is a discussion on how a set of cubes can be made as disjoint as possible. The approach is similar to that in MIS [4] except that a different algorithm is used.

A cube divisor exists for a function f if it can be expressed as $f = c \cdot D$, where c is a cube and D is a disjunction of cubes. A cube c should have a maximum size since this minimises the size of the input set of D and ensures that D is composed of disjoint cubes.

Clearly, any literal in c must occur in each of the cubes of f. This fact makes it easy to determine the maximal cube divisor of f. Such a cube is simply the conjunction of all the literals that occur in all the cubes of f. A procedure FIND-DIVISOR computes c, which may be the logical constant 1. It then performs the division on f, i.e., $D = f/c$.

Consider a variable x that occurs in different phases in two separate cubes, i.e., one cube contains the literal x and another contains $\overline{x}$. By definition, x or $\overline{x}$ cannot be part of c. Moreover, since both x and $\overline{x}$ occur in some of the cubes in D, then D does not have disjoint cubes. At least two cubes in D share a common variable x. This fact imposes a limitation on the efficacy of cube division. If it is possible to invert some cubes, it might be possible to extract x or $\overline{x}$ from all the cubes. Unfortunately, the decomposition of a function with respect to a variable such as x is non-trivial. Consequently, FIND-DIVISOR will not always find a common cube divisor nor a disjoint set of cubes.

As a further refinement, the intersection of various cubes is examined. The intersection of cubes p and q is a cube composed of the literals where p and q agree. For instance, if $p = x_1x_2\overline{x}_3\ \overline{x}_4x_5$ and $q = \overline{x}_2\overline{x}_3\ \overline{x}_4x_5x_6x_7$, then $r = p \cap q = \overline{x}_3\ \overline{x}_4x_5$. The two cubes can be re-written as $p = x_1x_2r$ and $q = x_6x_7r$. Such a substitution has the effect of decomposing the cubes (provided $|r| > 1$) and possibly decomposing the node as well. To achieve maximum benefit from this substitution, r should be shared by as many cubes as possible as well as having a maximal number of literals.

In this case, the only cube intersections of interest are those that have at least κ_β literals. This prevents the accumulation of under-utilised LUTs in the decomposition. Not all cubes that satisfy this condition can be extracted from the node's function at once. Therefore, a way of finding a maximal subset of these cubes for substitution is required. The selection of a cube r_i before r_j is based on the condition that:

- r_i is shared by more cubes in D than r_j; or
- r_i and r_j are shared by an equal number of cubes and $|r_i| \geq |r_j|$.

A procedure FIND-COMMON-CUBES works as follows. First, the set of common cubes C based on the pairwise matching of the cubes in the node's function is evaluated, i.e., for $f = p_1 + \cdots + p_m$,

$$C = \{c \mid c = p_i \cap p_j \wedge |c| > \kappa_\beta \wedge 1 \leq i < j \leq m.\}$$

This is an estimation; it could fail to detect a cube shared by three or more cubes. The computational effort required to find accurate common cubes precludes it. Next, the best cube in C, based on the criteria given above, is sought iteratively. Such a sub-cube c has a new node formed to represent it. The literal of c is then substituted into the original node function. This process may lead to some other sub-cubes in C ceasing to be sub-cubes of the node – such sub-cubes would have been derived initially from the same cubes as c. Therefore, these sub-cubes are removed

from C. Because not all the sub-cubes in C are extracted from the node, it may be necessary to call FIND-COMMON-CUBES again. In the special case where C has one cube only, this cube is extracted and the procedure should not be called again. The pseudo-code is listed in Section A.2.1 in the appendix.

EXAMPLE 4.4.1 *Consider a function*

$$w = a\overline{f}\,\overline{h}p\overline{z} + a\overline{f}\,\overline{h}\,\overline{r}\,\overline{z} + a\overline{y}\,\overline{z} + \overline{f}\,\overline{h}\,\overline{m}p + \overline{f}\,\overline{h}\,\overline{m}\,\overline{r} + \overline{m}\,\overline{y} + \overline{y}z.$$

The first and second cubes intersect to give a cube $c_1 = a\overline{f}\,\overline{h}\,\overline{z}$. *Similarly, the fourth and fifth cubes intersect to give a cube* $c_2 = \overline{f}\,\overline{h}\,\overline{m}$. *Smaller intersections are ignored. The new function is then*

$$w = \overline{y}z + \overline{m}\,\overline{y} + c_1p + c_1\overline{r} + c_2p + c_2\overline{r} + a\overline{y}\,\overline{z}.$$

A procedure CONVERT-CUBES is defined that does the following to a node f.

1. Computes a common cube divisor q of f and the kernel $p = f/q$.
2. Employs FIND-COMMON-CUBES repeatedly on p while changes occur.
3. Decomposes all infeasible cubes of p using DECOMPOSE-CUBE.
4. Returns (q, C), where C is the set of cubes of the new p.

4.4.2 Cascading Nodes

This section discusses a method of cascading nodes in the decomposition tree so as to minimise the size of the tree. Cube-packing decomposition involves packing a set of cubes into a set of bins (nodes) and then connecting the nodes to form a decomposition tree. Cascading applies during the second phase.

Suppose that a set of nodes need to be joined together in a tree-like fashion, where the number of edges incident to any node in the tree does not exceed a given value, k. The primary aim is to limit the number of new nodes that would be created in the resulting tree. Constructing the tree usually involves dividing the nodes into two sets: those that have been connected and those that are disconnected. A connected node is one whose out-going edge is connected to some other node. Initially, all nodes are disconnected. As the tree is built, some nodes are moved from the disconnected set to the connected one until the disconnected set contains one node, which is the root of the tree. If m ($\leq k$) nodes are removed from the disconnected set, they can be use to create a new node v with inputs from the m nodes. The function of v is a disjunction over m literals from the the m nodes. Then v is inserted into the disconnected set of nodes. So the size of the disconnected set decreases by $(m-1)$.

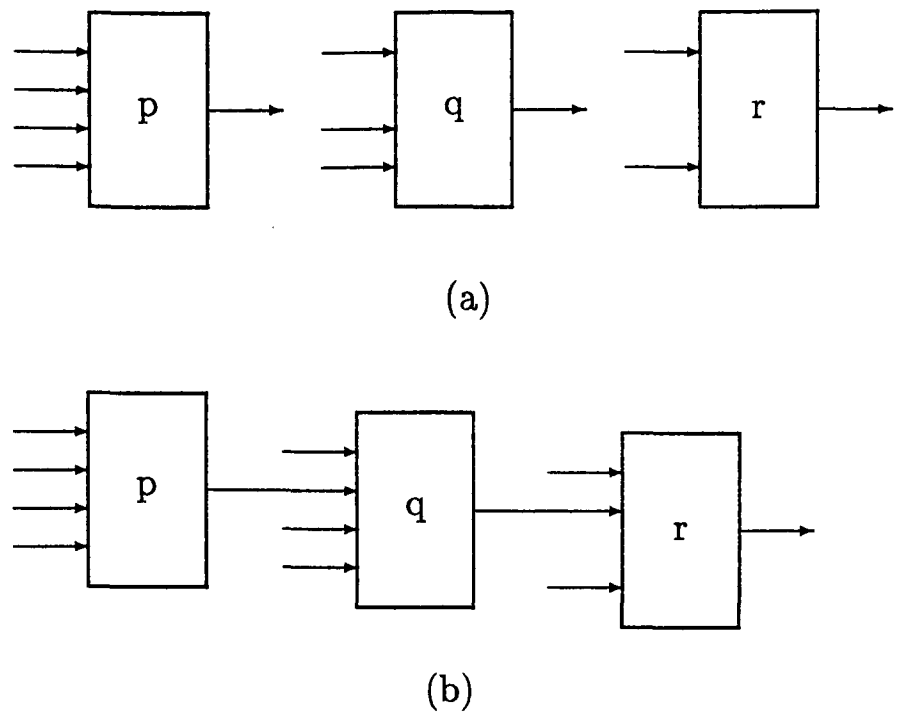


Figure 4.3: Cascading of nodes: (a) original nodes; (b) after cascading ($k = 4$).

To limit the size of the decomposition tree, it is necessarily to maximise the size of any new node that is created. In other words, each new node must have k inputs. Another way of creating a minimal number of new nodes is to maximise the usage of edges to existing nodes.This means that all nodes should have k inputs if possible. Let p and q be two nodes in the disconnected set such that q has $(k-1)$ inputs. If p is connected to q, then the new node for q, say q', will have k inputs. Let $f(x)$ represent the function of a node x and z_x be its literal. In terms of functions, $f(q') = f(q) + z_p$. Figure 4.3 shows an example of cascading three nodes p, q and r for $k = 4$. In this example no new nodes would be required. Without cascading the nodes, an extra node would be required with inputs from the outputs of all three nodes.

An arbitrary set of k-feasible nodes S would offer several cascading possibilities. Mindful of the need to limit the levels of all nodes, two criteria are used to determine the order of the cascade.

1. Larger sized nodes feed into smaller ones. This is because smaller nodes can accommodate more new connections. It is therefore better to leave them in the disconnected set so that more nodes can be connected to them.

2. Nodes with lower levels feed into those with higher levels. The level of a node is determined by its fanin nodes. If a node u has a lower level than a node v, then connecting the output of u to v does not increase the level of the new v.

These criteria may not coincide, however. If both nodes have a free input slot, then the second criteria applies; otherwise, the first applies. Assume that a procedure CASCADE(p, q) places the two arguments in cascade and returns the resulting trailing node. The procedure CASCADE-LUTS is defined to compute a cascade of the nodes in S if any is possible.

Let $S = \{s_1, \ldots, s_n\}$. Then partition S into two sets: $F = \{s_i \in S \mid |s_i| = k\}$ and $E = S - F$, with the order of S preserved in both sets. An iteration is then used to match the first node $f \in F$ to the first one $e \in E$ by applying the procedure CASCADE to them. Since f is no longer usable, it is not considered again. However, e may still have a free input slot and could be used once again. If e is full, it cannot be used again and so it is transferred to F. The iteration ends when either F or E is empty.

4.5 Heterogeneous Cube-Packing

Section 4.2 shows how a single cube function can be decomposed. This section considers multi-cube functions only. The aim is to decompose a function f, given by $f = c_1 + \cdots + c_m$, where c_i is a cube (constant value cubes are disallowed) and $m > 1$, into a tree of heterogeneous nodes. The tree will consist of α and β nodes so that they can be mapped onto logic blocks directly.

The general scheme is to partition the cubes into two sets and then pack them into the two types of nodes. Infeasible cubes can be decomposed as explained in Section 4.2. Given two sets of α and β nodes, the tree is built by carefully matching up the nodes.

Packing cubes into nodes has been discussed in Section 4.3. The approach used here is to consider the two identical problems. One is packing a set of cubes into κ_α-sized bins; the other is packing another set of cubes into $(\kappa_\beta - 2)$-sized bins. The result is two sets of nodes. Partitioning the cubes is simply guided by their sizes. A crucial step is matching the nodes, which is discussed next.

4.5.1 Matching Nodes

One way of optimising the usage of logic blocks is to maximise the number of LUTs in a block. This corresponds to using two α LUTs and one β LUT per block. Minimising the number of logic blocks therefore reduces to minimising the number of α LUTs, since a β LUT only exists when matched to an α LUT.

Let the nodes be partitioned into two sets. Set S_A has nodes with more than $(\kappa_\beta - 2)$ variables; while set S_B has all the other nodes. The figure $(\kappa_\beta - 2)$ is used as a cut-off because the intention is to connect the outputs of two nodes in S_A to two of the inputs of a node in S_B, which becomes a β node (since it will have no more than κ_β inputs). A procedure MAKE-LB assigns two α nodes and one β node to a logic block and returns the literal of the output. Consider, the case where $a_1, a_2 \in S_A$ and $b \in S_B$. The logic block is implements the function $f = f(a_1) + f(a_2) + f(b)$. Its output will have to be connected to the input of another node, unless it is the final output.

The problem is to connect all the nodes in S_A and S_B using as few logic blocks as possible. Applying MAKE-LB repeatedly has two effects. Firstly, it reduces the number of nodes in both S_A and S_B. But it also introduces new outputs which need to be connected to some node, either

an existing one or a new node. Providing there is at least one new logic block filled in an iteration of the matching procedure, the eventual pairing of all nodes in S_A is guaranteed. This arises from the fact that each logic block generates a new literal. This literal is simply a single input node, which can be taken as a new member of S_B. However, the problem with this approach is that it could end up with very long paths from the original inputs to the final output. To diminish this problem, a heuristic is proposed.

Ideally, there should be twice as many nodes in S_A as in S_B to achieve perfect matching. Often, this will not be the case. Supposing $|S_A| > 2|S_B|$. Let there be y logic blocks formed in the matching and the number of nodes left in S_A be m. Now, $S_B = \emptyset$ and the set of new literals, L, has y members (one for each logic block created).

For the next iteration, group $(\kappa_\beta - 2)$ literals in L to form a new node in S_B. Therefore, there are $\frac{y}{\kappa_\beta - 2}$ nodes in S_B and m in S_A. The nodes in S_B will match with twice as many nodes in S_A. Hence, the number of unmatched nodes in S_A after the this iteration is given by

$$u = m - \frac{y}{\kappa_\beta - 2}.$$

In order that $u < m$ (the number of nodes in S_A reduces), y must be greater than zero (at least one logic block must be created). What if $S_B = \emptyset$? This is easily solved solved by introducing a node with a constant Boolean expression (0 for disjunction). Now a logic block can be created such that for $a_1, a_2 \in S_A$, it implements the function $f = f(a_1) + f(a_2) + 0$.

What if $|S_A|$ is much larger than $2|S_B|$? Can the same be done for x pairs of S_A? If this were done, $2x$ nodes would be removed from S_A. In addition, $\frac{x}{\kappa_\beta - 2}$ new members of S_B will be formed, which will match with $\frac{2x}{\kappa_\beta - 2}$. The equation for unmatched nodes in S_A that arises is then

$$u - 2x = 2\frac{x}{\kappa_\beta - 2}$$

$$\Rightarrow \qquad x = \frac{u(\kappa_\beta - 2)}{2(\kappa_\beta - 1)}$$

What should the value of x be? It is not possible to resolve this immediately and so the following heuristic function is applied to x:

$$N(x) = 1 + \xi(x - 1), \tag{4.1}$$

where $0 \leq \xi \leq 1$. Equation 4.1 is a linear function on x. On one extreme ($\xi = 0$), only one pair of nodes from S_A is used (this being the most economical approach). On the other ($\xi = 1$), as many as x pairs are used. The effect of various values of ξ on the decomposition will be demonstrated later.

The matching phase is summarised by the following steps.

1. Find as many perfect matches as possible. These matches yield a set of literals L and are implemented by MAKE-LB. Note that $|S_A| \geq |S_B|$.

2. Compute how many pairs of nodes in S_A should be mapped onto logic blocks without a corresponding node from S_B. The β-node in the logic block implements a disjunction over the literals from the two α-nodes. The number of pairs used in this fashion is determined by Equation 4.1. All the new output literals generated by MAKE-LB are appended to L.

3. Use the literals in L to fill any under-utilised node remaining in S_A. Because these literals are unique, the extended bin-packing of Section 4.3 offers no advantage over the conventional bin-packing. In fact, an FFD strategy works equally well and it is a little faster than the BFD one. Therefore, a procedure is used that given a set of nodes S and a set of literals L, fills each node in S to its capacity by connecting the literals to an unused input slot of the node.

4. Append the remainder in L to S_B.

At the end of this phase, some unmatched nodes remain in S_A and S_B. Moreover, the nodes in S_B may not be utilised economically. This situation is resolved later.

4.5.2 Constructing the Decomposition Tree

This section describes how the decomposition tree is constructed for a node v with the set C of feasible cubes. The procedures introduced previously are used for this purpose. The algorithm executes the following steps.

1. Sort C according to the sizes of the cubes.

2. Define two sets C_A and C_B such that:

   - $C_A = \{c \in C \mid \kappa_\alpha \geq |\mathcal{F}_{\text{IN}}(c)| > (\kappa_\beta - 2)\}$; and
   - $C_B = C - C_A$.

3. Use PACK-BINS to pack C_A into a minimal set $\mathcal{B}_1$ of bins of capacity κ_α.

4. Fill the bins in $\mathcal{B}_1$ with as many cubes in C_B as possible.

5. Use PACK-BINS to pack the remainder in C_B into a minimal set $\mathcal{B}_2$ of bins of capacity $(\kappa_\beta - 2)$.

6. Define S_A and S_B as the sets of nodes implementing $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively.

7. Cascade the nodes in S_A, using CASCADE-LUTS, retaining in S_A the returned set.

8. Balance the nodes such that $|S_A| \geq |S_B|$. The reason for this is that there must be at least one κ_α-node for every κ_β-node in the mapping. A procedure BALANCE-LUTS moves some nodes from S_B into S_A. It fills the existing nodes where possible using the BFD strategy of Section 4.3.2.

9. Until either $|S_A| \leq 1$ or $|S_B| \leq 2$ perform the following.

   (a) Match the nodes in S_A and S_B (see below). The resulting literals from the matching are appended to S_B. Any node that is matched is removed from its set.

   (b) Pack S_B using an FFD strategy. Since most nodes in S_B have disjoint fanins an FFD, BFD or extended bin-packing strategy would yield the same result. The FFD strategy is the natural choice since it is the fastest.

   (c) If $|S_A| < |S_B|$, then balance the sets again using BALANCE-LUTS.

10. The terminal condition is $(1 \leq |S_A| \leq 2) \wedge (0 \leq |S_B| \leq 1)$. The last remaining nodes are packed into a logic block so that there is only one final node, r, that marks the root of the decomposition tree. A variable k indicates its maximum permissible input size, depending on whether it is a κ_β-node or a κ_α-node since r should be filled later on. The possible conditions in terms of $(|S_A|, |S_B|)$ are as follows.

    - (1, 0): r is the remaining node and $k = \kappa_\alpha$.
    - (1, 1): r is the disjunction of the two remaining nodes and $k = \kappa_\beta$.
    - (2, 0): r is either the result of cascading the two remaining nodes and $k = \kappa_\beta$, *or* r is the disjunction of the two remaining nodes and $k = \kappa_\alpha$.
    - (2, 1): r is the disjunction of all three remaining nodes and $k = \kappa_\beta$.

    If r has k inputs, then the result is its output's literal and k is reset to κ_α. The tuple (r, k) is returned.

The above is implemented as a procedure GENERATE-LUTS.

4.5.3 The Overall Algorithm

This section finalises the decomposition of an infeasible node $f = f_1+, \ldots, +f_m$, where $m \geq 1$. The process is accomplished by applying the procedures devised so far. These are the steps employed in the decomposition.

1. Find the largest cube divisor q of f such that $f = p \cdot q$, where $p = p_1+, \ldots, +p_m$, and for $i \in \{1, \ldots, m\}$, $p_i = f_i/q$.

2. Find common cubes among the cubes $p_1, \ldots, p_m$ and decompose any infeasible cube p_i using DECOMPOSE-CUBE (see Section 4.4.1).

3. Decompose q using DECOMPOSE-CUBE (since it is a cube). If q fills an α-node completely, then define t as its output literal; otherwise, $t = q$.

4. Generate the node tree for the cubes of p using GENERATE-LUTS (see Section 4.5.2). Let it return the tuple (r, k). Recall that r is a node and k is its maximum permissible input size.

5. Compute the root of the decomposition tree by merging t and r. This is done by padding r with the sufficient literals from t to makes its input size k (t is a single-cube node). The remaining literals in t are conjoined with the output literal of the new node (if any) to give the root node g.

6. Replace f with g.

The procedure DECOMPOSE-NODE below performs the above.

```
1  proc DECOMPOSE-NODE (f, κα, κβ, ξ, τ)
2    comment: Let f = c1+, ..., +cm.
3    comment: Single cube functions are decomposed specially.
4    if m = 1 then
5      f ← DECOMPOSE-CUBE1(f, f, κα, κβ)
6      return
7    endif
8    Θ ← ∅
9    (q, C) ← CONVERT-CUBES(f, Θ)
10   comment: p = f/q and C = {p1, ..., pm}.
11   comment: First, make sure that q is feasible.
12   if |F_IN(q)| > κα then
13     g ← DECOMPOSE-CUBE1(q, q, κα, κβ)
14     if |F_IN(g)| = κα then
15       Let t be the literal representing the output of g.
16     else
17       t ← g
18     endif
19   elseif |F_IN(q)| = κα then
20       Let t be the literal representing its own output.
21   else
22     t ← q
23   endif
24   (r, k) ← GENERATE-LUTS(p, C)
25   if t ≠ logical constant 1 then
26     if |F_IN(t) ∪ F_IN(r)| ≤ k then
27       g ← r ∨ t
28     elseif k − |F_IN(r)| > 0 then
29         comment: Pick enough literals in t to fill r.
30         Let t = t1 ... tn and i = k − |F_IN(r)| > 0.
31         h ← r ∧ (t1 ... ti)
32         Let h be the literal representing its own output.
33         g ← h ∧ (ti+1 ... tn)
34     else
```

```
35              Let r be the literal representing its own output.
36              g ← r ∧ t
37          endif
38      else
39          g ← r
40      endif
41      Replace f with g.
42      comment: Decompose any new infeasible nodes.
43      foreach v ∈ Θ do
44          v ← DECOMPOSE-CUBE1(v, v, κ_α, κ_β)
45      endfor
46  end.
```

The example below illustrates how the decomposition is achieved.

EXAMPLE 4.5.1 *Let* $f = u\overline{w}x\overline{z} + uy\overline{z}$. *It can be rewritten as* $f = u\overline{z}(\overline{w}x + y)$. *Thus, the common cube divisor,* $q = u\overline{z}$ *and* $p = f/q = \overline{w}x + y$. *Now,* $|\mathcal{F}_{\text{IN}}(p)| = 3$, *so forming a node out of* p *will leave it under-utilised. Therefore, some literals from* q *can be padded on to* p *to give* $h = \overline{w}x\overline{z} + y\overline{z}$. *This* h *makes a new node and is substituted into* f. *Finally, the decomposed function is* $f = uh$.

4.6 Mixing Decomposition Methods

It is now accepted that no single decomposition algorithm can outperform all others for all circuits [59]. To improve the performance of the decomposition presented above, a pre-processing step is desirable. One of the most effective, technology-independent decomposition methods is *kernel or split decomposition* of MIS [4]. The algorithm extracts kernels from a node function and then re-expresses the original nodes as a function of the extracted kernels. Not all nodes will have good kernels that could be extracted, nor are the kernels necessarily feasible. Therefore, kernel decomposition does not always result in feasible DAGs. In combining kernel and bin-packing decompositions, better results are obtained than using either method on its own.

Below is a procedure that takes a DAG D and decomposes it using the two methods. A given node v in D is decomposed in two ways:

- using the kernel-extraction method followed by bin-packing; and
- using the bin-packing method alone.

Whichever method results in fewer decomposition nodes is accepted.

```
1  proc HETERO-DECOMP (D, κ_α, κ_β, ξ, τ)
2      comment: D = (V, E) is a Boolean DAG.
3      foreach infeasible internal node v ∈ V do
```

```
4       success ← false
5       Let G1 be the subgraph of v and its immediate fanins.
6       Decompose G1 using kernel-decomposition.
7       n1 ← number of internal nodes in G1
8       if n1 > 1 then
9           comment: The decomposition worked.
10          success ← true
11          Find the largest cuts less or equal to κα of all nodes in G1.
12          Decompose each infeasible node in G1 using DECOMPOSE-NODE.
13          Assign types to the nodes in G1.
14          Eliminate nodes G1 by partial collapse.
15          n1 ← number of internal nodes in G1
16      endif
17      comment: Decompose v normally.
18      DECOMPOSE-NODE(v, κα, κβ, ξ, τ)
19      Let n2 be the number of nodes introduced by the decomposition.
20      if (success = true) then
21          comment: Determine which method is better.
22          if n1 < n2 then
23              Replace v with G1.
24          endif
25      endif
26  endfor
27  Assign types to the nodes in D.
28 end.
```

Line 11 relies on algorithms for computing min-cuts in DAGs. Similarly, line 14 relies on algorithms for eliminating nodes in DAGs. These algorithms are discussed in Chapter 3 and Chapter 5, respectively.

4.7 Experiments

The following experiments were conducted on a set of circuits optimised using the standard SIS optimisation scripts. The principal objective is to determine the effect of various parameters and methods on the bin-packing procedure. These are listed below.

1. What effect does ξ have on the matching process of Section 4.5.1? The experiments need to determine an optimal value for ξ.

2. How does pre-processing the function by cube division affect the overall decomposition?

3. How does cascading nodes when building the tree affect the overall decomposition?

4. What effect does the attribute $Count(v)$, used during the decomposition of a cube, have on the overall decomposition?

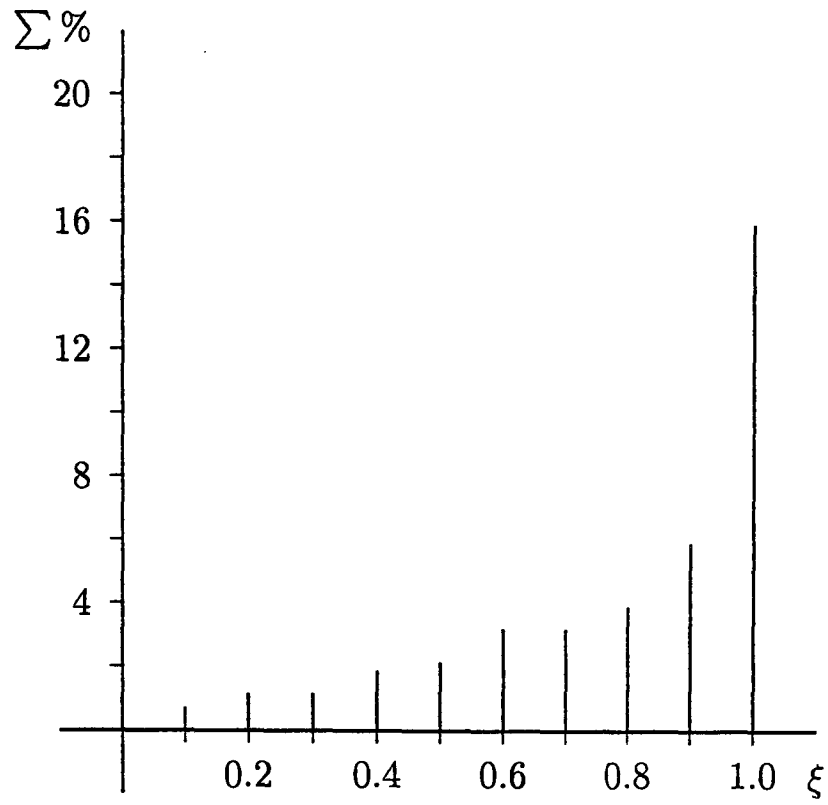


Figure 4.4: Effect of ξ on the number of nodes after cube-packing decomposition. The y-axis represents the total % increase in the number of nodes for $\xi > 0$ compared to $\xi = 0$.

5. What improvements arise from mixing decomposition levels?

6. What is the effect of varying τ when using the extended bin-packing algorithm?

In answering the questions above, the average results are considered. It is the trends that are examined since different benchmarks yield varying results.

4.7.1 Effect of ξ

In Section 4.5.1, a function was introduced for a parameter ξ that influences the matching of nodes (Equation 4.1). The most economical value for ξ is 0, though it leads to an increase in the circuit depth. Here, the effect of ξ is shown on the number of CLBs in the decomposition of several circuits.

In Figure 4.4 it is seen that, as expected, the number of nodes generated increases with ξ. The increase is especially noticeable for $\xi = 1$, where it rises to 16% from 151 circuits. However, most circuits display no changes with the variation in ξ.

The conclusion is that if the size of the decomposition tree is to be minimised, the matching must avoid using two α nodes and an under-utilised β node per CLB. This is what happens when $\xi > 0$. Once all the nodes in S_B have been matched, the sets S_A and S_B must be reconstituted

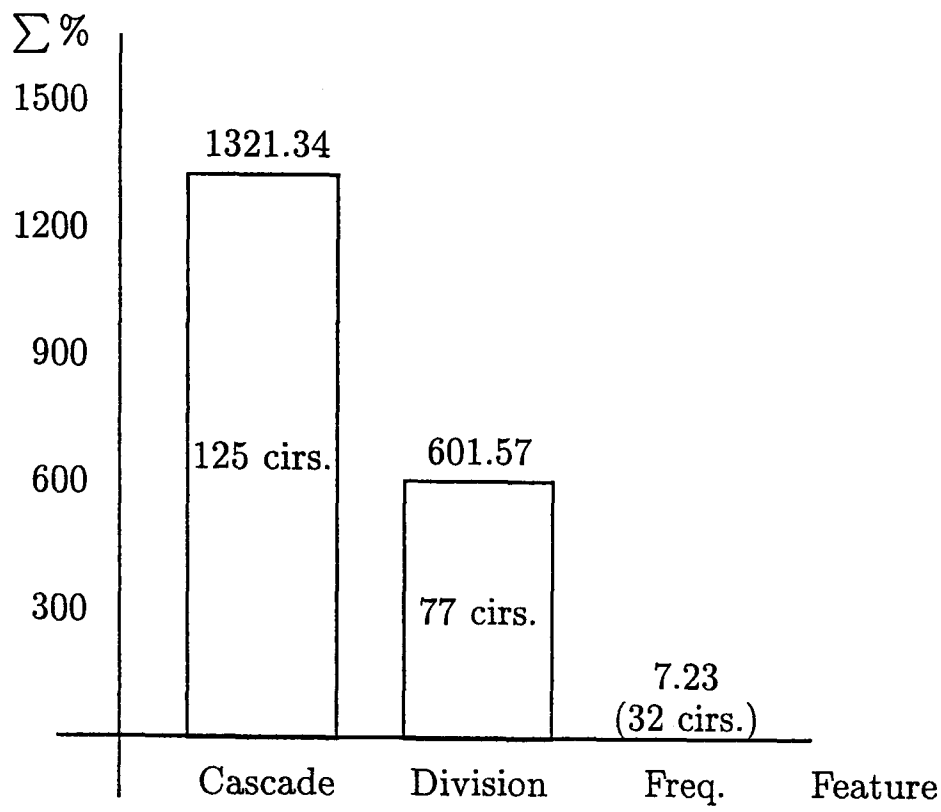


Figure 4.5: The sums of the increase in number of nodes after decomposition with various features disabled.

before the matching is repeated. A value of $\xi = 0$ is used by default.

4.7.2 Effect of Various Features.

The following features were investigated:

- cascading nodes,
- common cube division, and
- cube decomposition by counting the frequency of occurrences of the variables.

The experiments are divided into two sets for each feature: the control experiment has the feature enabled; the second experiment has the same feature disabled. The two sets of results are then compared. The chart in Figure 4.5 shows the *total* % increase in the number of nodes once a particular feature was disabled. Altogether, 151 circuits were tested.

Cascading has the greatest effect, affecting 125 circuits, with a total percentage increase of over 1321. Next, cube division affects 77 circuits with a total percentage increase of over 601. Finally, the use of the variable frequency only affects 32 circuits and the change in the results is marginal (the bar is not visible in the chart for this reason).

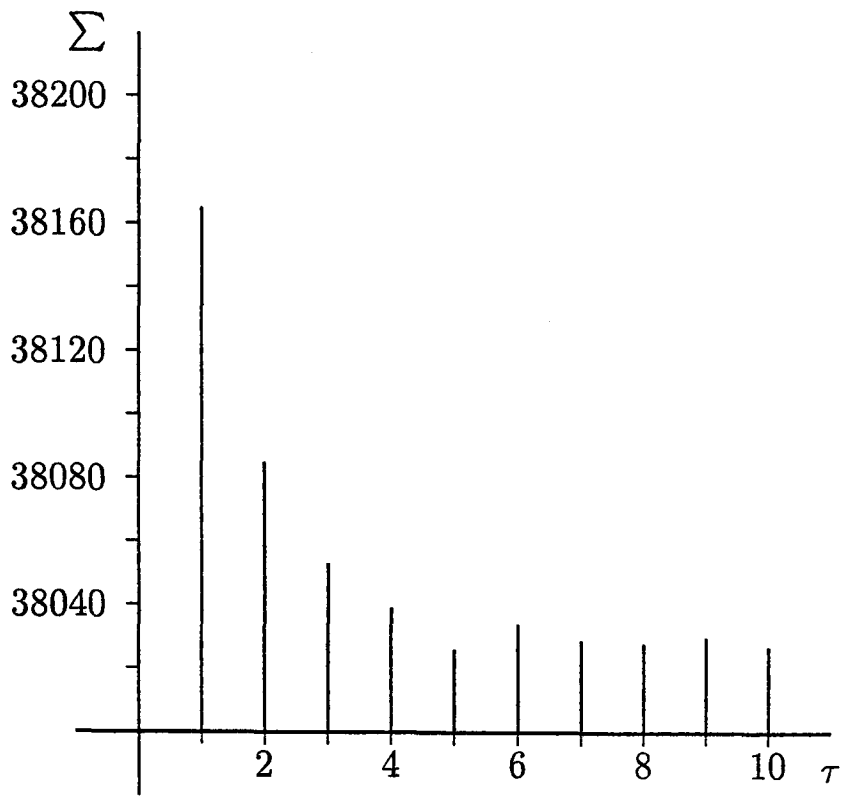


Figure 4.6: The number of new nodes needed by the bin-packing decomposition for various values of τ.

This shows that the greatest effect on the decomposition results is attributable to the construction of the decomposition tree, i.e., via cascading. Whereas pre-processing the nodes (by cube division) has some effect, the way in which individual cubes are decomposed seems to have little effect. Greater attention must therefore be paid to the construction of the decomposition tree.

4.7.3 Improved Bin-Packing

Here the benefits of the extension to bin-packing introduced in Section 4.3 is assessed. The effect of τ on the number of decomposition nodes is examined.

Figure 4.6 shows a general reduction in the number of nodes as τ increases. The number of new nodes is used rather than the percentage change because this shows the trends more clearly. A surprising result is that the minimum occurs at $\tau = 5$. This result is not entirely attributable to the the algorithm PACK-BINS and the effect of τ on it. It is coincidental that at $\tau = 5$ the bins packed in PACK-BINS give better results with the subsequent algorithms. A value of $\tau = 5$ is proposed as the default. In the experiments conducted, there is no noticeable difference in times required for various values of τ. This is due to the small sizes of the bins. For larger bins, the times can be expected to increase by a constant factor for larger τ.

4.7.4 Mixing Decomposition Methods

The last experiment compares the algorithms HETERO-DECOMP and DECOMPOSE-NODE. The same parameters were used for both algorithms. The tabulation of the results is given below.

Method	$\sum$ *nodes*
(1) Mixed	35171
(2) Bin-packing	38026

The table above shows that the results of the decomposition methods in terms of the nodes found for the feasible DAGs. Out of 151 benchmarks, 94 differed between the two methods and are always better with the mixed decomposition method. Some of the DAGs are already feasible so only the 94 that display differences are considered. The mixed method is 8.2% better than pure bin-packing on average. □

Chapter 5

Graph Restructuring

This chapter covers some aspects of restructuring Boolean DAGs. These are based on functional decomposition, partial collapsing, node elimination and graph simplification.

5.1 Introduction

The traditional approach to FPGA synthesis requires the optimisation of a circuit represented as a Boolean DAG. The optimisation techniques used are technology-independent. The measures of optimality usually include the literal count as the pre-eminent measure to be minimised. When dealing with LUT-based systems, these measures are not entirely relevant, though they may lead to simpler designs, which are then easier to map. This chapter covers the restructuring of optimised Boolean DAGs so that they correspond to the relevant optimisation goals.

Restructuring is useful because it helps the covering step to achieve good mappings. The restructuring operations presented here are divided into three parts.

One aspect of restructuring is referred to as re-synthesis and is presented in Section 5.2. This involves collapsing a subgraph into a single node and then decomposing it again. The aim is to produce a better subgraph than the original one. This new subgraph should be covered by fewer logic blocks. Partial collapsing is essentially grouping together nodes and converting them into new nodes. Finding subgraphs in a DAG is related to the work in Chapter 3. The decomposition in this chapter is largely based on functional decomposition, unlike the cube-packing of Chapter 4. Decomposition plays a very important rôle in restructuring DAGs. In order to achieve better results, the decomposition methods are combined.

The second aspect concerns the elimination nodes from the graphs and is presented in Section 5.3. This is related to partial collapsing. It involves collapsing a node into all its fanouts, which eliminates it from the DAG. From the viewpoint of the fanout nodes, the collapses usually increases their fanin support size and almost always increases their functional complexity. If this makes them infeasible, or increases their infeasibility, then they must be decomposed. The

objective would be to introduce fewer nodes in the decompositions than were eliminated or than might have been introduced by the decomposition schemes on the original nodes.

Finally, a simplification method is presented in Section 5.4. The basic objective is to find nodes in the DAG that share a common function. By re-expressing the functions of these nodes, it might be possible to eliminate a few nodes.

5.2 Re-synthesis

This section presents the concept of re-synthesising a DAG by partially collapsing it and then decomposing it. First, the decomposition method used is described. Then the collapsing techniques are presented. Finally, the overall approach is described.

5.2.1 Functional Decomposition

This section is concerned with decomposition methods. An approach to the functional decomposition of heterogeneous Boolean DAGs is presented. This decomposition method requires the variables in a function to be partitioned into two sets. The partitions have a considerable influence on the decomposition result. Issues on the partitioning of variables in a function and the factorisation of functions are addressed here. Much of the background on relevant material can be found in Section 2.3. However, the discussion here is on "heterogeneous" decomposition, where one seeks more than one size of node in the feasible decomposition graph.

Heterogeneous Graphs

Equation 2.1 in Section 2.3 is well-suited to homogeneous graphs. Finding the decomposition of f is easily accomplished by choosing a subset A whose size equals the limit of a feasible node's size and then recursively applying the decomposition on the composition functions. This is the method used by Murgai et al. [52].

Heterogeneous DAGs present a more difficult problem. These DAGs dictate a decomposition of the form

$$f(X) = f(A_1, A_2, B), \tag{5.1}$$

where $X = A_1 \cup A_2 \cup B$. For a disjunctive decomposition, all three subsets are mutually disjoint and mutually exhaustive subsets of X. Ideally, it should have $|A_1| = |A_2|$. Karp describes such a decomposition as a *multiple disjoint decomposition* but he considers up to m subsets of X of varying size [40]. One of Karp's theorem is re-stated below.

THEOREM 1 (KARP-1) *For a total function f, if $f(A, B, C) = F(\gamma(A), \omega(B), C)$, then there must be two functions $g(a, B, C)$ and $h(A, b, C)$ such that $f(A, B, C) = g(\gamma(A), B, C) = h(A, \omega(B), C)$.*

Karp's theorem shows that f can be decomposed with respect to either A or B independently and that a sequence of these decompositions achieves the desired multiple disjoint decomposition. A corollary of the theorem states that if $f(A,B,C) = g(\gamma(A),B,C)$, then the number of equivalence classes with respect to B in g is the same as that in f. This establishes the fact that a sequence of decompositions of f with respect to A and B results in the same number of decomposition functions irrespective of order.

Essentially, there are two ways to go about decomposing f with respect to the two types of nodes. Lai et al. suggested the following [44].

1. Decompose f with respect to A_1 and then A_2 so that $f_1 = f_1(\gamma_1(A_1), A_2, B)$ and $f_2 = f_2(\gamma_1(A_1), \gamma_2(A_2), B)$.

2. Now split B into two: $B = B_1 \cup B_2$. Then $f_3 = f_3(\omega(\gamma_1(A_1), \gamma_2(A_2), B_1), B_2)$.

This method relies on the existence of a simple disjunctive decomposition with respect to γ_1, γ_2 and ω. From experience, this is rarely the case. Hence, a different method is proposed.

A New Approach: Firstly, assume that $|X| = n \leq \kappa_{\text{LB}}$. An iterative application of Equation 2.1 with $|A| = \kappa_\alpha$ achieves an economical decomposition of f in terms of heterogeneous nodes. Karp's theorem is the basis of this claim. Moreover, if $f(X) = g(\gamma_1(A_1), \gamma_2(A_2), B)$, where A_1, A_2 and B are disjoint subsets of X and $|A_1| = |A_2| = \kappa_\alpha$, then g has at most κ_β variables. To see this, assume that g has m variables. Therefore,

$$\begin{aligned} m &= 2 + |B| \\ &= 2 + n - 2\kappa_\alpha \\ &\leq 2 + \kappa_{\text{LB}} - 2\kappa_\alpha \\ &\leq \kappa_\beta \end{aligned}$$

Now consider the case when n exceeds the input limit in a logic block, κ_{LB}. First, sub-functions of size κ_{LB} are sought. Therefore, Equation 2.1 is applied with $|A| = \kappa_{\text{LB}}$. Let

$$f(A,B) = g(\gamma_1(A), \ldots, \gamma_k(A), B).$$

Each γ_i is then decomposed by seeking economical decompositions limited to size κ_α as described above. The composition function g is then decomposed recursively. A second theorem by Karp [40] informs this approach.

THEOREM 2 (KARP-2) *Suppose a total function f has the following decompositions:*

1. $f(A,B,C) = g(\gamma(A),B,C)$, *and*

2. $f(A,B,C) = h(\omega(A,B),C)$.

If the second decomposition is strict, then there must be a function $\chi(\gamma(A), B) = \omega(A, B)$.

This theorem states that γ can either be obtained directly from f or ω found first and then γ obtained from it. The approach chosen here is equivalent to the latter.

The method is justified by the observation that often the difficult part is to find a decomposition that minimises the number of sub-functions for large functions. Therefore, it is more important to find economical sub-functions of size κ_{LB} in the first instance before going on to decompose them. Smaller functions are more easily and economically decomposed by other schemes. The selection has the added benefit in that it can reuse the algorithms used in the Roth-Karp decomposition and apply them directly.

Partitioning Variables

A very important step in functional decomposition is deciding on how to partition the variables of a function. Given a function of the form $f = f(X)$, where $X = \{x_1, \ldots, x_n\}$, a bi-partition of X into a *free set* B and *bound set* A is required (see Section 2.3). Hence, f can be decomposed into the functional form given by Equation 2.1. In general, only the size of A is specified. Since, $X = A \cup B$ and $A \cap B = \emptyset$, the size and composition of B follows immediately from the definition of A. Clearly, there are an exponential number of possible partitions of X.

The choice of the bound set has a significant effect on the cost of the decomposition. The optimal (minimal) decomposition can only be found by exhaustive trials of partitioning possibilities. The aim here is to show how the information from the function can be used to sieve out partitions that would obviously lead to uneconomical decompositions. A previous attempt on variable partitioning was made by Shen et al. [65]. The approach here is entirely different from theirs.

The cases of the function considered are in terms of its *sum-of-products* form as well as its factorized form. It is assumed that the function does not have redundant cubes and that it is not simply a single-cube function. The arguments are based on Karp's theorem's presented above. The cases considered are:

- the extraction of a common cube from the function;
- the separation of the cubes into disjoint sets; and
- the factorisation of the function.

Common Cubes: Without loss of generality, let

$$f(X) = (x_1 \ldots x_j) \cdot d(x_{j+1}, \ldots, x_n), \tag{5.2}$$

for $1 \leq j < n$. Therefore, $(x_1 \ldots x_j)$ is a common cube divisor (co-kernel) of f and d is its corresponding kernel. The following is conjectured.

CONJECTURE 1 *Let* $A_1 = \{x_1, \ldots, x_l\}$ $(1 \leq l \leq j)$ *and* $A_2 = \{x_{j+1}, \ldots, x_m\}$ $(j+1 \leq m \leq n)$ *so that* $A = A_1 \cup A_2$. *If*

$$f(X) = f(A, B) = g(\gamma_1(A), \ldots, \gamma_k(A), B),$$

then a similar decomposition can be achieved by a sequence

$$f(A, B) = g_1(\omega(A_1), A_2, B)$$

and

$$g_1(\omega(A_1), A_2, B) = g_2(\omega(A_1), \mu_1(A_2), \ldots, \mu_{k-1}(A_2), B).$$

In the second form, the product term of A_1 is given by ω, which divides f to give

$$g_1(\omega(A_1), A_2, B) = \omega[(x_{l+1} \ldots x_j) \cdot d(x_{j+1}, \ldots, x_n)].$$

All that is required for g to be similar to g_2 is for $\gamma_i = \omega$ for some $i \in \{1, \ldots, k\}$. Hence, g can be expressed as

$$g(\gamma_1(A), \ldots, \gamma_k(A), B) = \gamma_1(A_1) \cdot g^*(\gamma_2(A_2), \ldots, \gamma_{k-1}(A_2), B),$$

which is similar to g_2, where 1 is assigned to i (the γ sub-functions are not ordered).

It is concluded from the above that the decomposition of f can be accomplished by considering the co-kernel and kernel separately. This gives the first partitioning criterion.

Disjoint Cubes: The second case involves a function of the form

$$f(X) = f_1(X_1) + f_2(X_2), \tag{5.3}$$

where $X = X_1 \cup X_2$ and $X_1 \cap X_2 = \emptyset$.

CONJECTURE 2 *Let* $A_1 \subseteq X_1$, $A_2 \subseteq X_2$ *and* $A = A_1 \cup A_2$. *If*

$$f(X) = f(A, B) = g(\gamma_1(A), \ldots, \gamma_k(A), B),$$

then a similar decomposition can be achieved by a sequence

$$f(A, B) = g_1(\omega_1(A_1), \ldots, \omega_p(A_1), A_2, B)$$

and

$$g_1(\omega_1(A_1), \ldots, \omega_p(A_1), A_2, B) = g_2(\omega_1(A_1), \ldots, \omega_p(A_1), \mu_1(A_2), \ldots, \mu_q(A_2), B),$$

where $p + q \leq k$.

The intuition leading to this conjecture arises from the fact that all variables in A_1 are contained in f_1 and those in A_2 are contained in f_2. So, in fact, γ is an interleaving of ω and μ. This means that the decomposition of f_1 and f_2 can be accomplished separately; it provides the second partitioning criterion.

Factorized Form: A factorized form of a function based on common and disjoint cubes is now considered. For instance, the function might be

$$f(X) = f_1(X_1)[f_2(X_2) + f_3(X_3)], \tag{5.4}$$

where $X_1, \ldots, X_3$ are mutually exhaustive and disjoint subsets of X. The decomposition of f is based on the separate decompositions of $f_1, \ldots, f_3$ as shown in the previous sections. These sub-functions are decomposed in the same recursive fashion unless they cannot be factorized further. Note that f_1 is the largest cube divisor of f. The following example illustrates this point.

EXAMPLE 5.2.1 *Let*

$$f = v(x_1x_2x_3x_4 + y + z) + u.$$

It can broken down into two: $f_1 = v(x_1x_2x_3x_4 + y + z)$ *and* $f_2 = u$. *Similarly, the same can be done to* f_2 *to give* $g_1 = v$, $g_2 = x_1x_2x_3x_4$, $g_3 = y$ *and* $g_4 = z$. *This then gives*

$$f = g_1(g_2 + g_3 + g_4) + f_2.$$

If the requirement is for decomposition functions of four variables, then it will be seen readily that one such function is $\gamma_1 = g_2 = x_1x_2x_3x_4$; *this is the only non-trivial function in the current form. Thus,*

$$f^* = v(\gamma_1 + y + z) + u.$$

Where the sub-functions have fewer than the required number of variables, they may be combined to form a sufficiently large function. In the new function f^*, $\gamma_2 = v(\gamma_1 + y + z)$ *can be formed. Hence,*

$$f^{**} = \gamma_2 + u.$$

The concept of a *factorized tree* T of a Boolean function f is now introduced. A factorized tree is defined recursively as follows. Each node is associated with a set Δ, possibly empty, which contains the disjoint components of the largest kernel of f expressed in factorized tree form. The tree has three types of nodes.

1. If f is not a kernel itself but has a kernel k and co-kernel c such that $f = c \cdot k$, then $T(f)$ is a non-terminal node with two children $T_c(f) = c$ and $T_k(f) = k$, which are factorized subtrees of the co-kernel and kernel respectively. The disjoint component set Δ is empty.

2. If f is a kernel and $\Delta \neq \emptyset$, then $T(f)$ is a non-terminal node representing the function f without the children $T_c(f)$ and $T_k(f)$. Instead, the node has branches that leads directly to the elements in Δ.

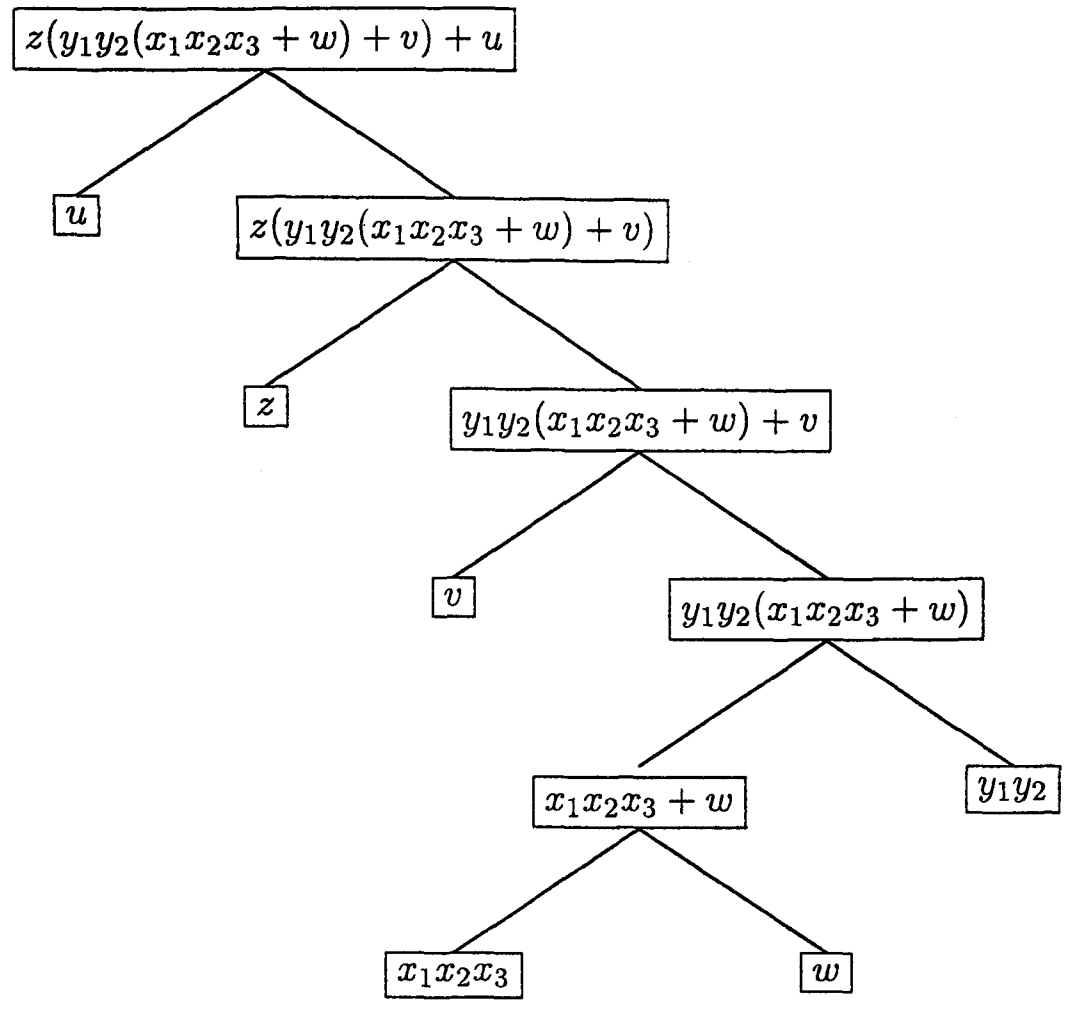


Figure 5.1: A factorized tree for $f = z(y_1y_2(x_1x_2x_3 + w) + v) + u$

3. If f is a kernel and $\Delta = \emptyset$ or f is a single-cube function, then $T(f)$ is a terminal node representing the function f.

EXAMPLE 5.2.2 *As an illustration, consider the function*

$$f = z(y_1y_2(x_1x_2x_3 + w) + v) + u.$$

Now f is itself a kernel, but it has two disjoint components:

$$f_1 = x(y_1y_2(x_1x_2x_3 + w) + v) \text{ and } f_2 = u.$$

Thus, $T(f_2)$ is a terminal node. However, f_1 is not a kernel; it can be re-written as $f_2 = c_2 \cdot k_2$, where $k_2 = y_1y_2(x_1x_2x_3 + w) + v$. Hence, the subtree of $T(k_2)$ is computed and so forth to give the structure in Figure 5.1.

In the variable partitioning scheme for a given function f, the factorized tree T is first constructed. Then T is traversed until a terminal node, g is found. If g has more than a given number of variables, then all its combinations of variables are included in the set of possible bound sets for the decomposition of f.

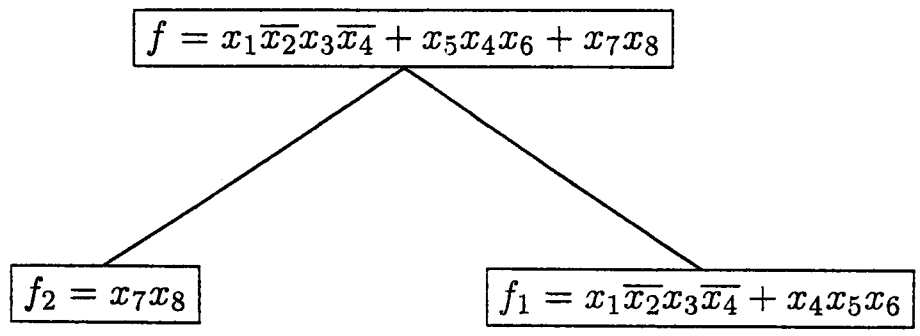


Figure 5.2: The factorized tree for $f_1 = x_1\overline{x_2}x_3\overline{x_4} + x_5x_4x_6 + x_7x_8$

EXAMPLE 5.2.3 *Consider the function*

$$f = x_1\overline{x_2}x_3\overline{x_4} + x_5x_4x_6 + x_7x_8.$$

Its factorized tree is shown in Figure 5.2. There are two terminal nodes, f_1 and f_2. Let the limit of the size of the bound set be 4, then the variables in f_2 need not be considered, since $|\mathcal{F}_{\text{IN}}(f_2)| = 2$. So the bound sets are formed from combinations of size four from $\mathcal{F}_{\text{IN}}(f_1) = \{x_1, \ldots, x_6\}$.

Decomposition with Factorized Trees

This section describes the application of factorized trees to the decomposition of functions. A factorized tree T is considered and it is assumed that the desired size of nodes is s variables. The sub-functions represented by terminal nodes in T may have varying numbers of variables. Where the number of variables exceeds the capacity s, then the node must be decomposed. If the sub-function is a single-cube function, its decomposition is easily accomplished by the cube-packing method of Chapter 4. On the other hand, kernels are decomposed by a method that is described below.

Let the root function be f. Since f is infeasible, $|\mathcal{F}_{\text{IN}}(f)| > s$. However, for the sub-functions of f, there are two possibilities. One is that they have less than s variables, and the other is that they are themselves s-infeasible. The former case is considered first.

Feasible Sub-functions: For any sub-function g at a terminal node, $\mathcal{F}_{\text{IN}}(g) \subset \mathcal{F}_{\text{IN}}(f)$. Moreover, each g has a unique subset of the variables of f. Therefore, providing that $|\mathcal{F}_{\text{IN}}(g)| > 1$, substituting for g with a literal representing it would decompose f. If $|\mathcal{F}_{\text{IN}}(g)| < s$, it would not be economical to introduce a node g into the DAG. Instead, it is better to combine g with some other terminal nodes so that the total input size is s. An algorithm similar to cube-packing achieves this end. All that is required is to define such sub-functions as the items and set the bin capacity to s.

Infeasible Sub-functions: Any infeasible sub-function needs to be decomposed. A single-cube sub-function is easily decomposed by extracting sub-cubes of size s from it. For a more general sub-function g, a functional decomposition is required. Since g is a terminal node in T, its variables are used to define a subset of the set of bound sets to be tried in the decomposition. If $A \subset \mathcal{F}_{\text{IN}}(g)$ such that $|A| = s$, then A will be used in a decomposition of the form

$$f(X) = h(\gamma_1(A), \ldots, \gamma_t(A), X - A).$$

For the moment, all infeasible sub-functions g as defined above are recorded in a set, say Ψ.

The recursive algorithm, called FIND-GROUPS, finds the sub-functions of size s in the tree and stores them in a set Φ.

FIND-GROUPS(T, s, Ψ)

1. Let f be the function of the root of T.

2. If $|\mathcal{F}_{\text{IN}}(f)| < s$, then the algorithm terminates with $\Phi = \emptyset$. Otherwise, if $|\mathcal{F}_{\text{IN}}(f)| = s$, then it terminates with $\Phi = \{f\}$. These two conditions cause the termination of the recursion – recall that only sub-functions of s variables or more are being sought.

3. Let $\Phi = \emptyset$. The co-kernel of f is represented by the sub-tree T_c. If the function of the root of T_c has more than s variables, then it is decomposed into s-sized nodes which are added to Φ.

4. The kernel of f is represented by T_k. For the reasons outlined above, if the kernel has s variables, it is inserted into Φ and the algorithm terminates.

5. At this stage, the algorithm turns to working with Δ. Recall that the disjoint components of f are given by the set Δ. If $\Delta = \emptyset$, then T is a leaf node. Therefore, the bound set will be constructed from the variables in f. Hence, f is inserted into Ψ and the algorithm terminates.

6. Where there are elements in Δ, they are processed separately. So for any subtree $T_i \in \Delta$, representing a root function g_i, the following applies.

   - $|\mathcal{F}_{\text{IN}}(g_i)| > s$: Let $\Theta \leftarrow$ FIND-GROUPS(T_i, s, Ψ). Then, $\Phi \leftarrow \Phi \cup \Theta$.
   - $|\mathcal{F}_{\text{IN}}(g_i)| = s$: No further decomposition is necessary, so $\Phi \leftarrow \Phi \cup \{g_i\}$.
   - $|\mathcal{F}_{\text{IN}}(g_i)| < s$: All such nodes are packed into s-sized nodes using a bin-packing strategy. The results are appended to Φ.

7. Φ now contains the decomposed sub-functions, and this is the set that is returned. The non-decomposed sub-functions are in Ψ.

The Decomposition Subgraph: Referring to Section 2.3, let a procedure ROTH-KARP$(f(X), \lambda, r)$ find the functional decomposition of f,

$$f(X) = f^*(g_1(\lambda), \ldots, g_t(\lambda), X - \lambda),$$

where $\lambda \subset X$. It then returns the set $\{f^*, g_1, \ldots, g_t\}$ only if $t \leq r$ and $\emptyset$ otherwise.

Given various bound sets λ, the procedure ROTH-KARP can be used to compute the decompositions. Only one such decomposition can be implemented, so that a way of selecting one from the various possibilities is required. One way is to select the decomposition that is estimated to yield the least number of decomposition nodes of size s. This is computed as follows. Given a function g, then the lower bound of sub-functions of size s that can be decomposed from the m variables of g is

$$N(m, s) = \begin{cases} 0 & \text{if } m \leq s, \\ d + N(d + r, s) & \text{otherwise,} \end{cases} \tag{5.5}$$

where $d = m/s$ and $r = (m \bmod s)$. Therefore, g needs at least $N(m, s)$ decomposition functions to make it feasible.

For a decomposition $\{f^*, g_1, \ldots, g_t\}$, the number of extra decomposition nodes required is given by

$$E(\{f^*, g_1, \ldots, g_t\}, s) = N(|\mathcal{F}_{\text{IN}}(f^*)|, s) + \sum_{i=1}^{t} N(|\mathcal{F}_{\text{IN}}(g_i)|, s). \tag{5.6}$$

Therefore, the best decomposition $\phi^* \in \Psi$ is one that minimises $E(\phi^*, s)$ as computed from Equation 5.6. If the decompositions ϕ_i and ϕ_j tie with this criterion, the tie is broken on the basis of the following criteria listed in order of decreasing strength.

1. Minimise the number of decomposition functions t;
2. minimise the number of cubes in the composition function; and
3. minimise the number of literals in the composition function.

The last two criteria help to select simpler composition functions which are often easier to decompose further if required.

An algorithm TRY-LAMBDAS that finds a decomposition from a set of bound set variables Λ is defined. It returns a vector ψ^* of the best decomposition found by trying each $\lambda \in \Lambda$. In practice, the size of Λ is limited in order to save computational effort.

Combined Decomposition Methods: To improve the decomposition results, it is necessary to try different methods. Hence, functional decomposition is combined with the cube-packing/kernel-extraction decomposition method introduced in Chapter 4. For the purposes of this decomposition step, a single sub-function size $s \geq \kappa_\alpha$ is sought. An iterative approach is used to continually decompose a node v and its decomposition tree until it is feasible.

A procedure MIXED-DECOMP first determines the possible bound sets and simple decomposition (using FIND-GROUPS). It then applies both the cube-packing/kernel-extraction decomposition and functional decomposition. It selects the better of the resulting decomposition subgraph. The growth in the decomposition of v and its derivatives is also measured by counting the number, m, of new nodes introduced into the DAG.

For practical reasons, the decomposition of large nodes is omitted, i.e., those with more than ρ_{IN} inputs. A value of $\rho_{\text{IN}} = 10$ is the general practical limit. It is assumed that the number of inputs to a logic block is $s \leq \kappa_{\text{LB}}$ and that the maximum number of new nodes permitted is given by *limit*. First, set a counter n to *limit* and count downwards as new nodes are created by the decomposition. So when n gets to zero, no further nodes will be permitted. This aspect is important in that it enables the whole process to be aborted once it is known that the decomposition that will be found will certainly be inferior to what is anticipated. Therefore, *limit* corresponds to upper bound of the cost of the decomposition of v.

A procedure ITER-MIXED-DECOMP (see Section A.3 in the appendix) is defined to perform the following on an infeasible node v.

1. Let H be a subgraph initially consisting of v only.
2. Use MIXED-DECOMP to decompose H until it is feasible or the decomposition fails.
3. Define the *cumulative degree of infeasibility* as $\sum \frac{|\mathcal{F}_{\text{IN}}(u)|}{\kappa_\alpha}$ and $|\mathcal{F}_{\text{IN}}(u)| > \kappa_\alpha$, where u is a node in H. Let the procedure return the cumulative degree of infeasibility assumed by H.

A further refinement would be to decompose a node using two alternative methods. One uses the cube-packing/kernel-extraction method while the other uses ITER-MIXED-DECOMP. Whichever produces the most economical decomposition is accepted.

ALT-DECOMPS$(v, \kappa_\alpha, \rho_{\text{IN}}, s)$

1. Let G_1 and G_2 be two copies of a subgraph comprising of v as the primary output and all its immediate fanin nodes as primary inputs.
2. Decompose G_1 using the cube-packing/kernel-extraction method. Let n_1 be the number of internal nodes in G_1. If the result is optimal, then the decomposed subgraph is returned.
3. Otherwise, decompose G_2 using ITER-MIXED-DECOMP. No more than n_1 nodes must be used. Let n_2 be the number of internal nodes in G_2 and the value returned by ITER-MIXED-

DEC be r. If r is negative or $(r + n_2) \geq n_1$, then return G_1 (recall that r is the cumulative degree of infeasibility of G_2).

4. Decompose G_2 using the cube-packing/kernel-extraction method if it is still infeasible. Recompute n_2.

5. Return the smaller of the two subgraphs.

In summary, the decomposition is carried out in two levels. Firstly, during the construction of the factorized trees, and secondly, after a suitable subset of the fanins have been identified as candidates for the bound set. In the latter stage, functional decomposition is combined with other methods either by interleaving them or by invoking them separately.

5.2.2 Partial Collapsing

This section addresses the issue of collapsing part of the DAG so that it can be re-structured, by decomposition and covering, in a way more suited to LUT-based FPGAs.

The Application of Min-Cut Algorithms

Chapter 3 showed how min-cut algorithms can be used to compute cuts within a DAG. For a cut $(X, \overline{X})$, any node contained within the partition $\overline{X}$ will be collapsed into the root node v. Hence, an application of a min-cut algorithm to all the internal nodes of a DAG effects a partial collapse of the DAG. A node that results from a partial collapse is referred to as a *reduced* node. This process has the benefit of decreasing the infeasibility of the nodes. However, the complexity of the functions implemented by each node often increases, since some of the optimisations are undone. It is observed that where nodes remain infeasible after the application of the min-cut operation, they are often even more difficult to decompose economically.

EXAMPLE 5.2.4 *Figure 5.3 shows an example of the partial collapse of a node v by computing its min-cut. Initially, v has four inputs: $\mathcal{F}_{\text{IN}}(v) = \{a, b, e, f\}$. Afterwards, its min-cut has the inputs $\mathcal{F}_{\text{IN}}(v) = \{a, b, c, d\}$. Therefore, if e and f did not fanout outside the fanin cone of v, then they would be eliminated from the DAG.*

Selecting Nodes for Partial Collapsing

Computing the min-cut of a node v within a DAG sometimes does not involve the collapse of any node into v. In other words, the min-cut is right across the immediate fanins of v. Yet, it is possible that by collapsing a subgraph of the fanin cone of v, some of its nodes could be eliminated. The collapsed form of v should introduce fewer nodes into the DAG when subsequently decomposed

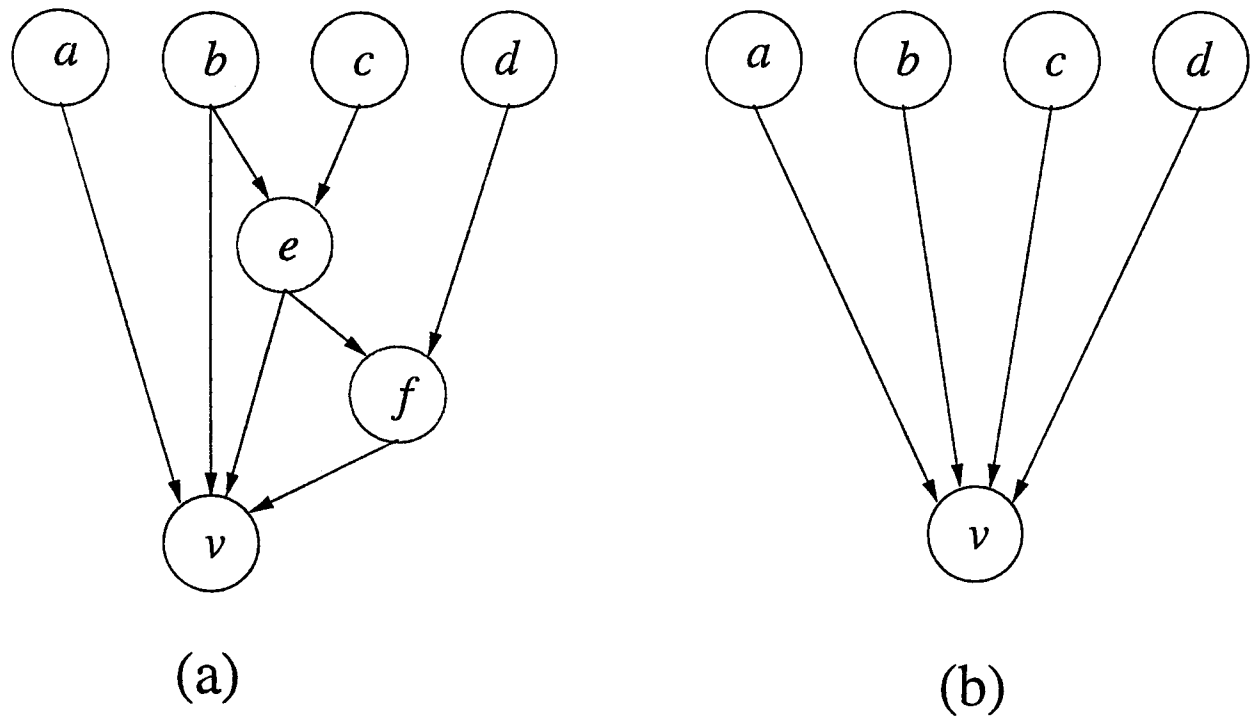


Figure 5.3: An example of partial collapse using min-cuts: (a) before and (b) after. In (b), the node function for v no longer depends on e and f.

than would have been the case with the original v. Therefore, a method is required for determining which nodes in the fanin cone of v should be collapsed into it.

The approach adopted here is to consider the immediate fanins of v. In order that a fanin u be collapsible into its fanout v, the following conditions must be satisfied.

1. Let $\theta = \mathcal{F}_{\text{IN}}(u) - \mathcal{F}_{\text{IN}}(v)$, i.e., θ is the set of fanins of u that are distinct from those of v. If u is collapsed into v, then the new fanin set of v would include θ. Practical considerations impose a limit to the fanin size of the collapsed v. Say this limit is ρ_{IN}, then the condition is $|\mathcal{F}_{\text{IN}}(v)| + |\theta| - 1 \leq \rho_{\text{IN}}$.

2. If $\mathcal{F}_{\text{IN}}(u) \cap \mathcal{F}_{\text{IN}}(v) = \emptyset$ (u and v have no fanins in common), two further conditions are imposed.

   (a) The fanin size of u must not exceed κ_α. There would be of no benefit in collapsing u into v only for it to be extracted latter by a decomposition process. Moreover, u has no unused input slots; there would be no chance of moving another fanin of v to the fanin set of u.

   (b) The number of fanouts of u must not exceed ρ_{OUT} fanouts (v is one of them). It is projected that the collapse of u into v would be unlikely to eliminate u from the DAG. Hence, there would be no advantage in doing so.

A maximal number of nodes in $\mathcal{F}_{\text{IN}}(v)$ that can be collapsed collectively into v without violating the conditions set out above is sought. To compute the sets of collections of nodes, a tuple $\psi = (A, \theta)$ is introduced, where A is a subset of $\mathcal{F}_{\text{IN}}(v)$ and θ is the union of the sets of distinct

fanins for all elements of A. Each such tuple gives a collapsing option, which needs to be maximised with respect to A. An iterative algorithm, FORM-OPTIONS, can be used to build the set of tuples Ψ, where each $\psi \in \Psi$ is maximal.

EXAMPLE 5.2.5 *Suppose that a node v and its fanins are as shown in the table below.*

Node	v	u_1	u_2	u_3
Fanins	$\{u_1, u_2, u_3\}$	$\{x_1, x_2\}$	$\{y_1, y_2\}$	$\{z_1, z_2\}$

Let $\rho_{\text{IN}} = 5$ and $|\mathcal{F}_{\text{OUT}}(u_i)| \leq \rho_{\text{OUT}}$ for $i = 1, \ldots, 3$, then FORM-OPTIONS *initialised Ψ to*

$$\Psi = \{(\{u_1\}, \{x_1, x_2\}), (\{u_2\}, \{y_1, y_2\}), (\{u_3\}, \{z_1, z_2\})\}.$$

In the first iteration, the pairwise matches found (in terms of the indices of Ψ) are: (1,2), (1,3), and (2,3). Therefore, Ψ becomes

$$\Psi = \{(\{u_1, u_2\}, \{x_1, x_2, y_1, y_2\}), (\{u_1, u_3\}, \{x_1, x_2, z_1, z_2\}), (\{u_2, u_3\}, \{y_1, y_2, z_1, z_2\})\}$$

In the next iteration, no matches are found, so that $\Phi = \Psi$. This means that, for instance, both u_1 and u_2 can be collapsed into v simultaneously.

The following discussion concerns the nodes created through various partial collapse options of a given node v. In order to achieve better results, the min-cut algorithm is invoked on the partially collapsed nodes so that the new nodes have as few inputs as possible and as many nodes as possible are eliminated. The number of nodes eliminated by each collapse is evaluated. In this way, the relative cost of the collapse and decomposition can be gauged. The following steps implement this scheme.

FIND-OPTIONS$(v, \kappa_\alpha, \rho_{\text{IN}}, \rho_{\text{OUT}}, R)$

1. Find the collapsing options of v. Say, FORM-OPTIONS yields the set Ψ.
2. For each $\psi_j = (U, \theta) \in \Psi$, where $1 \leq j \leq |\Psi|$:
   (a) Copy v to v_j.
   (b) Collapse all nodes $u \in U$ into v_j.
   (c) Find the min-cut of v_j.
   (d) Append v_j to a set Φ.
   (e) Let $R[j]$ be the number of nodes eliminated by the collapse.

Therefore, Φ contains partially collapsed copies of v. These are the nodes that will be used in the re-synthesis of v.

5.2.3 A Re-synthesis Algorithm

This section puts together the concepts introduced above. It shows how a DAG can be restructured by first collapsing some subgraphs and then decomposing them. Partial collapsing is performed using FIND-OPTIONS (see Section 5.2.2) and decomposition by ALT-DECOMPS (see Section 5.2.1).

The approach involves comparing the decompositions of various versions of a given node. These are:

- the original node v_0,
- the reduced node v (see Section 5.2.2), and
- various nodes v_i ($0 < i \leq m$ and $m \geq 0$) derived from the partial collapsing of v.

The decomposition of each of these nodes yields a subgraph. The third set of nodes is found using FIND-OPTIONS — call it A. Since each $v_i \in A$ is derived from v, a comparison between its decomposition subgraph G_i and that of v, G, can be made. This is achieved by noting how many node r_i are eliminated in forming v_i. Thus, if n and n_i are the numbers of internal nodes in G and G_i, respectively, then the cost of decomposing v_i is given by $(n_i - r_i)$. Hence, for v_i to be preferable to v, the requirement is that

$$(n_i - r_i) \leq n.$$

Using the above equation, the least costly node from the set A and v is selected. The best decomposition subgraph obtained from these nodes is denoted by $G^\star$. A similar comparison is made between v and v_0. Let G_0 be the subgraph representing the decomposition of v_0 with n_0 internal nodes. If $n_0 \geq n$, then G_0 is inferior in quality to G since v is the reduced form of v_0. Hence, G_0 can can be discarded safely. The opposite is not true, however. The global rather than local gains in partially collapsing a node are being considered here. Therefore, it is not possible to assert that G_0 is better than G even if $n_0 < n$. The number of nodes eliminated in forming v need to be taken into account.

Let the procedure COMPUTE-SUBGRAPHS compute the tuple $(G_0, G^\star)$, where G_0 may be NULL. For practical reasons, very complex nodes with too many cubes are not processed. The restrictions are: $|\mathcal{F}_{\text{OUT}}(v)| \leq \rho_{\text{OUT}}$ and $|\mathcal{F}_{\text{IN}}(v)| \leq \rho_{\text{IN}}$.

The Overall Approach

The re-synthesis algorithm can be summarised by these three basic steps:

1. find the min-cuts of all nodes in the DAG;
2. compute the decomposition subgraphs of the original nodes and the reduced nodes; and

3. select the best decomposition subgraphs.

The selection step arises when COMPUTE-SUBGRAPHS finds two subgraphs G_0 and G_1 for a node in its original form, u, and reduced form, v. In fact, COMPUTE-SUBGRAPHS returns a NULL G_0 if its size is not smaller than G_1. However, the fact that G_0 is smaller than G_1 does not make it automatically better. Any nodes eliminated when u is transformed into v must be taken into account. A table T that contains (internal) nodes in the original DAG that are eliminated due to the partial collapses is used for this purpose. Therefore, the extra cost of implementing G_0 is defined as the number of nodes in G_0 that are in T. These nodes would have to be included in the DAG again if G_0 is implemented. Whereas this measure gives a good approximation of the cost of implementing G_0, it would be more accurate to consider the entire transitive fanin of u. Due to the computational expense, only the fanins of u are examined.

The table T is filled by examining the fanouts of a given node. A node u is inserted into T if one of two conditions is satisfied:

1. u has no fanout nodes, i.e., $\mathcal{F}_{\text{OUT}}(u) = \emptyset$; or

2. all the fanouts of u are in T, i.e., $\forall w \in \mathcal{F}_{\text{OUT}}(u) \bullet w \in T$.

Because of the above definition, it is important to traverse the DAG in post-order when filling T. This ensures that when any node u is examined, all its fanouts will have been examined already. The selection process is accomplished by SELECT-SUBGRAPHS.

Given a DAG, $D = (V, E)$, the internal nodes are processed in a depth-first order, after finding their min-cuts. Using COMPUTE-SUBGRAPHS, the subgraphs for each node are computed as explained previously. If the selection between the subgraphs cannot be resolved immediately, they are inserted into a set Θ. Then the set $B \subseteq V$ is processed in reverse order so that T can be filled correctly. Hence, SELECT-SUBGRAPHS can be invoked with the table T to resolve the selection of the subgraphs in Θ. The procedure RESYNTHESISE below completes the re-synthesis algorithms.

```
1  proc RESYNTHESISE (D, κ_α, ρ_IN, ρ_OUT, s)
2    comment: The DAG D = (V, E)
3    Let D_0 = (V_0, E_0) be a copy of D.
4    Find the min-cut of D.
5    Let B be the set of internal nodes in V sorted in post-order.
6    m ← |B|
7    Θ ← ∅
8    for i = 1 to m do
9        v_i ← B[i]
10       Let u be the equivalent of v_i in V_0.
11       (G_0, G_1) ← COMPUTE-SUBGRAPHS(v_i, u, κ_α, ρ_IN, ρ_OUT, s)
```

```
12        if (G_0, G_1 ≠ NULL) then
13            Let n_0 and n_1 be the number of internal nodes in G_0 and G_1, respectively.
14            if G_1 = NULL
15                Replace v_i with G_0.
16            elseif G_0 = NULL then
17                    Replace v_i with G_1.
18            elseif n_1 > (n_0 + |F_IN(u)|) then
19                    Replace v_i with G_0.
20            else
21                Θ ← Θ ∪ {(G_0, G_1)}
22            endif
23        endif
24    endfor
25    for i ← 0 to m − 1 do
26        comment: Order of traversal is important - start with last first.
27        v ← B[m − i]
28        Update table T by checking v.
29    endfor
30    if Θ ≠ ∅ then
31        SELECT-SUBGRAPHS(Θ, T)
32    endif
33 end.
```

EXAMPLE 5.2.6 *Consider a DAG with the following functions.*

$$K = \overline{h} + \overline{i} + j$$

$$L = h\overline{j} + \overline{h}j + i$$

$$u_0 = a + b$$

$$u_1 = f\overline{h}ij + \overline{g}$$

$$P = KL\overline{u_0}ch + \overline{K}d\overline{e} + L\overline{u_0}c\overline{i} + u_1 + \overline{i}\,\overline{j}$$

The primary inputs are $\{a, b, c, d, e, f, g, h, i, j\}$ *and the primary outputs are* $\{K, L, P\}$. *There are two internal nodes:* $\{u_0, u_1\}$. *This example is concerned with* P. *Now,* $\mathcal{F}_{\text{IN}}(P)$ *has 10 nodes. However, its min-cut is*

$$P^* = \overline{u_0}chij + \overline{u_0}c\overline{h}\,\overline{i} + u_1 + d\overline{e}h\overline{j} + \overline{i}\,\overline{j}$$

which has only 8 nodes in its fanin set. Both the cube-packing and functional decompositions give 3 decomposition nodes. The former has the following subgraph G_1.

$$x_0 = d\overline{e}h + \overline{i}$$

$$x_1 = hij + \overline{h}\,\overline{i}$$

$$x_2 = x_1\overline{u_0}c + u_1$$

$$P^* = x_0\overline{j} + x_2$$

Next, consider the decomposition of the original node P. In this case, cube-packing gives an optimal decomposition with 3 decomposition nodes. It has the following subgraph G_0.

$$y_0 = L\overline{u_0}c$$
$$y_1 = \overline{K}d\overline{e} + u_1$$
$$y_2 = Ky_0h + y_0\overline{i}$$
$$P = y_1 + y_2 + \overline{i}\,\overline{j}$$

Since G_0 ties with G_1, G_0 can be discarded without loss of optimality.

Finally, P^* can be collapsed further to give P^{**}. First, collapse u_1 into P^* and then compute its min-cut. The result is

$$P^{**} = \overline{a}\,\overline{b}chij + \overline{a}\,\overline{b}c\overline{h}\,\overline{i} + d\overline{e}h\overline{j} + f\overline{h}ij + \overline{g} + \overline{i}\,\overline{j}$$

The decomposition subgraph for P^{**}, G_2 is:

$$z_0 = \overline{a}\,\overline{b}c$$
$$z_1 = f\overline{h}ij + \overline{i}\,\overline{j}$$
$$z_2 = d\overline{e}h\overline{j}$$
$$z_3 = z_1 + z_2 + \overline{g}$$
$$z_4 = z_0hij + z_0\overline{h}\,\overline{i}$$
$$P = z_3 + z_4$$

G_2 has 5 new nodes, but it comes with the potential elimination of two nodes, u_0 and u_1. If both nodes had no other fanouts, then they would be eliminated and G_2 would then selected. Otherwise, G_1 is the better subgraph.

Experiments on Re-synthesis

There are three parameters that influence the performance of the re-synthesis. The first is a fanin limit ρ_{IN}. Any node with more than this number of nodes is not processed. Also, ρ_{IN} is the fanin limit for the partial collapse operation. This value is generally limited by practical considerations to around 10.

A second parameter, ρ_{OUT}, ensures that nodes with more than this number of fanouts are not collapsed into any of their fanouts. The more fanouts a node has, the less likely it is for that node to be eliminated from the DAG, since it has to be collapsed into all its fanouts in the re-synthesis. Using a value of $\rho_{\text{OUT}} = 1$ is slightly restrictive, so other values are examined.

Thirdly, consider the input slot utilisation of a logic block. Ideally, each logic block should have all input slots utilised. However, if this requirement is relaxed, especially for the β-LUT,

a better packing of the logic blocks might be found after re-synthesis. Hence, a parameter s is introduced which is the number of inputs to a logic block, where $2\kappa_\alpha \leq s \leq \kappa_{\text{LB}}$. In other words, $s = 2\kappa_\alpha$ means that only the two α-LUTs have external inputs. On the other hand, $s = \kappa_{\text{LB}}$ implies that the β-LUT has $(\kappa_{\text{LB}} - 2\kappa_\alpha) = (\kappa_\beta - 2)$ external inputs. For $\kappa_\alpha = 4$ and $\kappa_\beta = 3$, $8 \leq s \leq 9$.

The experiments that follow determine optimal values for ρ_{IN}, ρ_{OUT} and s. The experiments assess the effect of running RESYNTHESISE with different values of the parameters. The DAGs are further decomposed using the mixed decomposition method presented in Chapter 4. This permits the LUT and CLB utilisation of each circuit to be estimated. The estimations will only be rough ones since no covering or graph simplification is performed.

The first table below shows the comparison of the number of LUTs and CLBs used for different values of ρ_{IN}. Each entry is the total percentage difference in the resource for two values of ρ_{IN}. For instance, the first entry (-171) indicates the total percentage difference in the number of LUTs used for $\rho_{\text{IN}} = 8$ compared to $\rho_{\text{IN}} = 7$. A total of 147 circuits are used. It is seen that the results improve as ρ_{IN} increases. In these experiments, $\rho_{\text{OUT}} = 2$ and $s = 9$.

ρ_{IN}	8		9		10	
	LUTs	CLBs	LUTs	CLBs	LUTs	CLBs
7	−171.0	−202.6	−449.6	−461.8	−467.8	−510.5
8			−278.2	−257.5	−296.5	−306.3
9					−18.4	−46.7

Next the effect of s on the same circuits used above is examined. Since $\rho_{\text{IN}} = 9$ and 10 give the best results, ρ_{IN} is restricted to these values. The table shows the total percentage difference in the resources using two sets of parameters compared to another, with $\rho_{\text{OUT}} = 2$ in all cases. It is seen that $s = 9$ gives the best results.

(ρ_{IN}, s)	$(10,8) - (9,8)$		$(10,9) - (9,8)$		$(10,9) - (10,8)$	
	LUTs	CLBs	LUTs	CLBs	LUTs	CLBs
$\sum$%	−54.5	−29.0	−177.5	−184.4	−117.6	−155.0

Finally, the influence of the value of ρ_{OUT} is assessed. Again, the results of the total percentage differences of the resources with various values of ρ_{OUT} compared to $\rho_{\text{OUT}} = 1$ are tabulated. In all cases $\rho_{\text{IN}} = 10$ and $s = 9$. Two circuits, `cm42a` and `t481`, were observed to influence the results greatly. The contributions of these circuits are included in the table so that the results without

them can be assessed.

ρ_{OUT}	2		3		4	
	LUTs	CLBs	LUTs	CLBs	LUTs	CLBs
$\sum$%	−125.2	−117.5	−194.6	−199.0	−215.6	−227.3
cm42a	0	0	0	0	−23.1	−28.6
t481	0	0	−71.4	−80.0	−71.4	−80.0
Revised	−125.2	−117.5	−123.1	−118.9	−121.1	−118.7

ρ_{OUT}	5		6	
	LUTs	CLBs	LUTs	CLBs
$\sum$%	−219.0	−228.2	−218.1	−226.6
cm42a	−23.1	−28.6	−23.1	−28.6
t481	−71.4	−80.0	−71.4	−80.0
Revised	−124.5	−119.6	−123.6	−118.1

The revised row indicates that if the two circuits are discounted, then $\rho_{OUT} = 5$ gives the best results. In fact, a direct comparison between $\rho_{OUT} = 3$ and $\rho_{OUT} = 5$ reveals that cm42a is the major part of the difference.

From the results above, the values

$$\rho_{IN} = 10; \quad \rho_{OUT} = 5 \text{ and } s = 9,$$

are proposed.

5.3 Node Elimination

An important graph restructuring operation is the elimination of nodes within a DAG by collapsing some nodes into others. This is useful because one wants to limit the number of nodes in a DAG. Quite often, the logic optimisation stage introduces small nodes into the DAG. Since an efficient usage of the LUTs is desired, such nodes should be removed from the DAG. This section describes algorithms that perform this operation. The novelty is its application to heterogeneous DAGs. The elimination of nodes is independent of the operations described in the preceding sections.

For a node v to be eliminated from a DAG, it must satisfy the condition that none of its fanout nodes has more than ρ_{IN} inputs once v is collapsed into it. More formally, after collapsing v,

$$\forall w \in \mathcal{F}_{OUT}(v) \bullet |\mathcal{F}_{IN}(w_v)| \leq \rho_{IN}, \tag{5.7}$$

where w_v is the node w with v collapsed into it. If the feasibility of the DAG is to be maintained, then ρ_{IN} must be set to κ_α.

The goal here is to eliminate as many nodes as possible given the above constraint. Let the Boolean DAG be $D = (V, A)$. The first step is to determine the set $V' \subset V$ of nodes which could be eliminated. In actual fact, only a subset of V' will be eliminated as the following argument shows. Consider two nodes $u, v \in V'$ where $v \in \mathcal{F}_{\text{OUT}}(u)$. For both nodes to be collapsible, it is required that:

- u is collapsible into all w_v for each $w \in \mathcal{F}_{\text{OUT}}(v)$; and
- v_u is collapsible into all $w \in \mathcal{F}_{\text{OUT}}(v)$.

Hence, selecting u for collapsing affects the viability of collapsing both its fanins and its fanouts. Quite often, collapsing u would preclude the collapsing of any node from either set.

5.3.1 A Collapsing Options Graph

Let $X \subseteq V'$ be the set of nodes that will actually be eliminated. The size of X should be maximised. Finding the optimal solution for X requires an exhaustive search. For tractability reasons, a heuristic approach that uses a greedy strategy is adopted.

This problem can be related to the *set-covering* or *vertex-cover* problems. The approach here is to construct an cyclic, *undirected* graph $G = (V', E')$ such that if $u, v \in V'$ and u is either a fanin or fanout of v, then $(u, v) \in E'$. Furthermore, if $v \in \mathcal{F}_{\text{OUT}}(u)$, then

$$\forall w \in \mathcal{F}_{\text{IN}}(v) \ \wedge \ w \in V' \bullet (u, w) \in E'.$$

Therefore, G is possibly disconnected. Now $X \subseteq V'$ with the property that any two nodes in X are mutually collapsible and none is a fanin (or fanout) of another. This restriction on X may be too restrictive since it is sometimes possible to collapse both a node and its fanin. Nonetheless, it serves as a good basis for computing a maximal X in an iterative manner.

The set X can be constructed as follows:

1. Select a node $v \in V'$ and add it to X.
2. For all edges $(u, v) \in E'$, delete the edge and u from G.
3. Repeat until $V' = \emptyset$.

The optimality of the above scheme depends on the selection of v at each stage. The greedy strategy selects a node v with the least number of edges incident on it. This means that if v has no edges incident on it, then it does not affect any other node in V'. Conversely, the more edges are incident on v, the more nodes it would preclude from being eliminated from the DAG.

EXAMPLE 5.3.1 *Consider a graph with* $V' = \{a, b, c, d, e, f, g, h\}$ *where:* b *is a fanin of* c*;* e*,* f *and* g *are fanins of* h*; and* d *is a fanin of* g*. The resulting graph is shown in Figure 5.4. First,*

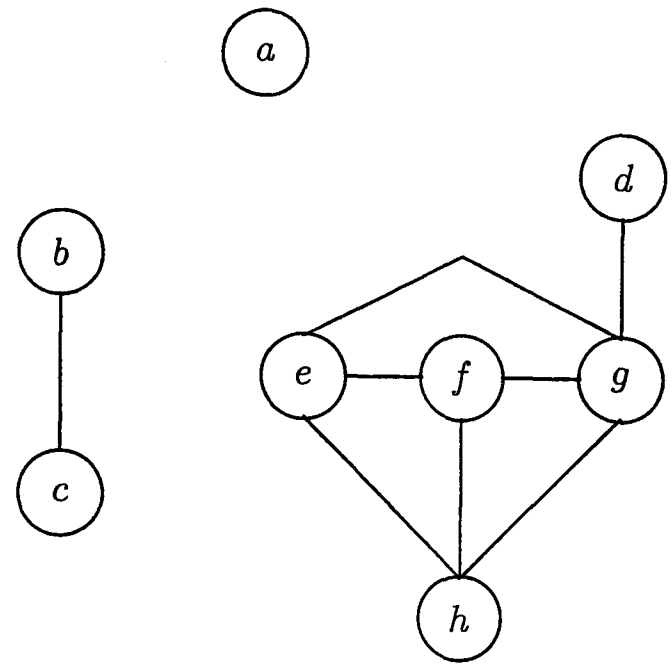


Figure 5.4: An example of a collapse options graph

select a because it has degree 0. Then, the algorithm might select b (degree 1) which eliminates c from the graph. Then d could be selected, eliminating g at the same time. Finally, e is selected, eliminating both f and h. In tabular form, the sequence of selections is shown below.

Step	V'	X
0	$\{a, b, c, d, e, f, g, h\}$	$\emptyset$
1	$\{b, c, d, e, f, g, h\}$	$\{a\}$
2	$\{d, e, f, g, h\}$	$\{a, b\}$
3	$\{e, f, h\}$	$\{a, b, d\}$
4	$\emptyset$	$\{a, b, d, e\}$

In the above example, it was assumed that, for example, e and f cannot be collapsed into h simultaneously, since they are both fanins of h. Hence, only e is selected for collapsing and put into X. It is possible that both could be collapsed into h. This can be checked by examining the DAG after e has been collapsed into h. This observation applies to all nodes in X.

A procedure SELECT-OPTIONS implements this scheme of computing and implementing X given the sets of vertices and edges in the undirected graph (see Section A.4 in the appendix). It returns the number of nodes collapsed. Note that X is built by eliminating any vertex whose degree is not zero. If a vertex u has degree 1, then there must be an edge $(u, v) \in E'$. Removing this edge from E' reduces the degree of u to 0.

5.3.2 Heterogeneous Nodes

With heterogeneous DAGs, there are two types of nodes, α and β. The value for ρ_{IN} in Equation 5.7 must reflect the difference between these nodes. Therefore, ρ_{IN} is set to κ_α for α nodes and to κ_β for β nodes. However, if the type of a node v has not been determined, ρ_{IN} can be set to the larger between the input size of v and a user-specified parameter σ, where $\sigma \geq \kappa_\alpha$. A larger value for σ allows more nodes to be eliminated but might yield rather complex functions which would be difficult to decompose. Therefore, a balance between eliminating nodes and not creating complex functions must be found. The value of σ is crucial in this respect.

It is assumed that a procedure MAKE-COLLAPSE-G constructs the collapsing options graph G from the set V of nodes in the Boolean DAG. For practical reasons, MAKE-COLLAPSE-G omits any node from G that has an excessive number of cubes. Such nodes are difficult or even impossible to collapse.

5.3.3 Iterative Elimination

In Section 5.3.1 it was shown that SELECT-OPTIONS finds a subset of V' that may be too pessimistic. This deficiency is overcome by iterating over the DAG, computing a new set X on each iteration. In this way, those nodes that could have been collapsed in a previous iteration will be collapsed in the current one.

```
1  proc ELIMINATE-NODES (D, σ)
2    comment: D = (V, A) is a Boolean DAG; σ is a parameter.
3    repeat
4      (V', E') ← MAKE-COLLAPSE-G(V, σ)
5      m ← SELECT-OPTIONS(V', E')
6      if m > 0 then
7          changed ← true
8      else
9              changed ← false
10     endif
11   until changed = false
12 end.
```

The loop in ELIMINATE-NODES terminates when no further nodes can be eliminated. This procedure is used on both a feasible DAG and an infeasible one. In fact, it turns out to be better to apply ELIMINATE-NODES before and after the application of decomposition and restructuring operations on a DAG. The experiments in the following sections will show the effect of ELIMINATE-NODES and of σ on the mapping of a DAG.

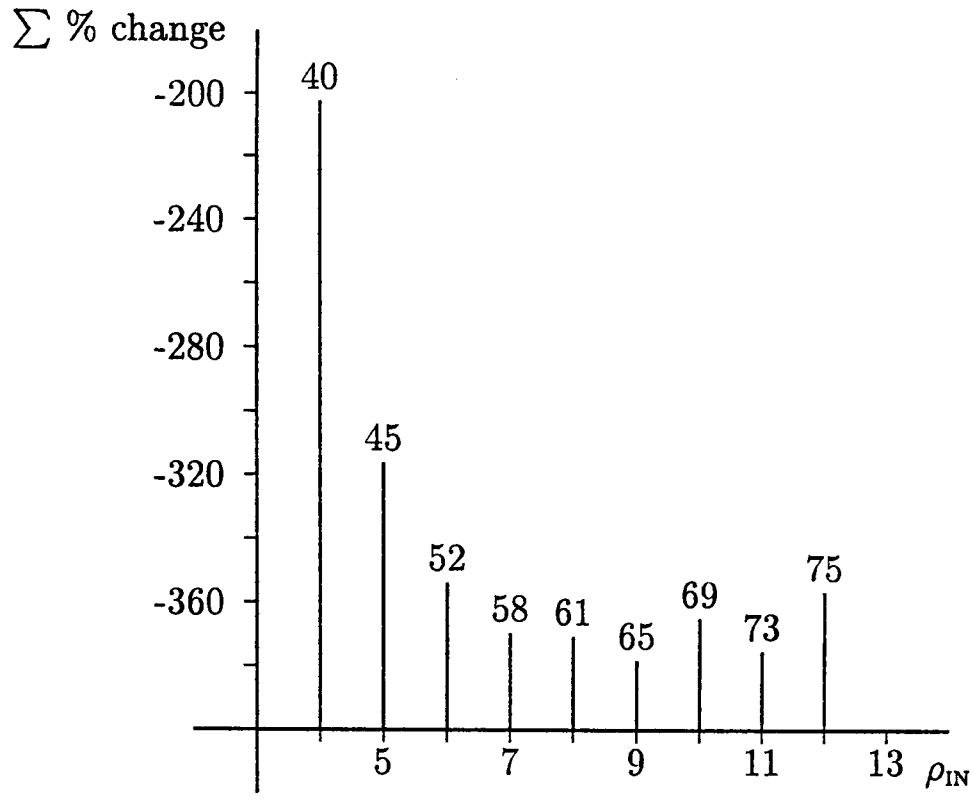


Figure 5.5: The total % change in the number of nodes due to various values for ρ_{IN} compared to no prior elimination. Above each bar is the number of circuits affected out of the 151 tested.

5.3.4 Experiments on Node Elimination

The procedure ELIMINATE-NODES should be applied after decomposition and before covering, where it has the maximum impact. However, it turns out that its application before decomposition results in smaller feasible graphs. Indeed, the infeasibility of the graph can even be allowed to increase before decomposing it. The following experiment tests various values of ρ_{IN} as discussed in Section 5.3 to determine its optimal value. Three steps are executed for each DAG.

1. Apply ELIMINATE-NODES for a given ρ_{IN}.
2. Apply HETERO-DECOMP (see Section 4.6) to decompose the DAG.
3. Apply ELIMINATE-NODES again without making the DAG infeasible.

The control experiment omits the first step. Figure 5.5 shows the sum (from all the circuits) of the percentage change in the number of nodes when prior elimination is used. Because the change is usually a decrease in the number of nodes, these sums are *negative*.

It is clear from Figure 5.5 that prior elimination benefits the decomposition process. A value of $\rho_{IN} > 4$ would make more nodes infeasible. This could cause the decomposition to introduce more nodes than are eliminated. It is therefore surprising that 9 is found to be an optimal value for ρ_{IN}.

Above this value, larger nodes are created which are decomposed inefficiently. Prior elimination undoes some of the optimisations. The decomposition process must defray this loss so that the number of nodes does not increase.

5.4 Graph Simplification

Simplification is part of the optimisation procedure which is beyond the scope of this thesis. Nonetheless, this section addresses the application of minimisation techniques to a Boolean DAG and its effect on LUT minimisation briefly. These techniques are independent of all others presented in other sections.

According to the authors of MIS, the minimal form of a DAG is one in which the factored form has the least literals [4]. Hence, the number of literals is the pre-eminent measure. Schlag et al. show that there is a linear correlation between the number of homogeneous LUTs needed to implement a set of Boolean expressions and the number of literals in the expressions [62]. However, for LUT-based FPGAs, the number of literals is not actually the most important measure. A k-input LUT can implement any function of a up to k variables regardless of how complex it is. Therefore, the most important attributes of a DAG are the number of nodes and their input sizes. The functionality of the nodes is irrelevant. Quite often, the application of the simplify command in MIS to a Boolean DAG increases the infeasibility of some nodes. In other words, the nodes are re-expressed in terms that reduce their literal counts but increase the fanins to the nodes. Therefore, the nodes have to be decomposed again leading to more nodes in the DAG and a more expensive mapping.

The simplification techniques can still be harnessed to improve the overall mapping for LUT-based FPGAs. The aim here is to find a function g within the DAG so that some other function f can be expressed in terms of g. However, a check is made to see that if f is κ_α-feasible, then it does not become infeasible as a result of the simplification. The other measures evaluated are:

- the number of literals in the factored form, $N_1(f)$;
- the number of literals in the *sum-of-products* form $N_2(f)$; and
- the number of cubes, $N_3(f)$.

Let $|\mathcal{F}_{\text{IN}}(f)| = N_0(f)$ in the following discussion. The decision on whether a function g should replace f for a given node in the DAG is taken in the following steps.

ACCEPT-SIMP(f, g, κ_α)

1. If the two nodes are infeasible, then check to see if the infeasibility is reduced. If the infeasibility is reduced, then accept the simplification. When the two functions are both infeasible, g may replace f even if it has more variables. The reasoning behind this is that a

simpler function (by other measures) will often be easier to decompose. However, f should not be re-decomposed if it is already feasible.

2. Compare each measure for f and g, accepting the simplification if $N_j(g)$ gives a smaller value and rejecting it if $N_j(g)$ gives a larger value. A comparison for $(j+1)$, is only made if there is a tie at j, where $1 \leq j < 3$.

Whereas the comparison order is not fixed, it has been determined by experimentation that the order: inputs, literals in factors, literals in *sum-of-products*, and cubes gives the best results.

The modified simplification procedure is shown below. The actual minimisation operation is carried out by standard minimisation routines.

```
1  proc GOOD-SIMPLIFY (G)
2    comment: G = (V, E) is a DAG.
3    foreach f ∈ V do
4      if |F_IN(f)| ≥ 2 then
5        DC ← don't care set of f
6        Trim the don't-care set.
7        g ← Minimisation of f using DC.
8        accept ← ACCEPT-SIMP(f, g, κ_α)
9        if accept = true then
10         Replace f with g.
11       endif
12     endif
13   endfor
14 end.
```

□

Chapter 6

Overall Algorithm and Results

This chapter presents comparisons of results obtained by the new synthesis method and other equivalent methods for sets of benchmark circuits. The individual synthesis steps are also assessed to see their effect on the overall synthesis process.

The experiments are conducted on a set of MCNC and LGSynth93 circuits. The circuits are pre-optimised in the customary way using a script in SIS [64]. All the algorithms have been implemented in the SIS tools. The SIS command `verify` is used to verify the circuits.

The results in Section 6.1 are crucial in determining the overall synthesis method. This method was then compared to other approaches in Section 6.2. It was also used to obtain the results listed in Section 6.3.

6.1 Effects of Various Synthesis Steps

The experiments in this section test the performance of various synthesis steps. These include the covering step and the re-synthesis step. The effect of node elimination was demonstrated in the experiments of Section 5.3.4.

The comparisons are based on the number of nodes in the DAGs after the synthesis steps have been invoked. The Xilinx tools are not used here since they would affect the results and obscure the actual influences of each process. Once each node has been assigned a type (either α or β), the number of logic blocks can be counted by traversing the DAG. In comparing two methods, the circuits are grouped according to the percentage improvement of one method over another. The tables show the number of circuits in each category.

6.1.1 Improved Cube-Packing

The first experiment tests the performance of the cube-packing algorithm presented in Chapter 4 (with kernel decomposition) compared to that of Murgai et al. [52]. The experiment involves

running each algorithm and then counting the number of nodes. For a fair comparison to be made, a homogeneous version of the cube-packing algorithm is used. The node sizes are set to five since this is what Murgai et al. considered.

% Improvement	-5 – 0	Equal	0 – 5	5 – 10	10 – 15	15 – 20	20 – 25	25 – 30
# circuits	4	77	33	17	2	5	1	1

The table above shows that for most circuits, the algorithms produce equivalent results. Since both algorithms are simply decomposition algorithms feasible DAGs will not be affected. For circuits with several functions composed of disjoint cubes, the two algorithms will produce the same results. Only four circuits give worse results with the new method. Most of the improvements are between 0 to 10%. On average, the improvement is over 2%. The longest time taken by any circuit is 5.9 seconds and the new method only requires an average of 0.26 seconds more per circuit.

6.1.2 Covering

The next experiment tests how much influence the covering step has on the mapping. Decomposition is indispensable to technology mapping. The cube-packing method of Chapter 4 was selected to decompose the DAGs. The results below are a comparison between:

- decomposition alone *vs.*
- decomposition followed by the covering algorithm of Chapter 3.

% Improvement	-15 – -10	-5 – 0	Equal	0 – 5	5 – 10	10 – 15	15 – 20
# circuits	1	1	39	34	20	12	8
% Improvement	20 – 25	25 – 30	30 – 35	40 – 45	45 –50	> 50	
# circuits	4	6	4	3	1	7	

The table above shows that the covering phase improves the mapping in most cases. The two times in which it produces worse results can be explained by the fact that the covering defines many nodes as β nodes which leads to more CLBs. On average, the covering leads to over 10% fewer CLBs.

How does the decomposition method effect the covering? To test this, the cube-packing decomposition is compared with a simple AND-OR decomposition in which all nodes are decomposed into either 2-input OR gates or 2-input AND gates. The results below are a comparison between:

- AND-OR decomposition and covering *vs.*
- cube-packing decomposition and covering.

The covering is done by the same algorithm.

% Improvement	-30 – -20	-20 – -10	-10 – 0	Equal	0 – 10	10 – 20
# circuits	1	1	2	20	7	15
% Improvement	20 – 30	30 – 40	40 – 50	50 – 60	60 –70	
# circuits	31	35	21	5	2	

The table above demonstrates that the decomposition method has a significant effect on the covering results. The cube-packing decomposition leads to a better mapping than the simple AND-OR decomposition (by over 24%). If the decomposition method produces a good structure for the DAG, then the covering will enhance it.

6.1.3 Re-synthesis

The final experiment tests the influence of the re-synthesis algorithm in Chapter 5. The DAGs are decomposed with the cube-packing algorithm and then covered. The results below are a comparison between:

- cube-packing decomposition and covering *vs.*
- re-synthesis, followed by cube-packing decomposition and covering.

% Improvement	< -100	-60 – -50	-40 – -30	-30 – -20	-20 – 10	-10 – 0
# circuits	1	1	3	2	2	23
% Improvement	Equal	0 – 10	10 – 20	20 – 30	30 – 40	40 – 50
# circuits	54	33	8	5	1	2
% Improvement	50 – 60	60 – 70	70 – 80	80 – 90		
# circuits	1	1	1	2		

As the table above demonstrates, re-synthesis has a varied effect on the mapping. On average the results are just over 2% better with re-synthesis. In order to get better results, it is necessary to decide when to apply re-synthesis and when not to. The synthesis method presented below does this.

6.1.4 Mapping Scheme

One of the most difficult problems is deciding on a good flow for the synthesis. The main question is what order the synthesis steps should be invoked. In fact, the only indispensable part of the synthesis is decomposition, since an infeasible DAG cannot be mapped onto the FPGA. The covering step does not make a DAG infeasible. Therefore, it should be the final step in the synthesis.

Decomposition introduces more nodes into the DAG. Sometimes this may be unnecessary if the DAG was restructured. Moreover, a restructured DAG can be decomposed in a more economical fashion. These arguments suggest that re-synthesis should precede decomposition.

The elimination of nodes should be done after decomposition. The experiments in Section 5.3.4 demonstrate that the DAGs can be decomposed in a better way if some nodes are eliminated prior to the decomposition.

With the these observations, a synthesis method has been devised. The new synthesis method is as follows:

1. Eliminate some nodes in the DAG as explained in Section 5.3. The DAG may increase its infeasibility.

2. Decompose the DAG using RESYNTHESISE (see Section 5.2.3) followed by HETERO-DECOMP (see Section 4.6). Use HETERO-DECOMP alone and pick the better of the two approaches.

3. Simplify the DAG as explained in Section 5.4.

4. Eliminate some more nodes in the DAG again, but without making it infeasible.

5. Cover the DAG with LUTs (see Section 3.5.3).

The flow can be explained as follows. Firstly, the DAGs have some nodes eliminated. These are usually small nodes with a few fanouts. Whereas the infeasibility of the DAG might increase, the decomposition methods have a great deal more freedom to structure the DAG in a way that fits the FPGA architecture. It was shown above that re-synthesis does not benefit all DAGs. Hence, the next step ensures that the result is not adversely affected by the re-synthesis algorithms. Simplification improves the sharing of nodes among the existing nodes. In practice, only a few DAGs are affected by this step. The next step eliminates any superfluous nodes. The preceding steps may re-structure the DAG to an extent that some nodes become superfluous. Finally, the covering step completes the mapping. For the circuits tested, this scheme takes at most a few minutes to complete on a SUN SPARCstation 5.

6.2 Xilinx FPGAs

This section is concerned with experiments carried out on Xilinx FPGAs. The DAGs are mapped onto a Xilinx 4013PQ240 device, speed version 4. For a given DAG, the synthesis method presented above is used. Then the Xilinx place and route tools ppr are used to map the DAG. The results of the placement are reported by ppr and the number of occupied CLBs is quoted. It is emphasised that this value is not usually the minimum number of CLBs that were needed. The Xilinx tools place the LUTs so that the routing can be made easier. Hence, quite often the LUTs

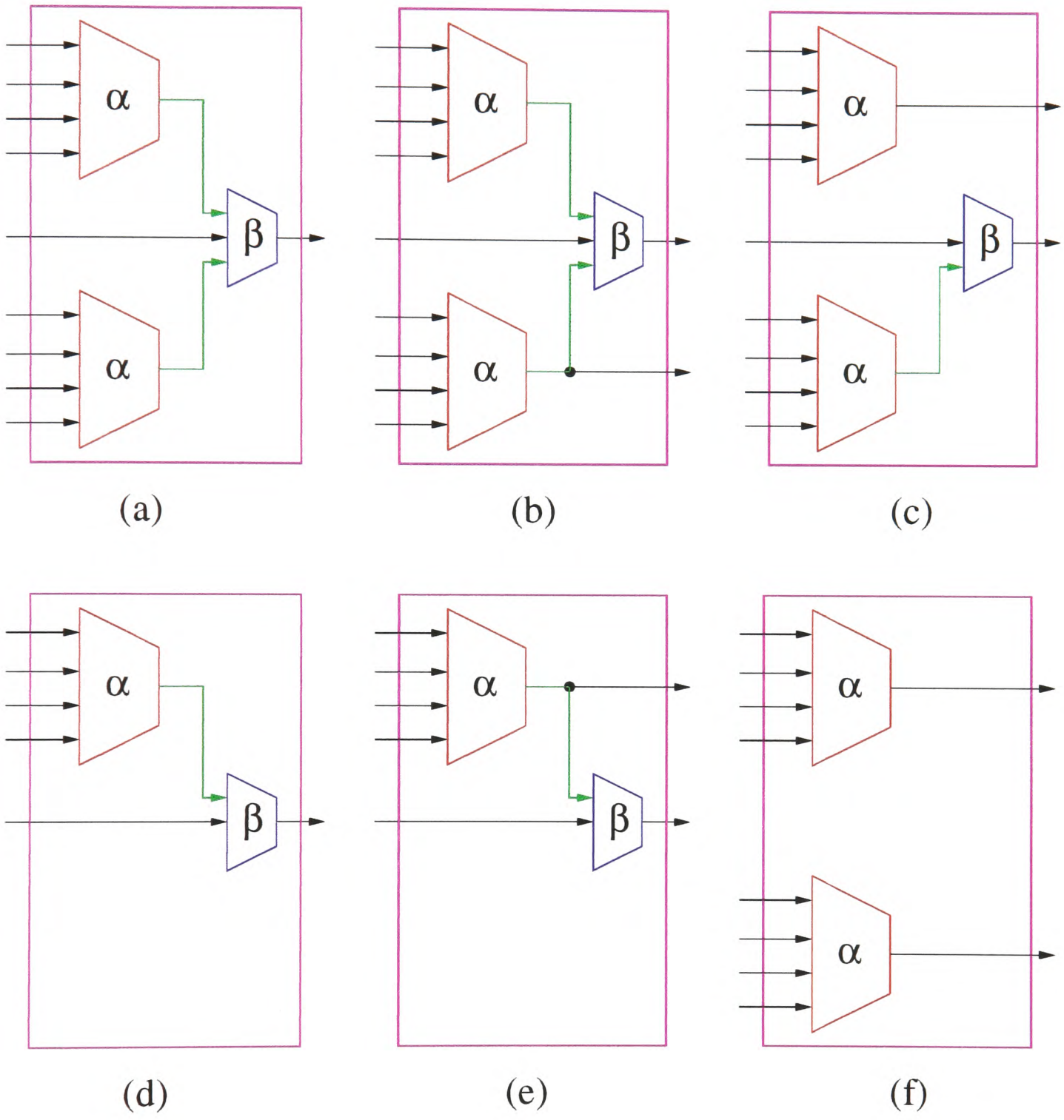


Figure 6.1: The CLB patterns of an Xilinx 4K device. A seventh pattern is the α LUT on its own in a CLB and is not shown above.

may be dispersed among several CLBs. Figure 6.1 shows the various patterns in a Xilinx 4K device. So far, only patterns (a), (d), (e) and (f) have been considered, since a general architecture was being considered. In the results that follow, all patterns are represented.

The delay results were analysed using the Xilinx tool XDelay. The number of levels indicates the maximum number of LUTs through which a signal propagates from an external input to an external output. Both *H* LUTs and and *F/G* LUTs are counted. Hence, the levels do not give an accurate indication of the circuit delay since the hard-wired link is not treated specially. The actual delay results are also given. These are greatly influenced by the placement and routing which is beyond the control of the synthesis method. Therefore, the delay values should be regarded as more a product of ppr than the mapping tools.

Three comparisons are made with other authors' tools whose results were available. The synthesis method is referred to as X.

6.2.1 FGSyn

The CLB utilisation results of FGSyn, taken from [43], are the best ones currently available. They are better than using the XACT tools alone. Table 6.1 shows that on average X outperforms FGSyn by over 11% in terms of percentage differences or 4% in terms of the total number of CLBs. The circuit `alu4` requires 51 CLBs with FGSyn and 100 CLBs with X. The results of FGSyn cannot be replicated using their method. It is likely the circuit `alu4` used here is different from that used by the authors of FGSyn. This possibility is further reinforced by the fact that another tool ASYL [1] uses 211 CLBs for `alu4`. It is also not clear whether the authors of FGSyn report the actual number of CLBs used or the minimum number that could have been used. For instance, the circuit `decod` is feasible after optimisation. All that FGSyn does is to decompose the DAG and therefore, it would not alter the structure of `decod`. With this structure it seems improbable that the 19 nodes in `decod` could be mapped onto 9 CLBs as reported in [43]. If the `alu4` results were omitted, X would outperform FGSyn by over 15%.

6.2.2 ASYL

The second comparison is with ASYL [1]. The authors of ASYL had no other results to compare theirs with which may have disadvantaged them. From Table 6.2 it is seen that X outperforms ASYL by over 27% in terms of percentage differences or 30% in terms of the total number of CLBs. In fact, in only one circuit, `alu2`, does ASYL do better than X.

6.2.3 MOFL

The third comparison is made with MOFL by Lee and Shragowitz [45]. Table 6.3 principally shows the number of CLBs used for the benchmarks used in [45]. The columns contain the raw

Table 6.1: Comparison of CLB utilisation between X and FGSyn (24 circuits compared).

Circuit	FGSyn CLBs	X CLBs	Difference	% change
alu4	51	100	49	96.08
decod	9	10	1	11.11
e64	43	45	2	4.65
alu2	52	54	2	3.85
mux	5	5	0	0.00
duke2	70	70	0	0.00
c8	18	17	-1	-5.56
misex2	19	17	-2	-10.53
vda	109	97	-12	-11.01
b9	23	20	-3	-13.04
count	19	16	-3	-15.79
apex7	37	31	-6	-16.22
9symml	6	5	-1	-16.67
9sym	6	5	-1	-16.67
c1355	47	39	-8	-17.02
comp	17	14	-3	-17.65
vg2	16	13	-3	-18.75
misex1	9	7	-2	-22.22
5xp1	15	11	-4	-26.67
rd73	7	5	-2	-28.57
rd84	10	7	-3	-30.00
sao2	29	19	-10	-34.48
f51m	10	6	-4	-40.00
clip	23	13	-10	-43.48
Totals	650	626	-24	-268.63
Average	–	–	-1.00	-11.19
Std. Dev.	–	–	11.03	26.17

Table 6.2: Comparison of CLB utilisation between X and ASYL (17 circuits compared).

Circuit	ASYL CLBs	X CLBs	Difference	% change
alu2	51	54	3	5.88
duke2	73	70	-3	-4.11
vg2	15	13	-2	-13.33
e64	52	45	-7	-13.46
5xp1	13	11	-2	-15.38
sao2	23	19	-4	-17.39
apex7	38	31	-7	-18.42
misex2	21	17	-4	-19.05
misex1	9	7	-2	-22.22
z4ml	4	3	-1	-25.00
count	22	16	-6	-27.27
9sym	9	5	-4	-44.44
rd84	14	7	-7	-50.00
rd73	10	5	-5	-50.00
f51m	12	6	-6	-50.00
alu4	211	100	-111	-52.61
clip	29	13	-16	-55.17
Totals	606	422	-184	-471.99
Average	–	–	-10.82	-27.76
Std. Dev.	–	–	25.33	18.31

Table 6.3: Comparison of CLB utilisation, levels, and delay between X and MOFL (25 circuits compared).

Circuits	CLBs	%	Levels	%	Delays (ns)	%
pcle	-3	-23.08	0	0.00	-24.4	-38.67
i9	-29	-27.88	-1	-14.29	-9.8	-10.54
frg1	-11	-32.35	2	40.00	7.4	10.24
vda	-51	-34.46	3	42.86	7.8	7.29
b9	-11	-35.48	1	25.00	16.1	29.81
c8	-11	-39.29	1	33.33	-16.9	-26.28
x2	-5	-41.67	-1	-33.33	-4.4	-11.22
c432	-26	-42.62	6	50.00	36.3	24.17
pcler8	-10	-43.48	0	0.00	-29.2	-35.65
alu4	-88	-46.81	5	31.25	57.8	29.34
x1	-45	-47.37	1	33.33	9.3	14.88
count	-15	-48.39	0	0.00	-20.5	-19.64
sct	-9	-50.00	3	75.00	7.9	14.13
lal	-12	-50.00	1	25.00	-8.1	-14.34
example2	-49	-50.00	-1	-20.00	0.0	0.00
alu2	-60	-52.63	2	15.38	-9.4	-5.24
apex7	-36	-53.73	3	75.00	0.7	0.91
c880	-60	-55.05	2	25.00	6.8	6.00
z4ml	-4	-57.14	0	0.00	0.9	2.09
c1908	-75	-58.14	0	0.00	-0.3	-0.19
c1355	-79	-66.95	-7	-53.85	-11.8	-10.50
term1	-39	-68.42	-1	-20.00	-12.2	-18.32
c499	-89	-70.08	-4	-40.00	-21.0	-17.62
9symml	-55	-91.67	-2	-40.00	-42.9	-52.00
t481	-192	-98.97	-6	-75.00	-103.6	-75.79
Averages	-42	-51.43	7	6.99	-6.54	-7.89
Std. Dev.	–	17.38	–	37.48	–	24.43

difference as well as the percentage difference. The results from X are on average 51% better in terms of percentage differences or an average of 42 CLBs less. It should be noted, however, that MOFL optimises for delay as well as area. The delays from X are on average 7% better. The delay results should be treated with caution since ppr has the greater influence over them, and moreover, the devices used may be different. It is also possible that the circuit `t481` as used by Lee and Shragowitz is different from the one used here. They need 194 CLBs compared to 2 obtained with X.

6.3 Detailed Results

The detailed results are presented in this section. These results are obtained by performing the synthesis method presented above on a set of circuits and then estimating how many CLBs and LUTs would be required to implement each circuit. Each β LUT is associated with one or two α LUTs, which are not associated to any other β LUT. Some α LUTs are not associated to any β LUT at all; such α LUTs are called *unassigned* LUTs. If the number of β LUTs is N_β and the number of unassigned α LUTs is N_u, then the estimated number of CLBs is:

$$N_{\text{CLB}} = N_\beta + \left\lceil \frac{N_u}{2} \right\rceil.$$

Note that pairs of unassigned α LUTs are packed into one CLB each. Single input nodes, such as buffers and inverters, are not counted.

The Xilinx tools were not used so that the results are unaffected by them. In reporting these results, it is hoped that other researchers will do the same and therefore fairer comparisons can be made. Such detailed results have not yet been reported by others. Tables 6.4 and 6.5 show the results.

Table 6.4: Results of the Mapping Scheme

Circuit	β LUTs	α LUTs	*CLBs*	*Levels*	*Circuit*	β LUTs	α LUTs	*CLBs*	*Levels*
5xp1	2	21	11	3	e64	0	86	43	23
9sym	1	9	5	4	ex4	13	158	75	8
9symml	1	9	5	4	ex5	17	118	65	8
C1355	4	74	38	7	ex5p	21	139	74	5
C17	0	2	1	1	example2	15	90	52	6
C1908	8	102	54	17	f51m	1	13	7	4
C3540	54	314	158	19	frg1	7	38	21	11
C432	5	68	34	19	i1	2	14	8	4
C499	4	74	38	7	i2	23	57	29	9
C880	22	90	51	16	i3	4	78	39	4
cm162	1	13	7	4	i4	24	54	33	9
Z5xp1	2	22	12	3	i5	0	66	33	9
Z9sym	1	9	5	4	i6	39	67	53	2
adder	0	32	16	16	i8	65	269	144	9
alu2	13	102	55	16	i9	49	145	73	6
alu4	24	190	98	22	inc	4	30	15	8
apex1	86	445	236	14	k2	46	366	191	11
apex2	4	48	26	12	lal	1	26	13	4
apex3	114	493	266	14	ldd	2	27	14	6
apex4	117	670	360	111	majority	1	1	1	2
apex5	48	194	108	15	misex1	1	13	7	4
apex6	11	185	96	10	misex2	2	33	18	4
apex7	10	54	32	9	misex3	32	168	91	17
b1	0	3	1	1	misex3c	18	125	68	30
b12	5	20	13	4	mm4a	4	33	18	10
b9	4	41	20	4	mm9a	14	79	43	19
bw	10	37	22	5	mm9b	19	110	59	25
c8	1	35	18	4	mult16a	0	32	16	16
cc	1	18	9	3	mult16b	0	31	16	1
cht	0	39	20	2	mult32a	0	64	32	32
clip	2	33	17	5	mux	1	10	5	4
cm138a	0	9	5	2	my_adder	0	32	16	16
cm150a	1	10	5	4	o64	16	34	17	8
cm151a	0	6	3	3	parity	0	5	3	2
cm152a	2	5	3	3	pcle	5	15	10	4
cm163a	2	9	6	3	pcler8	7	23	15	5
cm42a	0	10	5	1	pdc	14	95	52	7
cm82a	0	4	2	2	pm1	1	16	7	4
cm85a	5	6	6	4	rd53	3	5	3	2
cmb	3	12	6	3	rd73	2	9	6	3
comp	10	21	14	7	rd84	3	14	8	4
con1	2	3	2	2	s1196	17	169	88	11
cordic	1	14	7	6	s1238	16	180	95	15
count	4	38	21	7	s1423	16	142	78	21
cps	58	387	204	11	s208.1	3	15	9	8
cu	2	16	9	5	s344	4	42	18	5
daio	0	3	2	1	s349	4	42	18	5
dalu	43	189	98	11	s382	5	41	22	8
decod	0	20	10	2	s400	6	40	22	8
duke2	17	124	64	12	s420.1	8	29	19	16

Table 6.5: Results of the Mapping Scheme (*cont*)

Circuit	β LUTs	α LUTs	*CLBs*	*Levels*
s444	5	40	22	6
s526	8	36	21	6
s526n	7	35	21	6
s641	2	64	31	12
s713	2	64	31	12
s838	18	64	41	32
s838.1	16	62	39	32
s9234.1	38	247	131	12
s953	24	135	72	7
sao2	2	37	20	8
sbc	38	234	129	8
sct	1	17	9	6
spla	25	166	93	11
sqrt8	1	13	7	5
sqrt8ml	3	13	7	7
squar5	2	11	7	2
t481	0	5	3	2
table3	40	274	145	56
table5	58	296	159	13
tcon	0	8	4	1
term1	3	34	17	6
too_large	26	115	63	11
ttt2	8	46	27	6
unreg	0	32	16	2
vda	33	186	100	11
vg2	1	25	13	5
x1	17	99	52	5
x2	2	13	8	5
x3	41	173	107	6
x4	4	113	56	6
xor5	1	1	1	2
z4ml	0	6	3	3

Chapter 7

Conclusions and Future Work

7.1 Aims of the Work

The work presented here is an integrated approach to the technology mapping of LUT-based heterogeneous FPGAs. The thesis addresses primarily the problem of optimising the area utilisation. This is justified by the fact that the delay depends, to a large extent, on the interconnections external to the logic blocks. If these can be minimised by a compact mapping onto as few logic blocks as possible, then the delay will also be reduced. Furthermore, fewer programmable interconnections lead to an improvement in both the routability and power consumption of the designs. For heterogeneous LUT-based FPGAs, this becomes even more marked since the hard-wired link between LUTs in a logic block will lead to smaller propagation delays. It is advantageous to pack more LUTs into a logic block.

As heterogeneous LUT-based FPGAs are growing in importance in the IC design field, there is an added need to develop efficient design automation tools for them. This is especially so where their logic density is at a premium and where the speeds of the devices are critical. The technology mapping phase is crucial in the design process, but it has not yet been fully addressed for heterogeneous LUT-based FPGAs. This study aims to fill this gap and point to ways in which the problem can be solved.

Technology mapping can be split into different components. It is important that the general approach harmonises the various categories so that the overall results are optimal. Often researchers in this area have failed to recognise this aspect. An aim of this study was to overcome this deficiency.

7.2 Contributions

The problem addressed in this thesis has been examined as an extension of the technology mapping of homogeneous LUT-based FPGAs. Several methods have been proposed for solving the original

problem. Therefore, it is possible to find useful pointers to potential approaches to heterogeneous technology mapping.

The problem was also studied from a graph-theoretic and Boolean algebra approach. This gives the overall method its rigour. Quite often, methods have been devised that work very well in certain circumstances but perform poorly in others. For a method to perform well in the "average" case, it is important that it should be founded on sound theory. However, for tractability reasons, it is often necessary to resort to heuristics. The effectiveness of the heuristics can be demonstrated by experimentation.

The general contribution is that new algorithms have been developed that address the specific requirements of the new problem. Each of the algorithms dealing with different aspects presented here is devised so as to complement others. This gives the overall scheme its coherency. The effectiveness of the scheme is demonstrated by the overall results which improve on previous results obtained by other researchers.

A good approach is determined to be as follows. First, restructure the Boolean DAG by node elimination and partial collapsing. Then the DAG can be decomposed to make it feasible. After simplification and more node elimination, the DAG can be mapped into its final covering of LUTs.

7.2.1 Decomposition

In terms of the decomposition method, the contributions can be summarised as: an extension of the bin-packing strategy to heterogeneous decomposition in Chapter 4 and a new approach to using functional decomposition for heterogeneous graphs in Chapter 5. Not all aspects of technology mapping have equal importance. Indeed, Legl et al. show that the decomposition method has the greatest influence on the mapping [46]. More effort was expended on DAG decomposition than on covering. Moreover, various decomposition methods were explored to improve the mapping. It is unlikely that any one decomposition method will suffice for all circumstances. Each method performs well under a restricted set of circumstances. It was shown that it is possible to combine various methods in a harmonious fashion.

The decomposition methods have been extended to reflect the heterogeneous nature of the DAGs. The decomposition for heterogeneous DAGs is made harder by the fact that a good balance has to be sought for the different types of nodes. In the homogeneous case, it suffices to limit the number of nodes within the decomposition subgraph. In the new case, a preponderance of one type of node might actually increase the area measure, even if the total number of nodes is reduced. Along with balancing the nodes, the proposed algorithms also seeks to use more hard-wired links at the expense of the programmable links.

The new decomposition schemes proposed here include an extended bin-packing approach based on the conventional best-fit decreasing (BFD) strategy. It has been enhanced with new

features, including the extraction of sub-cubes, and cascading nodes in the decomposition tree. The functional decomposition method was extended to heterogeneous DAGs. A variable partitioning method was devised that was based on the functional structure of a node. Additionally, the decomposition methods are combined so that the a better decomposition is achieved than is possible using only one method. The improvements are as follows.

Cube-Packing: The major contribution is a general extension to cube-packing in which a multiplicity of solutions were computed simultaneously. From these, a near-optimal solution is computed for the packing. It turns out that the number of solutions that need to be held is relatively small (around 5). Hence, the extra computational effort required is marginal. The packing is made even better by making the cubes as orthogonal as possible. This is achieved by common cube division and extraction. Additionally, while decomposing an infeasible cube, the nature of the function at the node is taken into account. Thirdly, the nodes in the decomposition tree are cascaded as much as possible. It was shown that cascading improves the overall decomposition results most, followed by common cube division.

Functional Decomposition: Functional decomposition can be the most powerful decomposition method. A simple way that it can be applied to heterogeneous decomposition is by using varying bound set sizes in consecutive decompositions. It was shown that this works best if the nodes have been partially collapsed. However, functional decomposition is generally very computationally expensive. To reduce the number of decompositions that are examined, it is useful to form a disjoint decomposition on the node function. Chapter 5 shows how a function can be factorized and the factorisation used as a good indication on how to partition the variables. The factorisation sometimes even leads to a decomposition of the function.

Combined Methods: Mixing decomposition methods achieves the best results. Cube-packing can be used as a quick and effective guide on the limits of any decomposition on a function. The results presented in the thesis show that it is better to combine cube-packing with functional and kernel decompositions. Sometimes some clues can be gathered from the function on which method would perform best. Unfortunately, most functions are not amenable to this. Moreover, it is unlikely that a single method will ever suffice for all functions.

7.2.2 Covering

The main contributions on covering are: a mapping algorithm for heterogeneous covering; the use of flow network techniques to compute cuts in heterogeneous graphs; and the reduction in the sizes of flow networks derived from Boolean DAGs.

Clustering the nodes to form LUTs or for the purposes of collapsing a subgraph is an important part of the synthesis process. The flow networks technique is very useful in this respect and has been used by other researchers [52, 14]. With this technique, two useful things are possible. Firstly, the minimum cut of the transitive fanin of a given node can be identified. Thus, the fanin size of the node can be reduced to its minimal size by partial collapsing. If the number of fanins are sufficiently few, then no decomposition would be necessary, which would pre-empt the introduction of extra nodes into the DAG. A second use is in the computation of a cut of a given size, greater than the minimum cut. Thus, subgraphs of a desirable size for collapsing can be found. Clusters that use the LUT capacity more efficiently can also be computed.

A way of reducing the sizes of flow networks so as to reduce the complexity of the computation of cuts was shown in Chapter 3. Coupled with a dynamic programming approach, a set of feasible clusters for each node are computed. The selection of a subset of the clusters corresponds to the covering problem. An effective scheme for this was developed; it is an iterative one that maps nodes in various cones with regard to the mappings that they induce in other cones. A good cost function was also suggested for selecting a subset of the competing clusters to cover the DAG.

7.2.3 Restructuring

In terms of restructuring, Chapter 5 shows how computing min-cuts, partial collapsing, and decomposition can be combined to restructure a DAG so that the mapping is improved. Restructuring the DAG is important if the number of LUTs used is to be minimised. A simple and effective technique is the elimination of nodes in which a node is collapsed into all its fanouts. The fanin sizes of the nodes left must be constrained. Therefore, not all collapses are permissible nor are they mutually exclusive. It was shown that a graph-based strategy can be used to select a maximal set of nodes to eliminate. In Section 5.3, it was shown that the elimination technique is also useful if applied before decomposition. In this case, the input size constraint can be relaxed so as to give the decomposition more influence on the final DAG mapping.

Chapter 5 showed how flow networks can be used in the partial collapse of subgraphs. It also shows that by partial collapsing and re-decomposing a DAG, it can be re-synthesised so that the final result is a smaller DAG.

Another technique relies on the simplification strategies usually employed in the logic optimisation stage. These are beyond the scope of this work. However, it was shown that simplification should be applied after decomposition in order to avoid the duplication of logic among various nodes.

7.2.4 Mapping Scheme

A method was devised to perform the technology mapping of heterogeneous FPGAs, such as the Xilinx 4000 series devices. The algorithms have been incorporated into the SIS tool [64]. The comparisons of the mapping results on Xilinx devices show an improvement of over 11% over existing mapping tools for the same devices. Most researchers report results over a few benchmark circuits. The experiments were conducted over a large range of circuits. The author encourages other researchers to report results over as large a set possible. For heterogeneous FPGAs, it is not sufficient to report the logic block utilisation, but the number of the different types of LUTs should also be reported. This will enable a fairer and more accurate comparison between the various methods to be made. Additionally, a conventional way of reporting results is to quote the percentage improvements. However, a percentage depends on the base. For instance, in the comparison to MOFL, for `t481` the new method used 2 CLBs compared to 194 used by MOFL. In percentage terms, the difference is either $19200/2 = 9600\%$ or $19200/194 = 99\%$. There should be a standard set of results against which all methods can be compared. An extensive set of results is presented in Chapter 6.

7.3 Future Work

7.3.1 Placement

The work presented here has covered the entire area of technology mapping for LUT-based FPGAs with the aim of minimising area. More work needs to be done on the trade-off between area and delay optimisation and possibly power. This is important since the area measure is not always the most critical for some designs. Equally, the question of the allocation to logic blocks of the LUTs and the placement of the logic blocks relative to one another needs to be addressed. This can be expected to further improve on the overall design since the different phases affect the results of others. Most research has tended to view the mapping and placement phases as being completely distinct.

7.3.2 BDD-based Decomposition

There has been a lot of work done on the use of BDDs in functional decomposition [44, 43, 47, 71, 70]. The current implementation of the Roth-Karp decomposition method is based on the one in SIS. The use of the BDD method would improve the speed of computation. For very large industrial circuits, running the Roth-Karp algorithm is impractical due to the computational time and memory requirements. There is a possibility that the use of BDDs would ameliorate this.

An encoding method to select the decomposition functions is also needed for the functional decomposition method. A simple and widely-used encoding method is "serial encoding". A more

sophisticated encoding could improve the decomposition results. Some work has been done on this problem by Murgai et al. [54] and Huang et al. [35].

7.3.3 Multiple-Output Decomposition

A more direct application of Karp's theorem on multiple disjoint decompositions would be desirable [40]. This is likely to be very computationally expensive and may not be viable. Perhaps an inexpensive method for determining the existence of such decompositions could be found.

Another interesting aspect is multiple-output functional decomposition as devised by Wurth et al. [71, 70]. It does not present immediate benefits to heterogeneous decomposition but it might be adapted to this purpose.

7.3.4 Logic Optimisation

Most logic optimisation techniques have been developed with homogeneous architectures in mind. When applied to heterogeneous DAGs, they are still beneficial. However, these techniques might be modified to reflect more closely the heterogeneous architectures. Examples of these are decomposition, simplification and re-substitution. Furthermore, they might be coupled with technology mapping to achieve a better overall performance. It is conceivable that the logic optimisation stage and technology mapping need no longer be seen as distinct stages. □.

Bibliography

[1] P. Abouzeid, B. Babba, M. Crastes de Paulet, and G. Saucier, *Input-driven partitioning methods and applications to synthesis on table-lookup-based FPGAs*, IEEE Trans. Computer-Aided Design **12** (1993), no. 7, 913 – 925.

[2] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin, *Network flows: Theory, algorithms, and applications*, vol. 1, Prentice Hall, New Jersey, USA, 1993.

[3] Robert K. Brayton and Curt McMullen, *The decomposition and factorization of Boolean expressions*, International Symposium on Circuits and Systems (Rome), IEEE Computer Society Press, May 1992, pp. 49 – 54.

[4] Robert K. Brayton, Richard Rudell, Alberto Sangiovanni-Vincentelli, and Albert R. Wang, *MIS: A multiple-level logic optimization system*, IEEE Trans. Computer-Aided Design **6** (1987), no. 6, 1062 – 1081.

[5] Shih-Chieh Chang and Malgorzata Marek-Sadowska, *Technology mapping via transformations of function graphs*, International Conference on Computer Design: VLSI in Computers and Processors (Cambridge, MA, USA), IEEE Computer Society Press, October 1992, pp. 159 – 162.

[6] ———, *Technology mapping and circuit depth optimization for field programmable gate arrays*, IEEE Custom Integrated Circuits Conference (San Diego, CA, USA), IEEE Computer Society Press, May 1993, pp. 3.5.1 – 3.5.4.

[7] Shih-Chieh Chang, Malgorzata Marek-Sadowska, and TingTing Hwang, *Technology mapping for TLU FPGAs based on decomposition of Binary Decision Diagrams*, IEEE Trans. Computer-Aided Design **15** (1996), no. 10, 1226 – 1236.

[8] Chau-Shen Chen, TingTing Hwang, and C . L. Liu, *Low power FPGA design - a re-engineering approach*, 34th Design Automation Conference (Anaheim, CA, USA), ACM, June 1997, pp. 656 – 661.

[9] Chau-Shen Chen, Yu-Wen Tsay, TingTing Hwang, Allen C. H. Wu, and Youn-Long Lin, *Combining technology mapping and placement for delay-minimization in FPGA designs*, IEEE Trans. Computer-Aided Design **14** (1995), no. 9, 1076 – 1084.

[10] K. N. Chen, T. S. Wang, and Y. T. Lai, *A new strategy of performance-directed technology mapping algorithm for LUT-based FPGAs*, 1996 IEEE International Symposium on Circuits and Systems (Atlanta, GA, USA), vol. 4, IEEE Computer Society Press, May 1996, pp. 822 – 825.

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

[12] Amit Chowdhary and John P. Hayes, *Technology mapping for field-programmable gate arrays using integer programming*, IEEE/ACM International Conference on Computer-Aided Design (San Jose, CA, USA), IEEE Computer Society Press, November 1995, pp. 346 – 352.

[13] Jason Cong and Yuzheng Ding, *Beyond the combinatorial limit in depth minimization for LUT-based FPGA designs*, IEEE/ACM International Conference on Computer-Aided Design, 1993, pp. 110 – 114.

[14] ———, *FlowMap: An optimal technology mapping algorithm for delay optimization in lookup table based FPGA designs*, IEEE Trans. Computer-Aided Design **13** (1994), no. 1, 1 – 12.

[15] ———, *On area/depth trade-off in LUT-based FPGA technology mapping*, IEEE Trans. on VLSI Systems **2** (1994), no. 2, 137 – 148.

[16] ———, *On nominal delay minimization in LUT-based FPGA technology mapping*, Integration - The VLSI Journal **18** (1994), 73 – 94.

[17] ———, *Combinational logic synthesis for LUT based Field Programmable Gate Arrays*, ACM Trans. on Design Automation for Electronic Systems **1** (1996), no. 2, 145 – 204.

[18] Jason Cong, Yuzheng Ding, Tong Gao, and Kuang-Chien Chen, *An optimal performance-driven technology mapping algorithm for LUT-based FPGAs under arbitrary net-delay models*, Computers & Graphics **18** (1994), no. 4, 507 – 516.

[19] Jason Cong and Yean-Yow Hwang, *Simultaneous depth and area minimization in LUT-based FPGA mapping*, ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (Monterey, California, USA), ACM, February 1995, pp. 68 – 74.

[20] ———, *Structural gate decomposition for depth-optimal technology mapping in LUT-based FPGA design*, 33rd Design Automation Conference (Las Vegas, NV, USA), ACM, June 1996, pp. 726 – 729.

[21] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, MIT Press, 1993.

[22] H. Allen Curtis, *A generalized tree circuit*, Journal of the ACM **8** (1961), no. 4, 484 – 496.

[23] Frank Dresig, Oliver Rettig, and Utz G. Baitinger, *Logic synthesis for universal logic cells*, FPGAs (Oxford, UK) (Will R. Moore and Wayne Luk, eds.), Abingdon EE & CS Books, Abingdon, UK, September 1991, pp. 179 – 190.

[24] Amir Farrahi and Majid Sarrafzadeh, *TDD: A technology dependent decomposition algorithm for LUT-based FPGAs*, IEEE International ASIC Conference (Portland, OR, USA), IEEE Computer Society Press, September 1997, pp. 206 – 209.

[25] Amir H. Farrahi and Majid Sarrafzadeh, *Complexity of the lookup-table minimization problem for FPGA technology mapping*, IEEE Trans. Computer-Aided Design **13** (1994), no. 11, 1319 – 1332.

[26] ———, *FPGA technology mapping for power minimization*, 4th International Workshop on Field Programmable Logic and Applications (Prague, Czech Republic) (Reiner W. Hartenstein and Michal Z. Servit, eds.), Springer-Verlag, Berlin, Germany, September 1994, pp. 66 – 77.

[27] David Filo, Jerry Chih-Yuan Yang, Frédéric Mailhot, and Giovanni de Micheli, *Technology mapping for a two-output RAM-based field programmable gate array*, European Design Automation Conference (Amsterdam, Netherlands), IEEE Computer Society Press, February 1991, pp. 534 – 538.

[28] Robert J. Francis, *Technology mapping for lookup table-based FPGAs*, Ph.D. thesis, University of Toronto, 1991.

[29] Robert J. Francis, Jonathan Rose, and Kevin Chung, *Chortle: A technology mapping program for lookup table-based field programmable gate arrays*, 27th Design Automation Conference, 1990, pp. 613 – 619.

[30] Robert J. Francis, Jonathan Rose, and Zvonko Vranesic, *Chortle-crf: Fast technology mapping for lookup table-based FPGAs*, 28th Design Automation Conference (San Francisco, CA, USA), IEEE Computer Society Press, June 1991, pp. 227 – 233.

[31] ———, *Technology mapping of lookup table-based FPGAs for performance*, IEEE/ACM International Conference on Computer-Aided Design, 1991, pp. 568 – 571.

[32] Tong Gao, Kuang Chien Chen, Jason Cong, Yuzheng Ding, and C. Liu, *Placement and placement driven technology mapping for FPGA synthesis*, Sixth Annual IEEE International ASIC Conference (Rochester, NY, USA) (G. W. D'Luna, L. J. andBrown and P. P. K. Lee, eds.), IEEE Computer Society Press, October 1993, pp. 91 – 94.

[33] Matthias Groh, *Technology mapping for look-up table FPGAs*, FPGAs (Oxford, UK) (Will R. Moore and Wayne Luk, eds.), Abingdon EE & CS Books, Abingdon, UK, September 1991, pp. 191 – 200.

[34] Jianshe He and Jonathan Rose, *Technology mapping for heterogeneous FPGAs*, ACM/SIGDA International Workshop on FPGAs (Berkeley, CA, USA), February 1994.

[35] Juinn-Dar Huang, Jing-Yang Jou, and Wen-Zen Shen, *Compatible class encoding in Roth-Karp decomposition for two-output LUT architecture*, IEEE/ACM International Conference on Computer-Aided Design (San Jose, CA, USA), IEEE Computer Society Press, November 1995, pp. 359 – 363.

[36] ———, *An iterative area/performance trade-off algorithm for LUT-based FPGA technology mapping*, 1996 IEEE/ACM International Conference on Computer-Aided Design (San Jose, CA, USA), IEEE Computer Society Press, November 1996, pp. 13 – 17.

[37] Ting-Ting Hwang, Robert MichaelOwens, Mary Jane Irwin, and Kuo-Hua Wang, *Logic synthesis for field-programmable gate arrays*, IEEE Trans. Computer-Aided Design **13** (1994), no. 10, 1280 – 1287.

[38] Jie-Hong Jiang, Jing-Yang Jou, Juinn-Dar Huang, and Jung-Shian Wei, *BDD based lambda set selection in Roth-Karp decomposition for LUT architecture*, Asia and South Pacific Design Automation Conference (Chiba, Japan), IEEE Computer Society Press, January 1997, pp. 259 – 264.

[39] D. S. Johnson, A. Demers, J. D. Ullmans, M. R. Carey, and R. L. Graham, *Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM J. Computing **3** (1974), no. 4, 299 – 325.

[40] Richard M. Karp, *Functional decomposition and switching circuit design*, Journal of the Society for Industrial and Applied Mathematics **11** (1963), no. 2, 291 – 335.

[41] Kevin Karplus, *Xmap: A technology mapper for table-lookup field-programable gate arrays*, 28th Design Automation Conference (San Francisco, CA, USA), IEEE Computer Society Press, June 1991, pp. 240 – 243.

[42] Venkataramana Kommu and Irith Pomeranz, *GAFPGA: Genetic algorithm for FPGA technology mapping*, European Design Automation Conference (Hamburg, Germany), IEEE Comput. Soc. Press, Los Alamitos, CA, USA, September 1993, pp. 300 – 305.

[43] Yung-Te Lai, Kuo-Rueih Ricky Pan, and Massoud Pedram, *OBDD-based function decomposition: Algorithms and implementation*, IEEE Trans. Computer-Aided Design **15** (1996), no. 8, 977 – 990.

[44] Yung-Te Lai, Massoud Pedram, and Sarma B. K. Vrudhula, *BDD based decomposition of logic functions with applications to FPGA synthesis*, 30th Design Automation Conference, ACM, 1993, pp. 642 – 647.

[45] Jun-Yong Lee and Eugene Shragowitz, *Technology mapping for FPGAs with complex block architectures by fuzzy logic technique*, 1st Asia and South Pacific Design Automation Conference (Japan), 1995.

[46] Christian Legl, Klaus Eckl, and Bernd Wurth, *Performance-directed technology mapping for LUT-based FPGAs - what role do decomposition and covering play?*, 1996 International Workshop on Field Programmable Logic and Applications (Darmstadt, Germany) (Reiner W. Hartenstein and Manfred Glesner, eds.), Lecture Notes in Computer Science 1142, no. 6, Springer, September 1996, pp. 14 – 23.

[47] Christian Legl, Bernd Wurth, and Klaus Eckl, *A Boolean approach to performance-directed technology mapping for LUT-based FPGA designs*, 33rd Design Automation Conference (Las Vegas, NV, USA), ACM, June 1996, pp. 730 – 733.

[48] ———, *An implicit algorithm for support minimization during functional decomposition*, European Design & Test Conference (Paris, France), IEEE Computer Society Press, March 1996, pp. 412 – 417.

[49] Ilan Levin and Ron. Y. Pinter, *Realizing expression graphs using table-lookup FPGAs*, European Design Automation Conference with EURO-VHDL (Hamburg, Germany), IEEE Computer Society Press, September 1993, pp. 306 – 311.

[50] B. M. E. Moret and H. D. Shapiro, *Algorithms from p to np: Design and efficiency*, vol. 1, Benjamin/Cummings Publishing Co., California, USA, 1991.

[51] Rajeev Murgai, Yoshihito Nishizaki, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, *Logic synthesis for programmable gate arrays*, 27th Design Automation Conference, 1990, pp. 620 – 625.

[52] Rajeev Murgai, Narendra Shenoy, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, *Improved logic synthesis algorithms for table look-up architectures*, IEEE/ACM International Conference on Computer-Aided Design, 1991, pp. 564 – 567.

[53] ———, *Performance directed synthesis for table look up programmable gate arrays*, IEEE/ACM International Conference on Computer-Aided Design, 1991, pp. 572 – 575.

[54] Ranjeev Murgai, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, *Optimum functional decomposition using encoding*, 31st Design Automation Conference (San Diego, CA, USA), ACM, June 1994, pp. 408 – 414.

[55] S. S. Park, Y. H. Lee, S. H. Hwang, and C.-M. Kyung, *Cofactor packing algorithm for lookup-table based field programmable gate arrays*, Electronics Letters **30** (1994), no. 15, 1207 – 1209.

[56] Mariusz Rawski, Lech Jozwiak, Miroslawa Nowicka, and Tadeusz Luba, *Non-disjoint decomposition of Boolean functions and its application in FPGA-oriented technology mapping*, 23rd EUROMICRO Conference (Budapest, Hungary), September 1997, pp. 24 – 30.

[57] J. Paul Roth and R. M. Karp, *Minimization over Boolean graphs*, IBM Journal **6** (1962), 227 – 238.

[58] Alberto Sangiovanni-Vincentelli, *Some considerations on field-programmable gate arrays and their impact on system design*, Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping (Herbert Grünbacher and Reiner W. Hartenstein, eds.), Lecture Notes in Computer Science, no. 705, Springer-Verlag, 1992, pp. 26 – 34.

[59] Alberto Sangiovanni-Vincentelli, Abbas El-Gamal, and Jonathan Rose, *Synthesis methods for field programmable gate arrays*, Proceedings of the IEEE **81** (1993), no. 7, 1057 – 1083.

[60] Prashant Sawkar and Donald Thomas, *Area and delay mapping for table-look-up based field programmable gate arrays*, 29th Design Automation Conference, 1992, pp. 368 – 373.

[61] ———, *Performance directed technology mapping for look-up table based FPGAs*, 30th Design Automation Conference, ACM, 1993, pp. 208 – 212.

[62] Martine Schlag, Pak K. Chan, and Jackson Kong, *Empirical evaluation of multilevel logic minimization tools for an FPGA technology*, FPGAs (Oxford, UK) (Will R. Moore and Wayne Luk, eds.), Abingdon EE & CS Books, Abingdon, UK, September 1991, pp. 201 – 213.

[63] Martine Schlag, Jackson Kong, and Pak K. Chan, *Routability-driven technology mapping for lookup table-based FPGAs*, IEEE Trans. Computer-Aided Design **13** (1994), no. 1, 13 – 26.

[64] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli, *SIS: A system for sequential circuit systems*, Tech. Report UCB/ERL M92/41, University of California, Berkeley, USA, May 1992.

[65] Wen-Zen Shen, Juinn-Dar Huang, and Shih-Min Chao, *Lambda set selection in Roth-Karp decomposition for LUT-based FPGA technology mapping*, 32nd Design Automation Conference (San Francisco, CA, USA), ACM, June 1995, pp. 65 – 69.

[66] Hyunchal Shin and Chunghee Kim, *Performance-oriented technology mapping for LUT-based FPGAs*, IEEE Trans. on VLSI Systems **3** (1995), no. 2, 323 – 327.

[67] Ted Stanion and Carl Sechen, *A method for finding good Ashenhurst decompositions and its applications to FPGA synthesis*, 32nd Design Automation Conference (San Francisco, CA, USA), ACM, June 1995, pp. 60 – 64.

[68] Stephen M. Trimberger (ed.), *Field-programmable gate array technology*, Kluwer Academic Publishers, 1994.

[69] Chua-Chin Wang and Cheng-Pin Kwan, *Low power technology mapping by hiding high-transition paths in invisible edges for LUT-based FPGAs*, IEEE International Symposium on Circuits and Systems (Hong Kong), IEEE Computer Society Press, June 1997, pp. 1536 – 1539.

[70] Bernd Wurth, Klaus Eckl, and Kurt Antreich, *Functional multiple-output decomposition for lookup-table based FPGAs*, International Workshop on Logic Synthesis (Tahoe City, California), May 1995.

[71] ———, *Functional multiple-output decomposition: Theory and an implicit algorithm*, 32nd Design Automation Conference (San Francisco, CA, USA), ACM, June 1995, pp. 54 – 59.

[72] Xilinx, *The programmable logic data book*, Xilinx Inc., San Jose, California, USA, 1993.

[73] Honghua Yang and D. F. Wong, *Edge-Map: Optimal performance driven technology mapping for iterative LUT based FPGA designs*, 1994 IEEE/ACM International Conference on Computer-Aided Design (San Jose, CA, USA), ACM, November 1994, pp. 150 – 155.

[74] Shujian Zhang, D. Micheal Miller, and Jon C. Muzio, *Notes on 'complexity of the lookup-table minimization problem for FPGA technology mapping'*, IEEE Trans. Computer-Aided Design **15** (1996), no. 12, 1588 – 1590.

Appendix A

Pseudo-code for Algorithms

A.1 Graph Covering Algorithms

A.1.1 Altering Arc Capacities

```
1  function ALTER-CAPS (N̂_v, U, f, D)
2     comment: D is an f-tuple, (d_1, ..., d_f),
3     comment: and the cutset of v is U = {u_1, ..., u_f}.
4     admit ← ADMISSIBLE(u_1, D[1])
5     comment: admit is a variable that records the admissibility
6     comment:  of each flow in a u_i.
7     i ← 2
8     while (i ≤ f ∧ admit = true) do
9           admit ← admit ∧ ADMISSIBLE(u_i, D[i])
10          i ← i + 1
11    endwhile
12    if (admit = true) then
13       comment: flows are admissible for all u ∈ U.
14       Q ← ∅
15       for i = 1 to f do
16             if (D[i] > 0) then
17                ENQUEUE(Q, (u_i, 1 + D[i]))
18             endif
19       endfor
20       while (Q ≠ ∅) do
21             (u, d) ← DEQUEUE(Q)
22             c(u', u'') ← max(c(u', u''), d)
23             W ← cutset of u
24             D' ←  value of dis[d] for u
25             for j = 1 to |W| do
26                   if (D'[j] > 1) then
27                      ENQUEUE(Q, (w_j, D[j]))
```

```
28              endif
29            endfor
30        endwhile
31    endif
32    return (admit)
33 end.
```

As is seen above, the procedure ALTER-CAPS first checks that the distribution is admissible. The variable *admit* is set to *false* by any subnetwork of $\hat{\mathcal{N}}_v$, rooted at u_i, that does not admit an extra flow given by $D[i]$ (lines 4 and 9). The procedure ADMISSIBLE need only check a pointer to confirm this and hence takes constant time. Since the cutset is of size f, the admissibility of D can be confirmed in time linear to f. Next, the lines 12 to 31 are executed only if D is admissible. Line 16 tests if the capacity of a given node u_i needs to be increased. If so, u_i is placed in a queue. Notice that the queue holds pairs of nodes and their new capacities. The loop of lines 20 to 30 progressively increases $c(u', u'')$ for the node u at the head of the queue and then enqueues all nodes w in the cutset of u for which $c(w', w'')$ may have to increase. Line 22 must check that $c(u', u'')$ has not been increased to a larger capacity already than that required by d. Such a situation would occur if two or more subnetworks intersected at u.

A.1.2 Reforming a Flow Network

```
1  proc REFORM (N̂_v, v)
2    Q ← ⟨v⟩
3    while (Q ≠ ∅) do
4          w ← DEQUEUE(Q)
5          U ← cutset of w
6          foreach vertex u ∈ U do
7            if (c(u', u'') > f(u', u'') > 0) then
8                c(u', u'') ← f(u', u'')
9                ENQUEUE(Q, u)
10           endif
11         endfor
12   endwhile
13 end.
```

A.1.3 Feasible Cuts

```
1  proc FEASIBLE-CUTS1 (N̂_t, s, t, f, U, κ_α, κ_β)
2    comment: Assume that N̂_t = (V̂_t, Ê_t).
3    foreach node u ∈ U
4      if arc (u'', t) ∉ V̂_t then
```

```
5          comment: Introduce an arc (u'', t).
6          Create (u'', t)
7          c(u'', t) ← ∞
8          Ê_t ← {(u'', t)} ∪ Ê_t
9        endif
10     endfor
11     comment: Initialise M and CUT.
12     for i = (f + 1) to κ_α do
13         M[i] ← ∞
14         CUT[i] ← ∅
15     endfor
16     TD ← TABLE(κ_α − f, f)
17     G ← N̂_t
18     for j = 1 to (κ_α − f) do
19         foreach distribution D_j ∈ TD[j]
20           admit ← ALTER-CAPS(G, D_j)
21           if admit = true then
22               g ← MAX-FLOW(G, s, t, κ_α − f)
23               if (0 < g ≤ (κ_α − f)) then
24                   REFORM(G, t)
25                   W ← FIND-CUT(G, s)
26                   if (|W| = f + g) then
27                       z ← cost(t, W)
28                       if (z < M[g + f]) then
29                           M[g + f] ← z
30                           CUT[g + f] ← W
31                           Compute D
32                           R[g + f] ← (D, z)
33                       endif
34                   endif
35               endif
36               if (g = 0) then
37                   comment: Simply reform G.
38                   REFORM(G, t)
39               else
40                    comment: Reinstate G since it was altered.
41                    G ← N̂_t
42               endif
43           endif
44       endfor
45   endfor
46 end.
```

The first lines modify $\hat{\mathcal{N}}_t$ and initialise the arrays and table. Note that the arrays need only be of dimension $[f, \ldots, \kappa_\alpha]$ each. Line 17 makes a copy of $\hat{\mathcal{N}}_t$ since it may be altered. Rather than running a reverse flow algorithm, the choice is made to copy $\hat{\mathcal{N}}_t$ which is much easier. Since the copy G is reused several time, the space requirement is simply doubled. The time needed to copy

$\hat{\mathcal{N}}_t = (\hat{V}_t, \hat{E}_t)$ is $\Theta(|\hat{V}_t| + |\hat{E}_t|)$. The main loop between lines 18 and 45 iterates over all entries in the table. Each element, D_j of each list is processed separately. The max-flow algorithm is run whenever a viable distribution is found. At most $(\kappa_\alpha - f + 1)$ extra augmenting paths are sought. This means that line 22 takes time $\Theta((\kappa_\alpha - f)|\hat{E}_t|)$. The algorithm then checks that the new flow is within the bounds f and κ_α so that G is reformed using, REFORM, and a cutset found, using FIND-CUT. It is important that the new cutset be of the same size as the flow. Therefore, the conditional of lines 26 to 34 tests this and then goes on to compute a cost for the cut. If it is less than the existing one for an equal flow, the entries in R, M, and CUT are exchanged with the new ones. Finally, the conditional in lines 36 to 42 reinstates G to $\hat{\mathcal{N}}_t$ in preparation for the next iteration. Since REFORM is actually faster than copying $\hat{\mathcal{N}}_t$, it is invoked where no extra flow was admitted by G.

To work out the time complexity of FEASIBLE-CUTS1, the size of the table is taken to be bounded by a constant ν, which is a function of κ_α, as argued before. Based on this fact, there are at most ν iterations in the main loop of FEASIBLE-CUTS1. In the sequence of instructions that are executed, FIND-CUT has the greatest bound $O(|\hat{V}_t| + |\hat{E}_t|)$. Hence, the complexity of FEASIBLE-CUTS1 is $O(\nu(|\hat{V}_t| + |\hat{E}_t|))$.

An alternative procedure is given below.

```
1  proc FEASIBLE-CUTS2 (N̂_t, s, t, f, κ_α, κ_β)
2      comment: Assume that N̂_t = (V̂_t, Ê_t).
3      k ← f
4      comment: k records the size of the current cutset being considered.
5      extra ← κ_α − f
6      comment: extra is the maximum allowable extra flow.
7      U ← NIL
8      j ← 1
9      while k < κ_α ∧ j ≤ extra do
10         Let C be the cutset of size k for t.
11         if C ≠ U then
12             U ← C
13             foreach node u ∈ U
14                 if arc (u'', t) ∉ V̂_t then
15                     comment: Introduce an arc (u'', t).
16                     Create (u'', t)
17                     c(u'', t) ← ∞
18                     Ê_t ← {(u'', t)} ∪ Ê_t
19                 endif
20             endfor
21         endif
22         TD ← TABLE(κ_α − k, k)
23         foreach distribution D_j ∈ TD[j]
24             if k < κ_α then
```

```
25                  G ← N̂_t
26                  found ← true
27                  i ← 1
28                  while (i ≤ |U|) ∧ (found = true) do
29                      u ← U[i]
30                      if D_j[i] > 0 then
31                          if u is a primary input then
32                              found ← false
33                          else
34                              comment: Increase the capacity of the bridging arc.
35                              d ← D_j[i] + 1
36                              c(u', u'') ← max(c(u', u''), d)
37                          endif
38                      endif
39                      i ← i + 1
40                  endwhile
41                  if found = true then
42                      g ← MAX-FLOW(G, s, t, κ_α − k)
43                      if (0 < g ≤ j) then
44                          comment: Reform the graph first.
45                          REFORM(G, t)
46                          W ← FIND-CUT(G, s)
47                          if |W| = (f + g) then
48                              z ← cost(t, W)
49                              if (z < M[g + f]) then
50                                  M[g + f] ← z
51                                  CUT[g + f] ← W
52                              endif
53                              k ← max(k, f + g);
54                          endif
55                      endif
56                  endif
57              endif
58          endfor
59          j ← j + 1
60      endwhile
61  end.
```

The procedure FEASIBLE-CUTS2 works from a cutset U of size k. Initially, $k = f$, the max-flow, but increases at each stage of the iteration. As in FEASIBLE-CUTS1, all nodes in U are connected directly to the sink (lines 13 to 20), thus ensuring that any larger cut will bisect X_k, where $(X_k, \overline{X_k})$ is the cut that yields U. Given this modification, a larger cut is then sought by altering the capacities of the bridging arcs of nodes in U for various extra flows. For an extra flow of g, the cost of all cutsets of size $(f + g)$ is compared, and the best one C selected (lines 49 to 52). This cutset become the base cutset in the next iterations (line 12).

A.2 Cube-Packing Algorithms

A.2.1 Finding Common Cubes

```
1  function FIND-COMMON-CUBES (v, Θ)
2    comment: Returns a Boolean indicating whether the procedure should be retried.
3    comment: The set Θ records any new infeasible nodes.
4    Let n_c be the number of cubes of v.
5    if (n_c < 3 ∨ |F_IN(v)| < κ_α) then
6       return false
7    endif
8    changed ← false
9    C ← ∅
10   comment: C is a set of common cubes found.
11   for i ← 1 to (n_c − 1) do
12        p ← i^th cube of v
13        if variable size of p > κ_β then
14           comment: We consider only cubes larger than κ_β.
15           for j ← i + 1 to n_c do
16                q ← j^th cube of v
17                r ← p ∩ q
18                if variable size of r ≥ κ_β then
19                   Append r to C.
20                endif
21           endfor
22        endif
23   endfor
24   if C = ∅ then
25      return false
26   endif
27   if |C| = 1 then
28      Let g be a new node representing c ∈ C.
29      Re-express v in terms of g.
30      if |F_IN(g)| > κ_α then
31         Θ ← Θ ∪ {g}
32      endif
33      return false
34      comment: Don't repeat this procedure for v.
35   endif
36   repeat
37     comment: We now loop selecting the best cube at each iteration.
38     found ← false
39     Count the number of times each c_i ∈ C occurs in all other c_j ∈ C.
40     Let c the largest cube that occurs most frequently.
41     if c occurs more than once then
42        found ← true
```

```
43          changed ← true
44          Let g be a new node representing c.
45          Re-express v in terms of g.
46          if |F_IN(g)| > κ_α then
47              Θ ← Θ ∪ {g}
48          endif
49          Remove any c_j from C that is no longer a sub-cube of v.
50      endif
51  until found = false
52  return (changed)
53 end.
```

A.2.2 Balancing LUTs

It is assumed a procedure FIND-LUT works exactly like FIND-BEST-BIN except that it uses LUTs instead of bins. It also returns the index of the LUT found or -1 if none is found.

```
1  proc BALANCE-LUTS (S_A, S_B)
2   comment: |S_A| < |S_B|.
3   m ← |S_B|
4   i ← 1
5   while (|S_A| < m) do
6           comment: Invariant: i points to the next LUT in S_B.
7           comment: Variant: |S_B| − i.
8           index ← FIND-LUT(S_A, S_B[i], κ_α)
9           if (index < 0) then
10              comment: No LUT was found.
11              S_A ← S_A ∪ {S_B[i]}
12              comment: We might as well fill this latest LUT.
13              i ← i + 1
14              while (index ≥ 0 ∧ i ≤ |S_B|) do
15                      index ← FIND-LUT(S_A, S_B[i], κ_α)
16                      if (index > 0) then
17                          S_A[index] ← (S_A[index] ∨ S_B[i])
18                          i ← i + 1
19                      endif
20              endwhile
21          else
22              comment: An existing LUT was found.
23              S_A[index] ← (S_A[index] ∨ S_B[i])
24              i ← i + 1
25          endif
26          m ← |S_B| − i
27  endwhile
28  S_B ← S_B − {s_k ∈ S_B | 1 ≤ k < i}
29 end.
```

In BALANCE-LUTS above, the loop in lines 14 to 20 ensures that each new member of S_A is fully utilised by packing it with as many LUTs as will go into it. BALANCE-LUTS terminates when $|S_A| \geq |S_B|$.

A.3 Re-synthesis Algorithms

```
1  function ITER-MIXED-DECOMP (v, κ_α, ρ_IN, s, limit)
2    comment: The procedure returns the degree of infeasibility of the decomposition of v.
3    comment: A negative value signifies that v is unchanged.
4    comment: No node with more than ρ_IN inputs will be decomposed.
5    comment: s is the number of inputs to a logic block.
6    Θ ← {v}
7    infeasible ← 0
8    change ← false
9    n ← limit
10   comment: n counts the number of new nodes that may be introduced.
11   repeat
12     Γ ← ∅
13     foreach u ∈ Θ ∧ |F_IN(u)| > κ_α do
14       if |F_IN(u)| > ρ_IN then
15         infeasible ← infeasible + |F_IN(u)|/κ_α
16       else
17           success ← true
18         while (success = true ∧ |F_IN(u)| > κ_α ∧ n > 0) do
19           if |F_IN(u)| ≤ s then
20             r ← κ_α
21           else
22               r ← s
23           endif
24           m ← 0
25           comment: m counts the number of new nodes.
26           success ← MIXED-DECOMP(u, κ_α, r, m, Γ)
27           change ← change ∨ success
29           n ← n − m
30           comment: Update n.
31         endwhile
32         comment: Check for feasibility.
33         if |F_IN(u)| > κ_α then
34           infeasible ← infeasible + |F_IN(u)|/κ_α
35         endif
36       endif
37     endfor
38     Θ ← Γ
39   until Θ = ∅
40   if change = true
```

```
41        return (infeasible)
42    else
43        return (−1)
44    endif
45 end.
```

A.4 Node Elimination

```
1  function SELECT-OPTIONS (V', E')
2    comment: The procedure returns the number of nodes collapsed.
3    comment: V' is a set of vertices and E' is a set of undirected edges.
4    Sort the V' in descending order of degree.
5    comment: Select a subset X of V' for implementation.
6    m ← 0
7    for i = |V'| downto 1 do
8        v ← V'[i]
9        if degree of v is 0 then
10           Collapse v.
11           m ← m + 1
12       else
13           comment: Don't collapse v but delete from graph.
14           foreach (u, v) ∈ E' do
15               Delete v from V' and (u, v) from E'.
16           endfor
17       endif
18   endfor
19   return (m)
20 end.
```

```
1  function MAKE-COLLAPSE-G (V, σ)
2    comment: The procedure returns the collapsing options graph G.
3    comment: V is the set of internal nodes in a DAG; σ is a parameter.
4    V' ← ∅
5    foreach v ∈ V do
6      flag ← true
7      foreach w ∈ F_OUT(v) ∧ flag ≠ false do
8        if w is of type β then
9            ρ_IN ← κ_β
10       elseif w is of type α then
11           ρ_IN ← κ_α
12       else
13           ρ_IN ← max(|F_IN(w)|, κ_α)
14       endif
15       s ← |F_IN(w) ∪ F_IN(v)|
```

```
16          if  s > ρ_IN ∨ s > σ then
17              flag ← false
18          endif
19        endfor
20        comment: flag = false means that v cannot be collapsed.
21        if  flag = true then
22            V' ← V' ∪ {v}
23        endif
24      endfor
25      E' ← MAKE-EDGES(V')
26      G ← (V', E')
27      return (G)
28  end.
```

The conditional of lines 8 to 14 assigns a value to ρ_{IN}. Line 13 ensures that no node will increase its degree of infeasibility. Line 15 computes the fanin size of a fanout of w if v was collapsed into it. This must not exceed either ρ_{IN} or σ (line 16).

$\Diamond$