SURESH RAJGOPAL. Spatial Entropy – A Unified Attribute to Model Dynamic Communication in VLSI Circuits (Under the direction of Kye S. Hedlund and Akhilesh Tyagi)

Abstract

This dissertation addresses the problem of capturing the dynamic communication in VLSI circuits. There are several CAD problems where attributes that combine behavior and structure are needed, or when function behavior is too complex and is best captured through some attribute in the implementation. Examples include, timing analysis, logic synthesis, dynamic power estimation, and variable ordering for binary decision diagrams (BDDs). In such a situation, using static attributes computed from the structure of the implementation is not always helpful. Firstly, they do not provide sufficient usage information, and secondly they tend to exhibit variances with implementations which is not desirable while capturing function behavior.

The contribution of this research is a new circuit attribute called *spatial entropy*. It models the dynamic communication effort in the circuit by unifying the static structure and the dynamic data usage. Quantitatively, spatial entropy measures the switching energy in a physical (CMOS) implementation. A minimum spatial entropy implementation is a minimum energy implementation. For the purposes of this dissertation we restrict our scope to combinational circuits. We propose a simple procedure to estimate spatial entropy in a gate level circuit. It is characterized in extensive detail and we describe why it is difficult to compute spatial entropy accurately. We show how it can also be defined at other levels of abstraction.

We illustrate applications of spatial entropy in BDD variable ordering, a problem that has traditionally relied on static attribute based solutions. We also show empirically that spatial entropy can track function behavior through implementations, by using it to measure gate-count complexity in boolean functions.

Acknowledgments

I would like to take this opportunity to express my sincere gratitude towards my advisor Kye Hedlund and co-advisor Akhilesh Tyagi. This research would have been impossible without their advice, encouragement and guidance. I would like to thank them for the time that they have spent with me and the hours of discussions that we have had. Their comments and suggestions have gone a long way in helping me produce a comprehensive dissertation. They have taught me the essence of patience in research and the need to present ideas clearly and simply. Doing research under them has been exciting and challenging.

I am indebted to my committee member Doug Reeves whose advice and comments helped me keep this research in perspective. I gratefully acknowledge his patience, encouragement and willingness to help. I would also like to thank Don Stanat and David Plaisted for serving on my committee and for their suggestions and encouragement. I appreciate the assistance of Sujit Dey (of NEC Research Labs, Princeton) and Kris Kozminski of the OASIS group at MCNC during this research.

I am grateful to Yuki Watanabe for his support throughout my stay at UNC. I would also like to thank Peter Reintjes of DASIX/Intergraph for his help and encouragement during my graduate career, and my office mates at UNC, Jim Symon and Don Stone for making the department a better place to live/work in.

Finally, I would like to thank my wife Uju for her endless patience and support. Her strength and encouragement have been invaluable during hard times.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The increasing complexity of VLSI circuit design has continually motivated research in VLSI Design Automation and Computer-Aided-Design(CAD) tools for VLSI. The task of these CAD tools is to help automate and speed up the design process. Unfortunately almost all the problems that these tools attempt to solve are NP-complete. As a result many tools and their underlying algorithms can only achieve approximate solutions via heuristics. The quality of these solutions is often dictated by the information the algorithm can extract from the circuit, in the form of attributes. These circuit attributes can be broadly classified into *static* attributes and *dynamic* attributes. Static attributes are usually computed by examining the topology or static structure of the circuit. For example, the gate count attribute used in logic synthesis is a static attribute. Dynamic attributes on the other hand are computed by applying input data to the circuit. They capture circuit behavior or dynamic usage, in contrast to static attributes that capture circuit structure. Examples of dynamic attributes include 1-probability, observability, and controllability, used in test generation algorithms. This dissertation illustrates the use of a unified attribute that captures dynamic circuit usage *and* static circuit structure. As a physical attribute it provides a quantitative measure of switching energy in the circuit.

Static attributes, by themselves, do not always provide sufficient information for algorithms to make good decisions. Since they are computed by statically examining the circuit structure they are unable to estimate how various parts of the circuit get used over different input combinations. In addition, these attributes often use information from one implementation to capture the behavior of a function over all implementations. This over-reliance on the static structure can result in significant variations in the value of the attribute over different implementations. While this may seem desirable for some purposes, it is not helpful when the attribute is trying to capture a property of the function common to all implementations. In contrast to this, an attribute that can capture usage over all the input assignments is likely to show less variation over different implementations. This is because this usage is common to all implementations of a function. When an attribute can combine this dynamic usage information along with the static structure (in an implementation), it can answer questions about the function *and* the implementation.

We have quantitatively defined and characterized a circuit attribute called *spatial entropy* that can capture static structure and dynamic behavior in a circuit. We propose an efficient method to compute this attribute in a given circuit. We comprehensively analyze the attribute, describe the difficulties in computing it, the factors affecting its accuracy, and the effects of various approximations. In CMOS circuit implementations spatial entropy yields a measure of switching energy in the circuit by capturing the switching activity during dynamic power consumption. We illustrate its use in BDD variable ordering, a problem that has traditionally relied on static attribute based solutions. We also show that it can be used to capture function behavior through implementations by measuring of gate-count complexity in boolean functions. In this chapter we begin with an overview followed by an introduction to the problem and the motivation behind it. Then we state our thesis outlining its scope and assumptions. Finally, major contributions of the dissertation are outlined along with a description of the rest of the chapters.

1.1 Overview

VLSI circuit design is the process of transforming a high-level behavioral description into mask geometry that is mapped onto silicon. Due to the complexity of the design process, this task is typically performed by refining the initial behavioral description over several levels of abstraction. At each level of abstraction the circuit is described with increasing detail. This hierarchy of levels corresponds to phases in the design process: *architectural level*, *reg-*

ister transfer or *boolean function level*, *logic gate level*, and *physical level* or *layout level*.

The objective of research in CAD for VLSI is to develop tools to help automate the design tasks performed at the various levels of abstraction. A partial list of research problems in CAD includes: *high-level synthesis* - compiling behavioral HDL descriptions into RTL structures, *logic minimization*, and *logic synthesis* - minimizing two-level and multi-level boolean function descriptions to generate factored forms of the function, *netlist optimization* and *technology mapping* - mapping the factored boolean descriptions into gates belonging to a technology while performing area/time optimizations to the gate level netlist, *timing* and *critical path analysis* - estimating delay in circuit, *simulation* - propagating binary vectors through logic gates and transistors to verify function and timing, *verification* - using formal or symbolic techniques to verify function and timing across abstraction levels, *test generation*, and *testability analysis* - generating test vectors for the circuit and estimating its testability, *placement, layout*, and *routing* - placing the netlist of gates on a two-dimensional plane and connecting the nets with constraints of minimum area and delay.

Several of the research areas cited above use static circuit attributes to derive solutions. A static attribute is a structural attribute that is derived from a static examination of the circuit topology. It does not require the circuit to be exercised with data. For example, consider logic minimization and logic synthesis. The attribute used to guide two-level and multi-level logic minimization [BRSVW87] is *minimum literal count* (or *gate count*) that can be obtained by static examination of the boolean function description or the circuit topology. The phases of technology mapping [LBK88] and netlist optimization [TSB91] also use a static attribute (gate depth) for critical path removal and delay reduction. Timing analyzers [Ous83] work similarly. Placement, layout and routing [LP88] tools also use static attributes such as layout area, active area, net length, wire length and number of vias. In variable ordering for binary decision diagrams (BDDs), the level or depth of a node in a circuit has been used as a static attribute in various heuristics to generate orderings [BRM91].

All these CAD tasks are similar in that they are solved using static attributes defined at a given level of abstraction. These attributes do provide useful information about circuit structure and structural connectivity, and they are fairly easy to compute; but they have their drawbacks. They lack *dynamic usage* information, that is, information about how the various parts of the circuit get used over the different input combinations; and this is essential to solve some problems. For instance, the dynamic power consumption in a circuit is a function of the amount of switching that takes place in the circuit as the nodes change states ($1 \rightarrow 0, 0 \rightarrow 1$), by charging and discharging capacticances. This is difficult to capture when the circuit is not exercised over different input combinations.

It is also difficult to capture function behavior with a static attribute. The number of input combinations that influence a node is usually a measure of the minterms (or cubes) associated with that node, and it is unlikely that a static attribute can capture sufficient information about the 2^n minterms of a given function. A static attribute is computed on the circuit topology, and as a result its value is often influenced by the implementation itself. Two different implementations of the same function may often exhibit significant variances in their static attribute measures. When discriminating amongst implementations this may be beneficial. But this becomes a drawback when the attribute is trying to capture a fundamental characteristic of the function that is expected to remain invariant over different implementations of the function. In this situation such swings in the value of the static attribute are not desirable.

There are some areas of CAD like simulation (gate-level, switch-level, and fault), test generation and testability analysis, where the approach is inherently dynamic and the circuit is exercised for many input assignments. They use dynamic attributes like simulation vectors, signal probabilities, controllabilities, observabilities, and testability. But most of the other CAD tasks that require structure manipulation have restricted themselves to solutions based on static attributes. They have done so primarily because of efficiency considerations and because information from the static circuit structure is essential to solving several of these problems. Dynamic attribute computation requires the circuit to be exercised for several input assignments, and this usually requires more (time and space) resources than static attribute computation.

We believe that an attribute that can capture dynamic usage over various parts of the circuit along with static circuit structure can be useful in providing more information about the function *and* the implementation. We motivate the need for such an attribute in the next section.

1.2 Motivation

Why is such an attribute needed? Almost all problems in CAD are NP-complete [LP88]. It is computationally intractable to find an exact optimal solution for all but the smallest problems. The problem of two-level and multi-level logical minimization is NP-complete [BHMSV84, Law64]. Placement and layout tools face an NP-complete problem in trying to embed a non-planar graph on a plane with minimum arc crossing [GJ79]. Complete gate level or transistor level simulation over all the input assignments takes exponential time. The best known deterministic algorithm to arrive at an optimal variable ordering for a BDD is $O(3^n)$ [FS90]. This means that efficient solutions to all these problems are heuristics or approximate strategies that use information derived from the circuit to achieve desired criteria.

CAD problems that have relied on static attributes to achieve these solutions do not always capture sufficient information about the circuit to be able to make informed decisions. Typically, static attributes try to capture information about the behavior of a function from one particular implementation of the function. Since the attributes are derived from the static structure of the implementation they may fail to provide accurate information about the function as a whole. Consider timing analysis. The static sensitization process used to identify critical paths can erroneously identify *false critical paths* in the implementation that in reality may never be able to propagate a signal under any input combination. Variable ordering for BDDs is another area that relies on some (still unknown) fundamental attribute of the function for its solution. The use of static node depths or node levels (in an implementation) to discriminate between inputs and generate orderings is an attempt to capture function behavior from an implementation. But this may not always be accurate since an implementation represents only one of several possible interpretations of the given boolean function. Our hypothesis is that an attribute that can capture usage over various input combinations along with static circuit structure will be able to provide more information to help algorithms like these make better decisions.

Information about the structure and behavior of an implementation is also very helpful when computing (or estimating) the energy consumption in a circuit. With increasing circuit performance, energy and power consumption are becoming serious concerns for a designer. They are important metrics of circuit performance. The *switching energy* for a device [MC79] is equal to the power consumed by the device (at maximum clock frequency) multiplied by the device delay. In CMOS implementations, the *switching power* P_{sw} at a node is a direct function of the clock frequency and the charging and discharging of circuit capacitances at a node, $P_{sw} \propto C_{sw}V^2f_{clk}$. In order to measure this dynamic power consumption the switching activity over all the nodes in the circuit needs to be estimated. This requires dynamic usage information. The delay in a circuit is a function of the switching capacitance C_{sw} determined by the logic and the wires in the circuit. The time taken for the gate to charge or discharge the switching capacitance determines the delay. Estimating this requires structure and connectivity information of the logic and wires. Switching energy of a circuit is thus a complex function of not just the static structure and connectivity of the implementation, but also the dynamic behavior of the function being implemented. A single attribute that can unify these characteristics will be able to provide a quantitative estimate or measure of the energy consumption in a circuit.

Dynamic usage not only refers to the different combinations that the inputs can take, but also to the manner in which these combinations can cause data to combine and communicate through the circuit. Static structure on the other hand requires information about the gates, their arrangement, and the wires that connect the gates. What makes it difficult to define an attribute that can unify these two features? An important factor is the different levels of abstraction that distort the estimates needed to capture dynamic usage and static structure. The boolean function level estimates circuit usage in terms of cubes and minterms while the logic level might estimate it in terms of the logic gates. This is further complicated by the fact that information at one level of abstraction may be totally (or partially) absent at another. For instance distance estimates are absent at the boolean function level. Some level or depth information is present at the the logic gate level, but wire length information is absent. Usage estimates change again at the layout level when wire lengths are also included.

Another factor that makes it difficult to define such an attribute is the different levels of abstraction that distort the flow of information across the levels. This makes it difficult to unify the costs of the logic gates and the wiring together. Logic cost is estimated by attributes such as gate count, literal count, and cell area estimates, while wiring cost is provided by attributes such as wiring area, net length, number of nets, *etc.*. These costs span across different levels of abstraction (gate and layout) making it difficult to unify them.

The problems get more complicated for random logic than for regular logic. This is because the design-space in regular logic (like datapath circuits) is fairly well structured and can be explored in a methodical fashion. Circuit usage over various inputs is also quite regular. Hence, asymptotic area-time bounds can be derived for such circuits, *e.g.* adders, shifters, and multipliers. This helps simplify the task of modeling circuit usage and wiring. On the other hand the design space for random logic is very discontinuous and lacks structure. This makes it difficult to talk about dynamic usage or wiring complexity without tracking the logic and the wires together.

Finally there is also the fundamental problem that capturing dynamic usage of a circuit over all input combinations requires an exponential number of combinations to exercise the circuit.

In this dissertation we propose a new circuit attribute called *spatial entropy* that is capable of capturing the dynamic data movement and usage in a circuit along with the static circuit structure. In the next section we provide some background information and then state our thesis and describe its scope along with our goals and assumptions.

1.3 Thesis

The concept of spatial entropy is not new. It was first introduced by Carver Mead [MC79] to capture the communication activity in any physical computing system. He distinguishes between *logical entropy* and *spatial entropy* in the following way: while logical entropy is a measure of the effort needed to transform data from one form to another (computation), spatial entropy is a measure of the effort needed to transmit data from one place to another (communications).

Our thesis is that "*spatial entropy can be quantitatively defined and characterized as a dynamic circuit attribute, and it can be applied to CAD problems that have relied on static-attribute based solutions.*".

We defend this statement in three phases - definition, analysis, and application. In the first phase, our goal is to introduce the spatial entropy concept to the circuit domain. We define spatial entropy intuitively as the *dynamic communication effort* in a circuit. This intuitive definition is then followed by a quantitative definition of spatial entropy as a circuit attribute computable at every node in a circuit. This is then used to compute spatial entropy values over all nodes in the circuit. We describe its relation to a physical circuit attribute - switching energy. The objective here is to be able to compute and compare spatial entropy values for different circuits and obtain measures of their switching energies. This will help answer questions like: what does it mean for one circuit to have greater spatial entropy than another? What is the reason that a given circuit has high or low spatial entropy?

In the second phase we propose to analyze and further characterize the spatial entropy attribute. Our goal here is to study the problems faced in computing this attribute so that we can answer questions such as: How is spatial entropy best computed? How accurate is the computed value? What are the factors that affect this and how can they be controlled?

As a further characterization we plan to show that the definition of spatial entropy at a node can be extended by describing it as an accumulation of spatial entropy contributions from primary inputs affecting that node. We define this quantitatively so that it can be computed efficiently. This will help compare and contrast the communication effort of a pair of nodes at a finer granularity - in terms of the contributions of the primary inputs to the two nodes.

Finally we illustrate the usefulness of spatial entropy as an attribute that can capture function behavior and as an attribute that provides a quantified measure of the switching energy in a circuit. We compare the spatial entropy based approach with approaches that use static attribute based solutions to capture circuit behavior. Since spatial entropy models dynamic usage we hope to show that it captures a characteristic of a function that is invariant over its different implementations. This way spatial entropy will be able to track function properties through implementations. Our objective is to show that we can compute the spatial entropies of the implementations of two different functions and then use these values to contrast the two functions themselves.

Along with the effectiveness of spatial entropy as an attribute in CAD we also intend to study empirically the effects of the various approximations on spatial entropy computation and their influence on the quality of the solution. We would like to be able to determine whether spatial entropy is indeed a useful circuit attribute for solving CAD problems. What kind of problems is it best suited for and how does the accuracy of its calculation affect the quality of the solution?

The scope of circuit descriptions for this thesis is limited to combinational circuits. Circuit descriptions at the boolean function level and the gate level are acceptable. We assume a circuit model made up of the standard logic gate primitives: `and, or, not, xor, nand`, and `nor`. Transistors and lower-level structures are not treated. This thesis does not propose spatial entropy as a universal solution to CAD problems, at these levels of abstraction. What it does say is that there are several occasions where algorithms can benefit from information about dynamic usage in the function along with the static structure in the implementation. In such cases spatial entropy would be a useful attribute, and CAD tools that have traditionally used purely static attributes could benefit from this additional information on dynamic usage.

The next section discusses the major contributions of the dissertation, along with implications of this research.

1.4 Research Contributions

We briefly summarize the major contributions of this dissertation.

1. We have introduced the concept of spatial entropy to the VLSI domain as a circuit attribute.

2. We have defined it and characterized it as a measure of the switching energy (in CMOS implementations). An effective approach to compute it on VLSI circuits at different levels of abstractions is described.

3. We have illustrated its applicability in CAD.

4. We show that the spatial entropy of an implementation captures function behavior by modeling the dynamic communication effort needed to compute the function.

We have introduced spatial entropy as a circuit attribute that is capable of unifying the structure and behavior in a circuit. An intuitive definition has been refined to formulate a quantitative definition of spatial entropy as the *information-distance* product in the circuit. Information captures the dynamic switching of logic states at a node in the circuit. The distance that information has to travel is captured by the switching capacitance at the node. With this information-distance model spatial entropy measures the switching energy in a circuit implementation. A circuit with minimum spatial entropy is a minimum energy implementation that minimizes the dynamic power consumption and the delay over all the nodes in the circuit.

We have proposed a simple, efficient algorithm to compute spatial entropy for primitive gates and for an entire circuit. But it is difficult to compute spatial entropy accurately for a given implementation. Hence approximations that affect this accuracy have been identified and their implications were studied in detail, at different levels of abstraction. These effects were also studied empirically in the context of an application of spatial entropy, where we used spatial entropy to generate variable orders for binary decision diagrams (BDDs). We demonstrate that the spatial entropy based approach can generate variable orders competitive with existing static attribute based approaches, for the smaller benchmark circuits. But when the circuits get larger the effect of approximations made in computing the attribute become more significant leading to poor sizes. They have been outlined in detail with empirical results.

Simultaneously, we also explored a theoretical basis to connect spatial entropy and variable ordering. The BDD of a function depends on the characteristics of its minterms. If the spatial entropy of an implementation is indeed capable of generating good variable orders for the BDD of the underlying function then it must capture some characteristic of the function. But we observed that different implementations of the same function could differ significantly in their spatial entropy values due to different degrees of logic minimization in each implementation, *i.e.* minimality had a significant influence on spatial entropy.

So this posed the question: what is the best implementation to capture this function behavior and generate spatial entropy based variable orders and why? In a physical implementation spatial entropy is the switching energy expended by the circuit while computing the function. Over all implementations of a given function, the one with minimum spatial entropy has minimum switching energy and describes the minimum communication effort needed to compute the function. This minimum spatial entropy implementation is like a signature for the function since it provides a lower bound over all implementations of that function. But generating an implementation with minimum spatial entropy is difficult since it requires minimizing logic gates and wires, which the existing logic synthesis tools are not capable of doing. As an approximation to a minimal spatial entropy implementation we use a minimal gate count implementation. In order to study how spatial entropy on a minimal gate count implementation captures function behavior, we started by defining minimality and spatial entropy computation in cube space. This work also led to the second application of spatial entropy, its use as a gate-count complexity measure for boolean functions.

We first proposed an entropy based definition of the gate-count complexity of boolean functions called *information content*. Since the *complexity* of a function is a measure that provides a lower bound on some physical attribute over all implementations of that function this captures a fundamental characteristic of function behavior. We also show that the information content, which is defined over a minimal Karnaugh decomposition of the function, captures minimality of two-level function representations. This is because the minimal Karnaugh decomposition is equivalent to a minimized prime and irredundant two-level representation of the function. Minimality for multi-level function representations was then defined using a notation called decision tree to relate a function to its implementation. An implementation is constructed bottom up from a decision tree, and a minimal implementation corresponds to a minimal decision tree. Spatial entropy in cube space was then defined as the incremental contribution to the information content, over all nodes in an implementation derived from the decision tree. This definition of spatial entropy captures the communication between the minterms in the implementation.

We then empirically verified that spatial entropy of a minimal implementation does track function behavior in the form of the information content of the function. While it is computationally intractable to compute the cube space definition of spatial entropy, we showed that the gate-level spatial entropy computation procedure can be used as an approximation to it. We showed statistically that there is a strong correlation between the spatial entropy of a minimized implementation (as computed by the gate-level procedure) and the information content of the function. Our definition of cube space spatial entropy and information content was only defined for single-output completely specified functions. So we handled multi-output functions in our experiments by treating each of them as individual single-output functions.

The information content of the function is defined over a minimal Karnaugh decomposition and is thus an indicator of minimality in two-level functions. We showed that the information content also estimates the gate-count complexity of a multi-level implementation of the function, in terms of the logic gates required to implement it. An empirical correlation between the information content and the gate count (in a multi-level implementation) is shown for several single output functions.

The high degree of correlation between spatial entropy and information content shows us that spatial entropy can be used to measure the gate count of a multi-level implementation of the function. We also found a strong correlation between spatial entropy and gate count for our experimental data set. There is an explanation for this. Given that we limit our scope to only combinational circuits, the switching energy in these circuits is a function of the switching of states at the nodes and the delays at the nodes. In most combinational circuits this switching energy is equivalent to the circuit area since almost the entire circuit is switching as the circuit computes dynamically; since spatial entropy measures switching energy we find the high correlation between spatial entropy and circuit area. This correlation would probably be absent in sequential circuits where the switching energy is not a direct function of circuit area, or in other combinational circuits where only a fraction of the area is switched.

Spatial entropy is a unique attribute, since it is capable of capturing the structure in an implementation and the behavior of the function being implemented. One can use the structural aspect of spatial entropy to discriminate between two implementations of the *same* function, where an implementation with lesser spatial entropy has lesser switching energy. Since a minimum spatial entropy implementation acts as a signature of the underlying function, one can compute this for two *different* functions and compare the minimum switching energies of these two functions. The fact that spatial entropy can measure information content, a characteristic of the function that is invariant over different implementations, further lends credence to our hypothesis that spatial entropy, if computed accurately, can generate good variable orders for BDDs, on even the larger circuits. This is because BDDs too are a characteristic of the function definitions and not of its implementations.

1.5 Dissertation Outline

In the next chapter we describe work that is related to our research. This is discussed along three directions. We first describe traditional static and dynamic attributes in CAD and highlight the circuit properties that they capture. Then we discuss approaches that estimate complexity in terms of information flow or wiring area when data has to be communicated between various places in a circuit. Finally we discuss entropy based attributes in CAD.

In Chapter 3 we introduce the spatial entropy concept to the circuit domain. It is first defined intuitively, and then quantitatively as a dynamic circuit attribute that measures the switching energy in a circuit. We show that spatial entropy can be computed using existing circuit attributes, and we propose an algorithm to compute spatial entropy for primitive gates and for an entire circuit. In the remainder of this chapter we characterize the attribute thoroughly, explain why it is difficult to compute it accurately, and outline the approximations that affect its accuracy.

The subsequent chapters illustrate applications of spatial entropy. In Chapter 4, we begin by introducing the variable ordering problem for binary decision diagrams (BDDs). After a short survey of related research we describe our approach of using spatial entropy vectors to generate variable orders. The rest of the chapter describes our experiment to study spatial entropy based variable ordering empirically. The objectives are two-fold. First we study the effectiveness of spatial entropy in generating variable orders, and compare it with several existing approaches. Then we study the effect of the various factors and approximations and how they affect the accuracy of spatial entropy.

Chapter 5 draws on the conclusions in Chapter 4. It explains minimality and spatial entropy in cube space. It also illustrates empirically how spatial entropy can measure the area-complexity in boolean functions in terms of the multi-level gate implementations of these functions.

Finally we summarize conclusions and future directions in Chapter 6.

Chapter 2

Related Work

The spatial entropy attribute has three characteristics - its ability to unify static circuit structure and dynamic usage, its ability to capture function complexity, and its information theoretic basis. We discuss related work in this chapter by developing it along these three directions.

First we briefly describe traditional attributes that capture either static structure or dynamic usage and discuss how they have been used to solve CAD problems. Our purpose here is to illustrate circuit properties that are captured by these attributes in order to contrast them later to the properties captured by the spatial entropy attribute. In the second part of this chapter we discuss approaches that estimate the complexity involved when data has to communicate from various points in the circuit. The objective here is to illustrate how these approaches have been used to derive indicators such as information flow, wiring area *etc.* to estimate this complexity. This will help us contrast them with the use of spatial entropy to capture wiring complexity and boolean function complexity which we discuss in later chapters. Finally we describe other entropy based attributes that rely on an information theoretic definition (like spatial entropy), and outline their application to problems in CAD.

2.1 Static and Dynamic Attributes

We begin by briefly contrasting static and dynamic circuit attributes. *Static* attributes are computed by examining the circuit structure or topology. In contrast *dynamic* attributes are computed by exercising a circuit representation such that the circuit computes over a distribution of input values. While static attributes capture circuit connectivity or structure, dynamic attributes capture usage of various parts of the circuit over a distribution of input data values. Static attributes can be usually computed quickly; dynamic attributes on the other hand can be expensive to compute. We now consider a subset of CAD problems and describe in turn, the role static and dynamic attributes play in solving them. We also highlight the circuit properties captured by these attributes.

Static attributes are typically used when information about the circuit connectivity or structure is essential to solving the problem. Consider the problem of timing analysis. The objective here is to identify the critical (or longest sensitizable) path in the circuit to estimate the worst-case delay. Since the actual delay along a path in the circuit depends on the propagation delay and the number of gates in the path, classical static timing analysis [Ous83] uses attributes like static gate depth and fanout along with propagation delay to arrive at an estimate of the worst-case delay. But this is not always a true estimate. This is because some paths in the circuit are generated with sub-paths that require input combinations that can never occur. These logically incompatible paths are called *false paths* and accumulating gate delays along such paths would be erroneous. On the other hand an exact estimate of the critical path delay in the circuit would require simulating the circuit over all possible inputs to examine the effects of all input combinations. Since this is expensive the static attributes are used to obtain a quick approximation of the actual delay in terms of gate connectivity and gate depth. A lot of the work in timing analysis now concentrates on eliminating false paths [BI86, MK89, DYG89, BMCM90, PCD89].

Logic minimization and logic synthesis is another area of CAD where solutions are guided by static circuit structure information. One of the objectives here is to map a two-level boolean function representation into a set of gates (from some library) that occupies minimum area [BHMSV84, BM82, BRSVW87, LKB87, LBK88]. Ideally one would like to achieve this objective of minimum area with respect to not just the gates in the circuit but also the wiring between them [Sau92]. But since the notion of wires is absent at the logic level of abstraction the objective is restricted to minimum gate area. The static circuit attribute of literal count becomes an approximation to the gate area. Hence algorithms in logic minimization and logic synthesis use minimum literal count as the area criterion. Literal count is not only used as a measure of performance of the algorithm, but it is also used to guide the algorithm during the minimization process. It assists in searching and selecting candidate factors for substitution while decomposing and factoring the boolean function.

Finally let us look at static attributes at the physical layout level. CAD tools in the area of placement and routing are faced with the goal of placing circuit components on a 2-dimensional plane and connecting nets between them such that area and delay of the resulting circuit layout is minimized [LP88]. Again static structure and connectivity information play an important role here. They are captured by attributes like wire length, component dimensions, user-defined or pre-routed obstructions *etc.*. These yield minimization criteria such as minimum total net length, minimum circuit area, minimum number of vias, routeability for all nets, prior routing of selected (power and ground) nets *etc.*. The algorithms are guided towards desired solutions by a suitably weighted version of these criteria.

Dynamic attributes are typically used when circuit behavior over all possible inputs needs to be captured to solve a problem. Simulation is one such instance. Regardless of the kind of simulation (function, gate, switch, circuit, fault), it is complete only when the circuit behavior for all possible inputs is studied. In such a case the input vectors that represent various input combinations are dynamic attributes that capture circuit usage as they propagate through the various nodes.

Testability analysis [BPH84, SDB84, JA84, LBdGG87] which is sometimes viewed as an alternative to fault simulation, is another problem that relies on a dynamic attribute. The objective here is to project the cost of testing by predicting the number of random test patterns needed to achieve high fault coverage. High fault coverage is usually defined by predicting a large percentage $(95 - 98\%)$ of faults in the circuit with a high probability $(0.98 - 0.99)$. To solve this problem the probability of detecting a fault is expressed as a function of two probability based attributes: *observability* and *controllability*. Controllability is computed by assigning signal probabilities at the inputs to the circuit and propagating them forward through all the nodes in the circuit. Observability is computed similarly except that it is computed backwards from the output. These probability based attributes are dynamic because they capture circuit usage over different input combinations. In this particular instance they are being used to detect a given list of stuck-at faults (at nodes in the circuit) by looking at the value of the node over all possible input combinations. We now look at research in communication complexity.

2.2 Communication Complexity Prediction

With VLSI design being performed in sub-micron technology with smaller feature sizes there is a realization that the area-time and power performance of a circuit will be dominated less by the logic or the gates in the circuit and more by the *wires and the communication* between the logic. This has resulted in efforts to estimate communication complexity by the wiring area or information flow in a circuit.

In 1979 Thompson introduced a model [Tho79] for computing lower bounds on the complexity of VLSI implementations of functions. The complexity of the computation was measured in terms of bounds (AT^2, $A^{\alpha}T^{\beta}$) on chip area and computation time. The model highlighted the difficulty of communicating information across the surface of a chip. Given any partition of the input set into two equal halves on two disjoint regions of the chip, these area-time bounds defined lower bounds on the communication complexity of the function.

The notion of communication complexity across a partition was introduced by Yao to provide lower bounds on the worst case information complexity of many functions [Yao79]. Yao's model assumed a particular partitioning of the input set into two equal halves (as against the VLSI complexity model that assumes any partition). This captured a local information flow across the partition. The communication complexity was defined as the number of bits of information that needed to cross the partition to correctly compute the function. These bits of information were computed by a two-way protocol that dictated how the bits would be exchanged by the two input halves. The maximum number of bits exchanged over all input values for computing the function f was defined as the *communication complexity* $c(p)$ for a given protocol p. The worst-case complexity was then defined as the *minimum two-way communication complexity* over all protocols for that partition.

More recently Hwang, Owens and Irwin [HOI89, HOI92] have used communication complexity for multi-level logic synthesis. They have also provided ways to compute the communication complexity whose bounds were estimated in [Yao79]. In their approach the "decomposition and factoring" operations in multi-level logic minimization are performed by partitioning the boolean function f into three functions f_t, f_l, f_r such that

$$f(X) = f_t(f_l(X_l), f_r(X_r))$$

where X_l and X_r denote a disjoint partition of the input set X ($X_l \cup X_r = X$). The partition is generated using heuristic partitioning techniques that try to minimize the communication complexity or the number of interconnections between the functions (f_t, f_l) and (f_t, f_r). The communication complexity for a given partition, which acts as a cost function for their partitioning algorithm, is computed by computing the number of compatible classes [Ris82] of a given function. Two approaches were illustrated. The first approach uses a communication matrix while the second approach, which is more efficient, uses cubes and cube overlaps [BHMSV84].

Another partially related approach is discussed in [ASSP90]. Here wiring complexity in a synthesized circuit is minimized by controlling input dependency with lexicographic expressions of a boolean function. A lexicographic expression of a boolean function is a sum of product terms in which the input literals (that every product term depends upon) conform to an ordering called the reference order. This ordering is used to extract a set of lexicographically compatible kernels [BM82]. Kernel filtering computes the intersection of the extracted kernels to find shared parts amongst the functions. By tightly controlling this filtering process the logic cones can be prevented from intersecting with each other. The objective is to reduce the wiring between logic cones that manifests itself as wires in the layout.

We now discuss the application of entropy based attributes in CAD.

2.3 Entropy Based Attributes

2.3.1 Background

We begin with some background on the concept of entropy. There are two popular definitions of entropy. *Information theory* [SW49] defines it as the measure of information content in a system. *Thermodynamic* [Sea53] defines it as the thermodynamic probability of the internal particles of a system while holding the external properties constant. We consider each definition in turn and show how they both view entropy as - " the measure of disorder in a system".

Consider a system with N possible output events. In information theory this is usually a communication system where the N events are messages to be communicated to a receiver. Suppose each event i has a certain probability of occurrence p_i with respect to its inputs. Then the information-theoretic definition of entropy is the measure of information produced when one event is chosen from this set N [SW49]. It is defined as

$$\sum_{i \in N} p_i \log \frac{1}{p_i}$$

When all the events are equally likely, *i.e.* $p_i = \frac{1}{N}$, the expression reduces to $\log N$.

As an example consider a system where the messages are represented by a bit string of length n. The set of output events N is the set of all possible messages that can be represented by the n-bit string. If all bit combinations are assumed likely then the n-bit string can represent at most $N = 2^n$ different messages. On the other hand suppose we insisted that only a single bit combination, that of all 1s in the string, is possible. Then only one message ($N = 1$) will be possible. In the former case the information content (entropy) of the bit-string is $\log(2^n) = n$, while in the latter case it is $\log(1) = 0$. Thus if greater number of bit combinations (or messages) are possible this implies greater disorder in the system. This in turn implies greater entropy or information content.

In the same vein a bit-string that is twice as long (of length $2n$) will have an information content (or entropy) equal to $2n$. This conforms to our intuitive notion that a message that is twice as long should be able to contain twice as much information.

In thermodynamics, entropy is defined as being proportional to the logarithm of the number of ways of arranging the particles in a system while maintaining external conditions constant. As an intuitive example [MC79], consider a system with two containers holding a total of 10 red and 10 blue molecules. If we do not distinguish between containers there is only one way in which the molecules can be arranged so that all the 10 blue ones are in one container, and all the 10 red ones in the other. On the other hand there are a large number of ways of arranging 5 of each color in each container. The second arrangement of molecules has much more disorder than the first, and therefore has much more entropy than the first. The actual definition [Sea53] of this entropy S is in terms of the *thermodynamic probability* of the internal particles in the system. It is defined as $S = k \log W$, where k is Boltzman's constant, and W is the thermodynamic probability.

Thus in both information theory and thermodynamics, entropy captures the measure of disorder in a system. The information-theoretic definition of entropy has found applications in a few areas of CAD. We begin by reviewing work in these areas.

2.3.2 The Entropy function in CAD

One of the first applications of information theory was to use the information-theoretic definition of entropy to predict boolean function complexity. The relationship between function complexity and entropy was first conjectured by Cook and Flynn [CF73]. The complexity of a boolean function is expressed by the cost of implementing the function as a combinational network. Cook and Flynn demonstrated empirically that the average cost behavior of a single output combinational network could be modeled by a formula that captured the entropy of the boolean function implemented by the network. It was defined as

$$H(f) = \frac{u}{2^n} \log_2 \frac{2^n}{u} + \frac{2^n - u}{2^n} \log_2 \frac{2^n}{2^n - u}$$

where n is the number of input variables and u is the number of ON-terms in the cube space. Subsequently Hellerman [Hel72] proposed a definition of computational work based on the entropy function. Suppose a function $f : X \rightarrow Y$ performed some computation over a domain of inputs X and a range of outputs $Y = \{y_1, \ldots, y_n\}$. Then for $X_i \subset X$ and $X_i = f^{-1}(y_i)$, the work done by the function was expressed as $\sum_{i=1}^{n} \mid X_i \mid \log \frac{|X|}{|X_i|}$ where $\mid X \mid$ denotes the number of elements in the set.

The relationship between the works of Hellerman and Cook & Flynn was later observed by Mase[Mas78]. He showed that the complexity of a boolean function can be expressed in terms of an entropy-based definition of (computational) work performed by the combinational network.

In 1977 Pippenger further refined this entropy definition to handle don't cares in the function [Pip77]. More recently in [CA90], the entropy formulation was generalized to multi-output functions, both completely specified functions and partially specified ones. They also showed statistically that using the literal count as a measure of circuit area a linear relationship can be observed between entropy and average number of literals in a multi-level implementation.

Information theory has also found application as a testability measure. This was first proposed by Dussault [Dus78]. He presented observability and controllability measures based on information theory for gate level circuits. In [TA89], Thearling and Abraham extended this idea to estimate testability at the function level. They use a measure called the information transfer coefficient (ITC) [Koo87] to enable relative testability measures to be computed as against the absolute measures computed by Dussault [Dus78]. Agrawal [Agr81] has also applied information theory to test pattern generation. He shows that by choosing test patterns that maximize the information at the output the probability of fault detection can be maximized.

2.3.3 Entropy as a basis for Computation

In [MC79], Carver Mead proposed the idea of computation based on entropy. This has formed the basis for our definition of spatial entropy in the circuit domain that we discuss in Chapter 3.

He begins by suggesting that computation can be viewed as a process that reduces the disorder (or entropy) in the solution space while arriving at a result. Every computation finds an answer by making decisions on a solution space. With each decision the usually huge initial solution space is cut down to some fraction of its former size. The number of decisions required to specify one correct answer in the solution space is the *entropy of the computation*, defined as $\log \frac{|Total\_Solns|}{|Answer|}$. This definition is analogous to the information-theoretic definition of entropy where the solution space is all possible messages with a given length and bit-string format. The correct answer is one such message and the entropy is the number of bits required to specify this correct answer.

The description of entropy outlined above captures algorithmic computation and Mead terms this as *logical entropy*. This is because it depends on the logical operations required to perform the computation. The objective of a computation is to reduce the logical entropy of the data to zero. The study of algorithm complexity analysis is the study of these logical operations modeled by logical entropy. Mead also proposes another form of entropy, called *spatial entropy*, that is usually seen in situations when the computation has to be mapped onto a domain where data travels over a physical distance. The contrast between the two forms of entropy is best captured by the following quote from [MC79] :

> *" In any physical system, the logical entropy treated by classical complexity theory is only part of the story. There is also a spatial entropy associated with a computation. Spatial entropy may be thought of as a measure of data being in the wrong place, just as logical entropy is a measure of data being in the wrong form. Data communications are used to remove spatial entropy, just as logical operations are used to remove logical entropy."*

Entropy is the measure of disorder in a system. So spatial entropy is the measure of spatial disorder in a system. This spatial disorder (or spatial entropy) in a system captures a form of spatial distance between the inputs and the outputs in a system. The spatial entropy S of a system quantifies the spatial effort needed to bring the data at the input location to the output location. When a system computes, the data communications in the system are carrying data from the input to the output. This reduces the spatial distance between them, or removes spatial entropy in the system.

One scenario that illustrates spatial entropy is a communication network. Here messages or communication events are transmitted over communication pathways that remove spatial entropy by routing data between various spatially distributed source and destination sites. Another example is circuit computation. Here the input data travels through the wires in the circuit. These wires remove spatial entropy in the circuit by carrying the input data to the outputs. It is this latter model that is of interest to us. In the next chapter we characterize the spatial entropy concept in the circuit domain and define a quantitative measure that relates it to the switching energy in the circuit. We describe an algorithm that computes spatial entropy for a gate-level circuit implementation, and study the factors affecting the accuracy of this computation.

Chapter 3

Spatial Entropy - A Circuit Attribute

In the previous chapter we introduced the concept of spatial entropy. In this chapter we illustrate how spatial entropy can be characterized as a dynamic attribute in the circuit domain. We define spatial entropy quantitatively, and show how it can be computed on primitive gates and over an entire circuit. We also explain how it measures the *switching energy* in a physical implementation. Computing the attribute accurately is unfortunately a difficult task, and there are different factors that affect its accuracy. In the latter part of this chapter we discuss these factors in detail and explain how they result in various approximations while computing the attribute.

We begin with an intuitive notion of circuit spatial entropy followed by a quantitative definition. In Section 3.2 the technique to compute spatial entropy for gate level primitives is described. This becomes the basis for an algorithm to compute circuit spatial entropy, which we outline in Section 3.3. Section 3.4 further characterizes the attribute by introducing spatial entropy vectors. In Section 3.5 we talk about the difficulties in computing this attribute accurately, and how these factors force approximations to be made during spatial entropy computation.

3.1 Introduction and Definitions

Spatial entropy can be intuitively defined as the *communication effort* required to compute the circuit function. In a circuit both the logic gates and the wires contribute effort towards computing the circuit function. The gates compute boolean values or bits and the wires transmit these bits. Spatial entropy models the dynamic communication taking place in the circuit versus the static communication modeled by the wires. Over all the input combinations, it tries to capture the distribution of bits at the gate outputs, and the communication of these bits from one gate output to another. While the wires determine how far the bits have to travel, the gate types determine the distribution of boolean values that these bits take at the various nodes in the circuit. Together they determine the dynamic communication effort in the circuit. We use an information-theoretic definition to capture this effort through the circuit.

We start with a description of our circuit model. A circuit is represented as a directed weighted graph $G = \langle V, E, L \rangle$. Each primary input, primary output, and logic gate in the circuit is represented by a node $v \in V$ in the graph. An edge $(v, w) \in E$ represents a wire in the circuit. Each such edge has a length attribute $l_{(v,w)} \in L$ that is the length of the wire. $L : E \rightarrow \mathcal{R}$, where $\mathcal{R}$ is the set of real numbers. The direction of the edge, from v to w, represents the direction in which the bits travel in the wire. v is the *source node* and w is the *destination node* for the edge. The directed edges from v to other nodes are called *fanout* edges. v is the source node of these edges and the bits leave from v along these edges to go to destination fanout nodes. The directed edges that come into node v are called *fanin* edges. v is the destination node of these edges and the bits enter v along these edges from source fanin nodes. The primary input nodes have no fanin edges and the primary output nodes have no fanout edges. The number of fanin edges of a gate node is equal to the number of inputs to the gate. A *path* in the graph is a sequence of vertices from a source node to a destination node. The *support set* of a node v is the set of all primary inputs from which there is a path to node v. Figure 3.1 illustrates a simple circuit and its corresponding directed graph model. (The edge lengths are not shown).

We now refine the idea of dynamic communication effort by quantifying it with circuit attributes. To capture the distribution of boolean values at the node, we use the classical entropy function $H(\{p_i\})$ from information theory

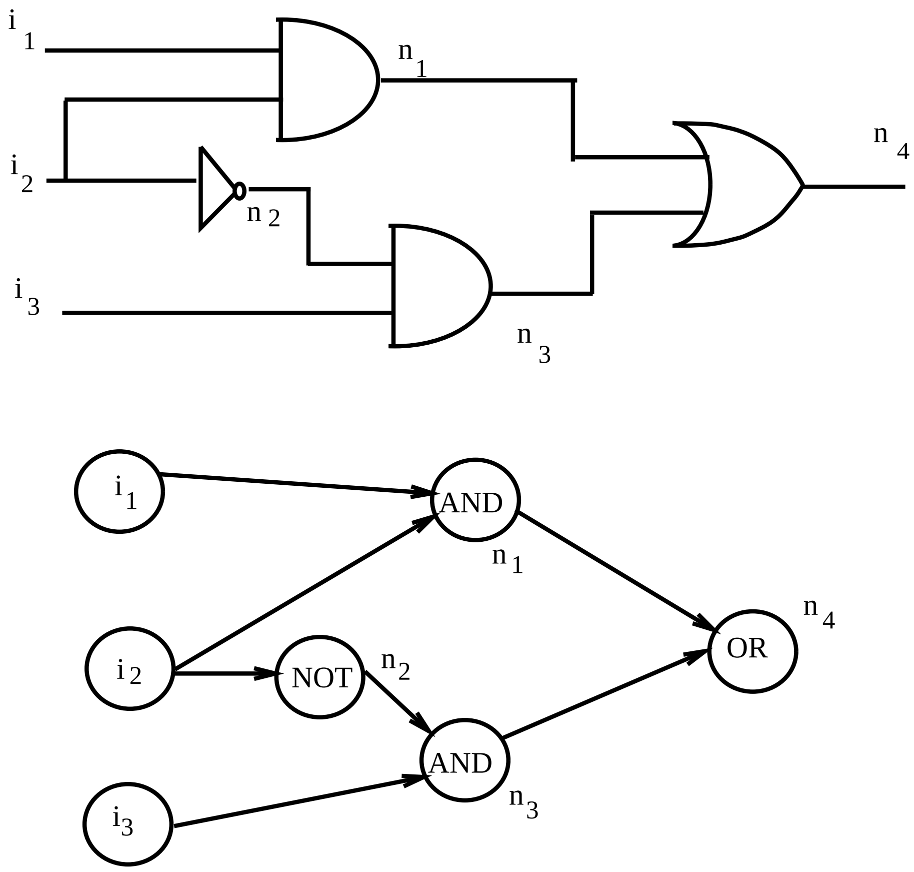


Figure 3.1: An Example Circuit and its Digraph Model

[SW49] $H(\{p_i\}) = \sum_{i=1}^{N} p_i \log(\frac{1}{p_i})$. N represents the number of possible values (or events) in a given system. Since there are only two values possible in a digital circuit we can model the distribution of boolean values computed at a node w by the binary entropy function $H(\{p_w^1, p_w^0\})$ at the node,

$$H(\{p_w^1, p_w^0\}) = p_w^0 \log(\frac{1}{p_w^0}) + p_w^1 \log(\frac{1}{p_w^1})$$

where p_w^1 is the 1-probability of a node w, and $p_w^0 = (1 - p_w^1)$ is the 0-probability of the node. The probability of a node gives the distribution of 1 and 0 values computed by the node. This binary entropy function $H(\{p_w^1, p_w^0\})$ denotes the *information* computed at the node over this probability distribution. We shall denote this function as H_w.

The function H_w quantifies the dynamic communication effort contributed by the node. Consider its plot shown in Figure 3.2. As we look at the extremities of this function we observe that the information H_w is a minimum when the 1-probability or 0-probability of a node is 1 or 0, while it is a maximum when $p_w^0 = p_w^1 = 0.5$. Assuming all inputs are equally likely, a 1-probability of 0.5 at a node w implies that for half the input values the node will have a value of 1 and for the other half the node will have a value of 0. This means that a gate modeled by this node will have to expend more effort in order to distinguish the ON-terms from the OFF-terms in order to compute the output value for a given input(s). On the other hand, a 1-probability of 1.0 implies that the node will have a value of 1 for all input values. This means that the gate does not have to expend any effort to compute the output since, given an input value, the gate automatically knows that it belongs to the ON-set. Thus the function H_w plotted in Figure 3.2 captures the effort expended by the gate in computing the boolean value at its output.

The distance traveled by the boolean values computed at a node is quantified by the length of a fanout edge from that node. The edge lengths in the graph quantify the wire lengths in the circuit in a straightforward manner. Long edge lengths imply long wires indicating more effort is expended in carrying the information from the gate. We now define the spatial entropy S for a circuit as follows.

Definition 1 *The spatial entropy S at the output node of a single output*

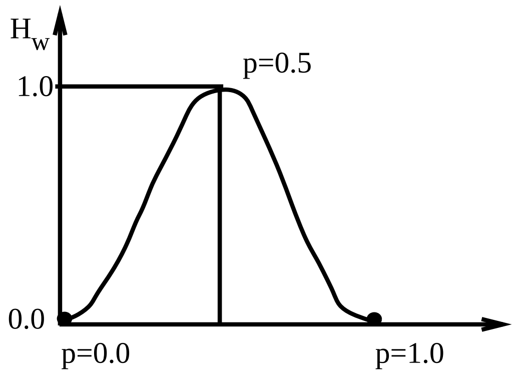


Figure 3.2: The Entropy Function H_w

circuit is the information-distance product over all the nodes in the circuit.

$$S = \sum_{v \in V} \sum_{w \in V} H_v * l_{(v,w)}$$

H_v is the information computed at the node v over its input probability distribution, and $l_{(v,w)}$ is the length of the fanout edge $(v, w) \in E$, from node v to node w.

The spatial entropy S of a multi-output circuit is expressed as $S = \sum_{i=1}^{m} S_{o_i}$, where m is the number of outputs and S_{o_i} is the spatial entropy at output o_i.

At each node v, the spatial entropy is computed by multiplying the information computed at v by the distance it has to travel along all the fanout edges from v. With this definition the intuitive notion of communication effort is captured by the total information flow in the circuit. The nodes compute the information while the edges communicate this information.

How does spatial entropy capture dynamic circuit usage? Spatial entropy is a dynamic circuit attribute in the sense that its value, computed from the entropy function H, is a function of the probability distribution at the primary inputs of the circuit. Thus H captures circuit usage defined by this probability distribution. Since it is symmetric it gives an accurate model of circuit usage capturing the propagation of both 1 and 0 probabilities. A circuit that propagates mostly 0s in its internal nodes could still be performing useful computation; but this would not be captured well with only 1-probabilities.

This definition of spatial entropy as the information-distance product is a good model to measure the *switching energy* in a physical circuit implementation. The switching energy of a device is defined as the dynamic power consumed by the device multiplied by the delay associated with the device. It gives a measure of the dynamic work performed by the device. In CMOS implementations the switching power P_{sw} depends on the frequency at which the circuit runs, and the number of times the switching capacitance C_{sw} gets charged and discharged while the nodes switch logic states ($1 \rightarrow 0$,$0 \rightarrow 1$). The delay is the time taken to charge or discharge the capacitance associated with a node, and it is a function of the resistance and capacitance of the device and the wiring associated with it.

In a physical implementation, the *information* at a node (computed by the entropy function H_w) captures the dynamic switching of logic states at the node. A node with high information has equal likelihood of acquiring a value of 1 or 0 ($p = 0.5$). This also implies that such a node is likely to undergo more switching of states ($1 \rightarrow 0$,$0 \rightarrow 1$) during dynamic computation and thus expend more switching energy. The *distance* information has to travel is a measure of the switching capacitance that needs to be charged and discharged, since this determines the delay incurred before the node changes state. Long edge lengths imply high switching capacitances causing longer delays for a node to change state and longer delays for information to travel to the next node. This definition of spatial entropy thus provides a quantitative measure for the switching energy in a physical implementation.

In the next section we describe a way to estimate spatial entropy in a circuit. It is difficult to compute spatial entropy accurately, and we only illustrate how to compute approximations to the spatial entropy definition in Definition 1. We begin by describing this process for individual primitive gates that then evolves into a procedure to compute spatial entropy for an entire circuit.

3.2 Spatial Entropy Computation

We restrict our treatment of spatial entropy computation to the domain of combinational circuits at the gate level. Spatial entropy of sequential circuits can be computed by expressing their next state and output functions as blocks of combinational logic. In this case the spatial entropy computation will capture the information flow through the circuit for only a single clock cycle. This would have to be repeated over successive clock cycles with new probability values to capture the entire computation of the sequential circuit. The circuits are multi-level implementations and technology mapping may or may not have taken place.

We begin by describing spatial entropy computation for implementations of single output boolean functions. Subsequently we extend our procedure to compute the spatial entropy of multi-output implementations. In order to compute spatial entropy for circuits at the gate level, the spatial entropy for primitive logic gates needs to be defined. Spatial entropy is a function of the information H_w at a node, and the fanout edge length at the node. But edge lengths are absent at the gate level. We currently assume *unit edge length* in our graph, *i.e.* *unit wire length* in our circuit, and compute an approximation to the spatial entropy of the circuit. Later in this chapter we describe ways to refine this approximation by obtaining estimates of the edge length.

The *local spatial entropy* at a gate node $g \in V$ is defined as:

$$\delta S_g = \sum_{g' \in V} H_g * l_{(g,g')}$$

where H_g is the information computed at the gate node g, and $l_{(g,g')}$ is the length of the fanout edge from node g to node g'. Since we have assumed $l_{(g,g')} = 1$ we can only compute an approximation $\delta S_g = H_g$. We illustrate this for some simple 2-input gates.

Consider a 2-input AND gate with 1-probabilities of p^1_x,p^1_y at its inputs x, y. The 1-probability at the output of the AND gate is $p^1_{and} = p^1_x * p^1_y$, since the only event yielding a 1 at the output is $p^1_x * p^1_y$. The local spatial entropy at the AND gates, δS_{and}, is equal to the information at the gate $H_{and} = p^0_{and} \log \frac{1}{p^0_{and}} + p^1_{and} \log \frac{1}{p^1_{and}}$.

For a 2-input XOR gate the 1-probability $p^1_{xor} = p^1_y(1 - p^1_x) + p^1_x(1 - p^1_y)$, since there can be a 1 at the output only if $x = 0$ and $y = 1$ or $y = 0$ and $x = 1$. The local spatial entropy $\delta S_{xor} = H_{xor}$, where $H_{xor} = p^0_{xor} \log \frac{1}{p^0_{xor}} + p^1_{xor} \log \frac{1}{p^1_{xor}}$.

For a 2-input OR gate the 1-probability $p^1_{or} = 1 - (1 - p^1_x)(1 - p^1_y)$. This is obtained by subtracting from 1, the probability of the event that would yield a 0 at the output. Similarly the local spatial entropy $\delta S_{or} = H_{or}$, where $H_{or} = p^0_{or} \log \frac{1}{p^0_{or}} + p^1_{or} \log \frac{1}{p^1_{or}}$.

For a NOT gate the 1-probability p^1_{not} is equal to $1 - p_x$, and the 0-probability p^0_{not} is equal to p_x. The local spatial entropy $\delta S_{not} = H_{not}$ is $p^0_{not} \log \frac{1}{p^0_{not}} + p^1_{not} \log \frac{1}{p^1_{not}}$. The local spatial entropy values for 2-input NAND, NOR and other (3,4)multi-input (AND,OR,EXOR) gates can be computed similarly.

To see what these definitions mean let us assume for the moment that the 1-probabilities at the inputs of all the above gates are 0.5, that is, 0 and 1 are equally likely at the inputs. Then the 1-probabilities (0-probabilities) at the outputs of the above gates are AND-0.25(0.75), OR-0.75(0.25), XOR-0.5(0.5), and NOT-0.5(0.5). Computing the information content H of these probability distributions we discover that the AND and OR gates will have the same amount of information. This is because both have the same distribution of 1/0 events at the output. In the AND, three input events yields a 0 and one input event yields a 1 at the output. It is exactly the reverse in the OR gate. In the case of NOT and XOR gates, assuming an input probability of 0.5, both gates have the same amount of information $H = 1$. This implies maximum information. The reason for this is that both gates have equal probabilities for output events of 1 and 0, and thus the output has as much chance of being a 1 as being a 0.

We now have rules to compute the spatial entropy for the primitive gates in our circuit. What is now needed is a way of accumulating these spatial entropy values over the various nodes in the combinational circuit. This would give us the spatial entropy at the output of the circuit. We illustrate this with a simple example first and then describe a simple algorithm for spatial entropy computation. As we indicated before, this computes an approximation to the actual spatial entropy since we assume unit edge lengths. In a physical implementation, such an approximation to spatial entropy will yield a measure of the dynamic power consumed in a circuit. It captures the rate at which the switching capacitance at each node charges and discharges as the logic states change at the nodes, without capturing the delay associated with the node.

Consider the example combinational circuit in Figure 3.3. The circuit has 4 inputs and 1 output. We assume the input values are distributed such that each input has equal probability of being a 0 or 1. So the circuit is initialized by assigning all primary inputs with a probability of 0.5. This initializes the spatial entropy of every primary input to 1.0 ($p = 0.5 \Rightarrow H = 1.0 \Rightarrow$

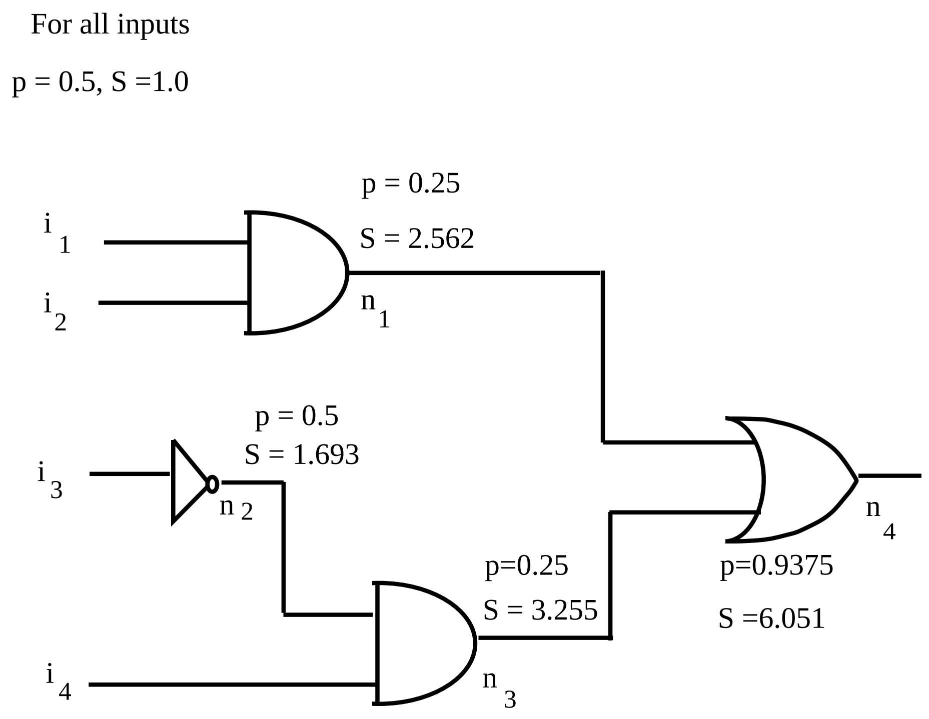


Figure 3.3: Spatial Entropy Computation for a Simple Circuit

$S = 1.0$ (for unit edge length)). The spatial entropy computation process begins with a breadth first search from the input nodes of the circuit. The probability of each node in the graph is calculated using the primitive gate rules. The spatial entropy at each gate's output node S_g is the local spatial entropy at that node plus the cumulative sum of the spatial entropy values at the fanin nodes of the gate. This is calculated as follows for a k-input gate g.

$$S_g = \sum_{i=1}^{k} S_{g_i} + \delta S_g$$

where δS_g is the local spatial entropy at the output node of the gate g and S_{g_i} is the spatial entropy value at fanin node g_i of gate g.

In the example in Figure 3.3 the spatial entropy at node n_1 is calculated as $S_{n_1} = S_{i_1} + S_{i_2} + \delta S_{n_1}$. Since i_1 and i_2 are primary inputs with probabilities of 0.5, their spatial entropies S_{i_1} and S_{i_2} are 1.0. The local spatial entropy at n_1, δS_{n_1}, is calculated using the formula for an AND gate to yield $\delta S_{n_1} = .25 * \log(\frac{1}{0.25}) + .75 * \log(\frac{1}{0.75}) = 0.562$. Similarly $S_{n_2} = S_{i_3} + \delta S_{n_2} = 1.693$. The spatial entropy at the internal node n_3 is calculated as $S_{n_3} = S_{n_2} + S_{i_3} + \delta S_{n_3}$. In this case while $S_{i_3} = 1.0$, $S_{n_2} = 1.693$. δS_{n_3} is calculated as it is calculated for n_1. Therefore, $S_{n_3} = 1.0 + 1.693 + 0.562 = 3.255$. Finally the spatial entropy at the output node n_4 is calculated to obtain a value for the total spatial entropy of the circuit. $S_{n_4} = 2.562 + 3.255 + \delta S_{n_4}$. The local spatial entropy at n_4 is calculated as $0.9375 * \log(\frac{1}{0.9375}) + .0625 * \log(\frac{1}{0.0625}) = 0.234$. This gives $S_{n_4} = 6.051$.

In the next section we describe an algorithm to compute spatial entropy for a given combinational circuit.

3.3 Algorithm

The following algorithm computes the spatial entropy for a given combinational circuit. The computed spatial entropy is an approximation to the spatial entropy in Definition 1 since we have assumed unit edge lengths in the circuit. The same algorithm can also be used to compute spatial entropy if the actual edge lengths or edge length estimates are available. We illustrate this later when discussing different estimates of the edge length.

We first describe the procedure informally and then outline the algorithm. The input is a logic gate level description of a combinational circuit modeled as a connected directed graph (V, E). The algorithm itself can fit in the framework of any event-driven system. It performs a breadth-first traversal of the graph by visiting nodes in successive levels. At the outset the level is 0, and the only nodes visited are the primary input nodes. These are initialized with their probability and spatial entropy values. The procedure then collects all nodes that are reachable from them. This makes up the nodes at the next level. A gate node is added to a level only if all its fanin nodes have been visited. The spatial entropy and probability values for these nodes (at the next level) are calculated and the nodes are merged with the existing set of visited nodes. The search then proceeds to the next level to collect the next set of gate output nodes. This continues till all the nodes in the graph have been visited (or reached). This indicates breadth-first-search is complete. At this point the spatial entropy and probability values of all the nodes have also been calculated. Shown below are some definitions and the outline of the algorithm.

`Total_NodeSet`: This is the set of all nodes (V) in the graph. It is unchanged over all levels.

`Reached_Set`: This is the set of all nodes reached till a particular level. At level 0 the `Reached_Set` is the set of primary inputs nodes (`Inputs`).

`NewAffected_Set`: This is the set of new nodes generated at each level.

In the following algorithm, `+`,`-` are used to distinguish the inputs and outputs respectively, to the individual procedures.

```
Procedure Compute_Spatial_Entropy(+Network, +Inputs)
begin
    collect_all_nodes(+Network, +Inputs, -Total_NodeSet);
    collect_input_nodes(+Network, +Inputs, -Start_NodeSet);
    Reached_Set = Start_NodeSet;
    initialize_attributes(+Reached_Set);
    Level = 0;
    while (Reached_Set ≠ Total_NodeSet) do
    begin
        Level = Level + 1;
        collect_new_affected_set(+Reached_Set, -NewAffected_Set);
        compute_attributes(+NewAffected_Set);
        Reached_Set = Reached_Set ∪ NewAffected_Set;
    end while;
end Compute_Spatial_Entropy;
```

The algorithm begins (`Level=0`) by initializing the `Reached_Set` to the set of primary input nodes `Inputs`. These nodes are assigned initial probability and spatial entropy values. Then the search proceeds to the next level (`Level=1`) in the graph. It collects all nodes that are reachable from the existing `Reached_Set`. This makes up the set of newly affected nodes `NewAffected_Set` at this level. For each node v in the `NewAffected_Set` the attributes probability (p_v), and spatial entropy (S_v) are calculated by procedure `compute_attributes`. The `NewAffected_Set` is then merged with the current `Reached_Set` to obtain an updated `Reached_Set`. This starts the next level of the breadth-first-search. This process repeats till the `Reached_Set` contains all the nodes in the graph (`Total_Node_Set`) indicating that breadth-first-search in the graph is complete.

The run time of the algorithm depends on the number of nodes n in the combinational circuit. Since the entire graph is connected, in the worst case, the breadth first search will add at least one new node for each iteration of the **while** loop. Thus there will be at most n iterations of the loop.

The *frontier* nodes $\mathcal{F}_l$ of the `Reached_Set` at some level l consist of all nodes that were added at level $l - 1$. In order to generate the `NewAffected_Set` at level l we examine the fanout nodes $F(f)$ from each frontier node $f \in \mathcal{F}_l$. Only a subset of nodes $F'(f) \subset F(f)$ are eligible to enter the `NewAffected_Set`. Nodes whose fanins are not in the `Reached_Set` are discarded from $F(f)$. For a pair of frontier nodes $f, g \in \mathcal{F}_l$, the eligible fanout sets $F'(f), F'(g)$ are not necessarily disjoint. Two nodes can fanout to the same node at the next level. Hence we need a union to compute the `NewAffected_Set`. At level l, `NewAffected_Set` $= \cup_{f \in \mathcal{F}_l} F'(f)$. In the worst case an $F'(f)$ can be the entire set of nodes n. So the `NewAffected_Set` is collected as an ordered set (without duplicates), and the union operation $\cup$ is implemented as an ordered set union whose total cost is the cost of sorting (to order the sets), followed by the cost of merging the two ordered sets. While the sorting requires $O(n \log n)$ time, the merging can be done in time proportional to the sum of the sizes of the two ordered sets ($O(n)$). Since the union is invoked for each level of the breadth first search, the algorithm has a worst-case run time complexity of $O(n^2 \log n)$.

The above algorithm computes an approximation to the spatial entropy of a combinational circuit. The same algorithm can be used to compute the actual spatial entropy as defined in Definition 1, if edge length information were available. The only change would be to the `compute_attributes` procedure that computes the local spatial entropy at each node in the circuit. In the next section we introduce *spatial entropy vectors* by extending the basic spatial entropy computation algorithm to further characterize the spatial entropy attribute.

3.4 Spatial Entropy Vectors

The spatial entropy of a circuit is a measure of the communication effort required to compute the circuit function. We obtained this amount by accumulating the spatial entropy at the internal nodes in the circuit. We can go one step further and characterize the spatial entropy of the circuit and the spatial entropy at the internal nodes of the circuit in terms of spatial entropy contributions of the primary inputs. This will give us the effort contributed by each primary input in computing the circuit function. In Chapter 4 we use this idea to study the problem of variable orderings for binary decision diagrams (BDDs).

We begin with an outline of the approach. The extension to the spatial entropy computation procedure to capture spatial entropy contributions of the primary inputs is implemented by adding another attribute, called the *spatial entropy vector* to each node in the network. The spatial entropy vector at any node is an n-element vector containing the spatial entropy contribution of each of the n primary inputs to the spatial entropy at the node. The same algorithm outlined in 3.3 is used to compute the spatial entropy vector at the circuit outputs. The value of this vector at any internal node w is computed by adding the spatial entropy vectors at the node inputs to the local spatial entropy δS_w at the node. Since δS_w is a single value it is distributed amongst all the primary inputs in proportion to the spatial entropy contribution of each primary input at the fanins of node w. The spatial entropy contribution of each primary input at node w's fanins can be computed from the spatial entropy vector at each of these fanins. We now express this idea formally.

For a circuit with n primary inputs the spatial entropy at a node w, S_w, and the spatial entropy vector at the node, $\vec{S_w}$, are related as follows,

$$S_w = \sum_{j=1}^{n} \vec{S_w}[x_j]$$

The spatial entropy at the node is the cumulative sum of the spatial entropy contributions of each primary input at the node.

The spatial entropy vector at each primary input x_j, $\overrightarrow{S_{x_j}}$, consists of 0s at all input positions in the vector, except for position j, which contains a 1. This indicates that the spatial entropy at this node is entirely contributed by the primary input x_j.

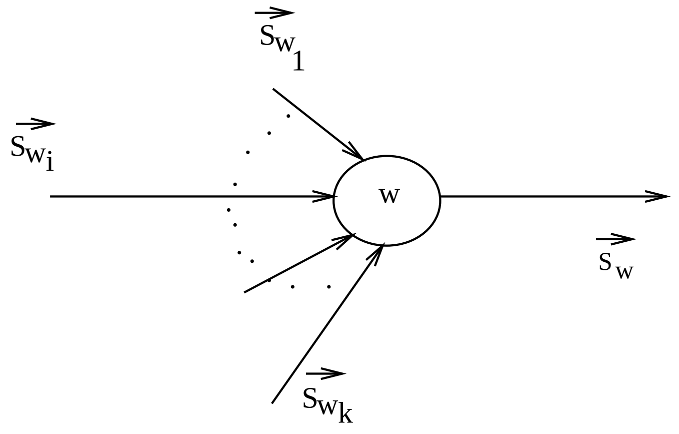


Figure 3.4: Spatial Entropy Vector Computation

How is the spatial entropy vector $\vec{S}$ computed at a node w? The spatial entropy vectors at the k fanins of a node w (Figure 3.4) can be accumulated by adding them together to obtain $\overrightarrow{S_{inw}}= \sum_{i=1}^{k} \vec{S_{w_i}}$. This yields a single accumulated vector consisting of the spatial entropy contributions of the primary inputs to all the fanins of node w. Since the local spatial entropy at w, δS_w, is a single value it needs to be distributed as contributions over all the primary inputs. This is accomplished by distributing δS_w amongst each primary input in proportion to its value in the accumulated vector $\overrightarrow{S_{inw}}$. If the total spatial entropy contribution of primary input x_j over all the fanins of node w is $\overrightarrow{S_{inw}}\ [x_j]$, and $S_{inw} = \sum_{j=1}^{N} \overrightarrow{S_{inw}}\ [x_j]$, is the total spatial entropy over all the fanins of node w, then the local spatial entropy contribution of primary input x_j is given by

$$\delta S_w * \frac{\overrightarrow{S_{inw}}\ [x_j]}{S_{inw}}$$

The intuitive justification behind this step is as follows. The local spatial entropy at node w (δS_w) is a measure of the effort required to compute the value at w. We distribute this effort amongst the primary inputs using the reasoning that a primary input that has contributed more effort to the inputs of w will be rewarded by receiving a greater contribution of the local spatial entropy. This gives us the equation for computing the spatial entropy vector at node w.

$$\vec{S_w} = \overrightarrow{S_{inw}} + \left(\frac{\delta S_w}{\sum_{j=1}^{N} \overrightarrow{S_{inw}}[x_j]}\right) * \overrightarrow{S_{inw}} \tag{3.1}$$

If we now extend the algorithm so that it computes the spatial entropy vectors during the breadth first search, we can obtain spatial entropy vectors at the outputs that reflect the spatial entropy contributions from the primary inputs. The following example will illustrate this extended algorithm on a simple gate-level network. It must be mentioned again that since we have assumed unit edge lengths, only an approximation to the actual spatial entropy vectors is being computed in this example.

Consider the combinational network shown in Figure 3.5 The circuit has 4 inputs and 1 output. We assume that the input values are uniformly distributed, *i.e.* all inputs have 1-probability of 1/2. The spatial entropy values at all the input nodes $S_{i_1}, S_{i_2}, S_{i_3}, S_{i_4} = 1.0$.

The spatial entropy vector at each input is initialized to a vector that has zeros at all positions and a 1.0 at that input position. $\vec{S_{i_1}} = [1.0, 0.0, 0.0, 0.0]$, $\vec{S_{i_2}} = [0.0, 1.0, 0.0, 0.0]$, $\vec{S_{i_3}} = [0.0, 0.0, 1.0, 0.0]$, $\vec{S_{i_4}} = [0.0, 0.0, 0.0, 1.0]$,

The first level of the breadth-first-search yields nodes n_5 and n_8 in the `New_Affected_Set`. So we compute the 1-probabilities and spatial entropy values at these nodes. n_5 is the output of an OR node, hence the 1-probability at n_5 is 3/4 (1 - 1/2*1/2). The 1-probability at n_8 is 1/2, since it is the output of an inverter node. At n_5 the spatial entropy value is calculated as follows

$$S_{n_5} = S_{i_1} + S_{i_2} + \delta S_{n_5}$$

The terms S_{i_1} and S_{i_2} provide the spatial entropy value from the inputs i_1 and i_2, which in this case is 1.0. δS_{n_5} provides the local spatial entropy contribution at the OR node n_5. This works out to 0.562. Hence $S_{n_5} = 2.562$. Similarly, $S_{n_8} = 1.693$.

The spatial entropy vector at node n_5 is

$$\vec{S_{n_5}} = \vec{S_{i_1}} + \left(\frac{\delta S_{n_5}}{S_{i_1} + S_{i_2}}\right) * \vec{S_{i_1}} + \vec{S_{i_2}} + \left(\frac{\delta S_{n_5}}{S_{i_1} + S_{i_2}}\right) * \vec{S_{i_2}}$$

Here $\vec{S_{i_1}}, \vec{S_{i_2}}$ are the initialized input vectors, since i_1 and i_2 are inputs. The other terms are calculated as follows: The term $\frac{\delta S_{n_5}}{S_{i_1}+S_{i_2}}$ is the proportion

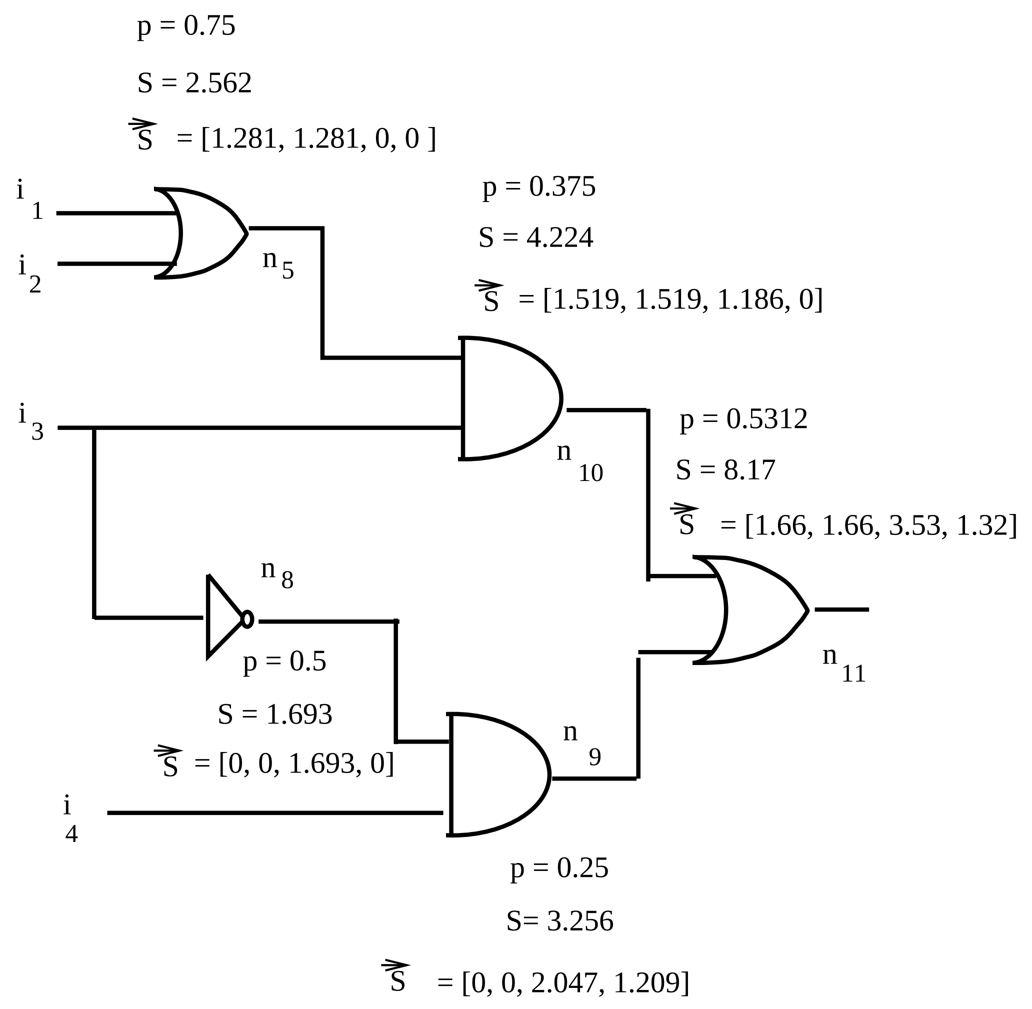

p = 0.75
S = 2.562
$\vec{S}$ = [1.281, 1.281, 0, 0]
i_1
i_2
n_5
p = 0.375
S = 4.224
$\vec{S}$ = [1.519, 1.519, 1.186, 0]
i_3
n_{10}
p = 0.5312
S = 8.17
$\vec{S}$ = [1.66, 1.66, 3.53, 1.32]
n_8
p = 0.5
S = 1.693
$\vec{S}$ = [0, 0, 1.693, 0]
n_{11}
n_9
i_4
p = 0.25
S= 3.256
$\vec{S}$ = [0, 0, 2.047, 1.209]

Figure 3.5: Example Spatial Entropy Calculation
$\forall\, i_j\ j = 1,4$, $p_{i_j} = 0.5, S_{i_j} = 1.0$, $\vec{S_{i_j}}$ is a vector of zeros, with a 1.0 at position j

in which the local spatial entropy at n_5 will be distributed amongst i_1 and i_2. This is 0.281. When multiplied by $\vec{S_{i_1}} = [1.0, 0.0, 0.0, 0.0]$, and $\vec{S_{i_2}} = [0.0, 1.0, 0.0, 0.0]$ it gives the local spatial entropy vector contribution due to i_1, and i_2 respectively, which are $[0.281, 0.0, 0.0, 0.0]$ and $[0.0, 0.281, 0.0, 0.0]$ respectively. Updating this with the spatial entropy vector values at i_1, i_2 we have the spatial entropy vector at $n_5 = [1.281, 1.281, 0.0, 0.0]$.

At the next level of the breadth-first-search the `New_Affected_Set` consists of nodes n_{10} and n_9, both outputs of AND nodes with two fanin edges. The probability at nodes n_{10} and n_9 is $p_{n_{10}} = 0.375$ ($\frac{3}{8} = 3/4 * 1/2$), and $p_{n_9} = 0.25$, ($\frac{1}{4} = 1/2 * 1/2$). The spatial entropy at the nodes are: $S_{n_{10}} = S_{n_5} + S_{i_3} + \delta S_{n_{10}}$, which is 4.224. $S_{n_9} = S_{n_8} + S_{i_4} + \delta S_{n_9}$, which is 3.255.

$\delta S_{n_{10}}$ is the local spatial entropy contribution at the AND node n_{10}. The spatial entropy vector at node n_{10} is computed as:

$$\vec{S_{n_{10}}} = \vec{S_{n_5}} + \left(\frac{\delta S_{n_{10}}}{S_{n_5} + S_{i_3}}\right) * \vec{S_{n_5}} + \vec{S_{i_3}} + \left(\frac{\delta S_{n_{10}}}{S_{n_5} + S_{i_3}}\right) * \vec{S_{i_3}}$$

This yields a contribution of

$$[1.281, 1.281, 0.0, 0.0] + 0.1857 * [1.281, 1.281, 0.0, 0.0] = [1.519, 1.519, 0.0, 0.0]$$

from n_5, and a contribution of

$$[0.0, 0.0, 1.0, 0.0] + 0.1857 * [0.0, 0.0, 1.0, 0.0] = [0.0, 0.0, 1.1857, 0.0]$$

from i_3, making $\vec{S_{n_{10}}} = [1.519, 1.519, 1.1857, 0.0]$.

A similar exercise at node n_9 yields $\vec{S_{n_9}}$ as $[0.0, 0.0, 2.047, 1.209]$.

Finally, the last level in the breadth-first-search yields the output node n_{11}, which is the output of an OR node with two fanin edges. We have $p_{n_{11}} = 0.53125(17/32)$, $S_{n_{11}} = S_{n_{10}} + S_{n_9} + \delta S_{n_{11}}$, which is $4.224 + 3.255 + 0.691 = 8.170$. The spatial entropy vector at n_{11} is:

$$\begin{aligned} \vec{S_{n_{11}}} &= [1.519, 1.519, 1.1857, 0.0] + [1.519, 1.519, 1.1857, 0.0] * 0.0923 \\ &+ [0.0, 0.0, 2.047, 1.209] + [0.0, 0.0, 2.047, 1.209] * 0.0923 \end{aligned}$$

This evaluates finally to $[1.659, 1.659, 3.531, 1.321]$. Adding these up gives us $1.659 + 1.659 + 3.531 + 1.321 = 8.170$, the spatial entropy at the output

$S_{n_{11}}$. An inspection of the individual contributions of the primary inputs (in $\vec{S_{n_{11}}}$) to the spatial entropy at n_{11} shows that input i_3 has the largest contribution (3.531). i_1, i_2 have identical contributions (1.659) and $i4$ has the least (1.321).

In the next section we discuss in detail, the factors that affect the accuracy of spatial entropy computation.

3.5 Factors affecting Spatial Entropy Computation

Computing spatial entropy accurately at the gate level is difficult. As we have seen, in the absence of wire length information, we have already had to make the approximation of using unit lengths while computing spatial entropy. In this section we discuss the effects of three factors that make spatial entropy computation difficult: *logic minimization*, *wire length*, and *reconvergent fanout*. The effect of logic minimization on spatial entropy is discussed at length in a later chapter where we define the information content of a boolean function in cube space and relate this to the spatial entropy of such functions when minimized in the two-level and multi-level form. Therefore we treat minimization only briefly here showing intuitively how minimization can affect spatial entropy values. The effects of wire length and reconvergent fanout will be handled in more detail here.

3.5.1 Logic Minimization and Spatial Entropy

In a gate level circuit the spatial entropy is computed by accumulating the information-distance product over all the nodes in the circuit. Suppose the spatial entropy of a given circuit C, as computed with our unit wire length approximation, is S_C. Let C' be a functionally equivalent version of the same circuit C that has not been minimized and thus has more gate nodes than C. Since the spatial entropy accumulates over all nodes and is non-decreasing, $S_{C'}$ may be greater than S_C. Figure 3.6 shows how the spatial entropy can increase. The implementation that is not minimized will have a greater spatial entropy, due to the excess spatial entropy contribution from the additional gate. Excess spatial entropy is not desirable since it artificially

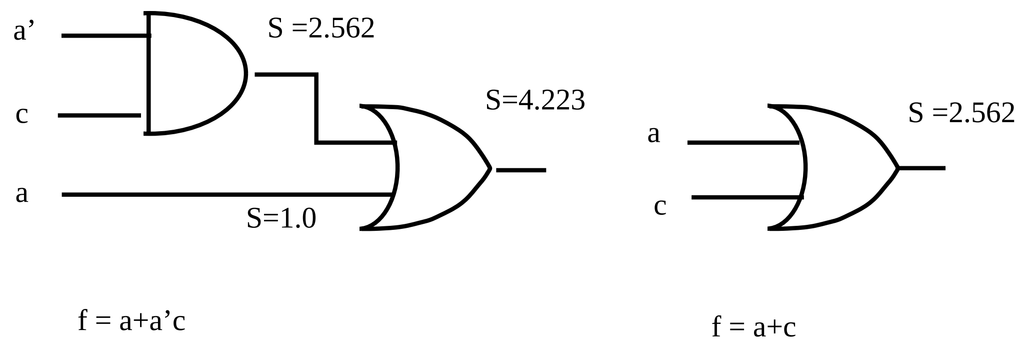


Figure 3.6: Effect of Minimization on Spatial Entropy

increases the spatial entropy of some nodes over others. Hence we require our circuits to be minimal in literal count before performing spatial entropy computation. In practice since minimization is a heuristic we only have approximately minimal circuits. The connection to switching energy can be seen here too. In an implementation with more gates there is likely to be more dynamic switching of states contributing to greater switching energy. We now discuss the relationship between wire length and spatial entropy computation.

3.5.2 Wire Length and Spatial Entropy

Computing exact spatial entropy as we have seen requires wire length information. Since a gate level netlist does not have length information, to this point, we have been approximating spatial entropy computation by using a unit wire length estimate. The problem here is that the flow of information across the different levels of abstraction in the VLSI design process make it difficult to compute spatial entropy accurately. These abstraction levels do not provide sufficient detail to compute the attribute. For instance at the gate level of abstraction one can compute 1 and 0 probabilities, and hence information H_w, but wire length information is absent. Actual wire lengths exist only at the layout level. But there are ways in which this length estimate can be improved at the gate level.

One approach is to estimate wire length at the gate level by using the *number of fanouts* at a node as the wire length at the node. Another approach is to use the *static levels* in the circuit for wire length information. The nodes in the circuit can be marked with levels corresponding to the gates, in a breadth-first prepass. Then the wire length at a node can be approximated by the cumulative sum of the level numbers of all its fanout nodes. So if one of the fanouts has a large level number, then the approximated wire length will be large signifying that a long wire was needed to get to that fanout node.

One can also visualize the extraction of length information from the schematic of a gate level netlist. These lengths can be computed as follows. Embed the netlist on a planar grid with the gates or nodes represented as points on the grid, and the wires or edges represented as Manhattan lines on the grid. (All input nodes would enter from one end and output nodes would appear at the other end.) Place the nodes so as to minimize edge crossing and edge length. Compute the Manhattan wire lengths between grid points. This placement problem itself is NP-complete [GJ79], but approximations can be obtained.

Sometimes instead of *point-to-point* lengths between nodes only a *lumped* length value may be available at the node. Lumped wire lengths introduce inaccuracies at *multiple fanout nodes*. When boolean functions share logic it is captured in an implementation by fanouts from the shared portion of logic. Consider Figure 3.7. Suppose the logic at the output of a k-input gate w is shared by 3 output functions This will be reflected by a node w with k fanins and 3 fanouts (f_1, f_2, and f_3).

The local spatial entropy at w, computed exactly, is expressed below

$$\delta S_w = \sum_{j=1}^{3} H_w * l_{(w,f_j)}$$

It is equal to the information H_w at the node multiplied by the distance traveled by the information along each fanout edge (w, f_j). Since the edge length from node w to each of its 3 fanout nodes will usually vary, each fanout node f_j is receiving a local spatial entropy contribution proportional to the length of the fanout edge (w, f_j). The reasoning here is that for a fanout edge with greater length the information H_w has to travel a longer distance to reach the fanout node.

The spatial entropy at each fanout node f_j (S_{f_j}) is computed as follows. The cumulative sum of the spatial entropies at the fanins of node w ($\sum_{i=1}^{k} S_{w_i}$)

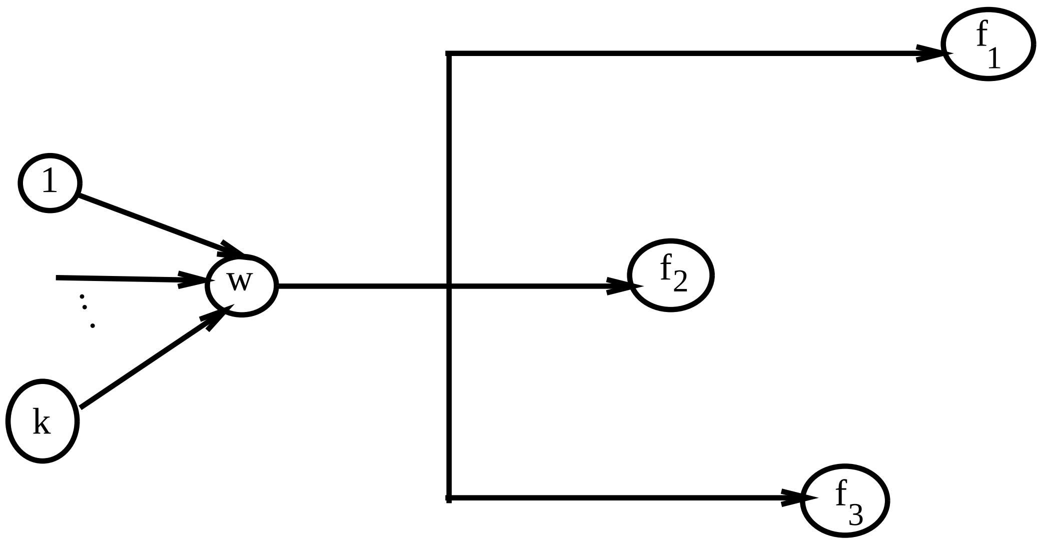


Figure 3.7: A Node w with Multiple Fanouts

represents the total effort required to compute the spatial entropy at the inputs to w. Since this represents the effort prior to reaching w it is divided equally amongst the three fanout nodes f_1, f_2, and f_3. On the other hand, the local spatial entropy contribution at each fanout node is proportional to the length of the fanout edge from w to f_j. So S_{f_j} is expressed below as

$$S_{f_j} = \frac{\sum_{i=1}^{k} S_{w_i}}{3} + H_w * l_{(w,f_j)} \tag{3.2}$$

where $l_{(w,f_j)}$, is the edge length between w and f_j.

With unit edge lengths or lumped edge lengths, instead of point-to-point edge lengths, we can only obtain approximate values for the spatial entropy at each fanout node. Consider the switching energy analogy. The switching energy at a multiple fanout node w is a function of the individual switching capacitances at each fanout node of w contributing to the delay at w. A lumped value will only yield an approximation. There are different ways in which to perform this approximation.

Option 1: We can compute the local spatial entropy at w as $\delta S_w = \sum_{i=1}^{3} H_w * l_{(w,f_i)}$, where $l_{(w,f_i)}$ is either equal to 1 or is the lumped wire length at w. This amount can then be divided equally amongst the 3 fanouts just as we divided the spatial entropy at the fanins of w (S_i). This is shown below.

$$S_{f_j} = \frac{1}{3} * (\sum_{i=1}^{k} S_{n_i} + \delta S_n) \tag{3.3}$$

But there is a disadvantage to doing this. It assumes that the information H_w travels an equal distance to each fanout node and each fanout node should get an equal fraction of the effort. This may not be the case. By dividing the local spatial entropy at w equally ($\frac{\delta S_w}{3}$) instead of determining it to be in proportion to the edge length, one fanout node may receive a greater proportion than it should actually get. Similarly another node may receive a smaller proportion than what it is entitled to receive. In single output functions this does not hurt because the fanouts are going to reconverge in the end into one single output; so even if we divide the spatial entropy at w disproportionately, they are going to recombine at the output. But the same is not true of multi-output functions. By dividing the spatial entropy at w disproportionately we are draining away some of the spatial entropy (effort) used to compute one function and attributing it to another.

Option 2: Another option is to first compute the local spatial entropy as above, $\delta S_w = \sum_{i=1}^{3} H_w * l_{(w,f_i)}$, and then obtain the total spatial entropy at w as $(\delta S_w + \sum_{i=1}^{k} S_{w_i})$. This is then *replicated* at all the fanout nodes. So

$$S_{f_j} = \sum_{i=1}^{k} S_{w_i} + \delta S_w \tag{3.4}$$

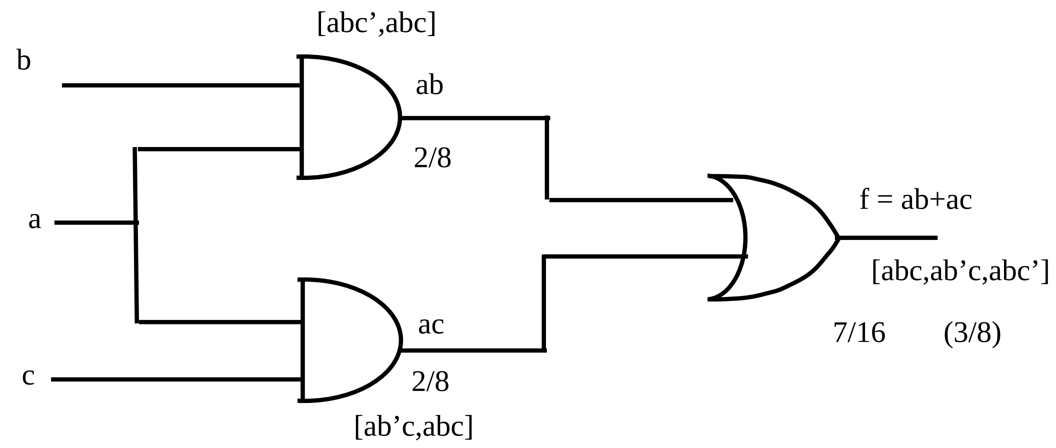
Replicating the first term $\sum_{i=1}^{k} S_{w_i}$ changes the Equation 3.2 by an equal amount at every fanout node and this does not affect the accuracy. Replicating the second term (δS_w) however, is an approximation to distributing it in proportion to the edge lengths amongst the different f_j. On the one hand, this does not drain away the spatial entropy meant for one function into another, because the entire amount (computed at node w) is replicated along each of the fanout edges. On the other hand, this approach can affect circuits with heavy reconvergent fanout. This is because when fanout nodes reconverge their spatial entropies get added. In this process we have artificially "created" more spatial entropy at the reconvergent node than necessary.

Option 3: A third option, which is a combination of the above, is to perform a prepass and compute for each output (of a multi-output function) the cone of logic that supports it. Then we perform spatial entropy computation on each output cone independently, either by equally dividing the spatial entropies at fanout nodes or by replicating them. This still is not an ideal solution. We are computing the spatial entropy over each output of a multi-output function, but with a unit edge length or lumped edge length estimate the local spatial entropy at the nodes are still being approximated Nevertheless performing a *cone-based* spatial entropy computation helps us concentrate on one output at a time preventing spatial entropy contributions meant for one output from getting drained away into another.

By combining lumped wire length information along with static level structure information one can further approximate the proportionate distribution of δS_w. Fanout nodes with greater level numbers will now receive a greater share of the lumped wire length. In Chapter 4, we use lumped wire lengths from an extracted layout to compute the spatial entropy at the gate level. This is used in the problem of spatial entropy based variable ordering for binary decision diagrams (BDDs). We also empirically study the effects of some other length estimates. Since all these estimates still yield only approximations to the spatial entropy we are interested in determining how useful they are and how they affect spatial entropy computation.

3.5.3 Reconvergent Fanout and Spatial Entropy

Reconvergent fanout affects the exact calculation of 1-probabilities at the nodes. Since the information H_w at a node w is a function of the 1 and 0 probability at the node, an error in the probability computation will affect H_w and consequently the spatial entropy. Exact 1-probability computation in circuits with reconvergent fanout has been extensively studied in the areas of fault simulation [BPH84, MJ90, AS87], test generation [HM86], and testability analysis [SDB84, SPA85, JA84].



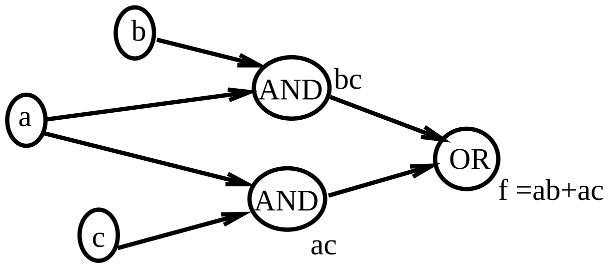


Figure 3.8: Reconvergent Fanout - An Example

We begin by reviewing the idea of reconvergence. This is illustrated with a simple example in Figure 3.8. The gate-level implementation and a simple digraph model of a 3 input function is shown. A *path* in the graph is a sequence of vertices from a source node to a destination node. Two paths are said to be *reconvergent* if their source and destination nodes are the same. The paths are said to *fanout* at the source node and *reconverge* at the destination node. The destination node is called the *reconvergent node*. In the example the output of the OR gate, which is also the output node of the function, is a reconvergent node for two reconvergent paths that fanout from the source node a.

Let us see how reconvergence affects the calculation of 1-probabilities. The 1-probability at a node is the fraction of onterms at that node. Figure 3.8

identifies the onterms at the two internal nodes and their 1-probabilities as fractions. The 1-probability at the OR node is obtained by combining the 1-probabilities (and hence the onterms) at its inputs, using the rule defined for an OR gate. But due to reconvergent fanout the two internal nodes share a common onterm abc. As a result the 1-probability at the reconvergent node no longer combines independent 1-probabilities (or onterms) at its inputs, as assumed by our probability rules for primitive gates. While the actual 1-probability at the OR node obtained by computing the fraction of onterms at the node is 3/8, the primitive gate rule calculates it incorrectly as 7/16.

Since 1-probabilities affect the number of information bits, error in the probability computation will be reflected as error in the spatial entropy expression S. But in addition to this error reconvergent fanout can introduce another form of error at fanout nodes depending on how the local spatial entropy at a node is attributed to its fanouts. For instance, if the spatial entropy and spatial entropy vector at a node is replicated amongst its fanouts (instead of dividing it equally or proportionately), then the replicated spatial entropy values add up at the reconvergent node. In the process the cumulative spatial entropy at the reconvergent node is artificially boosted. In this section we discuss approaches to remove these errors. We begin by briefly discussing existing approaches in the literature to compute exact 1-probabilities. We then extend this idea to removing error in the spatial entropy at a reconvergent node.

One of the ways to avoid the error in 1-probability computation due to reconvergent fanout is to apply a random patterns to the circuit and compute 1-probability at the internal nodes. A similar approach has been used in fault simulation and random test pattern generation [AS87, SB87], where a circuit is simulated with a sampling of random patterns until it is capable of detecting a high percentage $100 - \epsilon$ of faults with a high probability $1 - \delta$. Typically $\epsilon < 2, \delta < 0.001$. In this particular instance, if the distribution of probabilities at the inputs to the circuit are known, then it is possible to generate a sample of input patterns where each input is randomly selected to be a 1 with its known 1-probability. This statistical sample can then be applied on the circuit to generate 1/0 values at the internal nodes conditional to each input assignment. A weighted sum of these values would yield the 1-probability at the internal node. The main drawback is that to compute the probability exactly an exponential number of input patterns is required.

As mentioned earlier, exact 1-probability computation in reconvergent fanout circuits has been studied extensively. [BPH84, MJ90, AS87, HM86, SDB84, SPA85, JA84]. Parker and McCluskey [PM75] described an algorithm for exact computation of 1-probabilities in circuits with or without reconvergent fanout. But the exact procedure can require exponential space and time. Since then several approximate algorithms have been proposed for computing the 1-probability at reconvergent nodes. In all these approaches [SPA85, JA84, SDB84] there is a trade-off between accuracy of the probability computation and computational efficiency. This implies that in circuits with reconvergent fanout we again compute only an approximation of the spatial entropy in the circuit.

In the rest of this section we examine one of the approaches to compute 1-probability at reconvergent nodes in greater detail, and discuss how it can be extended to remove errors in spatial entropy. In [SPA85], Seth *et.al* propose a simple and elegant procedure to trade efficiency with accuracy in the probability computation phase. The procedure constructs *supergates* for reconvergent nodes and then recomputes the correct probability at the reconvergent node from the supergate. Informally a supergate $SG(v)$ for a reconvergent node v is a subgraph of the circuit that includes the edges and nodes on all paths that originate from nodes with independent probabilities. Figure 3.9 shows a simple circuit and one of its supergates. The procedure for 1-probability computation at the reconvergent node consists of the following steps.

1. Identification of reconvergent nodes.
2. Determining the supergate for each reconvergent node.
3. Calculating the probabilities at the supergate outputs.

We briefly elaborate on these steps.

Reconvergent Node Identification: Reconvergent nodes in a circuit are identified via a quick preprocessing step that sweeps through the graph in a breadth-first manner. This step labels each node v in the graph with a label set I_v of primary inputs occurring in its cone of influence. To check if v is a reconvergent node the procedure then examines $fanin(v)$, the set of fanin nodes of v. v is *not reconvergent* if and only if the following is true.

$$\forall x, y \in fanin(v) \; I_x \cap I_y = \{\}$$

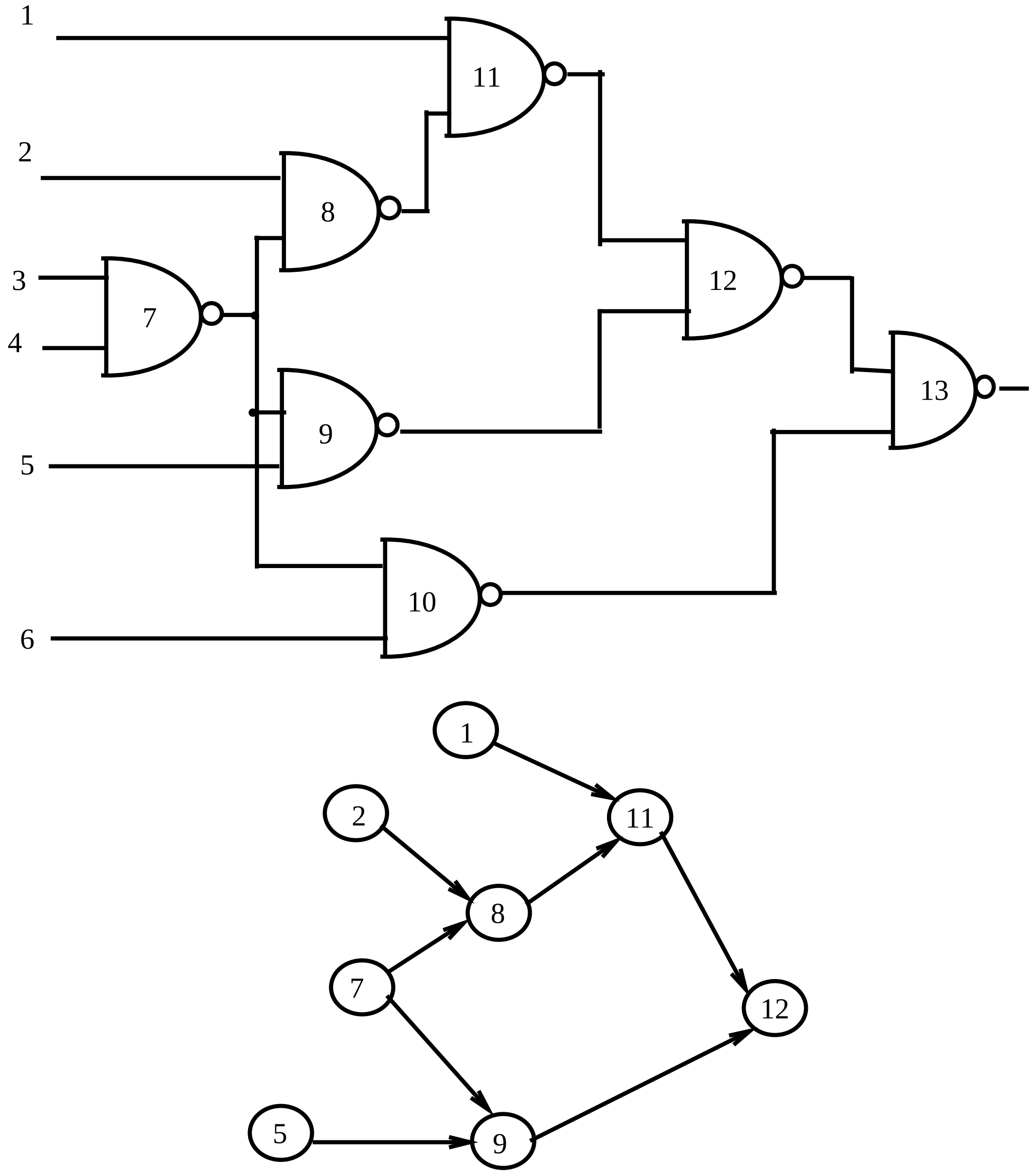


Figure 3.9: A Circuit with a Supergate $SG(12)$

So the label sets for all fanin nodes of v must be mutually disjoint for v to be non-reconvergent.

Supergate Construction: This step constructs a supergate for every identified reconvergent node. Working backwards from the reconvergent node, in a breadth-first manner, all nodes that make up the supergate are collected. The procedure stops when all inputs to the supergate are independent, *i.e.* their label sets are disjoint. It returns two mutually exclusive sets of nodes. The first is a set of *frontier nodes* that are input nodes to the supergate. The second is the set of *interior* nodes that are internal nodes of the supergate. There are two kinds of frontier nodes: *fanout input nodes*, and *non-fanout input nodes.* The fanout input nodes are supergate inputs from which reconvergent paths originate. Figure 3.9 has one fanout input node, 7, in $SG(12)$. There are three non-fanout input nodes - $1, 2$, and 5.

Supergate 1-Probability Computation: The procedure to compute the 1-probability of the reconvergent node v of a supergate $SG(v)$ has been described in [SPA85]. Values are assigned to the fanout and non-fanout input nodes of the supergate which is then simulated to factor out dependent events generated due to reconvergent fanout. The non-fanout inputs are assigned their existing 1-probability values. The fanout inputs are assigned a binary vector of assignments: $[000, 001, \ldots]$, where each fanout input is assigned a value of 1 or 0. If the number of fanout inputs is too large, then either only a sample of all the possible vector assignments is chosen, or a smaller supergate is constructed by limiting the number of fanout inputs it can have. In the latter case all the unexpanded nodes (which could have lead to fanout inputs) are marked as non-fanout inputs of the supergate.

The supergate is then simulated with the vector assignment to the fanout inputs and the probabilities at the non-fanout inputs. For each individual simulation the 1-probability at the reconvergent node v is calculated conditional to the given fanout input vector assignment. These conditional probability values are then weighed with the probability of the event corresponding to the fanout input vector assignment. Suppose $p^1_v(\vec{a})$ was the conditional probability at v for the binary assignment $\vec{a} = 001$, for fanout inputs x, y and z respectively. Then

$p_v^1(\vec{a}) * (p_x^0 * p_y^0 * p_z^1)$ is the weighted conditional probability at the reconvergent node for event 001. This weighted sum over all fanout input assignments yields the exact 1-probability at the reconvergent node.

The tradeoff between computational efficiency and probability accuracy arises when the fanout node set is too large. In such a situation when simulating with the set of all possible fanout input assignments becomes impractical, a sample subset is used. Accordingly, the final probability is also scaled by the size of this sample set.

Spatial Entropy Computation with Supergates:
While removing errors in the spatial entropy at a reconvergent node the steps of reconvergent node identification and supergate construction outlined above do not change. But having constructed the supergate and computed the 1-probability at the reconvergent node, its spatial entropy and spatial entropy vector must also be calculated. Informally this task can be described as follows. To start with, the constructed supergate is treated as an independent circuit. The existing spatial entropy S and spatial entropy vector $\vec{S}$ values at its frontier nodes are saved. In Figure 3.10 this happens to non-fanout nodes $8, 9$ and 11 and fanout node 7. Since the frontier nodes of the supergate are primary inputs of an independent circuit, these are now initialized with a supergate spatial entropy S^{sg} and spatial entropy vector $\vec{S^{sg}}$. So S_8^{sg}, S_9^{sg}, S_{11}^{sg} and S_7^{sg} are initialized, and the entropy vectors are initialized similarly. Then the local spatial entropy (δS^{sg}) and spatial entropy vector ($\delta\ \vec{S^{sg}}$) values for the supergate are computed using the normal spatial entropy computation procedure. At the end of this the supergate can be viewed as one large gate similar to the other (AND, OR, NAND) gates. Figure 3.10 shows this. These local (supergate) spatial entropy and spatial entropy vector values are combined with the saved spatial entropy and spatial entropy vector values at the supergate inputs to yield a cumulative expressions for the spatial entropy S_v and spatial entropy vector $\vec{S_v}$ at the reconvergent node. We now elaborate on these steps.

The procedure to calculate the spatial entropy S^{sg} local to the supergate is not as straight forward as the procedure to calculate the exact 1-probability. This is because the spatial entropy (and the spatial entropy vector) are aggregate concepts defined over a collection of input assignments or an entire probability distribution. A node with an assigned value of 1 or 0 has a spatial

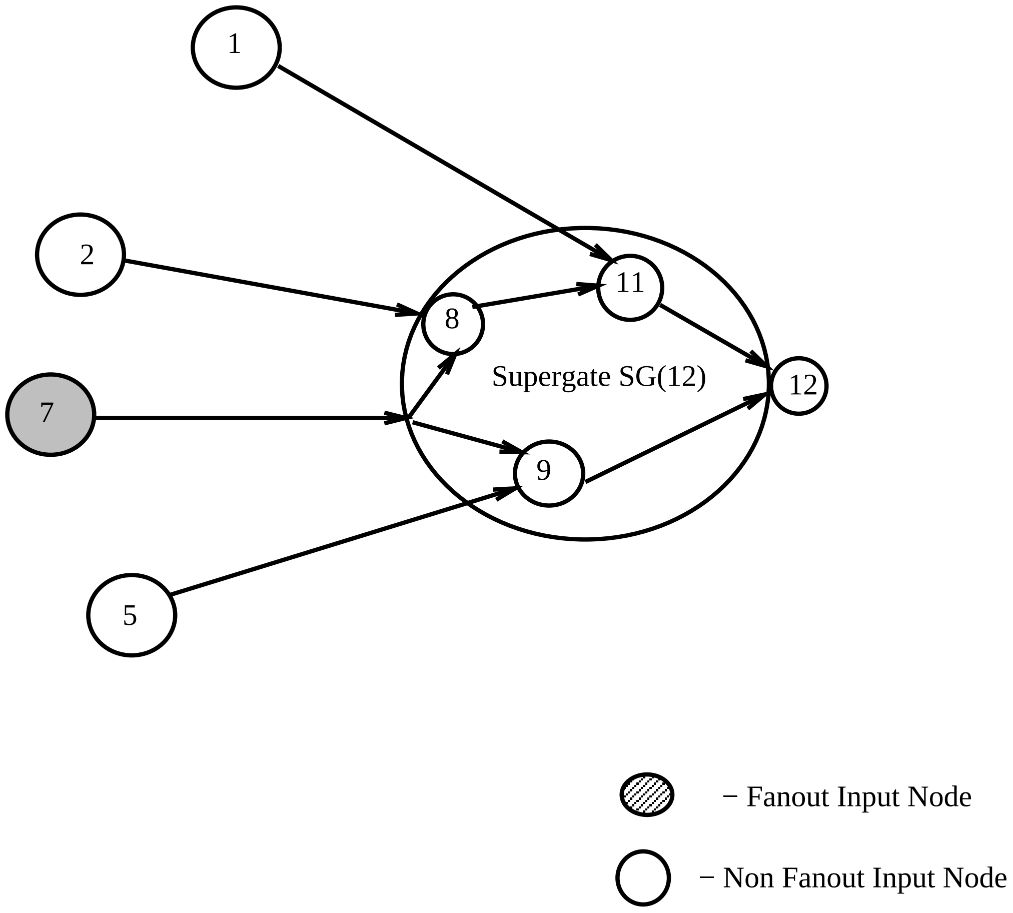


Figure 3.10: The Supergate $SG(12)$ denoted as one Large Gate Node

entropy of 0 since its information content H is 0. Hence the spatial entropy of the supergate cannot be calculated while simulating the supergate with binary (1/0) values. It can be done only after the exact 1-probability is computed at the reconvergent node.

The procedure to compute the spatial entropy at the reconvergent node consists of the following steps.

1. Compute exact 1-probability of the reconvergent node.

2. Compute spatial entropy and spatial entropy vector local to the supergate δS^{sg} and $\overrightarrow{\delta S^{sg}}$.

3. Add these values to the spatial entropy and spatial entropy vector values saved at the inputs of the supergate to obtain S_v and $\vec{S_v}$ at the reconvergent node.

We discuss each of these individually.

Step 1: This step is identical to the supergate 1-probability calculation procedure discussed earlier.

Step 2: Having computed the exact 1-probability at the reconvergent node another pass is made through the supergate to compute the spatial entropy δS^{sg} and spatial entropy vector $\overrightarrow{\delta S^{sg}}$ local to the supergate. For this the fanout inputs of the supergate are first initialized to a spatial entropy of 1. The non-fanout inputs are initialized to their local spatial entropy $(H * 1)$ which is computed from their 1-probabilities. Since the supergate is being treated as an independent circuit the spatial entropy vector for the supergate has as many components as the number of frontier nodes in the supergate (fanout inputs + non-fanout inputs). As in the normal spatial entropy computation procedure the spatial entropy vector at a frontier node x_i is initialized with a zero at all positions except at position i.

Next a breadth-first traversal of the nodes in the supergate is performed, and spatial entropy and spatial entropy vector values are calculated at all the nodes inside the supergate to finally obtain a value for δS^{sg} and $\delta\, \overrightarrow{S^{sg}}$.

Step 2: In this step the local spatial entropy and spatial entropy vector values at the supergate are combined with the existing spatial entropy and spatial entropy vector values at the supergate inputs to obtain the spatial entropy and spatial entropy vector at the reconvergent node. We express this formally as follows. Suppose we have an n-input circuit, with a reconvergent node v whose supergate has m inputs, and supergate spatial entropy and spatial entropy vector values of δS^{sg} and $\delta \overrightarrow{S^{sg}}$. The spatial entropy and spatial entropy vector of the reconvergent node v can be expressed as follows

$$S_v = \sum_{i=1}^{m} S_{x_i} + \delta S^{sg} \tag{3.5}$$

$$\vec{S_v} = \sum_{i=1}^{m} \overrightarrow{S_{x_i}} + \delta \overrightarrow{S^{sg}} \tag{3.6}$$

where S_{x_i} and $\vec{S_{x_i}}$ are the saved spatial entropy and spatial entropy vector values at the frontier node x_i of the supergate.

The vector $\overrightarrow{S_{x_i}}$ is n-elements long while $\delta \overrightarrow{S^{sg}}$ is m elements long, since the former is in terms of the primary inputs of the circuit, while the latter is in terms of the inputs of the supergate. n and m can be different. So the values in $\delta \overrightarrow{S^{sg}}$ have to be distributed proportionately in terms of the primary inputs of the circuit. A supergate input x_i will have a saved spatial entropy and spatial entropy vector of S_{x_i} and $\vec{S_{x_i}}$. It will also have a contribution $\delta \overrightarrow{S^{sg}}[x_i]$ in the local spatial entropy vector of the supergate. So if we distribute this contribution amongst the primary inputs in the vector $\vec{S_{x_i}}$, we can obtain the local spatial entropy contribution for a given primary input j as

$$\delta \overrightarrow{S_{x_i}}[j] = \frac{\overrightarrow{S_{x_i}}[j]}{S_{x_i}} * \delta \overrightarrow{S^{sg}}[x_i]$$

Since we have now expressed the supergate spatial entropy contributions in terms of the primary inputs, we can write Equation 3.6 as

$$\vec{S_v} = \sum_{i=1}^{m} (\overrightarrow{S_{x_i}} + \delta \overrightarrow{S_{x_i}})$$

At the end of this third step, the probability, spatial entropy and spatial entropy vector at the reconvergent node have been calculated.

To summarize, we have introduced the spatial entropy concept as a dynamic attribute in the circuit domain and defined it quantitatively using the entropy function in information theory. We explained how spatial entropy can provide a quantitative measure of the switching energy in a physical circuit implementation. A simple algorithm to estimate spatial entropy in gate-level circuits was described. The spatial entropy attribute was extended to describe spatial entropy vectors. Computing spatial entropy accurately is difficult. We discussed the various factors that affect the accuracy of its computation. The effects of minimization, wire length and reconvergent fanout are not necessarily independent. For instance, minimization reduces literal count but does not necessarily reduce wire lengths, and hence spatial entropy. Since minimization can also alter the structure of reconvergence in a circuit this too can affect spatial entropy. All this makes it all the more difficult to characterize the effects of each factor independent of the other. In the next chapter we discuss an application of spatial entropy to the problem of variable ordering for binary decision diagrams (BDDs). This application also helps us evaluate the various approximations in computing spatial entropy and their effects on the solution.

Chapter 4

Spatial Entropy Based BDD Ordering

In this chapter we study an application of spatial entropy - using it to generate input variable ordering for binary decision diagrams(BDDs). The problem of generating an ordering for the primary input variables of a boolean function in order to build small BDDs is an important one and has been studied extensively [MWBV88, FFK88, MIY90, Ber91, ISY91, JPHS91, BRM91, MKR92]. The ordering strategies in most of the literature rely on a static examination of the circuit topology. In this chapter we propose a dynamic approach to generate variable orders that uses spatial entropy to capture the aggregate dynamic data movement in the circuit. We begin by introducing BDDs and motivating the variable ordering problem. After a short survey of related research we outline our approach of using spatial entropy vectors, $\overrightarrow{S}$, to generate variable orders. In Section 4.3 we describe the experimental set-up to test our approach. After we state the objectives and assumptions we describe the data set, the software, and the experimental apparatus. Then in Section 4.5 we present our results followed by observations and conclusions.

4.1 Introduction and Motivation

In this section we begin with an introduction to binary decision diagrams and ordered binary decision diagrams. Then we introduce the variable ordering problem, motivate it and summarize related research in this area.

4.1.1 Binary Decision Diagrams

A binary decision diagram (BDD) [Bry86] is a directed acyclic graph (DAG) representation (V, E) of a Boolean function $f : I \rightarrow \{1, 0\}$. The input variables, I, of the function are represented by nodes in the graph . The node set V of the graph contains two types of nodes: *non-terminal* nodes and *terminal* nodes. A non-terminal node corresponds to a boolean input variable and is labeled by the name of the variable. A *terminal* node corresponds to the boolean constants 1 or 0 and is labeled T (true) or F (false) respectively. A fanout edge of a nonterminal node $x \in V$ is labeled with a 1 or 0. This label (0 or 1) denotes the value that is assigned to the boolean variable x along this path. The fanout edge $(x, y) \in E$ leads from x to y, where y is either another nonterminal node or a terminal node. In the latter case y is labeled T or F. The node y is called the *child* of x ($child(x) = y$) and x is called the *parent* of y. The value of the function for a given assignment to the input variables is determined by traversing the digraph (V, E) from the root down to a terminal node, following a 0 or 1 edge depending upon the value assigned to the boolean variable at that node. The value of the function equals 1 if the terminal node is T, and 0 if the terminal node is F.

Binary decision diagrams were first introduced in [Lee59], and further popularized by Akers [Ake78]. In [Bry86], Bryant introduced *ordered* binary decision diagrams (OBDDs), a restriction on the class of binary decision diagrams. In an ordered BDD a strict ordering π is imposed on the input variables. The labeling of the nodes along any path from the root to the terminal node must follow the order given by π. This ordering can be formally defined as follows. Suppose we are given a boolean function f with a set of input variables I, where each variable in I is assigned an index according to the ordering function $\pi : I \rightarrow \{1, \ldots, \mid I \mid\}$. A BDD for the function f under an ordering π has to satisfy the following constraint. Given non-terminal nodes w, x and y, $\pi(x) < \pi(y)$ if and only if the following relation holds.

$$[(child(x) = y) \Rightarrow \pi(x) \prec \pi(y)] \vee [(child(w) = y \wedge \pi(x) < \pi(w)) \;\Rightarrow \pi(x) \prec \pi(y)]$$

Figure 4.1 shows an example OBDD for the function $a_1.b_1 + a_2.b_2 + a_3.b_3$, with the ordering $\pi : a_1 \prec b_1 \prec a_2 \prec b_2 \prec a_3 \prec b_3$. Informally this ordering is specified by the sequence of variables in ascending order, $\{a_1, b_1, a_2, b_2, a_3, b_3\}$. Since we are only interested in ordered binary decision diagrams the rest of our discussion shall use the term BDDs to refer to ordered BDDs.

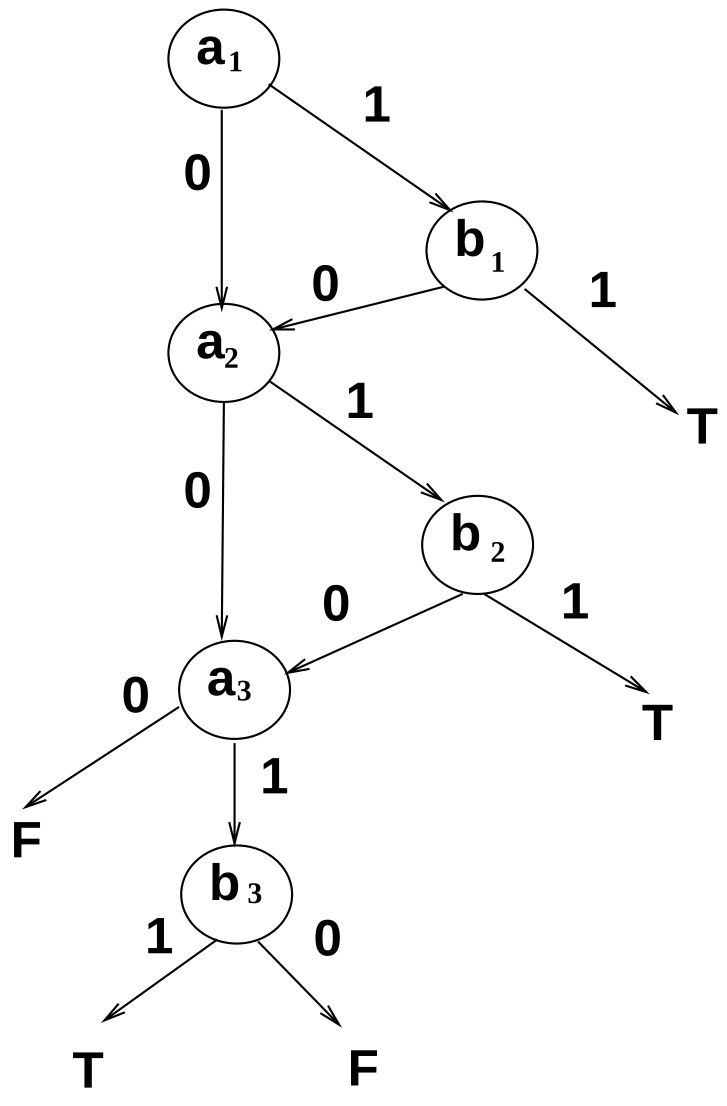


Figure 4.1: OBDD of $f = a_1.b_1 + a_2.b_2 + a_3.b_3$ with ordering $\{a_1, b_1, a_2, b_2, a_3, b_3\}$

The variable ordering restriction on BDDs has several desirable properties for symbolic boolean manipulation. In particular, for a given variable ordering π the smallest BDD for a function is unique. Any BDD can be reduced to yield this unique representation by a simple efficient algorithm [Bry86]. Furthermore many Boolean functions have efficient, polynomial size BDD representations making it efficient to manipulate such functions and perform operations on them. This in turn yields efficient algorithms [Bry86, BBR89] for tasks such as: determining if an input assignment satisfies a function (satisfiability testing), testing the equivalence of two functions (tautology checking), and combining two functions with a boolean operation. These properties led to the early use of BDDs in combinational logic verification [Bry86]. Since then they have been applied in several other areas of design automation research: sequential verification, test pattern generation, logic synthesis, and optimization. This is not surprising since the boolean function representation constitutes an important level of abstraction in the VLSI design process and an efficient representation/data structure benefits all the algorithms employed at this level of abstraction.

4.1.2 The Variable Ordering Problem

A key issue that determines BDD size is the input variable ordering. The BDD for a function can be very sensitive to the variable ordering. Figure 4.2 shows a BDD of the function represented in Figure 4.1, but with a different ordering, $\pi = \{a_1, a_2, a_3, b_1, b_2, b_3\}$. The size of the BDD has increased substantially.

Bryant [Bry91] has shown that while for certain classes of Boolean functions (integer multipliers) an efficient (polynomial size) BDD representation does not exist regardless of variable ordering, there are still a large class of boolean functions that have efficient BDD representations. The problem then becomes one of finding a good input ordering that will yield an efficient BDD representation. Since there are an exponential number ($n!$) of possible orderings for n input variables, it is hard to exhaustively determine the smallest BDD for a given function. The best algorithm so far to arrive deterministically at an optimal ordering for a function has a time complexity of $O(n^2 3^n)$ [FS90]. Several efforts [Ber91, MWBV88, FFK88, MIY90, ISY91, JPHS91, MKR92] have been devoted to developing heuristics or approximate strategies for finding a good

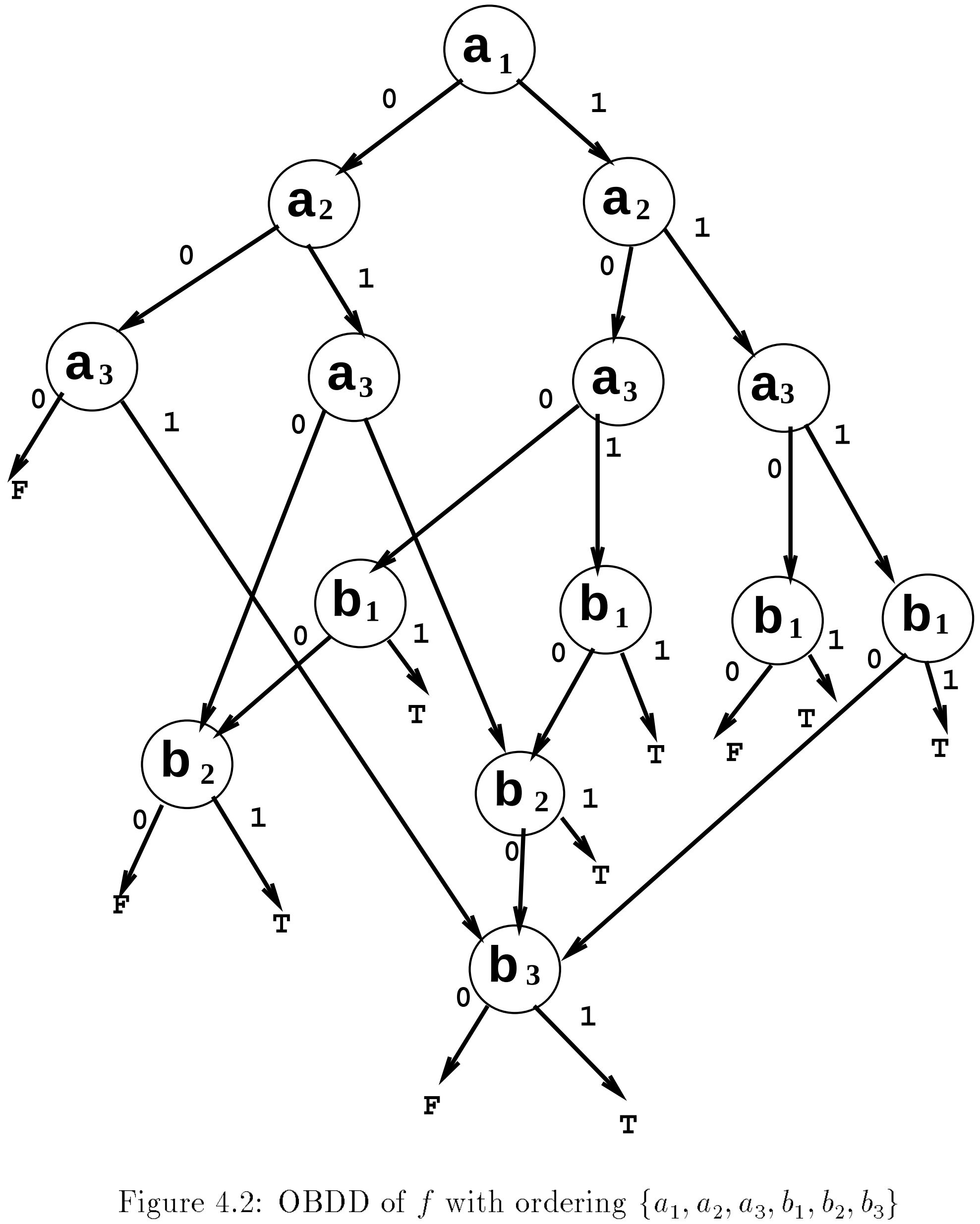


Figure 4.2: OBDD of f with ordering $\{a_1, a_2, a_3, b_1, b_2, b_3\}$

ordering to build small BDDs.

There are two broad directions along which these heuristics to solve the BDD ordering problem can be classified. In the first category are the approaches [MWBV88, Ber91, MIY90, JPHS91, FFK88, BRM91] that rely broadly on the idea that an input node further away from the output has a greater influence on the BDD size than an input node closer to the output. These approaches perform a static analysis over the topology of a multi-level implementation of the function. They define one or more attributes to capture the depth of nodes in the static structure, and then generate the orderings using these attribute(s). For instance, [MWBV88] uses node levels and transitive fan-in (TFI) depths as static attributes on a multi-level graph model of the implementation. The decreasing fanout count in the graph is used [FFK88] as a heuristic in an approach that generates orderings with the objective of minimizing the number of crossings of a net. An observability based approach is used in [MIY90]. Other approaches use techniques based on algebraic structure theory [JPHS91] and register allocation [Ber91]. All these strategies typically require a breadth or a depth first traversal of the network, and can consequently generate orderings in a short time. But on the other hand, they do not seem to be able to generate consistently good orderings. Butler *et al.* [BRM91] believe that a single algorithm might be inadequate for this purpose; so they use a suite of heuristics to solve the problem.

The second category consists of approaches that are more recent [FMK91, ISY91, MKR92]. They formulate the ordering problem as an optimization problem. For a given ordering a cost function is defined to estimate BDD sizes. Then starting with an initial ordering, and with the objective of minimizing the cost function, optimization techniques like simulated annealing and variable exchanging within a window are used to generate improved orders. These techniques are more exhaustive and they spend a lot more time examining different orderings before generating the best one. Thus they trade computation time for a more thorough search of the space of variable orderings. Consequently, they have yielded better BDD sizes than all the approaches in the first category.

In [PAS90] a slightly more fundamental approach is discussed where the variable ordering strategy is part of the factorization process. Starting with a two-level irredundant sum of primes form of the function, a procedure called *lexicographic factorization* is adopted. This procedure imposes a partial ordering on the input variables of the function while generating the multi-level factored form. The intuition behind using this factorization strategy is that if excessive factorization can be controlled by controlling input dependency, then it yields a structured logic cone implementation that prevents the wiring complexity from increasing across different cones in the implementation. The partial ordering of the input variables in the structured logic cone becomes the variable order for the BDDs. This ordering is further optimized with variable exchanging and then used to build the BDDs. We categorize this approach as more fundamental than the others because the variable orders are generated while keeping in mind how the 2^n cubes of the function combine with each other during the factorization. The approach has encountered problems while generating orderings for the larger, more complex benchmark circuit functions. This could possibly be due to the use of a greedy algorithm to generate the factors.

Our approach is similar to the first set of techniques in that it requires an existing circuit topology to find an ordering. But it differs from them in that it uses a dynamic attribute, *spatial entropy*, as a discriminator for variable ordering instead of a static circuit attribute. Such an attribute, we believe, can be related to a fundamental characteristic of the function. We elaborate on this in Chapter 5.

4.1.3 Motivation

Why do we think that spatial entropy can help generate good variable orders? A BDD is a representation for a boolean logic function. Its size is influenced by some characteristics of the literals and the minterms of the function and not by the implementation of the function. So when static attributes such as fanout depth or level (computed on an implementation of the function) are used to generate input variable orderings what characteristics of the literals and the minterms are they trying to capture and how do they use the implementation of the function for this purpose? The number of onterms and the spatial distribution of these onterms in the cube space of the function will influence the implementation of the function. The spatial distribution influences the degree to which the onterms can be combined to yield larger cubes. For instance in the multi-level (factored form) implementation of $f = a(b + dc(e + h'g)) + bi$ the literal a belongs to several onterms, not all of which can be combined. Hence it typically appears at the outermost position in the factored form representation of the function. In contrast, the few on-terms that literal g belongs to can be combined easily and it is nested deep inside the factored form. Since a factored form closely resembles a multi-level implementation of the function, the level or depth of nodes helps serve as a discriminator between input literals. So the depth or level of the nodes in the implementation is actually capturing some characteristic of the literals and minterms of the function that influence the BDD size. But the precise manner in which these literals and minterms influence the BDD size is still not clear. Hence this approach acts as a heuristic.

Ordering variables on static depth or distance (from the output) may not always help. This is because while the distance or depth measure in a static attribute may provide some information to discriminate between literals in a minterm, it may not capture sufficient information about how all the 2^n minterms of a given function combine with each other. Furthermore, when implementations fail to provide clear depth information it makes it more difficult to use static depth to discriminate between literals in different minterms. This is seen in a parallel-prefix adder where the least significant input bit that influences the maximum number of onterms is at the same static distance $O(\log n)$ from the output as the most significant input bit.

The spatial entropy attribute tries to capture the *spatial distribution of the minterms* in the cube space of the function. Like the other static attributes, it too is computed on an implementation of the boolean function. The attribute has two components. It relies on the entropy function H_w for information about the 2^n minterms of a function. H_w is probability based and it provides dynamic usage information that helps determine the influence of individual literals on the minterms. It relies on the distance attribute $l_{(w,w')}$ to capture the spatial distribution of these minterms in the cube space. When minterms (or cubes) are too far apart in cube space they cannot be combined to create larger cubes. The literals that make up these minterms/cubes also cannot be shared. As a result implementations of these minterms/cubes that are spatially apart in cube space will require wires or fanouts to bring them together in order to combine them. We also notice that in an implementation, wires are needed to bring together blocks of unevenly sized logic. This will also happen when unevenly sized cubes have to be combined. If the cubes themselves cannot be shared then wires would be needed in the implementation to stretch the output of the larger cube's implementation to combine it with the smaller cube. This issue and other issues related to function complexity are treated in further detail in Chapter 5.

Spatial entropy is attempting to capture the spatial distances that the cubes and minterms have to travel using its $l_{(w,w')}$ attribute. By combining usage with H_w and distances with $l_{(w,w')}$ the spatial entropy attribute is computing dynamic distances of the input nodes over all possible minterms. This we feel will yield a stronger discriminatory measure than static depth. For instance consider the addition function. Two static attributes - depth and area of different implementations (carry-ripple, carry-skip, carry-select and parallel-prefix) of an adder have a broad range of values. On the other hand the dynamic information about these circuits, such as average case delays to the output nodes, have a smaller variance [Tya91b]. This indicates that an attribute like spatial entropy that captures both the static structure and dynamic usage may have lesser possible variations for different implementations of the same function. Hence it might be a better discriminator for variable ordering than a static circuit attribute. In the next section we describe our approach of using spatial entropy to generate variable orderings.

4.2 Variable Ordering using Spatial Entropy

In Chapter 3 we described a procedure to compute the spatial entropy at the output nodes of a combinational logic function. Then in Section 3.5.1 we extended this procedure by characterizing the total spatial entropy of the circuit in terms of spatial entropy contributions of the primary inputs. This was done by adding an attribute called *spatial entropy vector* $\vec{S}$ to every node in the network. For a circuit with n primary inputs the spatial entropy vector at any node w, $\vec{S_w}$, is an n-element vector that reflects the contribution of each primary input to the spatial entropy value at node w. The spatial entropy at the node w is the cumulative sum of the spatial entropy contributions of each primary input.

$$S_w = \sum_{j=1}^{n} \vec{S_w}\,[x_j]$$

In the case of primary inputs, the spatial entropy vector at each input x_j, $\vec{S_{x_j}}$, consists of 0s at all input positions, except for position j, which contains a 1. This indicates that the spatial entropy at this node is 1 which is entirely contributed by the primary input x_j.

The spatial entropy vector at an internal node w is computed by distributing the local spatial entropy computed at the node amongst the primary inputs in the support set of every fanin of node w. For a node w with k fan-ins, $w_1, w_2, \ldots, w_k$, the spatial entropy contribution of primary input x_j at node w can be expressed as:

$$\overrightarrow{S_w}[x_j]=\overrightarrow{S_{inw}}[x_j] + \frac{\delta S_w}{S_{inw}} * \overrightarrow{S_{inw}}[x_j]$$

where δS_w is the local spatial entropy at node w, $\overrightarrow{S_{inw}}\ [x_j]$ is the total spatial entropy contribution of primary input x_j at the fanins of node w, and $S_{inw} = \sum_{j=1}^{n} \overrightarrow{S_{inw}}\ [x_j]$, *i.e.* the total spatial entropy over all the fanins of node w. This expression was discussed in detail in Section 3.5.1.

In order to generate an ordering for the primary inputs of a function using the spatial entropy vector ($\vec{S}$) at the output nodes, let us review the example in Section 3.5.1. The figure is reproduced in Figure 4.3 for easy reference. The circuit has a single output node n_{11}. The spatial entropy at this node is $S = 8.170$. This is expressed as the spatial entropy contribution of primary inputs $i_1, \ldots, i_4$ in the spatial entropy vector at n_{11} (`[1.659,1.659,3.531,1.321]`).

It may be recalled from Chapter 3 that spatial entropy was defined as the total information flow through the function, giving us a measure of the effort required to compute the function. The spatial entropy vector $\vec{S}$ at an output node contains the spatial entropy contributions of the primary inputs (at that node). This gives a measure of the effort required to compute the output function in terms of the primary input contributions. So from the vector `[1.659,1.659,3.531,1.321]` we find that inputs i_1, i_2 contribute equal amounts to the spatial entropy at n_{11}, i_3 contributes the most and i_4 contributes the least.

All our ordering strategies are based on the following premise. *A primary input that contributes less effort (or spatial entropy) towards the output function is less likely to influence the size of the BDD for the function. Likewise an input that contributes the most effort (or spatial entropy) is most likely to influence the size of the BDD.* A node that appears earlier in the ordering for a BDD has a greater influence on the size of the BDD. Hence an input with the highest spatial entropy contribution is given the highest position in the input order. Similarly an input that has less spatial entropy contribution

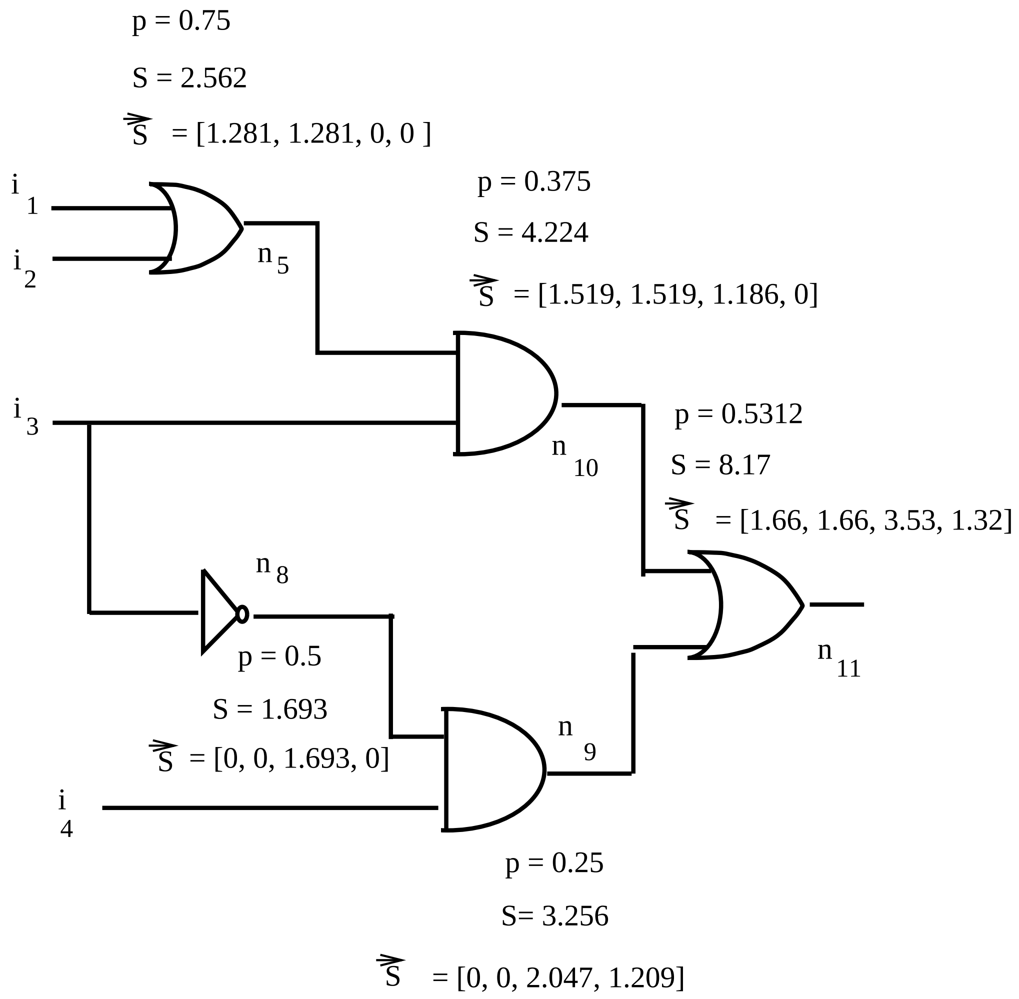

p = 0.75
S = 2.562
$\vec{S}$ = [1.281, 1.281, 0, 0]
i_1
i_2
n_5
p = 0.375
S = 4.224
$\vec{S}$ = [1.519, 1.519, 1.186, 0]
i_3
n_{10}
p = 0.5312
S = 8.17
$\vec{S}$ = [1.66, 1.66, 3.53, 1.32]
n_8
p = 0.5
S = 1.693
$\vec{S}$ = [0, 0, 1.693, 0]
n_{11}
n_9
i_4
p = 0.25
S= 3.256
$\vec{S}$ = [0, 0, 2.047, 1.209]

Figure 4.3: Example Spatial Entropy Calculation
$\forall\, i_j\ j = 1, 4,\ p_{i_j} = 0.5, S_{i_j} = 1.0,\ \vec{S_{i_j}}$ is a vector of zeros, with a 1.0 at position j

will appear lower down in the orderings. In Chapter 5 we develop a stronger basis for this premise.

For a single output function we can generate variable orderings by ordering the primary inputs in descending order of their spatial entropy contributions in the spatial entropy vector, $\vec{S}$, at the output node. But how does one generate variable orderings in a multi-output function? Consider a function with n inputs and m outputs. The effort contributed by primary input x_i in computing a given output function, o_j, is expressed by the spatial entropy vector at that output node $\vec{S_{o_j}}\,[x_i]$. So in order to determine the effort contributed by primary input x_i in computing all the m output functions, the m output spatial entropy vectors $\vec{S_{o_j}}$ $(j = 1, m)$ need to be combined. There are several ways in which this can be done. We have tested three approaches that we describe below.

Maximum: This approach is based on the intuition that an output with the highest spatial entropy will have maximum say in the variable ordering process. This is because the logic cone for the output node (with high spatial entropy) is contributing the maximum percentage of the total information flow over all the output functions. Hence the variable ordering required to build a BDD for this output function should have the maximum influence on the variable ordering required to build the BDD for all the output functions. Consider Figure 4.4 which illustrates an 8-input 3-output function with the distribution of spatial entropy values at each output. The spatial entropy or total information flow for the 3-output function is the cumulative sum of the spatial entropies of each output function. Since o_1 has the maximum contribution to this total information flow the spatial entropy vector associated with it should have the maximum influence on the ordering. This approach has also been used by some of the static attribute based heuristics to generate variable orderings; the difference is that instead of a dynamic attribute like spatial entropy, static depths and levels of the nodes in the logic cone were used.

The details of this strategy are as follows: First the primary output o_j with maximum spatial entropy (S_{o_j}) is considered. The input variables are ordered in descending order of their spatial entropy contributions

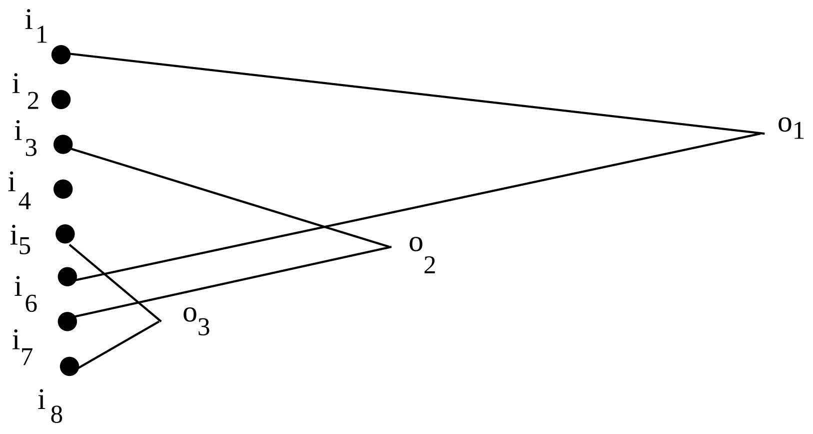


Figure 4.4: Spatial Entropy Distribution for a 3-output function (Maximum)

at o_j using the spatial entropy vector $\vec{S_{o_j}}$. So

$$\forall x, y \in I \; x \prec y \quad iff \quad \vec{S_{o_j}}[x] > \vec{S_{o_j}}[y] \tag{4.1}$$

In the above example the spatial entropy vector at output o_1 is examined first. This vector $\vec{S_{o_1}}$ will not order input variables that have equal spatial entropy contribution or zero spatial entropy contribution. The latter can happen when the input variables are not in o_j's support set, as is seen with i_7, i_8 in the above figure. As a result, the above ordering is really a partial ordering that partitions the input variable set I into a set E_{o_j} of ordered disjoint equivalence classes. For output o_j, $E_{o_j} = \{\alpha^j_{i_1}, \prec \alpha^j_{i_2} \prec \ldots \prec \alpha^j_{i_l}\}$ $l \leq n$ where each $\alpha^j_{i_k}$ consists of either a single input variable or a set of input variables such that

$$\forall x, y \in \alpha^j_{i_k} \;\; \vec{S_{o_j}}[x] = \vec{S_{o_j}}[y]$$

and

$$\forall x \in \alpha^j_{i_k}, \; \forall y \in \alpha^j_{i_{k'}} \; (S_{o_j}[x] > S_{o_j}[y]) \;\; \Rightarrow \;\; (\alpha^j_{i_k} \prec \alpha^j_{i_{k'}})$$

These classes, $\alpha^j_{i_k} \in E_{o_j}$, are then refined by consulting the spatial entropy vector of the output with the next-highest spatial entropy. In

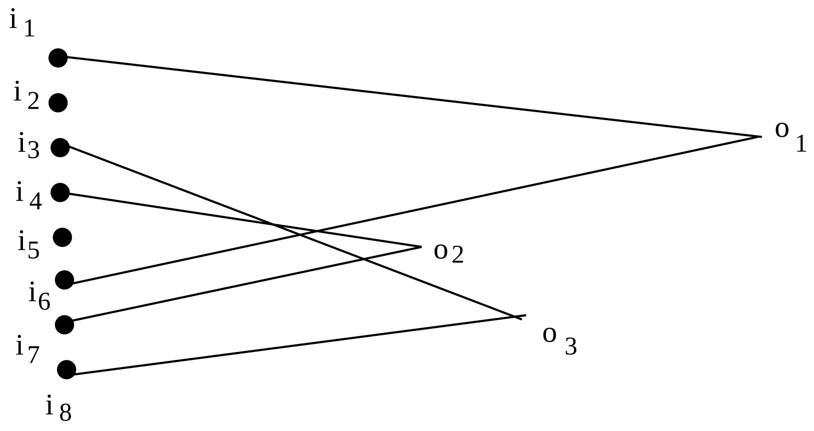


Figure 4.5: Spatial Entropy Distribution for a 3-output function (Weighted Multiply)

Figure 4.4, $\overrightarrow{S_{o_2}}$ is consulted and the ordering criteria in Equation 4.1 is used to generate a refined set of equivalence classes E_{o_2}. This process continues until either all the input variables are ordered, *i.e* $\mid \alpha^j_{i_k} \mid = 1, \forall \alpha^j_{i_k} \in E_{o_j}$, or the spatial entropy vectors of all the output nodes have been examined. In the latter case, if the input variables still appear together in an equivalence class ($\mid \alpha_{i_k} \mid > 1$) then this indicates that the relative order of the elements in α_{i_k} is not important.

Weighted Multiply: In the previous approach we generated orderings by giving maximum importance to the spatial entropy contributions of one output. This was the output with the maximum spatial entropy. The final ordering is thus very largely determined by this output function and its logic cone. But not all functions exhibit this scenario. For instance in some multi-output functions, an input that contributes heavily to one output may contribute very sparingly to all the other outputs. Consider Figure 4.5. Suppose, in the cone of output o_1, the spatial entropy contribution of input i_1 is greater than that of i_5, *i.e.* $\overrightarrow{S_{o_1}}[i_1] > \overrightarrow{S_{o_1}}[i_5]$. We notice that while i_1 does not contribute to any other output function, i_5 contributes to output functions o_2 and o_3 also. In such a situation, with respect to the multi-output function as a whole, input i_5 could be contributing greater cumulative effort than input i_1.

Hence a more global variable ordering strategy is needed.

While the spatial entropy vector at the output node with the highest spatial entropy must have a high influence on the final variable orderings, the spatial entropy vectors from the other outputs must also be able to influence the orderings. To do this for a given output o_j, every element in the spatial entropy vector, $\overrightarrow{S_{o_j}}[x_i]$, is weighted by multiplying it by the proportion of the spatial entropy that its output node o_j contributes to the total circuit spatial entropy, $\frac{S_{o_j}}{\sum S_{o_j}}$. Then the weighted spatial entropy vectors for all the outputs are added (in vector form) to obtain one single cumulative spatial entropy vector $\vec{S'}$. The contribution of input x_i in this vector is given by

$$\overrightarrow{S'[x_i]} \;=\; \sum_{j=1}^{m} \left[\overrightarrow{S_{o_j}[x_i]} * \frac{S_{o_j}}{\sum_{k=1}^{m} S_{o_k}} \right]$$

This strategy thus weighs the spatial entropy vector at each output in terms of the percentage contribution of that output's spatial entropy to the total circuit spatial entropy, and then accumulates the weighted vectors. The descending order of the contributions in this weighted vector determines the order of primary input variables for the BDD.

Weighted Divide: This strategy is similar to the weighted multiply except for the fact that instead of multiplying the elements of the spatial entropy vector by the output spatial entropy, we divide them by the output spatial entropy. Dividing each spatial entropy vector by its output spatial entropy normalizes all the spatial entropy vector contributions to the $[0, 1]$ range. This suppresses the amount of influence that each output spatial entropy value can exert due to its magnitude, since all the weighted vectors have a maximum value of 1. Thus it treats every output spatial entropy vector with equal importance. The effect of this is that a primary input that contributes a consistently high percentage to all the outputs of a multi-output function may appear higher in the ordering than an input that contributes a very high percentage to one output and negligible amounts to others. For example, given a pair of inputs where input i contributes a spatial entropy percentage of $0.2, 0.8$ and 0.1 towards output functions o_1, o_2, and o_3 respectively, and input

j contributes a spatial entropy of $0.5, 0.6$ and 0.5 respectively, the input j gets chosen as being more important ($1.6 > 1.1$).

In the next section we describe our experiment to study the use of spatial entropy vectors to generate variable orderings for BDDs.

4.3 Experiment: Objectives and Criteria

The primary objective of this experiment is to study the effectiveness of spatial entropy in generating good variable orders for BDDs. The criteria for good variable orders is determined by the size of the resultant BDD - the smaller the BDD size, the better the ordering. We compare the BDD sizes generated by the spatial entropy approach with those generated by other heuristics [Bra91, BRM91, ISY91, MKR92, PAS90]. The accuracy of the spatial entropy attribute, as defined in Chapter 3, is affected by factors such as logic minimization, wire length and reconvergent fanout. The study of the effect of some of these factors on variable orderings is another objective of this experiment. Finally we also study the different strategies that combine spatial entropy vectors. The results from this experiment will help evaluate the effectiveness of this attribute in generating good variable orders. In addition it will help characterize how the inaccuracies in computing this attribute affect variable orderings. We also hope that a study of the factors affecting spatial entropy will provide us with a better characterization of the spatial entropy attribute to assist its application in other areas. We begin by discussing our assumptions and limitations after which we describe the data set used for the experiment.

4.3.1 Assumptions and Limitations

Our first assumption is with respect to logic minimization. Computing the spatial entropy and the spatial entropy vector at the output nodes of a function requires an implementation. An unminimized implementation can affect the spatial entropy . Figure 4.6 shows how the spatial entropy can increase. The implementation that is not minimized will have a greater spatial entropy, due to the excess spatial entropy contribution from the additional gate. Even legitimate reasons for introducing redundancy, like delay reduction, are not

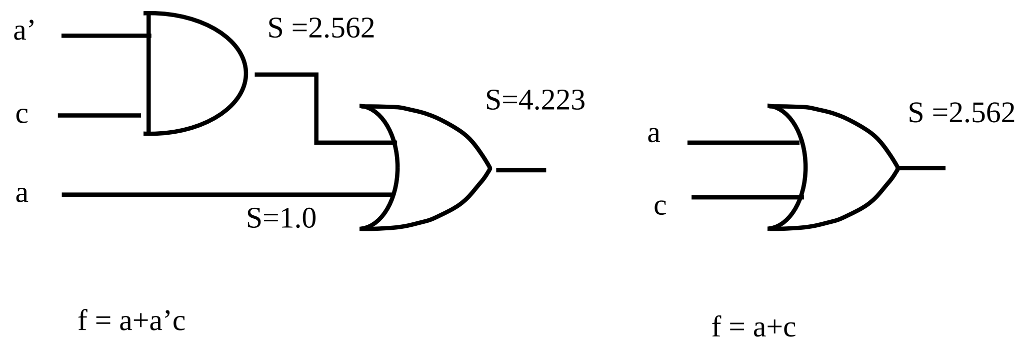


Figure 4.6: Two implementations of the same function with differing Spatial Entropy

desirable in this particular case since it artificially increases the spatial entropy of some nodes over others. Hence we require our circuits to be minimal in literal count before performing spatial entropy computation. In practice, since minimization is a heuristic we only have approximately minimal circuits. Minimizing the circuit by removing redundant literals will help reduce the spatial entropy, but the relationship between spatial entropy and logic minimization is not entirely straightforward. In larger complex functions the role of wire lengths becomes significant. Let us see how.

Given a two-level or a multi-level representation of a logic function, it can be minimized (for literal count) using logic-minimizers like `espresso` [BHMSV84] along with one of the minimization scripts in the UC, Berkeley MISII/SIS (Sequential Interactive System) system [BRSVW87]. But since exact multi-level logic minimization requires exponential time [Law64], most minimizers use heuristics to find "near-optimal" solutions. The optimality of such a solution is usually measured with respect to the minimal literal count in the multi-level implementation. The number of literals influences the number of nodes in the implementation. Since spatial entropy is computed over all the nodes in the implementation it is a function of this literal count measure.

But minimizing literal count does not necessarily imply that the excess spatial entropy (due to logic that is not minimized) is being removed. This is because wire lengths in the layout of such an implementation can reverse this effect. Figure 4.7 illustrates this in a very simplified form. `Version 1` factors out the expression $c+d$, and uses an extra fanout with a long wire to achieve the function. In contrast `Version 2` avoids the factoring and replicates the gate implementing $c + d$. In the process it saves on the long wire. While the redundant logic will contribute to the spatial entropy in `Version 2`, its absence in `Version 1` is offset by the long wire. This makes it difficult to evaluate the effectiveness of the multi-level logic minimizers and minimization scripts in completely solving the problem of generating implementations with minimum spatial entropy. With this scenario we assume for the present that minimizing literal count does remove excess spatial entropy, and we adopt the minimization script that does the best job in minimizing the literal count wherever possible.

Our next assumption deals with reconvergent fanout. In Chapter 3 we had characterized the effect of reconvergent fanout on spatial entropy, and outlined approaches to remove errors in spatial entropy due to reconvergent fanout. Almost all VLSI circuits are reconvergent. Removing errors in spatial entropy values in such circuits is an exponential time procedure. Hence, for most of the data in our experiments we elect to compute spatial entropy by ignoring the effect of errors due to reconvergent fanout.

Finally we discuss wire lengths. The effects of wire length on spatial entropy were described earlier in Chapter 3. We had first discussed inaccuracies due to the absence of wire lengths and then followed it by discussing inaccuracies in the distribution of the lumped wire length along multiple-fanout nodes. What role do wire lengths play in variable ordering? In Section 4.1.3 we had conjectured that the length attribute in the spatial entropy definition captures the spatial distance that cubes and sets of cubes have to travel in the implementation while computing the function. This distance is reflected in the implementation by wires that connect one set of gates (cubes) with another. One way of computing this wire length is to extract lengths from the schematic of a gate level netlist. But schematic generation for large netlists is non-trivial, and such tools are not easily available. The problem of gate placement and interconnection in a netlist schematic has similarities with the placement of cells in a layout synthesis system. We had access to a standard cell place and route system called VPNR (vanilla place-and-route) as part of the MCNC OASIS Silicon Compiler [KB88]. So we decided to use layouts from VPNR to extract our wire length estimates. This decision needs some justification.

A netlist at the gate level can be realized at the layout level by a large set

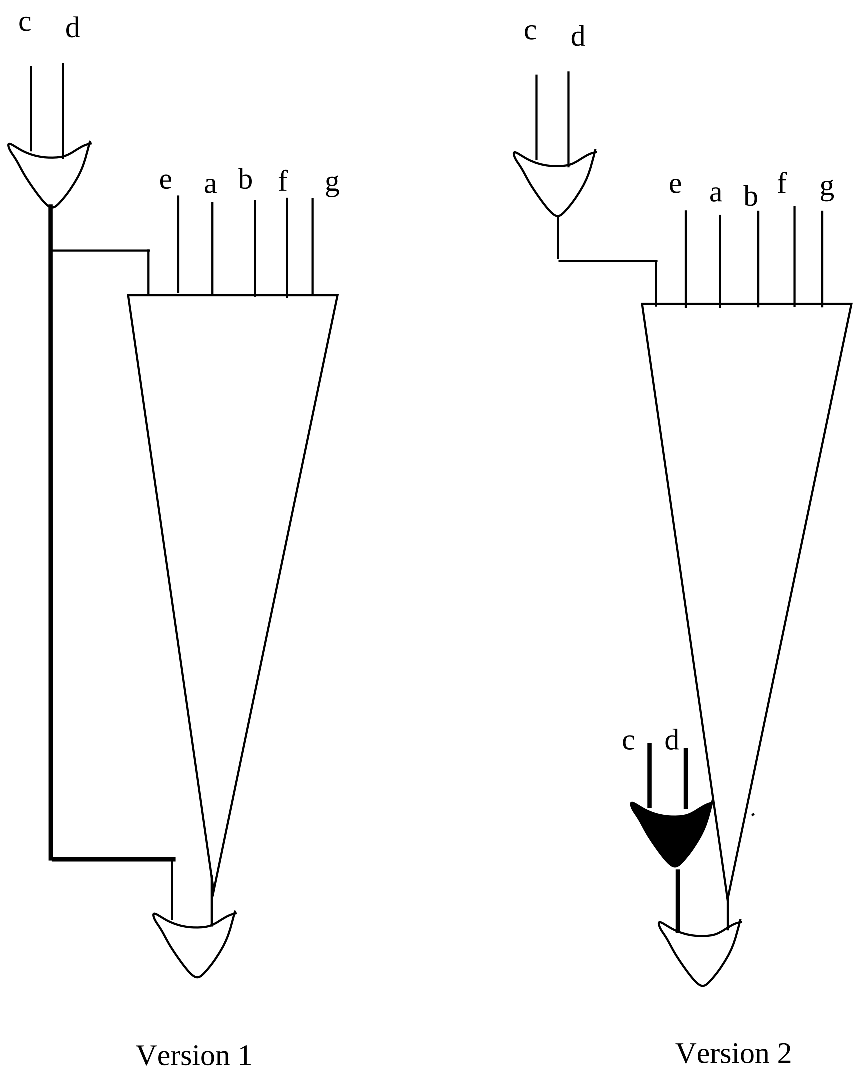


Figure 4.7: The tradeoff between redundancy and wire length

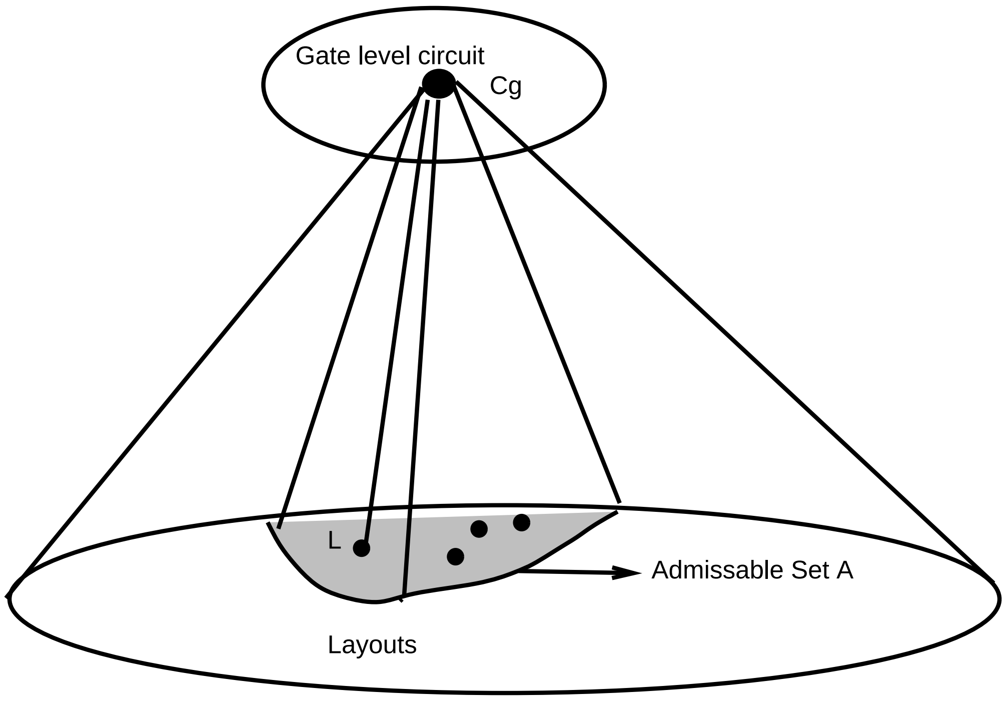


Figure 4.8: Mapping of Gate Level Circuits at the Layout Level

of layouts. All the layouts in this set will be functionally the same as the gate level implementation and will have the same set of nodes and gates. But the physical layouts will differ in other characteristics. The layout styles used to realize the gates (standard cell, full custom, gate array) may be different. The layouts may differ in the total area, the placement and routing between the cells, the number of layers in which routing is done, the number of wires, and the the length of these wires. These differences arise due to the different layout methodologies and styles available and the different criteria that they use while realizing the layout (minimize total area, total wire length, total number of vias *etc.*). We are only interested in a subset of the set of all feasible layouts called the **admissible set**, A. This is the set of layouts that yield a placement and interconnection pattern that is similar to our ideal schematic placement. Figure 4.8 captures this scenario. The experiment proposes to use the wire lengths extracted from one such admissible layout $L \in A$ of the gate level circuit C_g.

We also assume that the wire lengths of internal nodes in a layout correlate well with those in a gate-level schematic. By this we mean that for layouts $L_1, L_2 \in A$, $l_w^{L_1} > l_v^{L_1} \Rightarrow l_w^{L_2} > l_v^{L_2}$, where $l_w^{L_1}$ is the length of wire w in layout L_1. It is this relative ordering that we wish to capture. This is also argued with some empirical evidence. In [RT90] the VPNR place-and-route system was used to generate layouts for regular data path functions like adders, and shifters. It was shown (with statistical regression analysis) that VPNR generated layouts were a good fit to the expected analytical area of these functions. The coefficient of variance between the analytical curve and the area plots was within 10% for most cases. Since area is a function of wire length, this correlation ought to hold with respect to wire lengths too. This correlation can be generalized to random-logic circuits because the VPNR place-and-route system performs its placement and generates its wires without any knowledge of the circuit function. Hence its behavior should not be expected to change with circuit function. It must be emphasized here that extracted wire lengths from a layout are *not* necessary for this experiment. They just happen to be a convenient source for obtaining wire length estimates between gates in a circuit.

The next assumption we make is with respect to the process of wire length extraction itself. The layout synthesis system and the extraction software force us to make approximations. While the errors due to these approximations will not give us the exact wire length in the layout, our hope is that they are small enough not to affect the relative wire length distribution.

The first approximation is with respect to *point-to-point* versus *lumped* wire lengths. Since we are primarily interested in interconnect length we wish to obtain length estimates for only the wires used in the routing between cells in a layout. Wire lengths of layers inside a standard cell (gate) are not of interest to us. For a node that fans out to several other nodes ideally we wish to obtain *point-to-point* interconnect lengths for each individual fanout wire. This is difficult to do because of two reasons. Firstly multi-point nets in a layout are usually routed using a minimal Steiner tree that inserts additional points in the route that do not belong to the net. This makes point-to-point estimates difficult. This also makes it difficult for the the extraction software to extract wire lengths between nodes or points on a net. As a result we substitute point-to-point lengths by estimates of total *lumped* wire length at a node.

Our second approximation is with respect to the layers that constitute the lumped wire length estimate. The extraction software extracts area and not wire length. Dividing this area estimate by a constant factor should give us length information, since most rectangles in the layout have a fairly uniform width (3-4λ). The extractor generates the area of the rectangles by extracting the areas of all the different layers at a given node. This includes `contacts, metal1, metal2, poly, (n and p) diff` *etc.*. Assuming we can suppress area extraction inside a cell, we would still be including areas of several contacts in the inter-cell routing. We assume that retaining these areas along with the areas of the layers actually used for routing will not perturb the wire length estimate enormously. To summarize, our three main assumptions are

- We assume that aggressive literal count minimization helps reduce excess spatial entropy in redundant circuits.
- Reconvergent fanout removal is ignored.
- An estimate of the point-to-point interconnect wire length in a gate level netlist is obtained by using the *lumped* area of all inter-cell layers electrically connected at a given node in a (VPNR) layout of that netlist.

The primary limitation of our experimental set up is that it is difficult to evaluate the magnitude of our approximations in spatial entropy computation. The only way to determine if an approximation in the spatial entropy computation is good or bad is by generating variable orders, building BDD sizes, and comparing them. The fact that many approximations/assumptions (like redundancy and wire length) are interdependent also makes it difficult to isolate an approximation and characterize it separately. Two other limitations are the difficulty of computing exact spatial entropies in highly reconvergent circuits with large supergates, and the lack of more precise point-to-point wire lengths between nodes. The effect of redundancy on spatial entropy is also not very clear. The spatial entropy is a function of wire lengths. Although the effect of different minimization scripts on wiring area was documented in [Sau92], it is is not clear how the use of minimization scripts to remove redundancies will affect the wire length and what effect that has on spatial entropy.

4.3.2 Data Set

The requirements of the data set for this experiment are as follows. Firstly the data set should consist of benchmark circuits that can be used to compare the performance of the spatial entropy based BDD ordering with other approaches. The data set should also contain circuits that represent regular structured logic and random combinational logic to evaluate BDD sizes in these two domains. Finally it must contain small and large circuits; small circuits where the effects of the factors affecting spatial entropy can be studied more closely and large circuits to ensure that orderings for non-trivial circuit examples can also be obtained.

To satisfy these requirements, our data set consists of three classes of circuits. In all three classes we use the MCNC VPNR format [KB88] to represent the multi-level netlist as input to the spatial entropy computation procedure, `spent`. The VPNR format is a circuit description format that describes a technology mapped netlist in terms of the standard cells in the MCNC standard cell library.

ISCAS85 Circuits: This data set consists of multi-level circuits from the ISCAS 85 benchmarks. There are several large circuits here which can have significantly differing BDD sizes highlighting good and bad orderings. The circuits have well-defined characteristics - number of nodes, number of inputs, number of outputs. There is also a mix of circuit functions - adder, priority encoder, random logic, ALUs *etc.*. These circuits have also been used as the primary data set for comparing various BDD ordering strategies for combinational circuits. We use the UC Berkeley MISII/SIS system [BRSVW87] to further minimize these circuits and then translate them into the VPNR format. Table 4.1 illustrates some characteristics of these circuits.

LgSynth91 Circuits: These are benchmark circuits compiled at the MCNC Logic Synthesis Workshop (in May 1991). These circuits are in two-level and multi-level form. The circuits are a lot smaller than the ISCAS85 circuits giving us an opportunity to study the effects of factors affecting spatial entropy. They have also been used to compare BDD ordering strategies. They too consist of a mix of random and structured logic. The 2-level form of the circuits gives us an opportunity to study the effect of the different minimization scripts in SIS on

Circuit Name	Circuit Function	Inputs	Outputs	Total Gates
C432	Priority Decoder	36	7	160
C499[2]	ECAT[1]	41	32	202
C880	ALU and Control	60	26	383
C1355[2]	ECAT	41	32	546
C1908	ECAT	33	25	880
C3540	ALU and Control	50	22	1669

Table 4.1: ISCAS 85 Benchmark Circuit Characteristics

[1]ECAT stands for Error Correcting and Transmission.

[2]Circuits C499 and C1355 are functionally equivalent.

spatial entropy computation and variable ordering. The final netlist form is obtained by using SIS to minimize the circuit and then translating it to the VPNR format. The circuit characteristics are described in Table 4.2 and Table 4.3. The functions of the two-level circuits are unknown.

OASIS Generated Circuits: These are circuits generated using the OASIS silicon compiler tool set [KB88]. They are primarily circuit descriptions of regular structured circuits in various sizes. This includes adders, decoders, encoders, multiplexors, multipliers, shifters, and counters, with bit widths varying from 4 to 128. The circuits were described in LOGIC-III and OASIS was used to generated the VPNR gate-level netlist. The multi-level minimization strategy used here was MCNC's DECAF [LKB87], and not the Berkeley SIS scripts. This is because DECAF is part of the OASIS system. This data set helps characterize the behavior of the spatial entropy procedure on regular structured circuits. Their characteristics are described in Table 4.4.

Circuit Name	Circuit Function	Inputs	Outputs	Total Gates
alu2	ALU	10	6	335
alu4	ALU	14	8	681
count	Counter	35	16	143
f51m	Arithmetic	8	8	43
frg1	Logic	28	3	105
z4ml	2-bit Add	7	4	20

Table 4.2: Multi-level Circuit Characteristics from Logic Synthesis 91 Benchmarks

Circuit Name	Inputs	Outputs	Product Terms
5xp1	7	10	75
bw	5	28	87
clip	9	5	167
duke2	22	29	87
misex1	8	7	32
misex2	25	18	29
rd53	5	3	32
rd73	7	3	141
rd84	8	4	256
sao2	10	4	58
vg2	25	8	110

Table 4.3: Two-level Circuit Characteristics of Logic Synthesis 91 Benchmarks

Circuit Name	Circuit Function	Inputs	Outputs	Total Gates
mux32	32-to-1 Mux	32	1	191
mux64	64-to-1 Mux	64	1	384
dec32	5-to-32 Dec	5	32	65
dec64	6-to-64 Dec	6	64	130
bshift16	16-bit Barrel shifter	16	16	388
bshift32	32-bit Barrel shifter	32	32	965
ctr16	16-bit Counter	16	16	166
ctr32	32-bit Counter	32	32	326
ctr64	64-bit Counter	64	64	646
enc16	16-to-4 Encoder	16	4	102
enc32	32-to-5 Encoder	32	5	210
mult4	4-bit Multiplier	8	8	112
mult8	8-bit Multiplier	16	16	472

Table 4.4: OASIS generated Structured Circuit Characteristics

To summarize, the data set consists of about 36 example circuits more than 2/3rds of which are benchmark circuits. The remainder are structured circuits, generated by OASIS, that exhibit a more regular communication behavior. The circuits vary widely in size. The inputs range from 5 to 64 and the outputs range from 1 to 64. This yields a wide range of possible BDD sizes, from 2^5 to 2^{64}, in the worst-case. The circuit sizes range from 20 gates to approximately 2000 gates. The circuit function is also well distributed. There are datapath structures like ALUs, shifters, counters, multipliers, *etc.*; bus-like structures for select, transmitting and encoding data, like multiplexors, decoders, and encoders. Finally there is random glue logic of varying size.

4.4 Experiment Outline

Our objectives are three-fold. Firstly we would like to see how the spatial entropy attribute performs in generating variable orders for BDDs. For this we compare it with the other approaches: those based on static attributes and those based on optimization techniques. Secondly we would like to analyze the effect of some of the factors affecting spatial entropy by computing spatial entropy with different approximations. Finally we would like to compare the various strategies of combining the output spatial entropy vectors in multi-output functions. All these goals require the generation of BDDs using the spatial entropy approach. The experimental set up described in Figure 4.9 shows how this is done. The oval boxes indicate software while the rectangular boxes indicate data. The orderings generated by the spatial entropy computation procedure, `spent`, are fed to a BDD builder, `vpnr2bdd`, to generate BDD sizes. The orderings that we wish to compare with are also built using the same BDD builder. This is done in order to remove implementation dependent discrepancies in BDD sizes of the different approaches. We now describe the software construction for the experiment, and then outline the variables measured.

4.4.1 Software Construction and Variables Measured

As Figure 4.9 indicates the experiment required 3 pieces of software. The spatial entropy computation program `spent` was written to accept the input circuit description, compute the spatial entropy and spatial entropy vector for

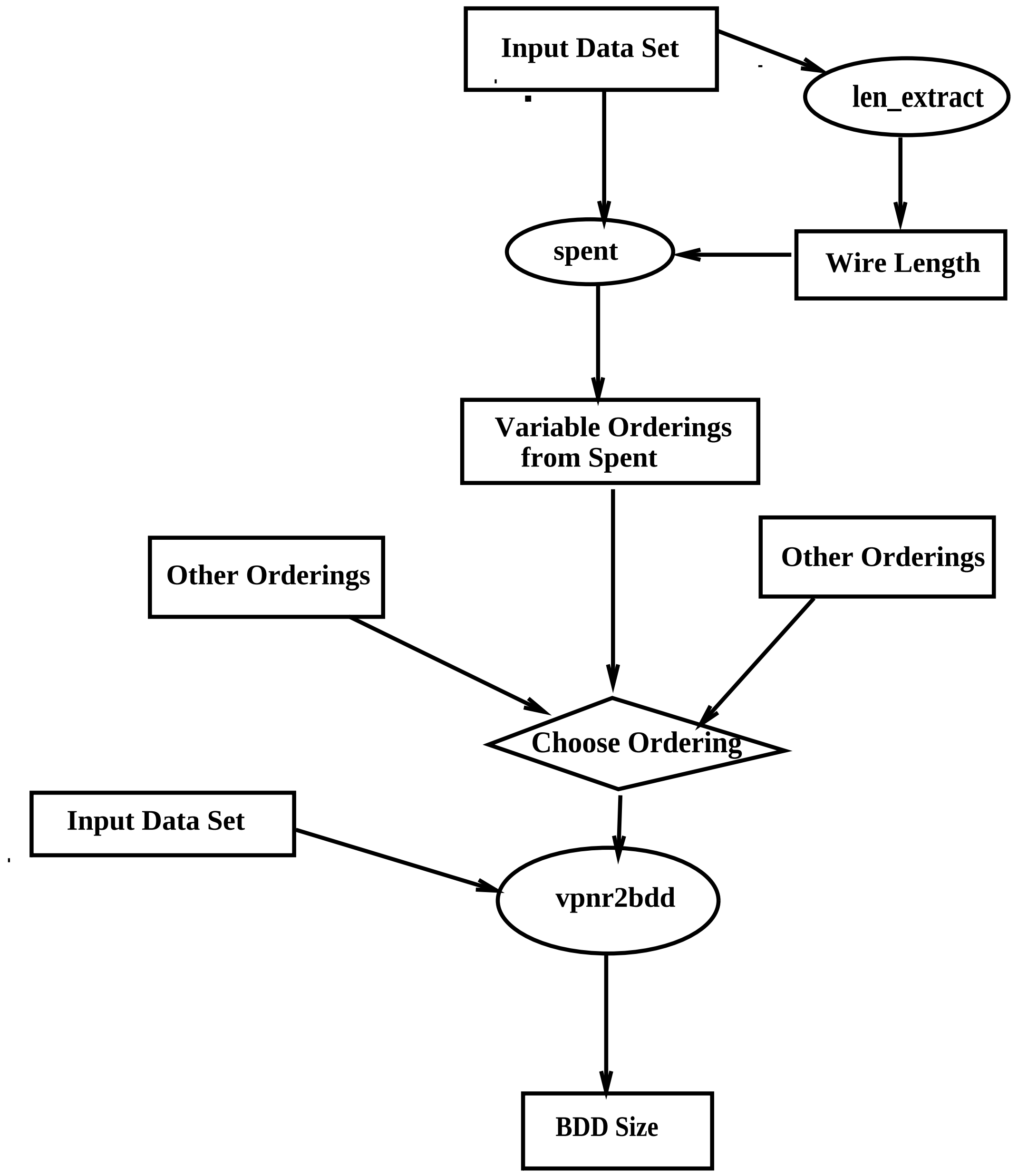


Figure 4.9: Experimental Setup

the nodes in the circuit, and generate variable orders for the primary inputs of the circuit using the different strategies discussed in Section 4.2. In order to incorporate wire length information during spatial entropy computation, wire lengths were extracted by `len_extract` and then fed to `spent`. Finally a program called `vpnr2bdd` was employed [Ree91] to build BDDs for the input circuit descriptions in VPNR using either the generated variable orderings from `spent`, or orderings contributed from other sources.

The criteria used to measure performance in this experiment was the size of the built BDD. Hence it was the main variable measured for the different circuits in the data sets.

4.5 Results and Observations

We present our results in three phases along the lines of the three objectives of the experiment. In the first phase we compare spatial entropy based BDD ordering with static attribute based approaches and optimization-intensive strategies. The same BDD builder, `vpnr2bdd`, was used to build BDDs for all the orderings. The BDD sizes are expressed as number of nodes. In the second phase we present results that illustrate the effects of some of the approximations - redundancy, wire length and reconvergent fanout, on BDD sizes. Results for all the 3 data sets are discussed here. Finally we present results comparing the three different strategies of combining spatial entropy vectors of multi-output functions to generate variable orders. All the experiments were run on a DECStation 5500 (32 MB) running Ultrix Version 4.2A.

4.5.1 Spatial Entropy and BDD Sizes

We illustrate our results with respect to two data sets. Table 4.5 compares BDD sizes generated by `spent`, for the MCNC Logic Synthesis 91 benchmark circuits, with 2 other sources. These sources were chosen because they are representative of related approaches to BDD variable ordering. They were also the only sources known to us that had generated orderings for this set of benchmarks. Finally these ordering sources are very recent giving us an opportunity to compare `spent` with the latest results.

`minwid91`: Uses the optimization intensive approach described in [ISY91]. Optimization is performed by improving an initial ordering with variable exchanges while using the BDD width as a cost function. Ordering source - [Min92].

`asyl90`: Uses the approach in [PAS90] where lexicographic partial ordering is followed by optimization via variable exchanging within a window to improve the size. Ordering source - [BA92].

`spent`: This is the spatial entropy based approach. Here we show the best sizes achieved over all approximations. In Section 4.5.2 and Section 4.5.3 we describe the effect of the various approximations in computing spatial entropy and the different strategies of combining the spatial entropy vector .

We have selected only those circuits from the Logic Synthesis 91 benchmarks for which orderings were available from both `minwid91` and `asyl90` sources.

Observations

We begin by comparing the optimization intensive approaches (`minwid91, asyl90`) to the spatial entropy based approach for the Logic Synthesis 91 benchmarks. In addition to Table 4.5 that illustrates the actual BDD sizes, in Table 4.6(page 86) we study the mean BDD size $\overline{\mid V \mid}$ (of these circuits) where V is the number of nodes in the BDD. For each of the three approaches we compare $\overline{\mid V \mid}$ with $\overline{\mid V \mid}_{smallest}$, where $\overline{\mid V \mid}_{smallest}$ is computed by taking the smallest BDD size for each circuit and averaging it over all the (17) circuits. The standard deviation $\sigma_{17-1}(\mid V \mid)$ of the sample is also shown.

We note that the average BDD size obtained from the `asyl90` approach is very close to the average obtained from the smallest BDD size. This is also borne out by the fact that the `asyl90` approach yields the best (smallest) BDD sizes in almost all the circuits. The `midwid91` and `spent` approaches are comparable to each other since they yield approximately the same number of nodes on the average $(176.94, 171.64)$. The standard deviation values also reflect this observation. The other observation that we can make from the standard deviation values is that the distribution of the BDD sizes about the mean is not very wide. This is a reflection of the circuit characteristics in the data set. For `asyl90` the smallest BDD has 25 nodes while the largest has

Circuit	# of Nodes		
Name	`minwid91`	`asyl90`	`spent`
alu2	222	204	198
alu4	1190	718	945
count	128	104	131
f51m	71	71	72
frg1	134	96	141
z4ml	28	33	33
5xp1	70	70	80
bw	114	97	112
clip	112	123	112
duke2	445	403	457
misex1	47	42	44
misex2	123	92	127
rd53	25	25	25
rd73	45	45	45
rd84	61	61	61
sao2	93	92	115
vg2	100	90	220

Table 4.5: Comparative BDD sizes for Logic Synthesis 91 Benchmarks

Statistic	Mean and Standard Deviation of BDD sizes			
	`minwid91`	`asyl90`	`spent`	`smallest`
$\overline{\lvert V\rvert}$	176.94	139.17	171.64	137.88
$\sigma_{16}(\mid V \mid)$	278.85	173.03	223.75	173.17

Table 4.6: Comparative Statistics for Logic Synthesis 91 Benchmarks

718 nodes. In fact, if `alu4` is excluded then the range of BDD sizes drops further ($25 - 500$).

While the `asyl90` approach definitely has the best performance for this data set, we would now like to see how the `spent` and `minwid91` approaches compare with it. Table 4.7(page 88) shows the normalized BDD sizes for the `spent` and `minwid91` approaches computed as a fraction of the sizes generated using `asyl90`. To analyze this better we compare average performance and standard deviation of these approaches with respect to that of `asyl90`. Since the BDD sizes are normalized with respect to `asyl90` sizes, we are comparing against a mean of 1.0 and a standard deviation of 0.0. The `minwid91` approach has an average normalized BDD size of 1.14, while `spent` has an average normalized BDD size of 1.19. The standard deviations are 0.196 for `minwid91`, and 0.35 for `spent`. While this indicates that these approaches yield about $15 - 20\%$ poorer sizes, we also notice that the circuit with the worst size in each of these approaches has a significant influence on these statistics. If we dropped `alu4` from `minwid91` and `vg2` from `spent`, the average normalized sizes drops for `minwid91` ($1.14 \rightarrow 1.08$), and for `spent` ($1.19 \rightarrow 1.12$). The standard deviation values drop too ($0.19 \rightarrow 0.14$ and $0.35 \rightarrow 0.16$ respectively). So we can conclude that for the Logic Synthesis benchmarks, the spatial entropy approach yields sizes that are within $10 - 12\%$ of the best size. Some of the circuits for which the spatial entropy approach yields poor sizes, *viz.* `vg2, frg1` are examined in greater detail when we study approximations in the spatial entropy computation. We now look at results for the ISCAS 85 benchmarks. Since we do not have comparative sizes for the OASIS generated circuits from other approaches we do not show results for them here. We will however discuss the BDD sizes for these circuits when we study the effects of spatial entropy approximations on BDD sizes.

Table 4.8 shows the BDD sizes for the ISCAS 85 benchmark circuits with orderings generated from 5 sources described below. The 4 other sources that `spent` was compared with with were chosen because, amongst the orderings that we had access to, they represented the most recent results on variable ordering for the ISCAS 85 benchmark circuits. Also some of them represent the best achieved orderings so far. In addition we were unable to obtain orderings for the `asyl90` approach [1] that did so well in the Logic Synthesis

[1]We were informed that they did not have good sizes for the ISCAS 85 circuits

Circuit	Normalized w.r.t. `asyl90`	
Name	`minwid91`	`spent`
alu2	1.08	0.97
alu4	1.65	1.31
count	1.23	1.26
f51m	1.0	1.01
frg1	1.39	1.46
z4ml	0.85	1.0
5xp1	1.0	1.14
bw	1.17	1.15
clip	0.91	0.91
duke2	1.10	1.13
misex1	1.11	1.04
misex2	1.33	1.38
rd53	1.0	1.0
rd73	1.0	1.0
rd84	1.0	1.0
sao2	1.01	1.25
vg2	1.11	2.44

Table 4.7: BDD sizes for Logic Synthesis 91 Benchmarks relative to `asyl90`

Circuit	# of Nodes				
Name	`cmu91`	`uta91`	`minwid91`	`uta92`	`spent`
c432	32,378	1,850	1,547	1,294	1,817
c499	52,961	35,461	42,367	34,977	37,764
c880	7,613	7,749	9,077	5,940	6,774
c1908	17,423	18,376	8,870	11,471	22,488
c1355	52,961	35,461	42,367	34,977	37,764
c3540	$> 600,000$	403,000	53,434	52,708	456,190

Table 4.8: Comparative BDD sizes for ISCAS 85 Benchmarks

benchmark set. The 5 sources are:

`cmu91`: Orderings from static attribute based heuristics discussed in [BBR89]. (Source - [Bra91]).

`uta91`: Orderings using a collection of several static attribute based heuristics [BRM91]. (Source - [Kap91]). The orderings in `uta91` are the best known orderings based on static-attribute sources.

`minwid91`: Orderings generated using an optimization intensive approach discussed above.

`uta92`: Orderings also generated using an optimization intensive approach. In this case ordered partial decision diagrams OPDDs [RBKM91] were used as cost functions and simulated annealing was used to conduct the optimization [MKR92]. (Source - [MKR92]).

`spent`: Spatial entropy based ordering.

We do not show the time to generate the spatial entropy based orderings for the individual circuits because they are not critical. As an example, for the largest circuit `c3540` the spatial entropy based approach took 257 seconds to generate orderings.

Observations

For these much larger ISCAS 85 benchmark circuits we note that the optimization intensive approaches `minwid91, uta92` are clearly superior and yield the smallest BDD sizes. The exponential manner in which the size of a BDD grows makes these effects markedly visible. The primary advantage that the optimization intensive approaches `minwid91, uta92` have over the static attribute based approaches `cmu91 uta91` and the spatial entropy based approach (`spent`) is the fact that they generate their orderings while the BDD is being built. On the other hand the other two approaches rely on an implementation for information about the function, making approximations in the process. The static attribute based approach uses node levels and depths as approximations, while the spatial entropy based approach makes approximations while computing the spatial entropy attribute. These approximations are responsible for the poorer orderings and poorer sizes. On the other hand, the drawback with the optimization approaches is that they take a lot longer to run since they make a through search of the space of $n!$ orderings. To cite a comparison, the spatial entropy approach took 80.42 seconds, on a DEC Station 5500, to generate an ordering for the ISCAS circuit c499. In contrast the `minwid91` approach was documented [ISY91] as taking approximately 35,000 seconds on a SPARC Station 1+.

We now compare the static attribute based approaches with the spatial entropy based approach. In Table 4.9 (page 91) we show how the the static attribute based approaches compare with the the spatial entropy approach by expressing the BDD size for each approach as a fraction of the best size generated for that circuit. The entries with a 1.0 indicate the best size for that circuit. The approaches `uta91` and `spent` yield better sizes for all but one circuit. Table 4.10 (page 92) illustrates the mean and standard deviation of these normalized BDD sizes. These are being compared with a mean of 1.0 and a standard deviation of 0.0. We note that while the approach `cmu91` does poorly, the static based approach `uta91` and `spent` are quite comparable. Their average normalized BDD sizes 1.04, 1.09 are within 5% of each other indicating comparable performance. The standard deviations too reflect this. `uta91` is an approach that uses a collection of static attribute based heuristics and this seems to help it do better. They use a combination of breadth-first and depth-first traversal options on topological circuit representations, with a combination of several heuristics [MWBV88,

Circuit Name	Normalized # of Nodes		
	`cmu91`	`uta91`	`spent`
c432	17.82	1.02	1.0
c499	14.93	1.0	1.06
c880	1.12	1.14	1.0
c1908	1.0	1.05	1.29
c1355	14.93	1.0	1.06
c3540	-	1.0	1.13

Table 4.9: Spatial entropy based ordering vs static-attribute based approaches

FFK88, MIY90]. This smorgasbord of strategies gives this approach greater applicability than `cmu91`, which does not have such uniform results.

The spatial entropy based approach appears to have the ability to capture the characteristics and communication pattern of several function types within the single unified framework of spatial entropy vectors. For instance it was observed that the distribution of spatial entropy amongst the output functions was different in each of these circuits. While `c499` showed an almost uniform spatial entropy distribution at its respective output functions, `c880, c432` showed a gradual progression of spatial entropy values (from low to high) over the individual outputs, and `c1908` had some outputs with uniform spatial entropy values and others with a progression of values. While this single algorithm yields competitive results while the BDD sizes are small, the approximations performed in computing spatial entropy unfortunately get more and more significant with larger circuits. With BDD sizes increasing exponentially for poor orderings, the sizes rapidly deteriorate. This leaves very little margin for inaccuracy. We now study the effects of some of these approximations on spatial entropy computation, the orderings, and the consequent BDD sizes.

4.5.2 Spatial Entropy Approximations and BDD Sizes

Wirelength:
Table 4.11 illustrates the effects of wire length approximation. For this we

Statistic	Mean & Standard Deviation of Normalized sizes		
	`cmu91`	`uta91`	`spent`
$\overline{\lvert V \rvert}$	8.63	1.04	1.09
$\sigma_5(\lvert V \rvert)$	8.0	0.05	0.1

Table 4.10: Normalized Mean and Standard Deviation of sizes for ISCAS85 Benchmarks

show BDD sizes for OASIS generated circuits. The orderings were generated using the spatial entropy approach with three wire length approximations described by three columns in the table. The first describes sizes generated *without* using wire length in spatial entropy computation. The second uses *lumped* wire length, while the third uses a *level-based* approach to distribute the lumped wire length at a node in proportion to the levels of the fanout nodes. We also illustrate the average BDD size and standard deviation in BDD size for each of the three wire length approximation strategies (Table 4.12, page 93).

The BDD sizes for each of the circuits and the average size statistics indicate that wire length information is beneficial. While lumped wire length does not have a significant effect, the level based wire length approach improves the average BDD size from ~ 1646 to ~ 1423 nodes. The standard deviation values also improve with wire length information, but they are relatively high. This is because of the characteristics of the data set. The 8-bit multiplier has a significant effect on this statistic. Computed without the `mult8` sizes the standard deviations reduce by almost a factor of 4, from $4274 \rightarrow 1180$ for no wire length, $4286 \rightarrow 1106$ for lumped wire length, and $3945 \rightarrow 705$ for level based wire length. The means sizes drop too. From $1645 \rightarrow 544$ for no wire length, $1624 \rightarrow 514$ for lumped wire length, and $1423 \rightarrow 384$ for level based wire length. Thus in the absence of the 8-bit multiplier this collection of circuits builds medium sized BDDs, upto about 4000 nodes.

While we observe that for most of these structured circuits wire length approximation in the spatial entropy computation does help improve the orderings, the amount of improvement varies from circuit to circuit. Table 4.13 (page 94) shows the percentage improvement using wire length approxima-

Circuit Name	# of Nodes		
	No W/length	Lumped W/length	Level based W/length
mux32	67	65*	76
mux64	131	129*	152
dec32	64	64	64
dec64	128	128	128
bshift16	100	82*	82*
bshift32	228	194*	194*
ctr8	98	98	76*
ctr16	322	310	267*
ctr32	1154	1098	923*
ctr64	4354	4082	2607*
enc16	70	69*	72
enc32	184	185	183*
mult4	180	180	172*
mult8	15,963	16,052	14,927*

Table 4.11: BDD Sizes for OASIS-generated circuits
*: Best Sizes

Statistic	Mean & Standard Deviation of BDD sizes(# of Nodes)		
	No W/length	Lumped W/Length	Level based W/length
$\overline{\lvert BDD \rvert}$	1645.92	1624	1423.07
$\sigma_{13}(\mid BDD \mid)$	4273.96	4286.60	3945.30

Table 4.12: Mean and Standard Deviation of sizes for OASIS-generated circuits

Circuit Name	% improvement in BDD Sizes
mux32	3%
mux64	1%
bshift16	18%
bshift32	15%
ctr8	22%
ctr16	17%
ctr32	20%
ctr64	40%
enc16	1%
enc32	0.5%
mult4	4%
mult8	6.5%

Table 4.13: Percentage improvement for OASIS circuits with wire length

tion for the OASIS generated circuits. Circuits like the barrel shifter and the counter benefit from using wire length approximation in the spatial entropy computation. But other circuits like the multiplexor, encoder and multiplier do not gain from this additional information. In the latter case the BDD sizes show less than 10% improvement. The cases in which wire length information does help, the level based wire length distribution approach seems to do better than using lumped wire lengths.

Some of the reasons for the varied responses to wire length information hinge on the communication pattern in these functions. In the case of a *counter* the communication of data from one bit slice to another is usually along one dimension. As a result the inputs enter the circuit at different locations. Thus some inputs end up traveling a longer distance to get to a bit slice; this skew is captured by the wire length information. In a *barrel shifter* the n *data inputs* typically enter together and then travel through $\log n$ levels communicating with each other before reaching the outputs. Thus input data communicates down and across, in *two* dimensions. The $\log n$ *control inputs* that determine the shift amount exhibit a different communication pattern since they only flow in *one* dimension - across the circuit. The wire length information captures some of this variation in the communication pattern between the control inputs and the data inputs. *Multiplexors* and *encoders* are variations of the selection function. In the multiplexor one of 2^n input combinations gets selected with the help of $\log n$ bits of select lines. In the encoder one of 2^n input combinations gets selected and encoded into $\log n$ output lines. These functions require two dimensional communication. Since all the 2^n input combinations are involved in each output function, the influence of each input variable is equal. They all end up traveling an approximately equal amount to get to the output causing wire lengths to be a nonfactor. *Multipliers* are also functions with two-way communication and with equal participation from each input variable, causing wiring skew to be absent. In addition the family of (integer) multiplier functions have been proven to have exponential BDD sizes regardless of ordering [Bry91]. So the changes caused by wire lengths can only be minor perturbations.

Let us now look at decoders. Table 4.11 (page 93) shows that decoders are immune to any kind of wire length approximation. Their BDD sizes are the same regardless of wire length information. The reason for this is fundamental to the nature of the decoding function. In a completely specified n input, 2^n output decoder each output of the decoder is specified by exactly one minterm of the n variable function. For example in a 2 to 4 decoder $o_0 = a'b', o_1 = a'b, o_2 = ab'$, and $o_3 = ab$. Hence the BDD for every output function o_i requires an ordering on the input variables in a single minterm. In such a situation the size of the BDD will be the same regardless of how the variables in the minterm are ordered, since every variable appears once.

Minimization:

Now we illustrate the effects of redundancy on spatial entropy based BDD ordering. As discussed in our assumptions earlier, since the effects of *minimization* and *wire length* approximations are not independent, we study them together. Table 4.14 (page 97) shows BDD sizes for the Logic Synthesis 91 benchmarks using different wire length approximation strategies. There are two rows for each circuit. The first row represents sizes when the benchmark circuits were minimized using the SIS script `algebraic` [2]. The second row represents BDD sizes when the benchmark circuits were minimized using the SIS script `rugged` [3]. In both cases no reconvergent fanout removal was per-

[2]this script uses algebraic division while generating kernels and common cubes in the factoring process.

[3]This is a more aggressive form of minimization using don't care extraction of various

formed. Table 4.16 (page 98) illustrates the same results for the much larger ISCAS 85 benchmark circuits. The average BDD size and the standard deviation in BDD sizes for the two sets of benchmarks is illustrated in Table 4.15 and Table 4.17.

In order to try and analyze the effects of wiring and minimization separately we illustrate some additional plots. These effects are studied in the following order.

A: Minimization with and without wire length for the Logic Synthesis 91 benchmarks (LgSynth91).

B: Minimization with and without wire length for the ISCAS 85 benchmarks.

C: Wire length with and without minimization for the Logic Synthesis 91 benchmarks.

D: Wirelength with and without minimization for the ISCAS 85 benchmarks.

Minimization effects are analyzed by comparing the effects of the `algebraic` script vs the `rugged` script. Wire length effects are analyzed by comparing the effects of *no* wire length, *lumped* wire length, and *level-based* wire length.

Minimization with and without wirelength (LgSynth91):

We begin by studying the effect of **minimization** with and with out the use of wire length in spatial entropy computation. This is illustrated in Figure 4.10 (page 99) for the Logic Synthesis 91 circuits. [For space reasons, the X axis uses Circuit Id numbers instead of circuit names]. The two plots depict percentage improvement (positive or negative) in BDD sizes when the spatial entropy computation was run on circuits minimized with the `algebraic` script versus circuits minimized with the `rugged` script. Wire length information was not used in one plot, while it was used in the second plot.

Let us look at Figure 4.10 (page 99) first. For the smaller Logic Synthesis benchmarks we observe that overall there seems to be little change in BDD size (for the better or for the worse) when using circuits minimized with the `algebraic` script versus circuits minimized with the `rugged` script. This is also seen in Table 4.15 (page 98) where the average BDD size varies by $\pm$ 10

kinds to reduce literal count.

Circuit Id	Circuit Name	# of Nodes		
		No W/length	Lumped W/length	Level based W/length
1	alu2^a	210$*$	214	250
	alu2^r	217	198$*$	241
2	alu4^a	1102	1000$*$	1064
	alu4^r	969	1070	945$*$
3	count^a	149	142	131$*$
	count^r	229	231	184$*$
4	f51m^a	78$*$	78	86
	f51m^r	78	78	72$*$
5	z4ml^a	35	35	35
	z4ml^r	33	33	33
6	5xp1^a	81	80$*$	87
	5xp1^r	81	80$*$	80$*$
7	bw^a	116	121	112$*$
	bw^r	116	116	114$*$
8	clip^a	125	125	125
	clip^r	117	112$*$	143
9	duke2^a	470	457$*$	474
	duke2^r	469	466$*$	493
10	misex1^a	49	49	44$*$
	misex1^r	51	51	49$*$
11	misex2^a	130$*$	134	133
	misex2^r	127$*$	130	142
12	sao2^a	115$*$	115$*$	133
	sao2^r	121$*$	123	137
13	vg2^a	278	296	220$*$
	vg2^r	312	291$*$	329

^{a} : `algebraic script`, ^{r} : `rugged script`

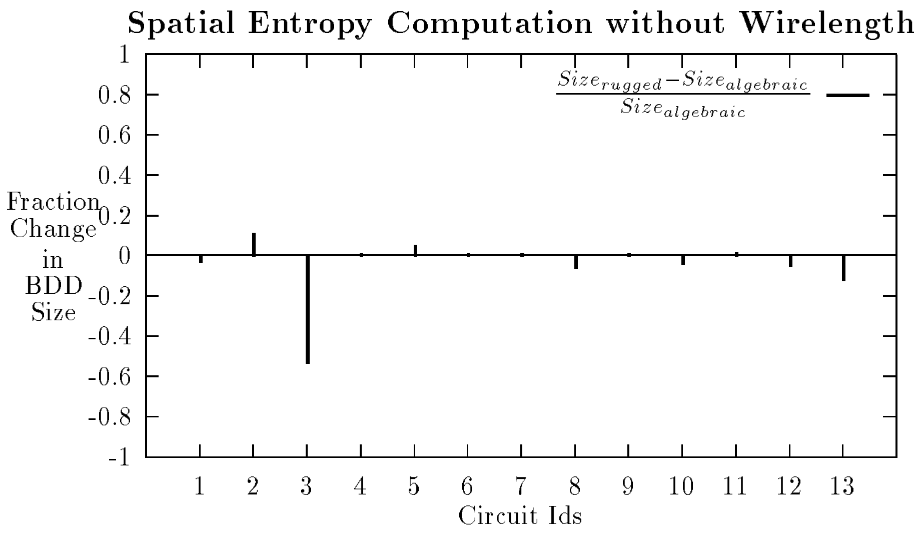
Table 4.14: Effect of wire length and the SIS scripts on the Logic Synthesis Benchmarks

Statistic	Mean & Standard Deviation of BDD sizes(# of Nodes)		
	No W/length	Lumped W/Length	Level based W/length
$\overline{\mid V \mid}^a$	226	218	222
$\sigma_{13}^a(\mid BDD \mid)$	287.28	260.9	277
$\overline{\mid V \mid}^r$	224.6	229	227
$\sigma_{13}^r(\mid BDD \mid)$	254	278.5	249.8

Table 4.15: $\overline{\mid V \mid}$ and σ of sizes for Logic Synthesis 91 circuits (with minimization and w/l approximations)
^{a}: `algebraic` script, ^{r}: `rugged` script

Circuit Name	# of Nodes		
	No W/length	Lumped W/length	Level based W/length
c432^a	43,262	37,025	41,953
c432^r	6,268	4,407	1,815
c499^a	37,764	61,462	70,489
c499^r	54,029	49,260	65,105
c880^a	27,963	34,514	11,133
c880^r	7,241	6,774	29,853
c1908^a	52,380	26,588	34,000
c1908^r	28,342	29,173	22,488
c3540^a	517,177	476,395	480,066
c3540^r	$> 600,000$	456,190	460,598

Table 4.16: Effect of wire length and the SIS scripts on the ISCAS 85 Benchmarks
^{a} `: algebraic script,` ^{r} `: rugged script`



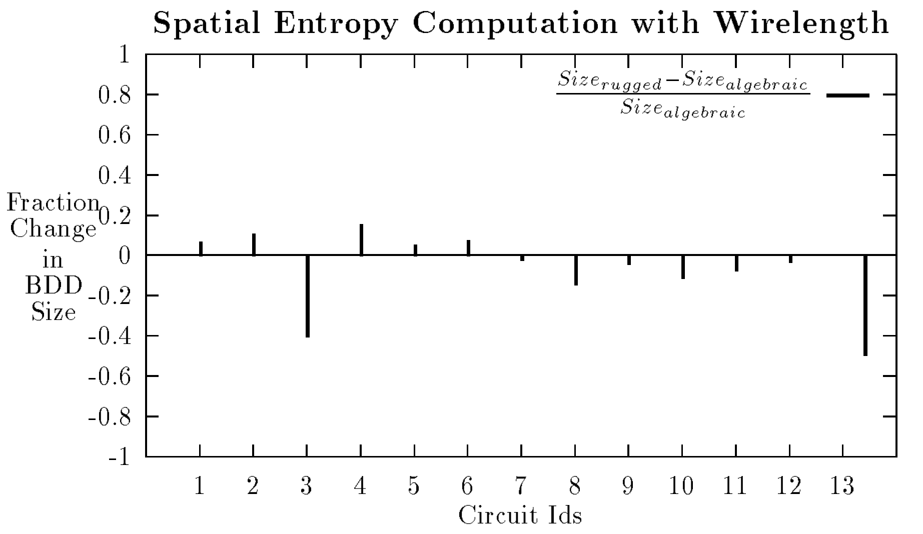
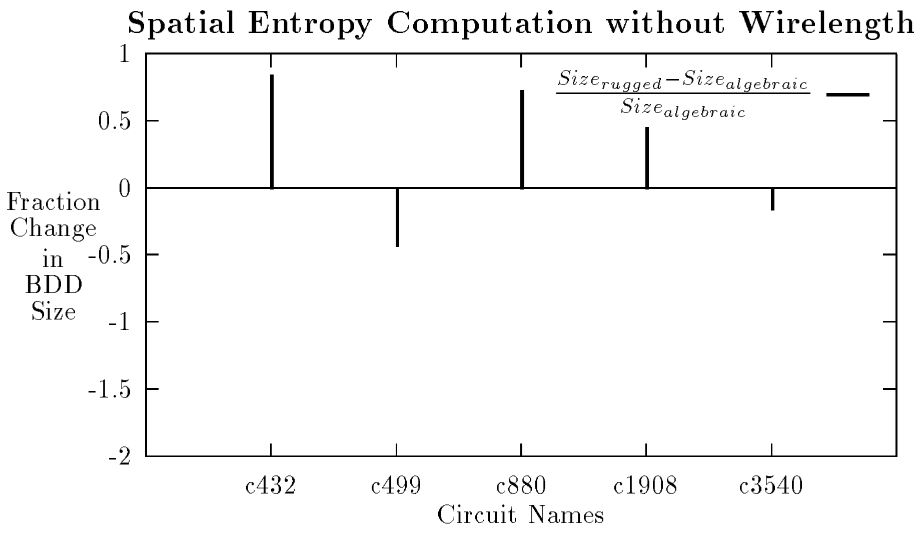


Figure 4.10: Effect of Minimization Scripts on the Logic Synthesis Benchmarks



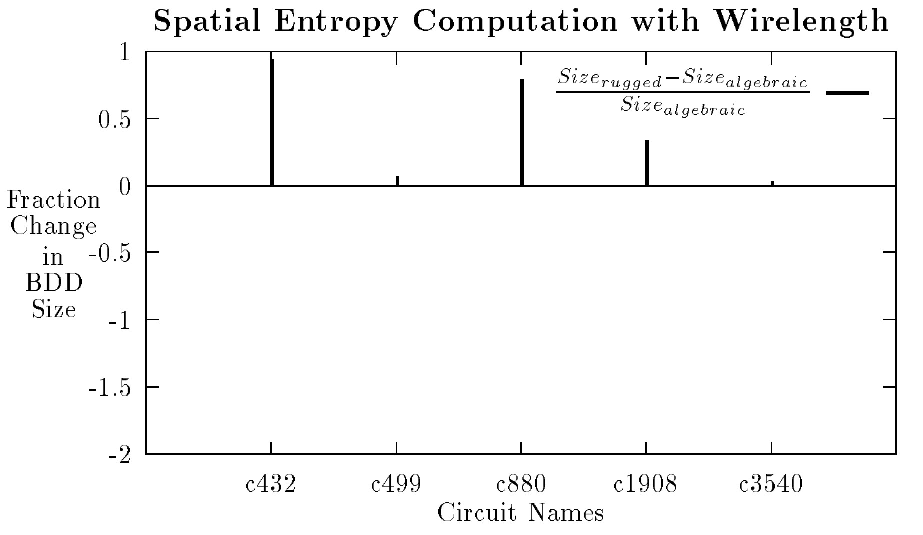


Figure 4.11: Effect of Minimization Scripts on the ISCAS 85 Benchmarks

Statistic	Mean & Standard Deviation of BDD sizes(# of Nodes)		
	No W/length	Lumped W/Length	Level based W/length
$\overline{\lvert BDD\rvert^{a}}$	135,709	127,196	127,528
$\sigma^{a}_{13}(\mid BDD \mid)$	213,430	195,642	198,213
$\overline{\lvert BDD\rvert^{r}}$	139,176	109,160	115,971
$\sigma^{r}_{13}(\mid BDD \mid)$	258,342	194,853	193,998

Table 4.17: Mean & Standard Deviation of sizes for ISCAS85 circuits (for wire length and minimization approximations)
^{a} : `algebraic script`, ^{r} : `rugged script`

nodes, while going from the `algebraic` script to the `rugged` script. The main reason for this could be that since the circuits are small, little improvement is gained (in literal count) when using the more aggressive minimization script.

With the inclusion of wire length information in spatial entropy computation there seems to be more overall perturbation in the sizes (as compared to the case when no wire length information was used). Circuits whose sizes improved with the use of the `rugged` script improved a little more when wire length information was used. At the same time circuits whose sizes deteriorated with the use of the `rugged` script worsened some more when wire length information was used. The standard deviation in Table 4.15 which captures the distribution of sizes about the mean fluctuates accordingly, increasing ($260 \rightarrow 278$) when lumped wire length is used but decreasing ($277 \rightarrow 249$) when level based wire length is used. With no wire length the standard deviation reduces from 287.28 to 254 when the `rugged` script is used. This effect could likely be due to the fact that the biggest circuit `alu4` yields a smaller size with the `rugged` script ($1102 \rightarrow 969$), thus reducing the variance.

Minimization with and without wire length (ISCAS85):

Now let us turn our attention to Figure 4.11 (page 100). This shows the results of minimization for the ISCAS 85 circuits. For the larger ISCAS 85 benchmarks the effects of minimization are more significant.

When wire length information is not used there seems to be a definite and substantial improvement in BDD sizes after minimizing with the `rugged` script. This is because with larger circuits there is a greater likelihood of performance improvement (in literal count) when the `rugged` script is used for minimization instead of the `algebraic` script. This is not reflected in the average and standard deviation values in Table 4.17 (page 101) because of the dominating effect that circuit `c3540` has on these statistics. Discounting its effect from the computation $\mid V \mid^a \rightarrow \mid V \mid^r$ goes from $135,709 \rightarrow 139,176$ to $40,342 \rightarrow 23,920$, indicating a clear improvement.

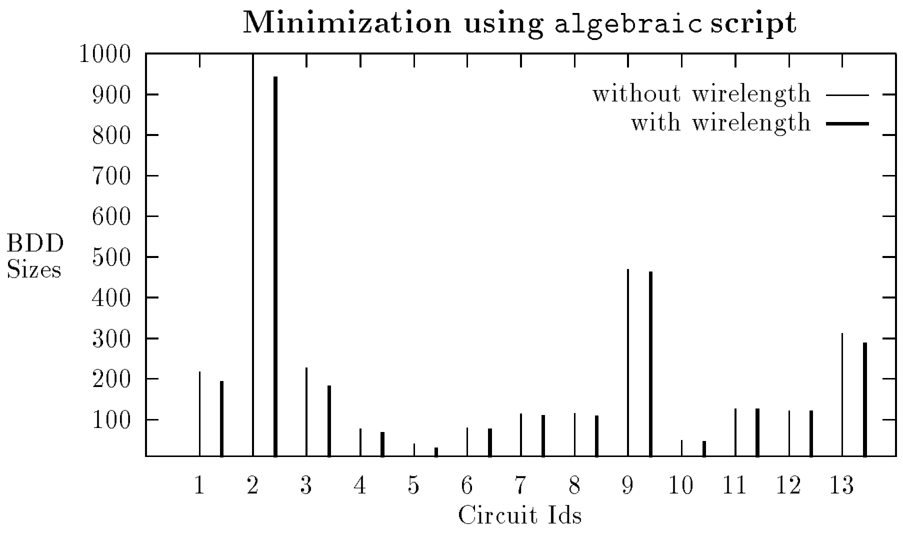
When wire length information *is* used the benefits of running spatial entropy computation on the `rugged` script minimized circuits appear mixed. While the lower plot of Figure 4.11 (page 100) shows sizes improving with wire length information this is not always the case. In Table 4.16 (page 98) we notice that for circuit `c880` minimized with the `rugged` script, the sizes improve when *lumped* wire length information is used, but they deteriorate when *level-based* wire lengths are used. Similarly in circuit `c1908` minimized with the `rugged` script, the sizes improve when *level-based* wire lengths are used but deteriorate when *lumped* wire length is used. The statistics in Table 4.17 (page 101) on the other hand show that there is an improvement in average BDD size while going from the `algebraic` script to the `rugged` script, regardless of whether lumped wire length is used $(127,196 \rightarrow 109,160)$, or level based wire length is used $(127,528 \rightarrow 115,971)$. This implies that on the average the BDD sizes improved more than they deteriorated. The standard deviation does not show much change $(195,642 \rightarrow 194,853)$, $(198,213 \rightarrow 193,998)$. because of the dominating effect of `c3540`.

The mixed size improvement results for minimization with wire length confirms our earlier hypothesis that the effects of minimization and wire length are not independent. While the `rugged` script does a better job of removing redundant literals it also alters the wire length distribution in the circuit with the excessive levels generated during factoring. This could affect the spatial entropy distribution in the circuit for the better or for the worse.

Wire length with and without minimization (LgSynth91):

We now try to analyze the effect of **wire length** on BDD sizes for circuits minimized with the `algebraic` script and circuits minimized with the `rugged` script. Figure 4.12 (page 103) shows two plots for the Logic Synthesis benchmarks. The first plot in the figure compares BDD sizes for circuits minimized with the `algebraic` script, when spatial entropy computation is performed with and without wire length information. The second plot does the same comparison for circuits minimized with the `rugged` script.

For the Logic Synthesis benchmarks minimized with the `algebraic` script



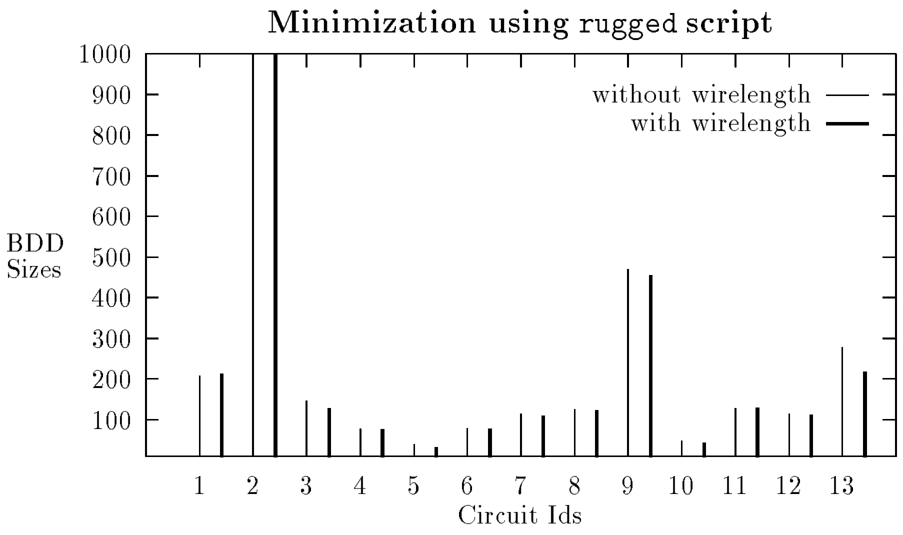
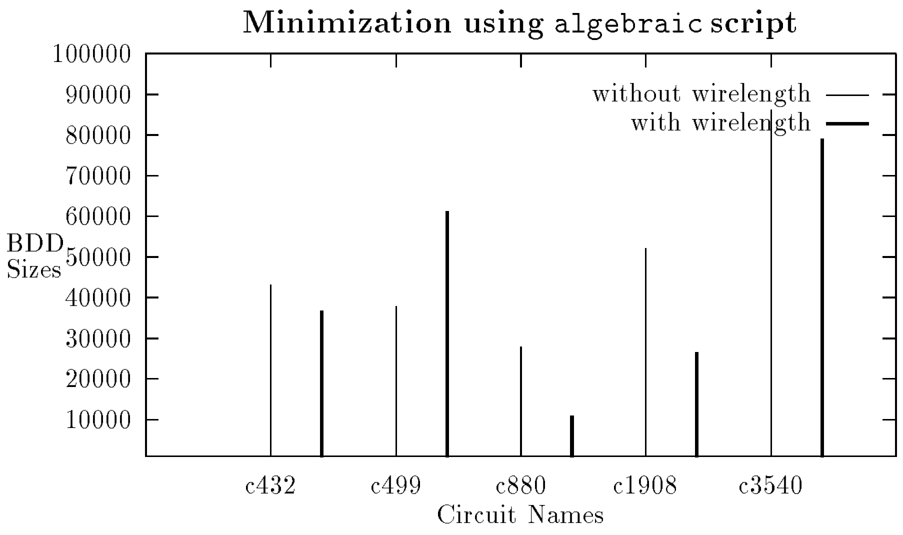


Figure 4.12: Effect of Wire Length on the Logic Synthesis Benchmarks



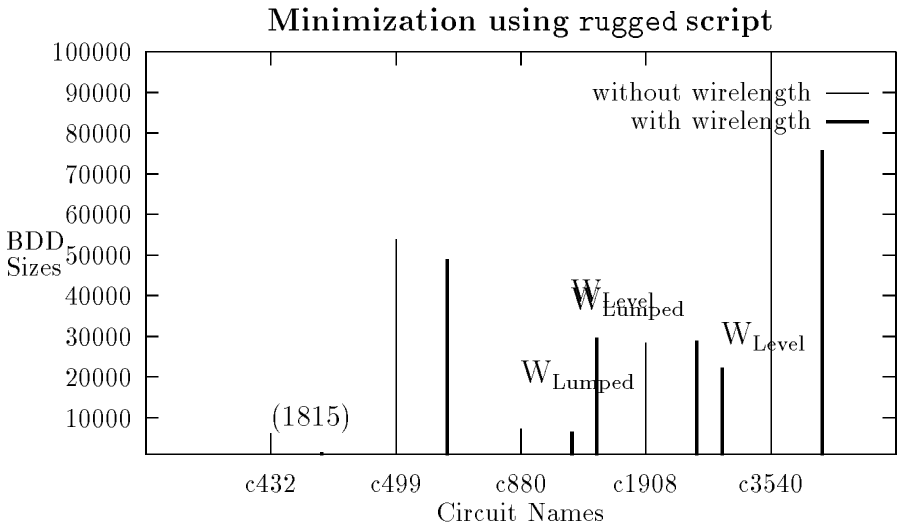


Figure 4.13: Effect of Wire Length on the ISCAS 85 Benchmarks

(Figure 4.12), wire length information appears to be useful only when the BDD sizes get bigger (see Circuit Ids $1, 2, 3, 13$). As Table 4.15 (page 98) indicates the average BDD size improvement is quite minimal for these circuits with wire length information ($226 \rightarrow \{218, 222\}$). The standard deviation varies accordingly ($287.28 \rightarrow \{260.9, 277\}$). When the same circuits are minimized with the `rugged` script even this benefit seems to disappear. The average BDD size values also indicate this ($224.6 \rightarrow \{229, 227\}$).

Wire length with and without minimization (ISCAS85):

Figure 4.13 shows the same set of results for the ISCAS 85 circuits. [The sizes for circuit `c3540` have been scaled to 100K nodes to help present results clearly. This should not affect the results here since we are only interested in comparative numbers.]

For the ISCAS benchmarks (Figure 4.13) minimized with the `algebraic` script, except for one circuit (`c499`), the wire length information does seem to benefit, but in varying amounts. Table 4.17 (page 101) confirms this, as sizes improve from $213, 430$ to $\{195, 642, 198, 213\}$.

When minimized with the `rugged` script, the results are again mixed; but due to the improvement in size of the `c3540` circuit, the overall average BDD size shows in improvement in Table 4.17. As mentioned earlier while wire length information does benefit the *lumped* and the *level-based* approximations, both fare differently.

Reconvergent Fanout

Finally we discuss the errors due to reconvergent fanout. As discussed in Chapter 3 the procedure to remove these inaccuracies in spatial entropy computation is very expensive and is exponential in the number of fanout inputs in the supergate. We implemented the procedure to construct supergates and remover errors in spatial entropy and 1-probability computation. But our results were unfortunately not very conclusive. For circuits with small supergates and a small number of fanout inputs, where reconvergent fanout removal was tractable, we found that there was not a need to apply the procedure since the spatial entropy computation already yielded good sizes. For the circuits where the BDD sizes were poor, it was very expensive to apply this procedure because the supergate sizes and fanout inputs were large in number. We tabulate in Table 4.18 the supergate sizes and number of fanout inputs for circuits in the Logic Synthesis 91 benchmark set. In the table we indicate the number of reconvergent nodes and the largest supergate size for each circuit. We also indicate the largest number of fanout nodes

Circuit Name	Total Nodes	Reconv. Nodes	Largest Sg Size	Max # of F/out Nodes	# of F/outs >= 10
alu2	318	69	205	10	10
alu4	1408	371	738	14	130
count	364	107	84	18	9
frg1	262	53	251	23	14
5xp1	150	39	65	7	0
bw	134	46	31	5	0
clip	336	90	131	9	0
duke2	327	91	103	17	32
misex1	50	11	26	4	0
misex2	103	7	29	7	0
rd73	299	80	221	7	0
sao2	169	41	86	10	9
vg2	247	66	117	23	16

Table 4.18: Reconvergent Fanout Information for Logic Synthesis 91 Benchmarks

found in a supergate. This is an indication of how wide the supergate is and how expensive it would be to simulate such a supergate. The last column is another indicator of reconvergent fanout removal expense. It counts the number of supergates that have 10 or more fanout nodes. A supergate with 10 fanout nodes requires 1024 simulations to compute the exact 1-probability and spatial entropy at its reconvergent node. Hence this is a measure of how many such large supergates exist in the circuit.

We observe from Table 4.7 (page 88) that some of the circuits for which `spent` had trouble in generating good BDD sizes were `vg2, sao2, frg1, count` and `alu4`. From Table 4.18 (page 106) we also notice that these are the circuits that have high reconvergent fanout with several supergates that have 10 or more fanout nodes. So it is likely that the errors due to the reconvergent fanout in these circuits is contributing to the poor orderings.

4.5.3 Spatial Entropy Vector Combination Strategies

We study three approaches to combining output spatial entropy vectors in multi-output functions: *maximum* (max), *weighted-multiply* (wmult), and *weighted-divide* (wdiv). These were discussed in detail in Section 4.2. Our results are presented in Table 4.19 and Table 4.20 for the two benchmark circuits with and without wire length approximations. Figure 4.14 and Figure 4.15 plot the BDD sizes for the different spatial entropy vector combining strategies, generated with and without wire length approximation. [The sizes for circuit `c3540` have again been scaled to 100K nodes.]

Let us look at the plot for the Logic Synthesis benchmarks first (Figure 4.14). For most of the small circuits there seems little to choose between one combining strategy versus the other. For a couple of the bigger circuits (Circuit Id #s 2,6), there seems to be a slightly clearer choice of the best strategy, though this is true only when wire length information is also provided. The choice of the strategy seems to vary from circuit to circuit. This is also illustrated in the following situation. Looking at Table 4.19 we observe that for the "arithmetic" circuits in the benchmark - `alu2, alu4, count`, the weighted divide strategy `wdiv` does not appear to yield good sizes. It may be recalled that `wdiv` is the approach where the spatial entropy vector values are combined such that the cumulative spatial entropy at every output node is normalized to the range $[0, 1]$. All the above arithmetic circuits probably have adder-like computing elements in them where the individual outputs have disproportionate contributions to the total circuit spatial entropy. As a result giving equal importance to all the output spatial entropy vectors using the `wdiv` approach does not seem to be the right thing to do. On the other hand if we look at circuits `sao2, duke2, clip` and `vg2`, in Table 4.19, they seem to benefit by the `wdiv` approach. One of the reasons for this could be the following. These are circuits with high reconvergent fanout. Since we ignore reconvergent fanout removal the errors in the computation process may make the spatial entropy at some output nodes unrealistically high. Using the weighted divide approach forces every output to be treated with equal importance, thus supressing errors that reconvergent fanout might have introduced.

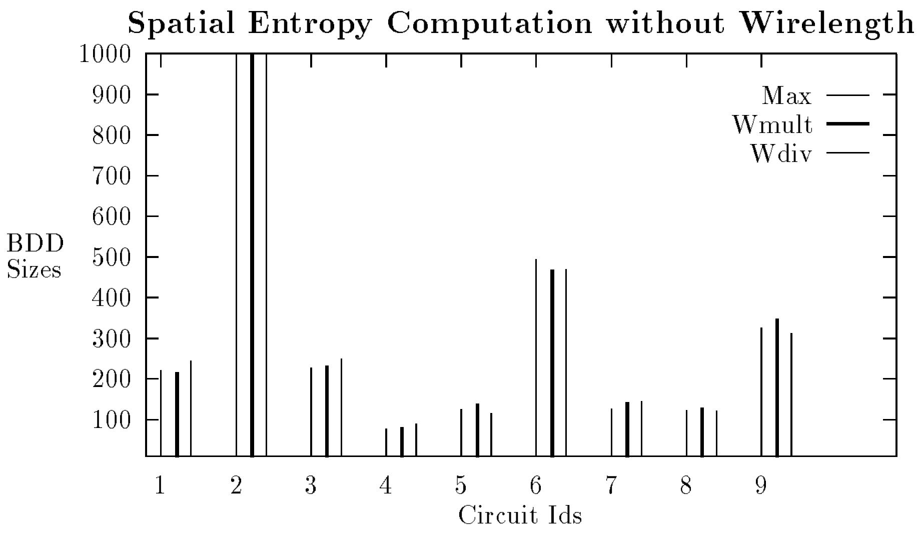
Now let us look at Figure 4.15 that shows plots for the ISCAS 85 benchmarks. Here we notice that there is more variation between sizes generated by one strategy versus sizes generated by another. But the choice of a par-

Id	Circuit Name	$\vec{S}$ Combining Strategy	# of Nodes (with Wire Length Approx.)		
			No W/len	Lumped	Level
1	alu2	max	223	217	241
		wmult	217	198	242
		wdiv	246	250	271
2	alu4	max	969	1070	1053
		wmult	1075	1080	945
		wdiv	1098	1071	1202
3	count	max	229	231	184
		wmult	233	232	211
		wdiv	249	246	228
4	f51m	max	78	78	72
		wmult	84	82	77
		wdiv	90	90	90
5	clip	max	125	125	147
		wmult	140	131	153
		wdiv	117	112	143
6	duke2	max	497	507	1021
		wmult	469	469	640
		wdiv	469	466	493
7	misex2	max	127	130	147
		wmult	145	143	142
		wdiv	147	147	142
8	sao2	max	124	140	147
		wmult	132	123	161
		wdiv	121	123	137
9	vg2	max	326	333	329
		wmult	352	326	405
		wdiv	312	291	446

Table 4.19: Comparision of $\vec{S}$ combination strategies for Logic Synthesis 91 Benchmarks

	Circuit	$\vec{S}$ Combining	# of Nodes (with W/length approx.)		
Id	Name	Strategy	No W/len	Lumped	Level
1	c432	max	18157	4407	44903
		wmult	6268	1815	42698
		wdiv	6461	5910	44588
2	c499	max	49058	68722	148370
		wmult	37764	72392	70489
		wdiv	38532	72392	71937
3	c880	max	7222	6774	29853
		wmult	7241	6774	153085
		wdiv	28233	30120	48725
4	c1908	max	28342	26453	30445
		wmult	28343	26319	31627
		wdiv	28342	26517	22488
5	c3540	max	>600,000	460598	456190
		wmult	>600,000	460598	456190
		wdiv	>600,000	593617	>600,000

Table 4.20: Comparision of $\vec{S}$ combination strategies for ISCAS 85 Benchmarks



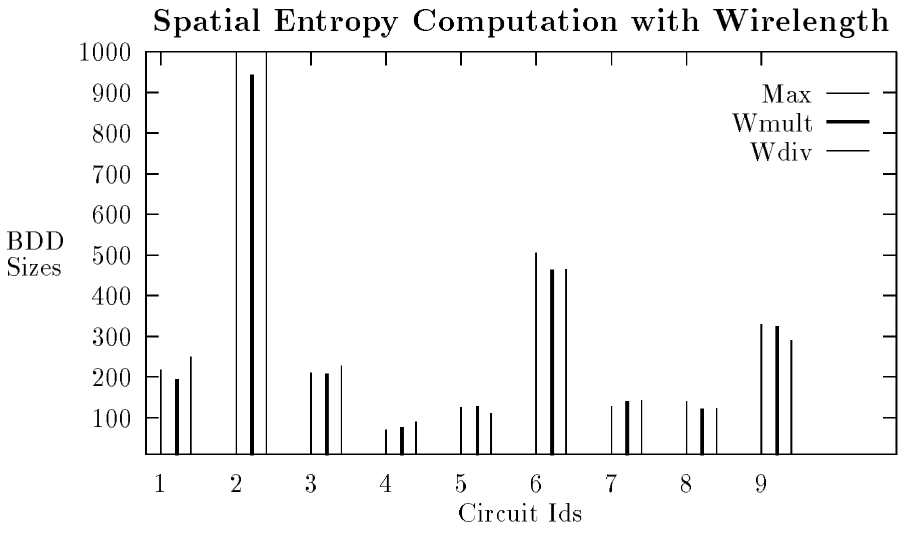
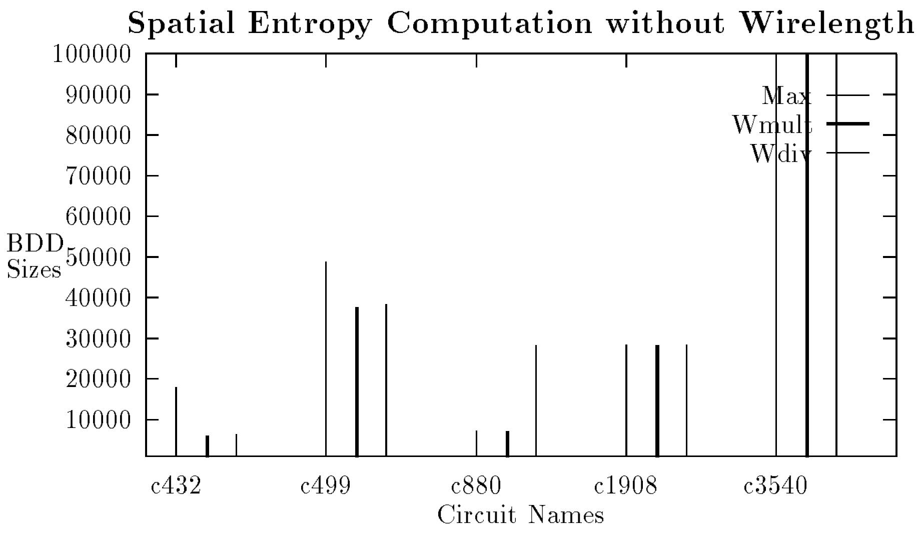


Figure 4.14: $\vec{S}$ Combining Strategies for Logic Synthesis Benchmarks



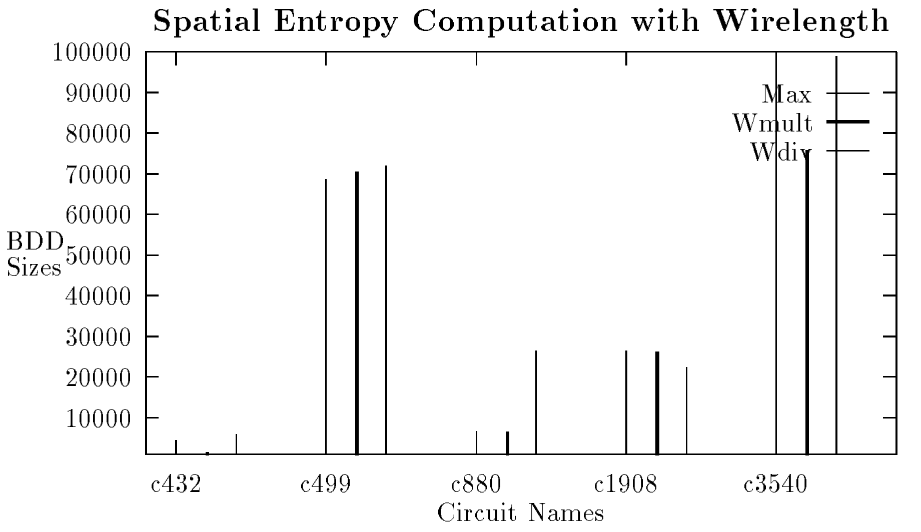


Figure 4.15: $\vec{S}$ Combining Strategies for ISCAS 85 Benchmarks

ticular strategy is unclear. The `max` and `wmult` approaches seem to perform equally well. We also observe that circuit `c880`, an ALU+control circuit, does badly with the weighted divide approach. This appears to confirm our earlier observations about using this approach on circuits with uneven contributions from the different outputs.

4.6 Conclusions

We are now ready to draw some conclusions. For the smaller circuits the spatial entropy approach generates BDD sizes competitive with the optimization intensive approaches and the static attribute based approaches. But as the circuits become larger the effectiveness of the optimzation intensive strategies becomes markedly apparent while both the static attribute based approach and the spatial entropy based approach do poorly but for different reasons. The effectiveness of the optimization approach comes from the fact that they work directly with the BDDs. In contrast, the other approaches make approximations while working with an implementation. The static attribute based approaches use level and depth based heuristics, while the spatial entropy based approach makes approximations to simplify the computation of the attribute. The effect of this is apparent as the circuits get larger. With larger circuits there is a greater chance that static attribute based heuristics may make wrong decisions. Similarly the effect of spatial entropy approximations such as reconvergent fanout, wire length, and redundancy is more significant in larger circuits. This results in poor orderings. The effect of a poor ordering on the BDD sizes is more significant in large circuits. This is because a BDD size can grow exponentially at successive levels in the graph, and a wrong variable choice at one level (in a large circuit) could explode into a huge BDD within a few levels. The optimization intensive approaches directly search the space of orderings while building the BDDs, thus avoiding the need to examine an implementation.

But optimization intensive approaches pay a big price in the time they spend searching for a good ordering. Most static attribute based approaches require a simple (breadth first or depth first) traversal through the network that consumes a few minutes. For instance, the spatial entropy approach took 80.42 seconds on a DEC Station 5500 to generate an ordering for the ISCAS circuit c499. In contrast the `minwid91` approach was documented

[ISY91] as taking approximately 35,000 seconds on a SPARC Station 1+.

How do inaccuracies in computing the spatial entropy attribute affect variable orderings and BDD sizes? We find that for regular structured circuits, where the communication pattern is well-defined, the effects of wire length can be observed clearly. These effects are a function of the communication behavior in these circuits. Consequently some circuits benefit from this information generating smaller BDDs while others do not. For the more irregular circuits (as in the benchmark sets) the effect of wire length is mixed. For the smaller benchmark circuits computing spatial entropy without wire length information does not hurt BDD sizes. For the larger circuits wire length information helps improve sizes, but not always. The degree of minimization in the circuit and the communication pattern in the circuit control this effect.

Lack of minimization affects spatial entropy computation and BDD sizes significantly in the larger circuits. Hence, in such circuits minimization is beneficial and improves BDD sizes. But when wire length information is also used in these minimized circuits, the approximations made in distributing wire lengths influence the effects of minimization. We had stated in an earlier section that the factors of logic minimization and wire length are interdependent and these results confirm it.

Reconvergent fanout seems to have a definite influence on spatial entropy computation for larger circuits, as we found a correlation between poor oderings and heavy reconvergent fanout in some of these circuits. But the extent of its effect cannot be clearly estimated. This is because it was computationally too expensive to remove the errors in spatial entropy computation (due to reconvergent fanout) in these large circuits. Since minimization can change the structure and organization of the gates in the circuit, it is likely that the minimization process can introduce or remove reconvergent nodes in the circuit. This also makes it difficult to attribute the cause of poor orderings in these circuits entirely to reconvergent fanout errors.

With respect to the spatial entropy vector combining strategies, the `max` and `wmult` approaches do not perform very differently from each other. This could be because both approaches combine output spatial entropy vectors in proportion to the output spatial entropy values. The `wdiv` approach forces every output node to contribute equally. Since this may not always be the case this approach does not do well in some circuits. On the other hand it does help yield good orderings in cases where excess spatial entropy contribution gets generated due to reconvergent fanout or wire length distribution approximations.

On the whole the spatial entropy computation approach does use approximations that affect the accuracy of the attribute. But the effects of these approximations are felt more significantly in the variable ordering problem, especially on large circuits, than they might be in other applications of this attribute. The first reason is the fine granularity with which the attribute is being handled. Orderings are generated on spatial entropy vectors and not spatial entropies themselves. Thus inaccuracies get further distributed from the spatial entropies on the nodes to the spatial entropy contributions in the vectors. The second reason is the extreme sensitivity of BDDs to variable orderings. A small error in spatial entropy computation that yields a different ordering can translate into a BDD that is several thousand nodes larger in size when the circuit is large.

Due to the approximations in spatial entropy computation, the experimental evidence for spatial entropy based variable ordering has not been conclusive. We simultaneously explored a theoretical basis for using the spatial entropy of an implementation to generate BDD variable orders. The size of a BDD is a characteristic of the function and is invariant over different implementations. Then how can the spatial entropy of an implementation of the function be capable of generating variable orders for the BDD? This can only be possible if the spatial entropy of an implementation captures a fundamental characteristic of the function. This is the focus of the next chapter.

Chapter 5

Spatial Entropy as a Measure of Area-Complexity

This chapter draws on some of the conclusions in Chapter 4. In order to explain a theoretical basis for using spatial entropy to generate BDD variable orders an important question that needed to be answered was: If a BDD is a characteristic of the function, then how can the spatial entropy of some implementation of the function generate variable orders? In this chapter we show empirical evidence that the spatial entropy of an implementation is capable of measuring the gate-count complexity of the function. This provides empirical evidence that spatial entropy is capable of capturing function behavior. We also show that our definition of gate-count complexity in boolean space, called information content, does a good job in estimating actual logic gate cost, and there is strong correlation between information content and gate count.

We begin with some motivation for the work in this chapter and then provide background information on boolean functions and complexity. In Section 5.3 we define our estimate of gate-count complexity in boolean space. In order to show that spatial entropy can be used to measure gate-count complexity we introduce a notation called the *decision tree*, in Section 5.4. This is an intermediate representation relating a function and its implementation. We then illustrate in Section 5.5 how the spatial entropy of an implementation can be computed using the decision tree, and how this computation is approximated by the gate-level spatial entropy procedure discussed in Chapter 3. Section 5.6 discusses experimental results to correlate spatial entropy and information content and gate-count complexity and information content.

5.1 Motivation

In the previous chapter we demonstrated empirically that the spatial entropy based approach can generate variable orders for binary decision diagrams. We also observed that factors that affect the accuracy of spatial entropy computation (in a given implementation), like reconvergent fanout and poor estimates of wire length, have a significant effect on BDD sizes. Spatial entropy is also influenced by logic minimization. Different implementations of the same function can differ significantly in their spatial entropy values due to different degrees of minimization in each implementation.

Along with the experimental work, we explored a theoretical basis to connect spatial entropy and BDD variable ordering. The size of a BDD is a characteristic of the function; it is invariant over different implementations. Then how can the spatial entropy of an implementation generate variable orders for the BDD? In order to do so, the spatial entropy of the implementation must be able to capture some characteristic of the function's minterms. But our experiments in Chapter 4 showed that there can be wide variations in the spatial entropies of the same function, when computed on different implementations. So this raised the question: what should be the best implementation to capture this function behavior and why?

Spatial entropy is the dynamic communication effort required to compute the function. In a physical (CMOS) implementation this becomes the switching energy expended by the circuit while computing the function. Over all implementations of a given function, the one with minimum spatial entropy has minimum switching energy and describes the minimum communication effort needed to compute the function. Thus the minimum spatial entropy over all implementations of a function acts like a signature of the function since it provides a lower bound on the switching energy over all implementations of that function. Minimum spatial entropy in an implementation implies simultaneously minimizing the logic gates and the wires in the implementation. This yields minimum switching energy because the switching activity at the gates (that contribute to the power consumption) is minimized due to lesser gates, and the delay at each gate is minimized due to reduced switching capacitance and shorter wires. So comparing a pair of implementations with minimum spatial entropy is equivalent to comparing the behavior of the underlying functions (in the two implementations).

But as we noticed in Chapter 4, generating minimum spatial entropy implementations is difficult with existing logic minimization tools. These tools only perform logic minimization and ignore the effect of wires. As Figure 4.7 showed, minimizing gate count can generate longer wires yielding greater spatial entropy (and switching energy). In such a situation, an approximation can be made. A *minimal gate count implementation*, on which spatial entropy is computed with unit wire length, can be used as an approximation to a minimum spatial entropy implementation computed with actual wire lengths. This minimal gate count implementation also acts like a signature of the function. This is because even this approximate spatial entropy (computed with unit wire length) provides a lower bound on a physical attribute, the gate count, over all implementations of that function.

How does the spatial entropy of a minimal gate count implementation capture minterm characteristics and boolean function behavior? We answer this question in this chapter by studying minimality and spatial entropy in boolean space in the context of boolean function complexity. The *complexity* of a boolean function is a measure that provides a lower bound on some physical attribute over all implementations of that function. Thus it is a fundamental characteristic of function behavior. Depending upon the physical attribute that it bounds, the complexity can be defined differently - wiring-complexity layout-area complexity, active-area complexity, energy-complexity *etc.*. We are interested in the *gate-count complexity*, the cost of gates in an implementation. In this chapter we define *information content*, an estimate of the gate-count complexity of boolean functions over its minterms, and illustrate empirically that spatial entropy is a linear function of information content. While this does not give us a theoretical basis for using spatial entropy for BDD variable ordering, it yields empirical evidence that such a basis can exist, by demonstrating that spatial entropy can track function behavior. We also show that information content and spatial entropy can act as measures of actual gate-count complexity.

We begin with some background on boolean functions and then discuss related work in function complexity.

5.2 Background

5.2.1 Definitions

We start by defining some frequently used terms. A *boolean variable* is a variable that takes a value from the set $\{0,1\}$. A *literal* is a (boolean) variable or its negation. For example a and a' are literals. A *single-output boolean function* of n *input* variables $f(x_1,\ldots,x_n)$, is of the form:

$$f : \{0,1\}^n \longrightarrow \{0,1\}$$

When the function f is of the form

$$f : \{0,1\}^n \longrightarrow \{0,1\}^m$$

then we have a multi-output boolean function (n-input, m-output). We shall restrict ourselves to single output functions for the present.

A *product term* is a conjunction (logical AND) of input literals. For example $ab'c$ is a product term with implicit conjunction. A *minterm* is a product term where every input variable is represented in its complemented or uncomplemented form. The *ON-set* of a function is the set of minterms for which the function has a value of 1. The *OFF-set* of a function is the set of minterms for which the function has a value of 0. The *DON'T CARE-set* of a function is the set of minterms for which the function can have a value of 0 or 1. A *completely specified function* f is one in which each minterm is assigned a unique value, either a 0 or 1. In an *incompletely specified function* there are minterms that may be a 0 or 1, *i.e.*, the DON'T CARE-set is not empty.

A *sum-of-products expression* is a boolean function representation expressed as a disjunction of product terms. Here is an example.

$$f = ac + ad'e + bc + bde$$

This is also called a *two-level* expression because it has two levels of logic (AND-OR). A function can also be represented as a *multi-level* expression with more than two levels of logic. For example the above function f would appear as

$$f = (a+b)(c+de)$$

A *minimal expression* is a multi-level expression with the fewest number of literals.

An alternative way to view an n-variable boolean function is as an n-dimensional hypercube ($\{0,1\}^n$). Figure 5.1 shows a boolean function of 3 variables a, b, c represented as a 3-dimensional hypercube. The vertices of the hypercube denote the minterms. For example the vertex (010) represents the minterm $a'bc'$. A *cube* is a collection of minterms of an n-variable function where the number of minterms is a power of 2, 2^m, for some $m \leq n$. If $m = 0$, then the cube has only 1 minterm which is a vertex of the hypercube. If $1 \leq m \leq n$ then for each of the 2^m minterms, the literals have the same value in $n - m$ positions and one of 2^m combinations in the other m positions. Algebraically, a cube is a product term; for example the cube with the minterms $\{000, 100\}$ is the product term $b'c'$. Viewed on a hypercube a cube is a projection of the hypercube along one or more dimensions, where each dimension corresponds to an input literal. This yields an m dimensional rectangular subspace of the hypercube $\{0,1\}^n$. For example projecting the 3-dimensional hypercube along the dimension of literal b, yields a cube of dimension 2 with $2^2 = 4$ minterms $\{011, 111, 010, 110\}$ whose algebraic product term equivalent is b. A cube c is a *subset* of another cube c', if all the minterms in c are contained in the minterm set of c'. For example the cube $\{010, 011\}$ is a subset of the above cube.

The Shannon expansion of a boolean function f on the variable x_i is

$$f(x_1, x_2, \ldots, x_n) = x_i'.f(x_1, \ldots, x_i = 0, \ldots, x_n) + x_i.f(x_1, \ldots, x_i = 1, \ldots, x_n)$$

where $f(x_1, \ldots, x_i = 0, \ldots, x_n)$ and $f(x_1, \ldots, x_i = 1, \ldots, x_n)$ are *partial functions* that represent the values of f when the input variable x_i has value of 0 and 1 respectively. These partial functions are called *boolean cofactors* or *Shannon cofactors* of the function, with respect to x.

We now briefly summarize related work in function complexity. The complexity of boolean functions has been defined over the years with bounds on different physical attributes, depending on the implementation technology. It was first investigated by Shannon [Sha49] while studying two-terminal switching networks with relay contacts. Muller [Mul56] then applied his result to networks with logical elements and showed that an upper bound on the complexity of single output networks was of the order of $\frac{2^n}{n}$ where n is the number of input variables. Then Kellerman [Kel68] derived an empirical

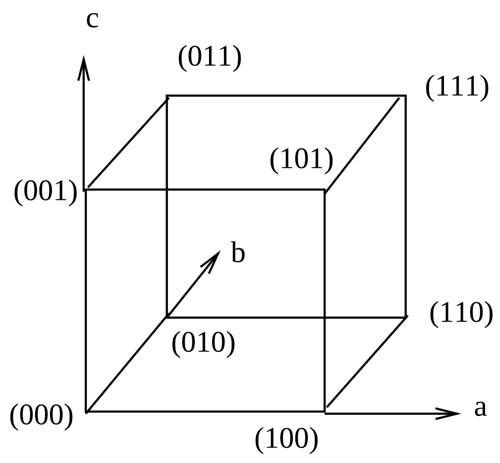


Figure 5.1: A Cube Space Representation of a 3 Variable Function

formula to estimate the average cost of a combinational logic network as a function of the number of ON-terms in the cube space of the function. He illustrated his results for single-output networks using diodes and modules as the implementation technology. This was followed by work on entropy based definitions of complexity by several researchers [Hel72, CF73, Mas78, Pip77, CA90] with bounds on attributes like diode counts, gate counts and literal counts. These were discussed in detail in Chapter 2. Yao [Yao79] used a communication model to to formulate lower bounds on the worst case information complexity of a boolean function f, and this was also discussed in Chapter 2.

How does our work relate to the above research? In the next section, we define the gate-count complexity of a boolean function by its *information content*. This definition differs from the existing definitions in that it emphasizes on the *spatial distribution* of the 1s and 0s in the cube space rather than just the *number* of 1s and 0s in the cube space. One of the motivations for doing so is to be able to predict the cost of implementing the function not just in terms of the literal count or gates but also in terms of the communication between the gates through wiring. Our definition is entropy based and is defined for completely specified single-output functions. The definition ensures minimality for two-level function representations and is an estimate of gate-count complexity in a multi-level implementation.

5.3 Information Content

We begin this section with an intuitive development of our idea. Our objective is to define the gate-count complexity of a boolean function in terms of the spatial distribution of minterms (1s and 0s) in the cube space. This will give us an estimate of the cost of implementing the function.

Let us start by trying to understand the spatial embedding of minterms and see how it contributes to the complexity of the function. Consider a set of completely specified, single-output, 3-variable functions. Figure 5.2 illustrates an example collection of such functions and their minterm distribution in cube space. The ON-terms or 1s are marked with a • at the cube vertex. The unmarked vertices represent the OFF-terms or 0s. The minimized expressions for these functions are also shown. Let us assume for the sake of these examples that the cost of the implementation is measured by the literal count in the minimized two-level expression. Consider functions A and F. Both are 3-variable functions that have exactly four 1s in their cube space. But the cost of implementing A is more than the cost of implementing F, using the literal count measure. The reason for this is the spatial arrangement of the 1s in the cube space. This was also observed in [Kel68] and [CF73]. Their experimental data showed that the maximum-cost data point (for a function of n variables with u 1s in the cube space) appeared when none of the 1s could be covered, while the minimum-cost data point appeared when all the 1s were optimally covered.

On a finer grain consider functions C and D. Both have exactly two 1s in their cube space. But the spatial arrangement of the 1s permits D to achieve a cheaper implementation, though this would require a finer cost discriminator than the simplistic measure of two-level literal count. Functions B and E exhibit a different feature. Here the number of 1s outnumber the number of 0s. The function E contains six 1s and two 0s, and the two OFF-terms are in almost identical spatial positions as the two ON-terms of function C. So with respect to spatial distribution the two functions (E and C) look very similar. Their implementations too have the same cost.

Now that we have an intuitive idea of why spatial embedding of minterms can affect gate-count complexity of functions we can formally define our measure for completely specified single-output functions. We begin with a few definitions.

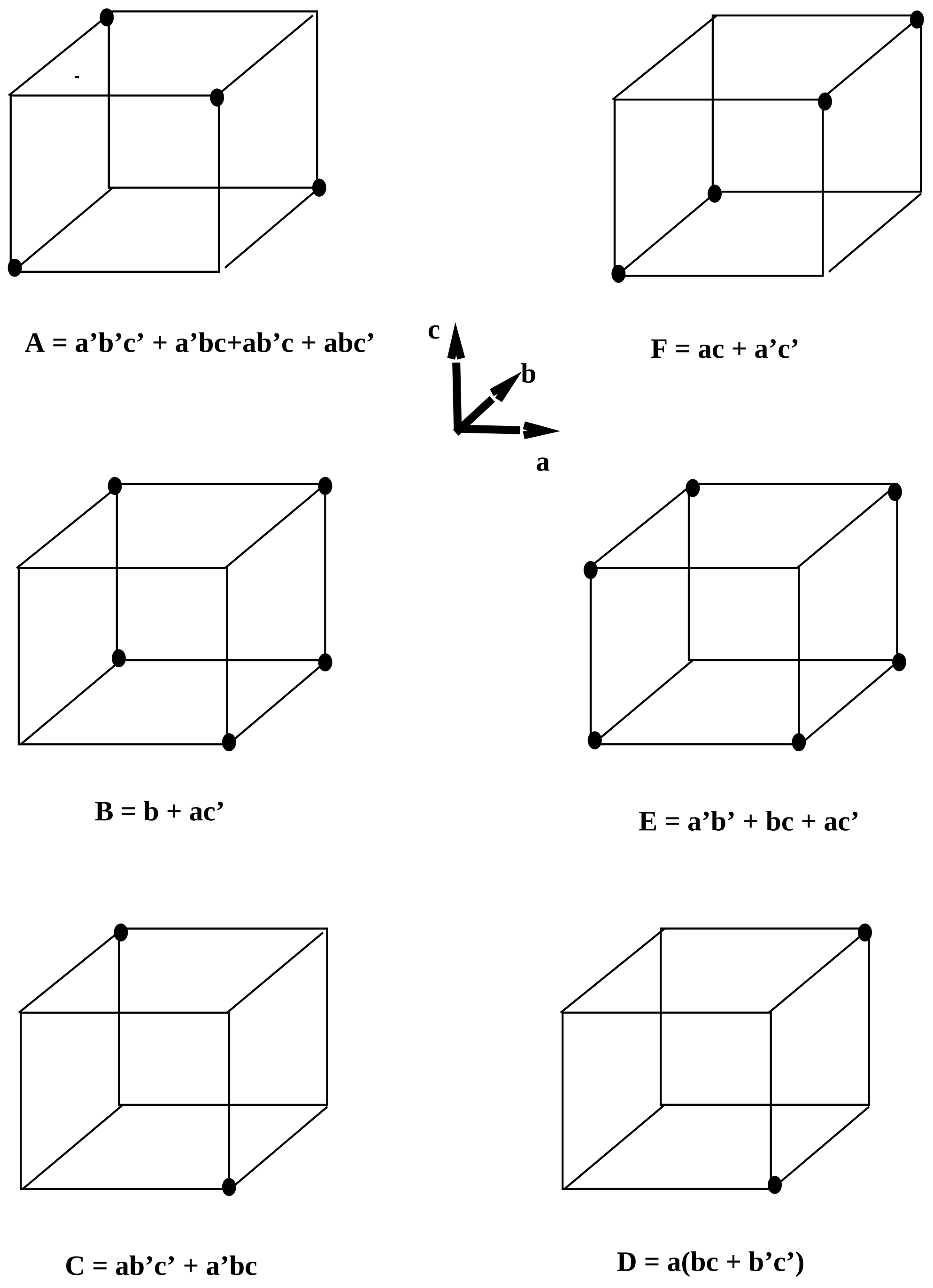


Figure 5.2: Example functions of 3 variables a, b, c

Definition 2 *A* **monochromatic cube** *c of a function f is a cube such that $\forall m \in c \quad (f(m) = 0) \;\vee\; (f(m) = 1)$, where m is a minterm of the function f that belongs to cube c, and $f(m)$ is the value of the function f for the minterm m. When $f(m) = 1$ then the cube is termed a* **one-cube**, *c^1. When $f(m) = 0$ the cube is termed a* **zero-cube**, *c^0.*

Definition 3 *The* **size** *of a cube $\mid c \mid$ is the number of minterms in the cube.*

Definition 4 *A* **k-decomposition** *D_k of a boolean function is a union, $O \cup Z$, of a set of one-cubes O, and a set of zero-cubes Z such that $k = \mid O \mid + \mid Z \mid$ and $O \cup Z$ is a partition of the function's cube space.*

Definition 5 *A* **minimum k-decomposition** *of a boolean function is a k-decomposition D_k with the smallest k such that for any other k'-decomposition $D_{k'}$ of the same boolean function, either $k < k'$, or if $k = k'$, then the sum of the cube sizes in k is greater than the sum of the cube sizes in k',*

$$\sum_{c_i \in \{O_k \cup Z_k\}} \mid c_i \mid > \sum_{c'_i \in \{O_{k'} \cup Z_{k'}\}} \mid c'_i \mid$$

Since cubes can overlap, the sum of the cube sizes in a minimum k-decomposition does not have to be bounded by the number of minterms 2^n.

The *monochromatic cubes* defined here are an n-dimensional equivalent of the monochromatic rectangles used by Yao in his communication complexity model [Yao79]. A monochromatic cube is a collection of minterms (in the n-dimensions) for which the function value is constant (0 or 1). The monochromatic rectangle defined by Yao also describes a collection of minterms with a constant function value. But the rectangle is defined differently. The input n bits are partitioned into $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ halves. The set M consists of the $2^{\lfloor n/2 \rfloor}$ values in one half, while N consists of the $2^{\lceil n/2 \rceil}$ values in the other half. The rectangle is a cross product $P \times Q \subset M \times N$, where $P \subset M$ and $Q \subset N$.

Before we move further let us review the examples in Figure 5.2 (on page 122). The *minimum k-decomposition* of the functions into monochromatic one/zero-cubes is shown in Figure 5.3 (on page 125). In function A all the one-cubes and zero-cubes are of size 1. The rearrangement of the ON-terms in the cube space of function F results in larger cubes (of size 2). Functions C and D differ in that the arrangement of minterms in D's cube space yields a zero-cube of size 4, while the minterm distribution in C only permits zero-cubes of size 2. This contributes to the sharing in the implementation of D. The other observation of note is that the monochromatic decomposition of cubes for C and E are similar, except for the one-cube and zero-cube distribution being reversed.

In the following subsection we show how the minimum k-decomposition of cubes relates to results in two-level logic minimization.

5.3.1 k-decomposition and Two-level Minimization

Two-level logic minimization addresses the problem of minimizing the number of literals in a two-level representation of a boolean function. The objective is to generate a minimal form of the function that is *prime* and *irredundant*. A function is *irredundant* if and only if no product term can be deleted without changing the function. Conversely, *redundant* terms can be removed without changing the function. A product term is *prime* if and only if all literals in the term are necessary. Any non-prime term can be expanded by removing one or more literals without affecting the logic function. Thus there are two basic strategies: *expansion* and *redundancy removal*.

In the early years of logic design, two-level logic minimization was performed with the use of Karnaugh-maps [Koh78]. The monochromatic one/zero-cubes in our definition bear a resemblance to them. Subsequently Quine and McCluskey [McC65] introduced more sophisticated techniques for two-level minimization, like prime implicant generation, and minimum prime cover extraction that have formed the basis for almost all of two-level logic minimization today.

Regardless of the technique, the objective in two-level minimization is to generate a minimal form of the function that is prime and irredundant. We now show that a minimum k-decomposition of a function into monochromatic one/zero-cubes guarantees a prime irredundant minimized form of the function.

Lemma 1 *A minimum k-decomposition guarantees an irredundant cover of the function.*

Proof by contradiction. Suppose there exists a minimum k-decomposition that has a redundant product term in it. A product term corresponds to a one-cube, and since it is redundant, we can remove it. This implies removing the

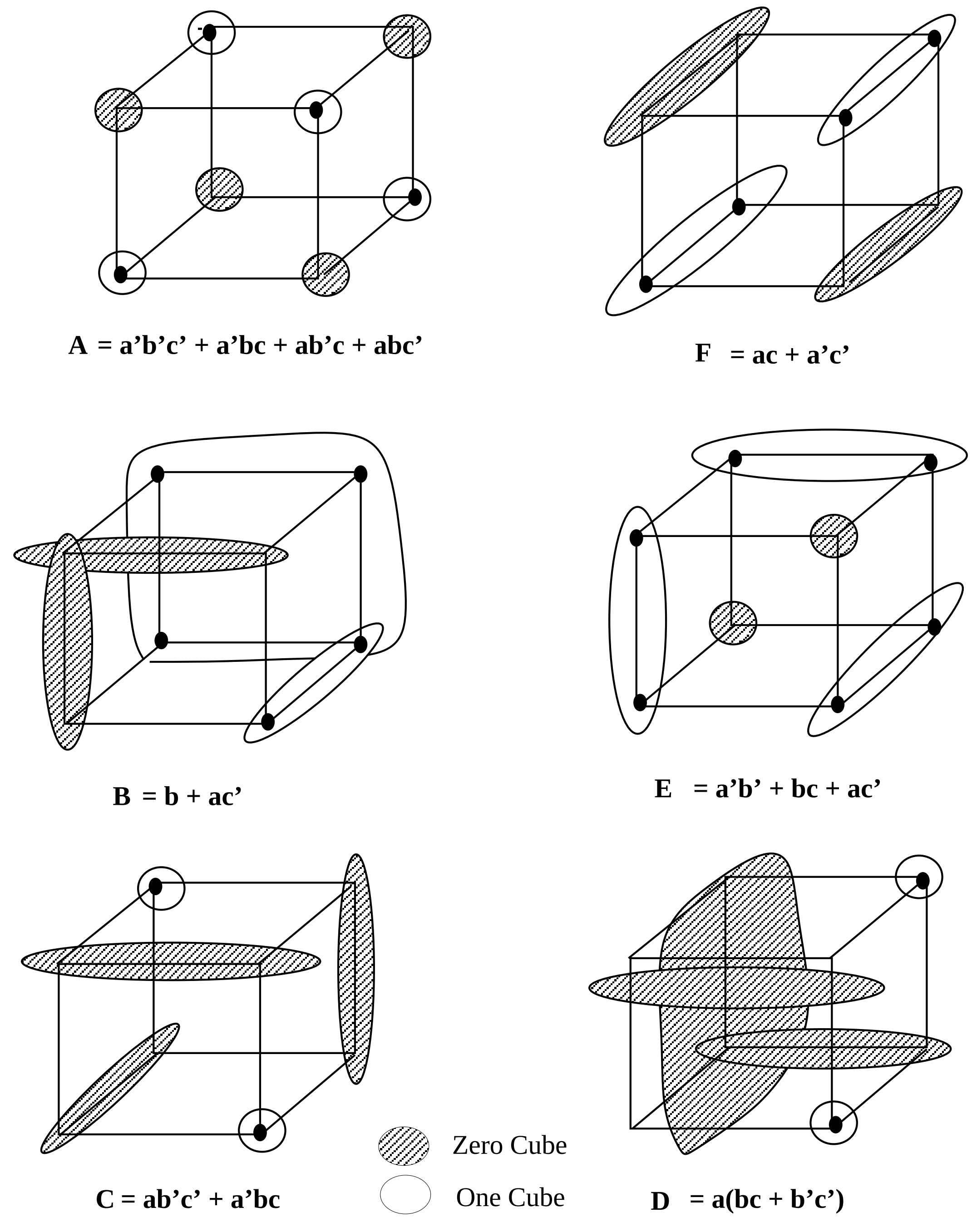


Figure 5.3: Monochromatic Cube Decomposition for 3-Variable Functions

cube from the decomposition set, giving us a decomposition (of the function) that is smaller than what we started with. This contradicts our assumption that we had a minimum k-decomposition initially.
□

Lemma 2 *A minimum k-decomposition guarantees that all the literals in the two-level minimized form of the function are necessary.*

Proof by contradiction. If a literal were not necessary in a minimum k-decomposition, then it means that it can be expanded out from its minterm. This would mean that we have just doubled the size of the one-cube containing that minterm. If the sum of the cube sizes before expansion was α, and if literal expansion does not introduce a redundant cube then the sum of the cube sizes after expansion will be at least $\alpha + 1$. This is because the smallest cube is of size 1 and it will produce an increase in size of 1 when doubled. But if the cube size did increase then this implies that we did not start with a minimum k-decomposition (see Definition 4).
□

Lemma 3 *A minimum k-decomposition of a boolean function guarantees a minimized two-level representation that is prime and irredundant.*

Proof. From the Lemma 1 on irredundancy, and Lemma 2 on primality.
□

Generating a minimum k-decomposition of monochromatic cubes for a given function is an NP-complete problem [GJ79] just as problems in traditional two-level logic minimization are NP-complete; generating all prime implicants is $O(\frac{3^n}{n})$ [BHMSV84], while extraction of a minimum prime cover is NP-complete [GJ79].

For a given function, the minimum k-decomposition need not be a unique decomposition. For example, Figure 5.4 shows a decomposition of a function into monochromatic one/zero-cubes in two distinct ways.

Before we define a formal expression to estimate gate-count complexity in terms of these monochromatic cubes, we describe certain characteristics of these cubes and intuitively motivate their role in defining this complexity. We will also use the examples in Figure 5.3 to illustrate the importance of some of these characteristics.

To start with we need to strengthen the simplistic two-level literal count measure that we assumed as the gate cost of an implementation. Ideally, the

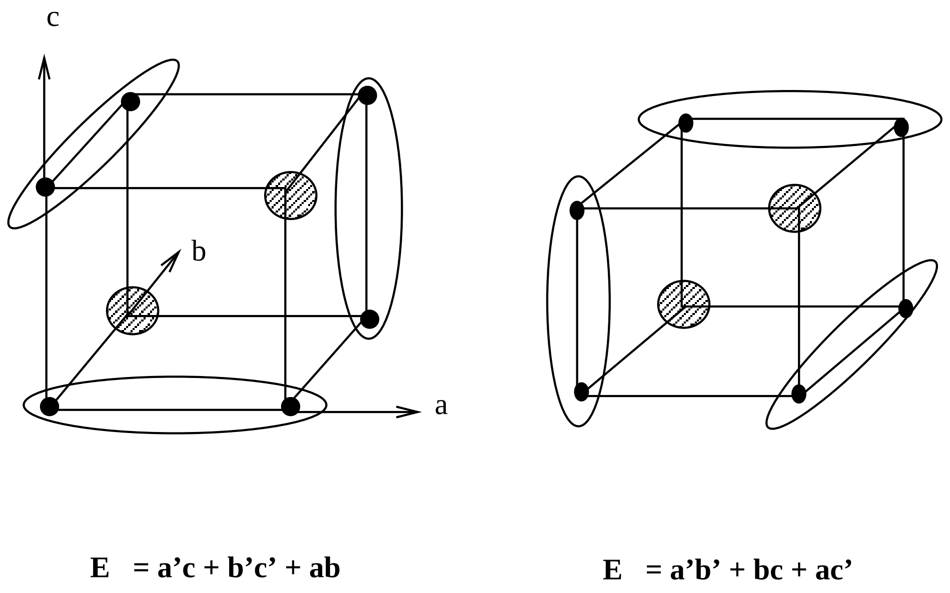


Figure 5.4: Two different Minimum Decompositions for a Function

implementation cost should include the gate-count and the total wiring area between the gates. We examine cube characteristics by studying both these costs but our formal definition is only in terms of gate-count complexity. In the following discussion we intuitively explain how cube characteristics can contribute to gate-count complexity and wiring complexity.

Distribution of One/Zero-cubes: Consider the *distribution* of the one and zero cubes in the minimum k-decomposition of a function. Since the one-cubes determine the onterms in the function and the zero-cubes determine the offterms, the distribution of one-cubes versus zero-cubes has a direct effect on the sizes of the ON and OFF sets of the function. This in turn can influence implementation cost. Suppose the number of one-cubes is much larger than the number of zero-cubes. This skew in the distribution will mean that there is a good chance that the onterms can be minimized to yield a cheap implementation. At the same time there is also a good chance that because there are few zero-cubes, implementing the OFF set of the function and inverting its output will also yield a cheap implementation. A more uniform distribution of one/zero-cube will make both these tasks more difficult. Thus a skewed cube distribution reduces the complexity of the function.

Sizes of One/Zero-cubes: The second factor is the *sizes* of the cubes in the minimum k-decomposition of a function. The sizes of the cubes determine the number of literals in a minimized form of the function. The number of literals in the minimized form in turn influences the gate count and thus the cost of the implementation. For instance, in the earlier examples, we notice that the literal count in B is less than that in A. While A has cubes that are very small in size, B has larger cubes; in fact one cube is of size 4 contributing only one literal to the minimized form.

Number of One/Zero-cubes: The third cube characteristic is the *number* of cubes in the minimum k-decomposition of a function. This is different from the issue of one-cube versus zero-cube distribution. We are concerned here about the total number of one-cubes and zero-cubes. Take functions A and F in the above example. Both have an equal ratio of one-cubes::zero-cubes - $2 :: 2, 4 :: 4$. But A has 4 one-cubes, while F has only 2 one-cubes. This is a consequence of the spatial distribution of minterms in cube space. The number of one-cubes is a reflection of the degree of sharing that is permissible between parts of logic in an implementation. The greater the number of one-cubes the more spatially distributed are the minterms of the function. As a result there is less sharing possible not only between cubes but also between the literals in the cubes. For instance in function A the cubes $ab'c'$ and abc can share the literal a in the implementation $a(bc + b'c')$; but the gain is not as much as compared to F where the single one-cube (consisting of $ab'c'$ and abc') yields ac' in the implementation. The lesser sharing permitted also implies more cost in combining the cubes and literals to implement the function. This means more gates or wires in the implementation to perform the combining.

Variance of One/Zero-cube Sizes: Finally we consider the effect of the *size distribution* of cubes in a minimum k-decomposition of the function. A function in which there is a greater variance in the sizes of the one-cubes (or zero-cubes) will tend to exhibit greater wiring complexity in the implementation. This complexity arises due to the spatial distances between gate implementations of differently sized cubes. The function B above has one-cubes of sizes 4 and 2, but it is too small an example to capture the effect. Consider a 10 variable function with a wide variance in the one-cube sizes $(2, 4, 8, 128, 256, \ldots)$ of its minimum k-decomposition. In order to implement a cube of size 2 (c_2) one would require 9 of the 10 literals to be present in the implementation. On the other hand a cube of size 256 (c_{256}) would only require 2 literals in its implementation. Regardless of how the 9 literals in c_2 are factored and implemented there is likely to be some disparity in the size of the logic required to implement c_2 versus c_{256}. Sharing amongst cubes and literals may reduce some of this disparity. But for the implementation as a whole, there is likely to be a need for more wiring when differently sized cubes are combined. This wiring effect will not be as evident in a function with a lesser variance on cube sizes. Whether the cube sizes are small or big, the cones of logic implementing them will approximately be of the same size. Again, sharing and factoring can change this to an extent.

While we have discussed each cube characteristic individually, it needs to be emphasized that the actual implementation cost is the result of a complicated interaction of all these factors along with variations introduced by multi-level logic minimization and technology mapping. We have tried to explain intuitively how cube characteristics can influence implementation cost in terms of gate-count and wiring. We now define a formal estimate for gate-count complexity, called *information content*, that captures the effects of factors 1, 2 and 3 above in the implementation cost. Since the definition does not capture the effect of factor 4 (variance in cube sizes), which contributes significantly to wiring-complexity, we only define gate-count complexity here.

Definition 6 *Given a* **minimum** *k-decomposition D_k of a boolean function f into a set of one-cubes O, and a set of zero-cubes Z, where $\mid O \mid + \mid Z \mid= k$, the* **information content** *$I(f, D_k)$ of the function is defined by:*

$$\begin{aligned} I(f, D_k) = \; & \left[\tfrac{|O|}{|O|+|Z|} * \log_2\left(\tfrac{|O|+|Z|}{|O|}\right) + \tfrac{|Z|}{|O|+|Z|} * \log_2\left(\tfrac{|O|+|Z|}{|Z|}\right)\right] * \\ & \left[\textstyle\sum_{c^1 \in O} \tfrac{|c^1|}{2^n} * \log_2\left(\tfrac{2^n}{|c^1|}\right) + \textstyle\sum_{c^0 \in Z} \tfrac{|c^0|}{2^n} * \log_2\left(\tfrac{2^n}{|c^0|}\right)\right] \end{aligned}$$

where c^1 is a one-cube and c^0 is a zero-cube.

We have defined information content as the product of two entropy functions. The *sizes of one/zero-cubes* are captured by the "log" sub terms in the entropy expression $\frac{|c^1|}{2^n}\log_2(\frac{2^n}{|c^1|})$ and $\frac{|c^0|}{2^n}\log_2(\frac{2^n}{|c^0|})$. The *number of one/zero-cubes* is captured by accumulating these sizes over $\sum_{c^1 \in O}$ and $\sum_{c^0 \in Z}$. The *distribution of one/zero-cubes* is captured by the two binary entropy functions in the definition.

Why were the two entropy functions multiplied? In information theory the entropy of two systems is usually expressed as the sum of their individual entropies. For instance suppose we wished to combine the entropies of a pair of bit strings, n-bits long and m-bits long respectively. If we denote an event by a combination of bits in a bit string, then there are 2^n possible events in the n-bit string and 2^m possible events in the m-bit string. Since entropy is expressed as the logarithm of the number of possible events, the individual entropy in the former case is n and in the latter case is m. Since the events in the n-bit string are independent of those in the m-bit string, taken together there are 2^{n+m} events in all and the entropy of the combined string pair is $\log(2^{n+m}) = n + m$. In our definition, the first entropy function describes the distribution of one/zero-cubes only in terms of the *number* of cubes, while the second entropy function describes their distribution in terms of the *minterms* in the cubes. These are not independent of each other. The distribution of the minterms amongst the cubes also influences the number of cubes, irrespective of whether they are one-cubes or zero-cubes. Since the two entropy functions have a combined effect on the gate-count complexity of the function, we take their product.

The entropy based definition of information content in cube space is an indicator of minimality in two-level function descriptions. This is because it is defined over a minimum k-decomposition of the cubes; and we proved that this is equivalent to the minimized (prime, irredundant) two-level representation of the function. In fact, the literal count in a two-level function description is part of the definition of information content in Definition 6. The term $\log_2(\frac{2^n}{|c^1|})$ represents the number of literals in the minimized cube c^1. Suppose $n = 3$, and the cube had 2 minterms, *i.e.* $\mid c^1 \mid= 2$, then the number of literals in the minimized cube would be 2. For example if a, b, c were the three variables in the function and the minterms were abc and abc', then the minimized cube is ab, which has 2 literals. This can be determined from $\log_2(\frac{2^3}{2})$.

As mentioned earlier, the objective of this definition was to capture the effects of the various cube characteristics that influence gate-count complexity in multi-level implementations. We experimented with various other estimates, before settling upon this definition as an estimate of gate-count complexity. Some of these estimates are shown below.

1. Literal count in a minimized two-level expression.

$$\log_2(\frac{2^n}{\mid c^1 \mid}) + \log_2(\frac{2^n}{\mid c^0 \mid})$$

2. Entropy function of one/zero cube sizes.

$$\sum_{c^1 \in O} \frac{\mid c^1 \mid}{2^n} * \log_2\left(\frac{2^n}{\mid c^1 \mid}\right) + \sum_{c^0 \in Z} \frac{\mid c^0 \mid}{2^n} * \log_2\left(\frac{2^n}{\mid c^0 \mid}\right)$$

3. Entropy function of number of one/zero cubes.

$$\frac{\mid O \mid}{\mid O \mid + \mid Z \mid} * \log_2\left(\frac{\mid O \mid + \mid Z \mid}{\mid O \mid}\right) + \frac{\mid Z \mid}{\mid O \mid + \mid Z \mid} * \log_2\left(\frac{\mid O \mid + \mid Z \mid}{\mid Z \mid}\right)$$

4. Weighted entropy function of cube sizes.

$$\begin{aligned} I(f, D_k) = \; & \left[\log_2\left(\tfrac{|O|+|Z|}{|O|}\right) * \textstyle\sum_{c^1 \in O} \tfrac{|c^1|}{2^n} * \log_2\left(\tfrac{2^n}{|c^1|}\right)\right] + \\ & \left[\log_2\left(\tfrac{|O|+|Z|}{|Z|}\right) * \textstyle\sum_{c^0 \in Z} \tfrac{|c^0|}{2^n} * \log_2\left(\tfrac{2^n}{|c^0|}\right)\right] \end{aligned}$$

In order to study minimality in multi-level functions and define spatial entropy in cube space we introduce a notation called the *decision tree* in the next section. It is an intermediate representation that we use to bring together a function and its implementation. The information content is a characteristic of the function and its minterm distribution. In Section 5.5 we use the decision tree to establish a relation between the spatial entropy of a given implementation and the information content of its underlying function and show how spatial entropy can capture function behavior.

5.4 Decision Tree

Given a k-decomposition for a function how does one construct an implementation for such a function? Before we answer this question let us try and visualize what the implementation must do. An implementation accepts as input a minterm m whose literals are spatially distributed. Its task is to decide if the given minterm m is an onterm, or if m is an offterm. If a naive approach is adopted this would require all the onterms or all the offterms to be examined. But since several onterms and offterms share literals and cubes, the implementation's task can be simplified. It would be sufficient if the implementation is able to decide that $m \in c^1$ or $m \in c^0$ where c^1 is a one-cube and c^0 is a zero-cube. One can intuitively see that such a decision would be easier to make when the function has a large number of onterms ($| ONset | >> | OFFSet |$) or a large number of offterms ($| ONset | << | OFFSet |$). The difficulty in making such a decision is captured by the complexity of the function. In Section 5.3 we defined the gate-count complexity as the information content defined over the k-decomposition of the one/zero-cubes. As we shall see later, spatial entropy is defined as the *effort* required to make such a decision, thus capturing the measure of difficulty, or the measure of information content.

Suppose we are given a k-decomposition of the one/zero-cubes of a function. We are now interested in understanding how an implementation uses this k-decomposition to make decisions of the above nature with respect to an input minterm m. We introduce a representation called *decision tree* that illustrates this decision process. The decision tree also gives us an insight into constructing an implementation for the function from its k-decomposition.

We first illustrate the decision process for a simple example, following which we formally define it. Consider a 4 variable function $f = ab + cd$. Figure 5.5 shows the k-decomposition for this function, where $k = 6$. The arrows indicate the one and zero-cubes ($k = 6$) which are numbered below.

1. $a'd'$ - zero-cube.
2. $b'd'$ - zero-cube.
3. $b'c'$ - zero-cube.
4. $a'c'd$ - zero-cube.

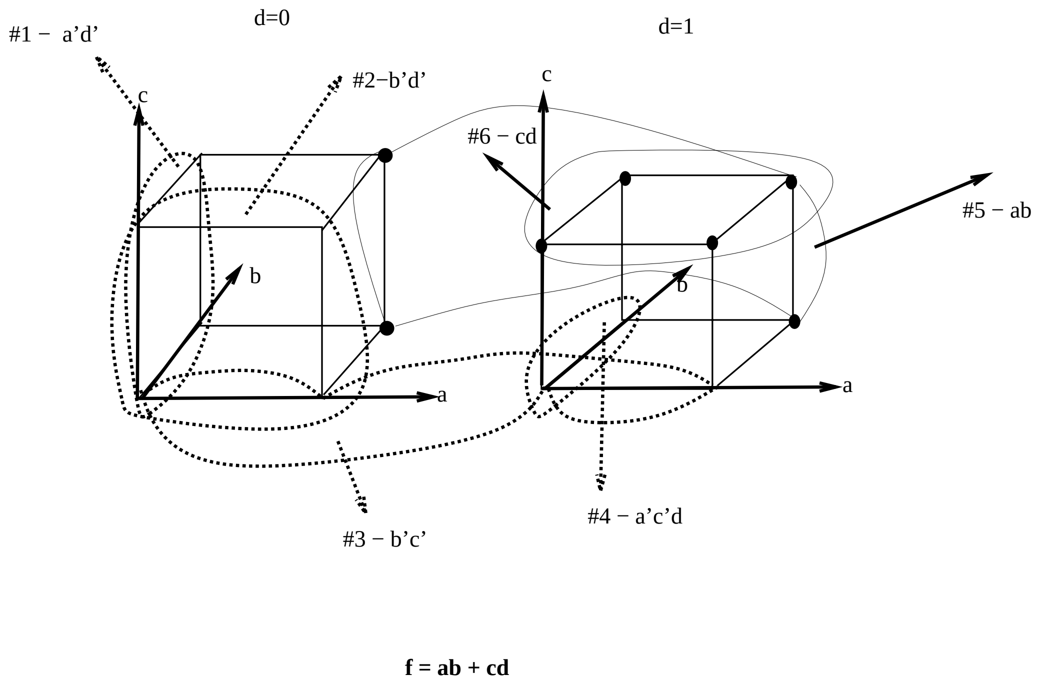


Figure 5.5: Minimum 6-decomposition of $f = ab + cd$

5. ab - one-cube.

6. cd - one-cube.

Using the cube numbers to identify them, we have a one-cube set $O = \{5, 6\}$ and a zero-cube set $Z = \{1, 2, 3, 4\}$.

The decision that we would like to make is: given an input minterm m with an assignment of boolean values to the literals a, b, c and d which of these 6 cubes does it belong to? More specifically, we are interested in determining a subset M of monochromatic cubes, where $M \subset O$ or $M \subset Z$ such that the input minterm $m \in M$. At the outset the input decision space of one/zero-cubes is represented by the 6-decomposition $D_6 = \{1, 2, 3, 4, 5^*, 6^*\}$, where the one-cubes are marked with a $*$. With successive assignments to the input literals a, b, c and d we can start making decisions to generate smaller sets of current candidate cubes. Suppose we started with the literal a. The assignment $a = 1$ tells us that the variable a is a 1 in the input m. From

Figure 5.5 this tells us that the input m must lie in one of the cubes $2, 3, 5^*, 6^*$. Similarly, if $a = 0$ then m must lie in cubes $1, 2, 3, 4, 6^*$. Thus the candidate cube sets now are $\{2, 3, 5^*, 6^*\}$ and $\{1, 2, 3, 4, 6^*\}$. Since neither of them is a monochromatic cube set we continue making assignments. This process is illustrated in Figure 5.6. We now consider assignments to the literal b. For $a = 1, b = 1$, we discover that the input must lie in cubes $5^*, 6^*$. Since we now have a monochromatic cube set the decision process terminates along this path. This tells us that if the input m is assigned $a = 1, b = 1$ then it is part of some minterm in the one-cubes 5^* or 6^* in the ON-set. The assignments $a = 1, b = 0$; $a = 0, b = 1$ and $a = 0, b = 0$ all yield subsets of D_6 none of which are monochromatic. Hence the literal assignment continues along these paths. The decision process completes when every path terminates in a monochromatic cube set M, where $M \subset \{O, Z\}$. As Figure 5.6 indicates, this yields a tree that has captured the decision process. This is called the *decision tree*. For a given k-decomposition D_k the decision tree is not always unique. A different sequence of literal assignments can yield a different decision tree. Figure 5.7 illustrates the decision tree for the assignment sequence a, c, b, d for the same 6-decomposition of $f = ab + cd$. This is different from the one in Figure 5.6 which used the assignment sequence a, b, c, d.

We are now ready to state some definitions. We first define an ordering π on the input variable set I. The ordering function π generates a permutation on the indices of the input variable set I by assigning an index to each variable in I. Given a set of one/zero-cubes representing the k-decomposition D_k for a function, and an ordering π on the input variable set I, a decision tree $T(D_k, \pi)$ is denoted by a directed acyclic graph (V, E). Every node $(v \in V)$ in the tree is associated with a set of cubes, $C_v \subset D_k$. We shall call this the *cube set* C_v. For the *root* node v_{root}, the cube set is the k-decomposition itself, $C_{v_{root}} = D_k$. There are two types of nodes in the tree: *internal nodes* and *leaves*. Every internal node v is labeled with a boolean input variable. The leaf nodes are not labeled. A directed edge $e = (v, w) \in E$ corresponds to an assignment of 0 or 1 to $label_v$. Every internal node v has two children - a one-child $child_{v^1}$, which is the node along the edge that has $label_v = 1$, and a zero-child $child_{v^0}$, which is the node along the edge that has $label_v = 0$. The leaves of the tree do not have any children and correspond to monochromatic cube subsets of the k-decomposition D_k. That is, if O and Z are the monochromatic one and zero-cube sets respectively for the decomposition D_k $(O \cup Z = D_k)$, then a leaf v of the tree is represented by

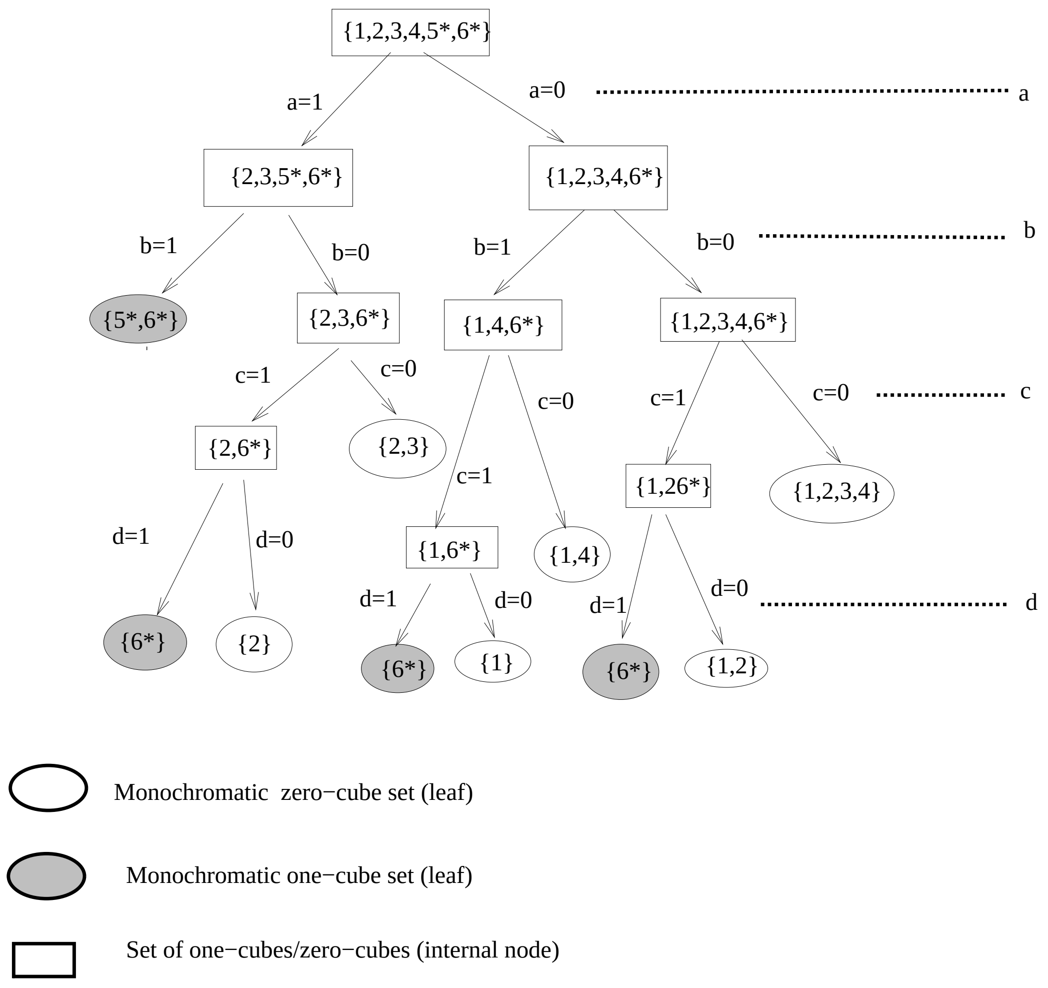


Figure 5.6: Decision Tree for $D_6(f)$ with ordering a, b, c, d

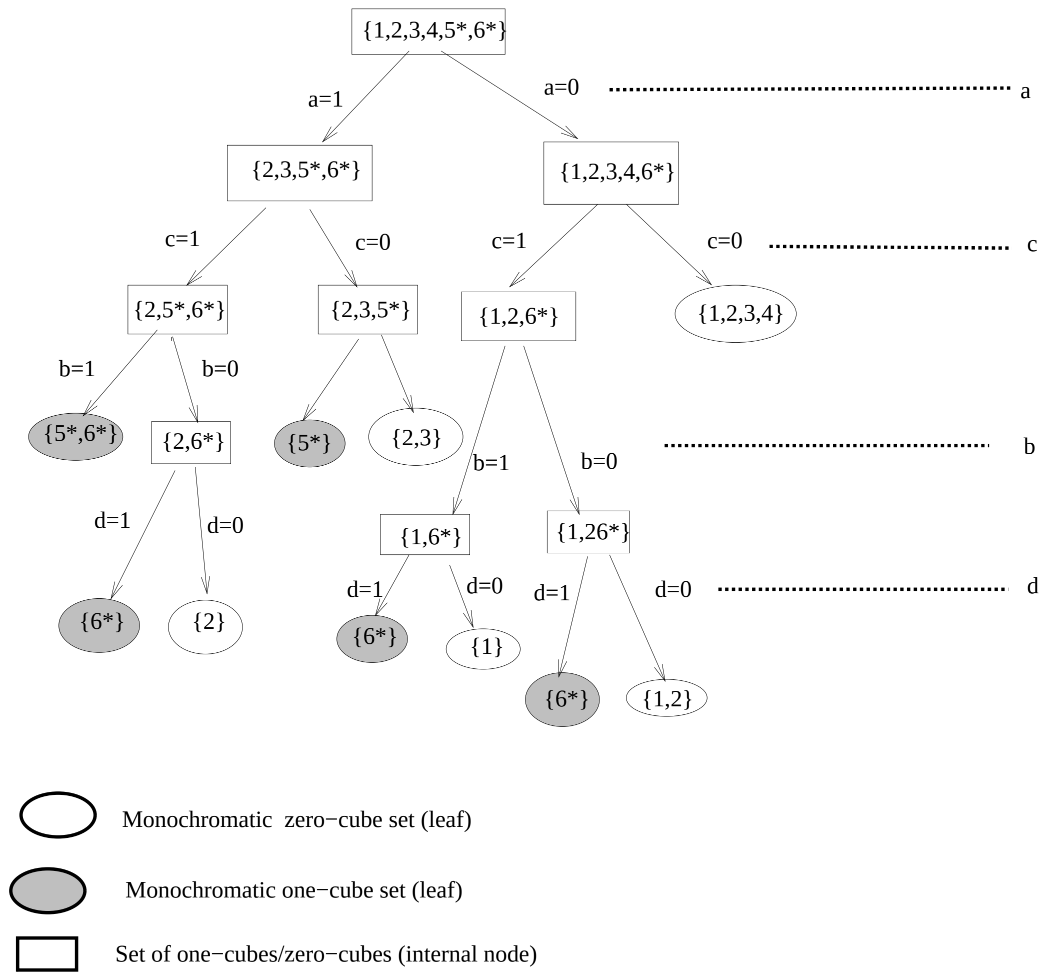


Figure 5.7: Decision Tree for $D_6(f)$ with ordering a, c, b, d

a set of monochromatic cubes M, where $M = C_v$, and $(C_v \subset O) \vee (C_v \subset Z)$.

The permutation of indices generated by the the function π for each variable in the input set I can be used to order the input variables. Thus in Figure 5.6 $\pi(a) = 4$, $\pi(b) = 3$, $\pi(c) = 2$ and $\pi(d) = 1$. For convenience we shall denote the imposed ordering $a \succ b \succ c \succ d$ by $\pi = \langle a, b, c, d \rangle$. Since internal nodes in the decision tree are labeled with input variable names $\in I$, the ordering π also imposes a partial order $\succ$ of *levels* l on the node set V and the cube sets C_v associated with them. The levels in the node set are distances from the leaves. The leaf nodes are at level 0 and the root node is at level n, where n is the number of input variables, $\mid I \mid$. This creates a set of ordered partitions of the node set $V = \{V_n \succ V_{n-1} \succ \ldots \succ V_2 \succ V_1\}$. Each element of the partition is a set V_i consisting of all nodes at level i. Since each node $v \in V_i$ is associated with a cube set C_v, we interchangeably refer to the elements of the set V_i as nodes (v) or cube sets (C_v). We define a *partition set*, $\mathcal{P}_{V_i}$ as the set consisting of the cube sets associated with each node in V_i. Therefore $\mathcal{P}_{V_i} = \cup_{v \in V_i} C_v$. Thus a partition set $\mathcal{P}_{V_i}$ is a set of sets. We can now define a maximal partition set.

Definition 7 *Given a set of nodes V_i at level i, the* **maximal partition set** $\mathcal{P}^{max}_{V_i}$ *at this level is defined as*

$$\mathcal{P}^{max}_{V_i} = \cup_{v \in V_i} C_v \ \ such\ that\ \ (C_v \in \mathcal{P}^{max}_{V_i})\ \ iff \quad C_v \not\subset C_w \ \ for\ w \in V_i.$$

It contains all the cube sets in V_i such that no cube set is completely contained inside another. For every cube set $C_v \in \mathcal{P}^{max}_{V_i}$, the cube sets C_w that are subsets of C_v are said to be *covered* by C_v.

Before we look at an example we make one other observation. Since every node $v \in V_i$ is labeled with the same input variable x_j, we can associate each element of the partition $V = \{V_n \succ V_{n-1} \succ \ldots \succ V_2 \succ V_1\}$ with a unique input variable. This means we can rewrite $\mathcal{P}^{max}_{V_i}$ as $\mathcal{P}^{max}_{i,x_j}$ - the *maximal partition set associated with an input variable $x_j \in I$ at level i, in a decision tree $T(D_k, \pi)$*. To avoid confusion, we shall adopt the convention of using the subscript j for input variables and i for levels as far as possible.

Let us now go back to the decision tree in Figure 5.6. The maximal partition sets at each level i associated with each variable in $\pi = \langle a, b, c, d \rangle$ are shown below.

$$\mathcal{P}^{max}_{4,a} = \{[1, 2, 3, 4, 5^*, 6^*]\}$$

$$\mathcal{P}^{max}_{3,b} = \{[2,3,5^*],\ [1,2,3,4,6^*]\}$$
$$\mathcal{P}^{max}_{2,c} = \{[1,2,3,4,6^*]\}$$
$$\mathcal{P}^{max}_{1,d} = \{[1,2,6^*]\}$$

The maximal partition set at a level consists of a set of cube sets at that level. Every cube set in the maximal partition set has the characteristic that the subtree below it generates monochromatic leaf cube sets that are *unique* (*i.e.* not generated by any other element of the maximal partition set at this level) and *maximal* (*i.e.* it generates the largest possible monochromatic leaf cube set).

We now define the *signature* $\mathcal{S}$ of a decision tree as follows.

Definition 8 *The signature $\mathcal{S}$ of a decision tree $T(D_k, \pi)$ is the union of the maximal partition sets associated with each input variable in the tree.*

$$\mathcal{S} = \bigcup_{x_j \in I} \mathcal{P}^{max}_{i,x_j}$$

where i is the level associated with variable x_j. The **size** *of a signature $\mid \mathcal{S} \mid$ is the cardinality of the set $\mathcal{S}$.*

The signature of a decision tree is a set of all the internal nodes that make decisions to generate monochromatic cube sets at the leaves , such that the cube set C_v associated with each such node v is unique and covers all other cube sets C_w that lie at the same level.

The signature for the decision tree in Figure 5.6 is given by $\mathcal{S} = \{[1,2,3,4,5^*,6^*], [2,3,5^*],\ [1,2,3,4,6^*][1,2,6^*]\}$, and $\mathcal{S} \subset 2^{D_k}$.

Having introduced the decision tree we now return to the question addressed at the beginning of this section. Rephrasing it, we ask how can an implementation for a function be constructed, given a decision tree $T(D_k, \pi)$? Before we describe the procedure to construct this implementation some assumptions about our implementation model are necessary. We assume that the implementation is derived from a multi-level minimal expression representing a boolean function. The implementation assumes primitive technology mapping and uses only 2-input AND/OR/NOT gates corresponding to the operators in the minimal expression of the function. A *node* in the implementation represents the output of a collection of gates that implement a partial function. The nesting depth in the minimal expression creates levels in the implementation. Level 1 consists of the input literals, and the levels are numbered in increasing order moving from nodes (or partial functions) formed at the innermost level to the output node that has the maximum level number. Thus these levels impose a partial ordering on the nodes in the implementation. Given a pair of nodes v, w belonging to levels i, i' respectively, if $i < i'$ then node v is closer to the inputs than node w. The implementation for the minimal expression

$$f = (c(a + b) + abd)$$

is shown in Figure 5.8 along with its partial functions.

We now describe the procedure to construct an implementation from a given decision tree $T(D_k, \pi)$. The procedure starts bottom up from the leaves (or monochromatic cube sets) of the tree. Using the ordering of the variables specified by π, the procedure examines the cube sets C_v associated with each node v at level i. Every cube set C_v at level i is replaced by a partial function

$$\delta f^i(C_v) = x_j.\delta f^{i-1}(child_{v^1}) + x'_j.\delta f^{i-1}(child_{v^0})$$

where x_j is the input variable associated with the node v, $\delta f^{i-1}(child_{v^1})$ is the partial function from the 1-child of v at level $i-1$, and $\delta f^{i-1}(child_{v^0})$ is the partial function from the 0-child of v at level $i-1$. Since the $child_{v^1}$ is a node where the value of $x_j = 1$ and the $child_{v^0}$ is a node where the value of $x_j = 0$, we can rewrite the partial function as

$$\delta f^i = x_j.\delta f^{i-1}_{x_j} + x'_j.\delta f^{i-1}_{x'_j} \tag{5.1}$$

This is like the Shannon expansion of the function at level i except that it is built bottom up from Shannon cofactors (with respect to x_j, x'_j) that already exist at the level below. The value of this partial function at any level i is governed by a few rules. These rules are just manifestations of simple boolean optimizations.

1. At the leaves of the tree (level 0) there are only monochromatic one-cube sets and monochromatic zero-cube sets. If the leaf is a monochromatic one-cube set then the partial function δf^0 has the value 1. If the leaf is a monochromatic zero-cube set then the partial function δf^0 has the value 0. This corresponds to optimizations of the form $x'.1 \Rightarrow x'$ or $x.0 \Rightarrow 0$.

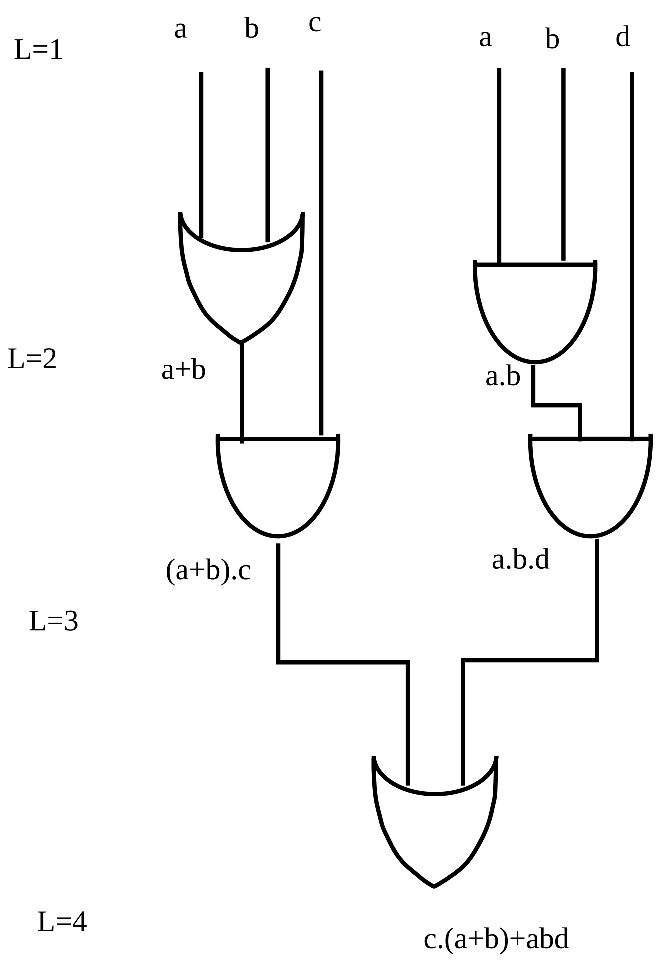


Figure 5.8: Implementation for f=(c(a+b)+abd)

2. For any level $i > 0$, suppose $C_{child_{v1}}$ is the cube set at the one-child of v and $C_{child_{v0}}$ is the cube set at the zero-child of v. Then the partial function δf^i defined in Equation 5.1 can be reduced to

$$\delta f^i = \delta f^{i-1}_{x_j} + x'_j.\delta f^{i-1}_{x'_j}$$

provided $\forall c^1;\ (c^1 \in C_{child_{v1}} \quad \Rightarrow \quad c^1 \in C_{child_{v0}})$
This says that if all the one-cubes that belong to $C_{child_{v1}}$ are contained in $C_{child_{v0}}$, then the literal x_i is not making any useful decision and hence is not part of the function. This corresponds to an optimization of the form $b' + bd \Rightarrow b' + d$. Similarly, if it is true that every $(c^1 \in C_{child_{v0}}) \quad \Rightarrow \quad (c^1 \in C_{child_{v1}})$ then the partial function δf^i reduces to

$$\delta f^i = x_j.\delta f^{i-1}_{x_j} + \delta f^{i-1}_{x'_j}$$

Here literal x'_j is not making any useful decision in the tree and hence does not appear in the partial function. The corresponding optimization is $b + b'd \Rightarrow b + d$.

3. For any level $i > 0$, if the cube set at the one-child of v, $C_{child_{v1}}$ is identical to the cube set C_v at v, then $\delta f^i = \delta f^{i-1}_{x_j}$ If the cube set at the zero-child of v, $C_{child_{v0}}$ is identical to the cube set C_v at v, then $\delta f^i = \delta f^{i-1}_{x'_j}$ This performs the optimization $b'cd + bcd \Rightarrow cd$, saying that neither x_j nor x'_j are making any useful decisions.

4. For any level $i > 0$ if $C_{child_{v1}} \notin P^{max}_{i-1,x_k}$ then δf^i can be reduced to

$$\delta f^i = x'_j.\delta f^{i-1}_{x'_j}$$

Similarly if $C_{child_{v0}} \notin P^{max}_{i-1,x_k}$ then δf^i can be reduced to

$$\delta f^i = x_j.\delta f^{i-1}_{x_j}$$

This says that if the cube sets at the one-child or zero-child of v are not elements of the maximal partition set at that level, then the partial functions associated with them can be ignored. Since there will be maximal partition set elements that covers these cube sets $(C_{child_{v1}} C_{child_{v0}})$, the partial functions associated with these elements will cover $\delta f^{i-1}_{x'_j}$ and $\delta f^{i-1}_{x_j}$, and hence they can be ignored here. This corresponds to the optimization $(ab + b'cd) + cd \Rightarrow (ab + cd)$.

In order to construct an implementation from the decision tree the partial functions computed at each level need to be replaced by gates. But our simple 2-input AND/OR/INVERTER gate model will not suffice for this purpose. This is because the partial function constructed at a level may sometimes require more than one such primitive gate. This happens in two instances.

1. If every component of the partial function $\delta f = x.\delta f_x + x'.\delta f_{x'}$ is making a contribution to the decision process, then a pair of AND gates and an OR gate would be needed.

2. Similarly when the partial function is of the form $\delta f = x.\delta f_x + \delta f_{x'}$ one AND and one OR gate would be needed.

So we extend our implementation model to include a 4-input AND-OR gate AOR_4 and a 3-input AND-OR gate AOR_3. Given that such complex gates do exist in practical implementations this extension does not seem unreasonable.

We can now construct the implementation bottom-up from the decision tree by replacing the partial function at every node by a gate in the implementation. Thus the implementation is built incrementally. The set of gates corresponding to the partial function at the root node is the multi-level implementation of the entire function. The internal nodes in the implementation correspond to maximal partition sets in the decision tree at all levels > 1. This is because it is only these nodes in the decision tree that contribute to the decision process, and thus do not get optimized only during construction. The correspondence between input nodes in the implementation and inputs in the decision tree is difficult to establish because of the decision tree structure that forces the inputs to enter sequentially, in contrast to an implementation where the inputs enter in parallel. So we shall restrict our attention to the internal nodes of an implementation. Let us look at an example.

Figure 5.9 shows the cube space distribution for the function $f = ac + bc + abd$ implemented in Figure 5.8. It has 7 one/zero cubes (not shown). The one-cube set O and zero-cube set Z in the minimum 7-decomposition are numbered below.

1. ac - one-cube.

2. bc - one-cube.

3. abd - one-cube.

4. $c'd'$ - zero-cube.

5. $b'c'$ - zero-cube.

6. $a'b'$ - zero-cube.

7. $a'c'$ - zero-cube.

Figure 5.11 shows the decision tree with the ordering $\pi : \langle c, d, a, b\rangle$, along with the constructed partial functions. Consider the nodes at the level 3 associated with d. For cube set $\{1^*, 2^*, 3^*, 6\}$, the partial function is computed as follows. Since the 1-child of this node has the same cube set $\{1^*, 2^*, 3^*, 6\}$, by rule 3, the partial function at this node is the same as the partial function at its 1-child $a+b$. The partial function associated with the other cube set at level 3 $\{3^*, 4, 5, 6, 7\}$ is computed as follows. Since the 0-child of this node is a monochromatic cube set, rule 1 applies, and $\delta f^i = x_j.f^{i-1}_{x_j}$, $\delta f = d.ab$. The factored form representation of the function at the root node yields the multi-level implementation for this decision tree. The nodes in the implementation correspond to the partial functions at the maximal partition sets at each level of the decision tree. Figure 5.10 illustrates the constructed implementation.

Since the implementation is derived from the decision tree, a different ordering π' will possibly yield a different decision tree and a different implementation. Figure 5.12 illustrates this for the ordering $\pi' = \langle d, c, a, b, \rangle$. Even though each ordering yields a different decision tree the trees may be similar in other respects. For instance we can define *cost-equivalent* decision trees as follows.

Definition 9 *Two decision trees $T(D(f), \pi_1)$, $T(D(f), \pi_2)$ have the same* **cost** *if their signatures $\mathcal{S}_1$ and $\mathcal{S}_2$ have the same sizes.*

For the class of implementations that are derived from decision trees, we can state the following.

Lemma 4 *If a pair of decision trees $T(D(f), \pi_1)$, $T(D(f), \pi_2)$ have the same cost, then the respective implementations $\mathcal{I}_{\pi_1}$ and $\mathcal{I}_{\pi_2}$ derived from them have the same number of nodes.*

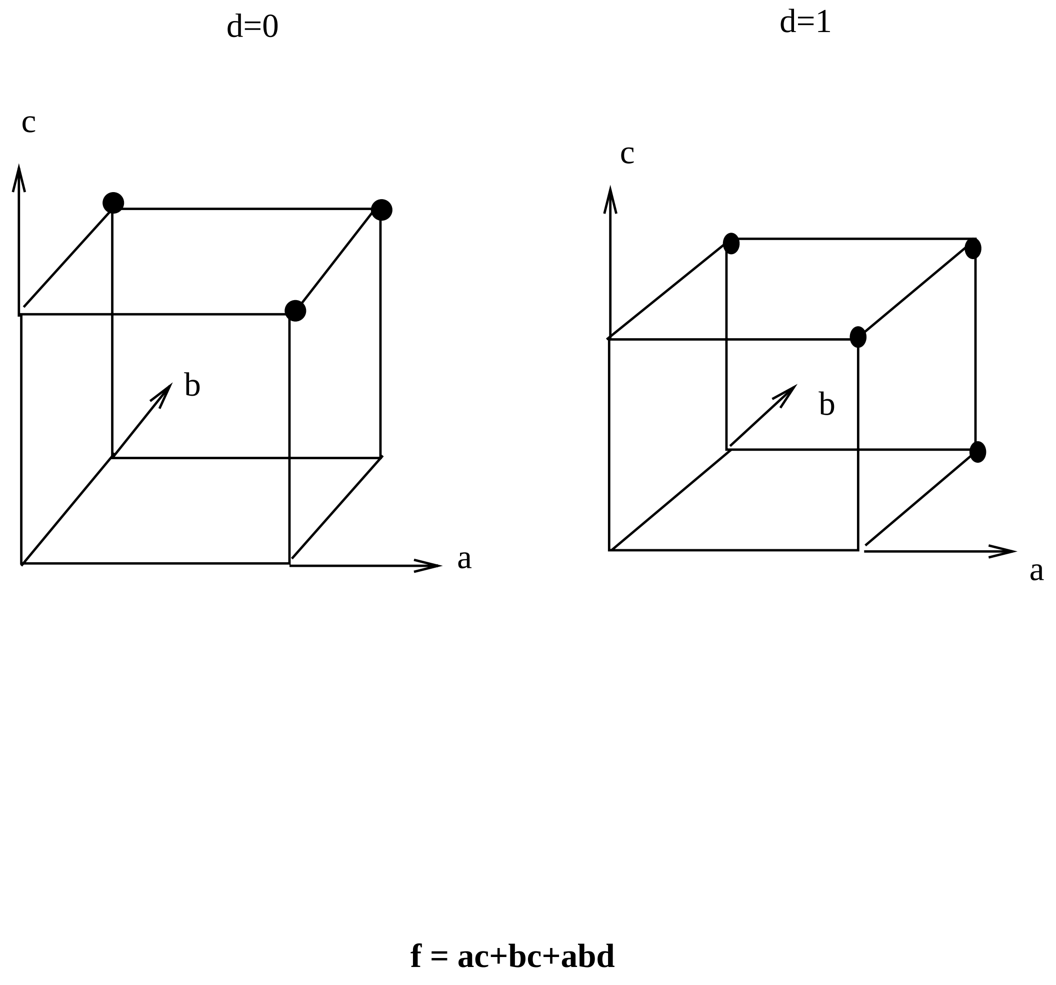


Figure 5.9: Cube Space Distribution for $f = ac + bc + abd$

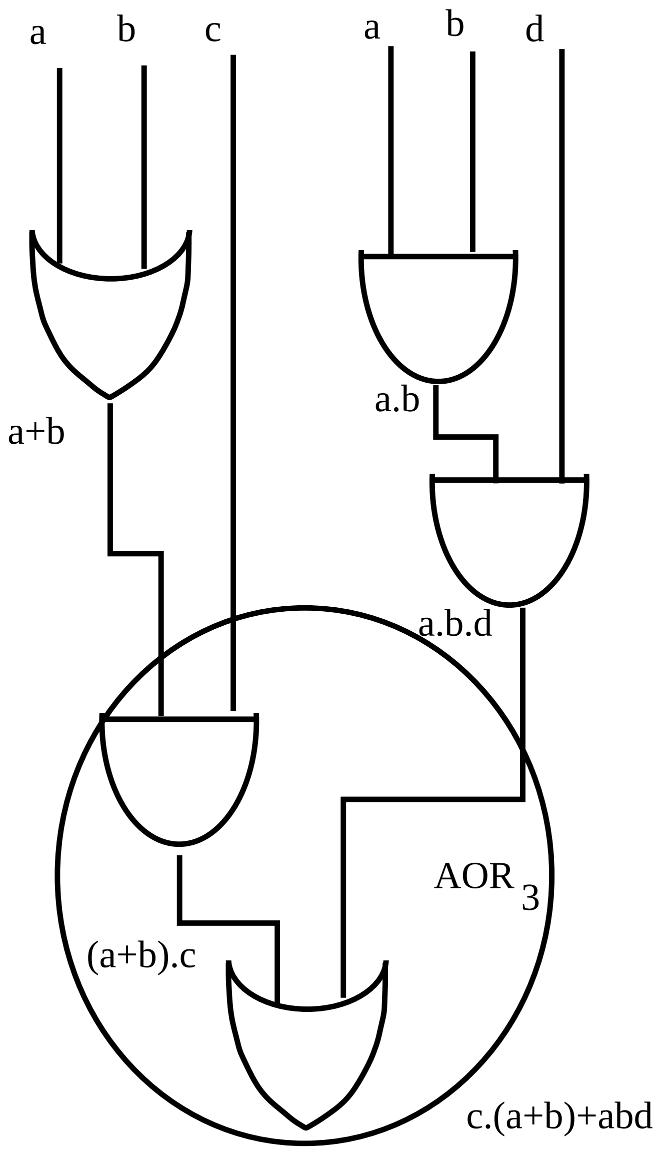
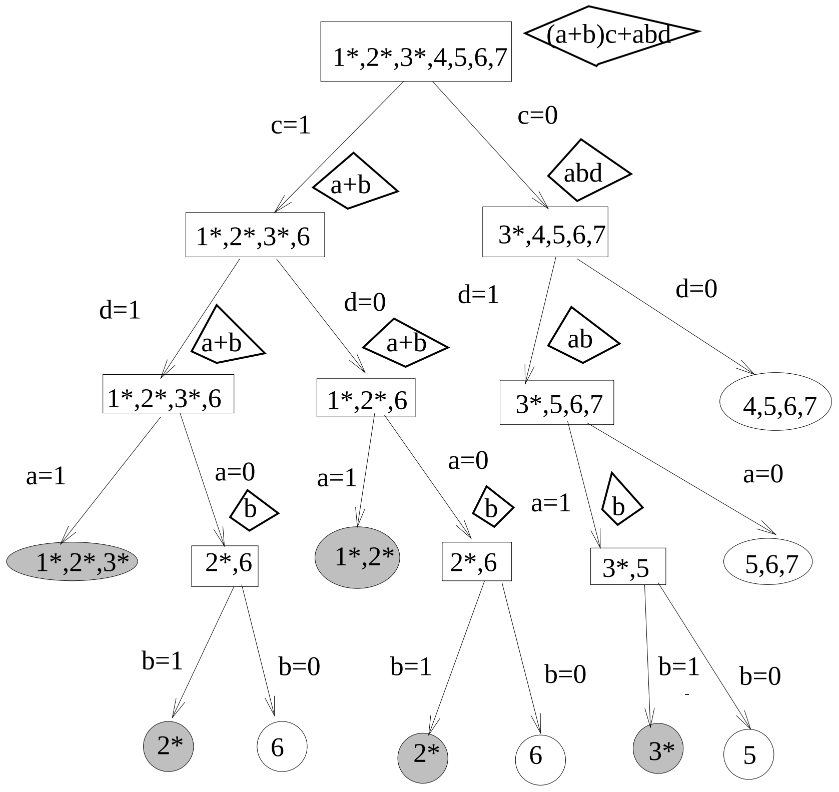
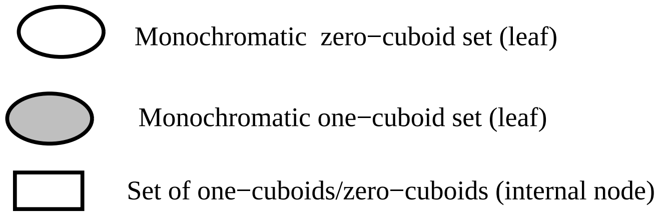


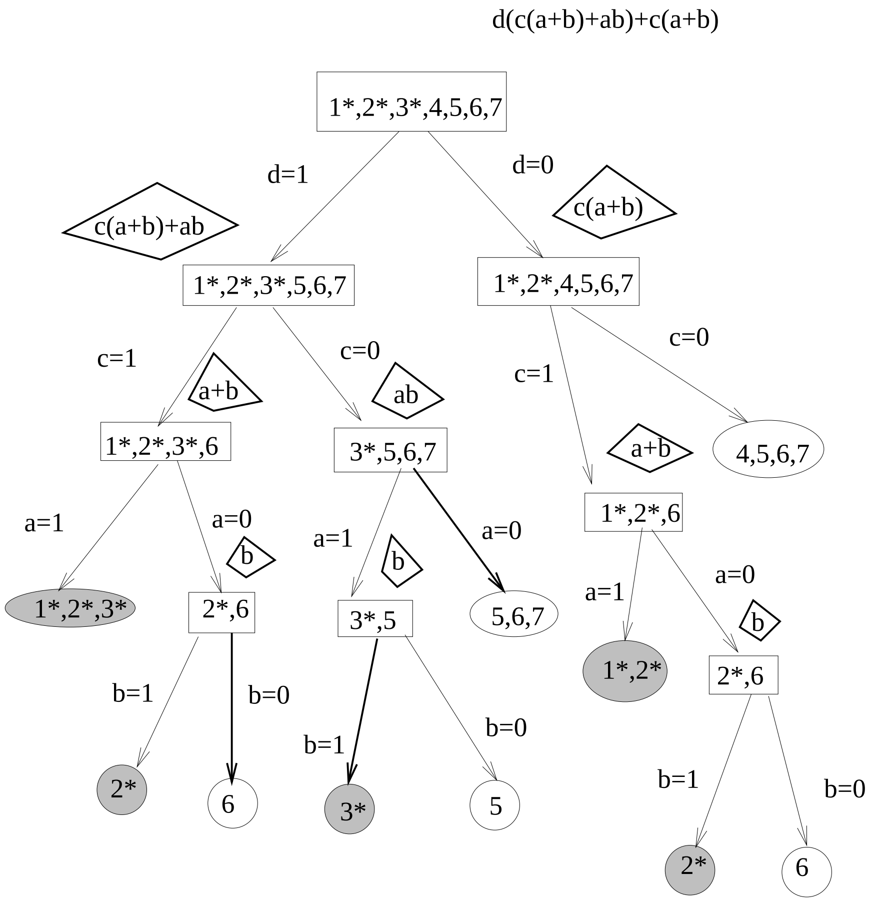
Figure 5.10: Implementation Constructed from Decision Tree in Figure 5.10





Monochromatic zero−cuboid set (leaf)
Monochromatic one−cuboid set (leaf)
Set of one−cuboids/zero−cuboids (internal node)

Figure 5.11: Decision Tree for $f = ac + bc + abd$ with $\pi : c, d, a, b$



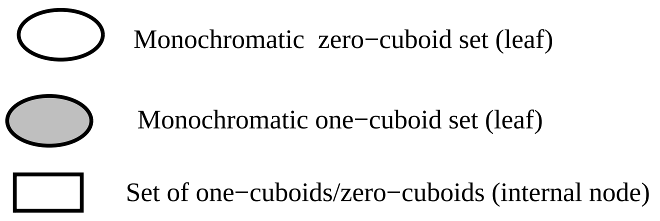

Monochromatic zero−cuboid set (leaf)
Monochromatic one−cuboid set (leaf)
Set of one−cuboids/zero−cuboids (internal node)

Figure 5.12: Decision Tree for $f = ac + bc + abd$ with $\pi : d, c, a, b$

Proof. By construction. Since the implementations are derived from the decision tree the number of nodes in the implementation is equal to the number of maximal partition sets in the decision tree (by construction). Since the decision trees have the same cost their signatures have the same number of maximal partition sets and hence their implementations have the same number of nodes.
□

We now refine this cost measure between a pair of implementations (or decision trees) to talk about cost-equivalence with respect to sub-trees in a pair of decision trees.

Consider an ordering $\pi = \langle x_{j1}, \ldots, x_{jn} \rangle$. By moving backwards in the ordering, we can create a series of partitions P_i^π that are subsets of the input variable set, consistent with the ordering π. The first partition P_1^π corresponds to $\{\{x_{jn}\}, \{x_{jn-1}, \ldots, x_{j1}\}\}$, the second corresponds to $\{\{x_{jn}, x_{jn-1}\}, \{x_{jn-2}, \ldots, x_{j1}\}\}$ and so on until the last partition P'_n equals $\{\{x_{jn}, x_{jn-1}, \ldots, x_1\}, \{\}\}$.

Consider two decision trees with orderings π_1 and π_2 such that $\exists k$ where $P_k^{\pi_1} = P_k^{\pi_2}$. That is, the subset of inputs in both these partitions is the same, even though the order in which they entered their respective partitions may be different. The k in the partition $P_k^{\pi_2} P_k^{\pi_1}$ corresponds to the level k in the implementation and is referred to as the *join level.*

Definition 10 *Given a pair of partitions $P_k^{\pi_1}, P_k^{\pi_2}$ defined as above, with $P_k^{\pi_1} = P_k^{\pi_2}$, their associated implementations $\mathcal{I}_{\pi_1}$ and $\mathcal{I}_{\pi_2}$ have the same cost with respect to this partition P_k if the total number of nodes in all the partial functions in $\mathcal{I}_{\pi_1}$ is equal to the total number of nodes in all the partial functions in $\mathcal{I}_{\pi_2}$ computed till join level k.*

In Figure 5.11 and Figure 5.12 the two implementations derived from the respective decision trees have the same cost with respect to the partition $\{\{a, b\}, \{c, d\}\}$ of the input set $\{a, b, c, d\}$.

We note that the partial functions at the join level k are functions of variables in the set $\{x_{j_k}, \ldots, x_{j_n}\}$. They represent the cofactors of the original function f with respect to the set $\{x_{j_1}, \ldots, x_{j_{k-1}}\}$. Regardless of the order in which the inputs $\{x_{j_1}, \ldots, x_{j_{k-1}}\}$ were cofactored, we would finally arrive with the same set of partial functions and the same number of maximal partition sets at the join level.

Unfortunately, our current notion of equivalence between implementations is much tighter than what we would ideally like. It needs to be expanded. If we can relax the assumption that identical partitions are needed at the join level, it will help us derive a class of *compatible* implementations at the join level giving us a broader notion of equivalence.

5.5 Spatial Entropy and Information Content

In the previous section we introduced the decision tree as an intermediate representation to relate a function with a broad class of implementations. The top-down reduction of the k-decomposition of the function into monochromatic cube sets generates the nodes and the leaves in the decision tree. The bottom-up construction of the k-decomposition from the leaves of the decision tree generates the implementation for the function.

We have defined the information content $I(f, D_k)$ of a function f in terms of its k-decomposition D_k that appears at the root of all decision trees for f. In order to establish a relationship between the spatial entropy of an implementation of f and its information content $I(f, D_k)$ we first define the spatial entropy of a decision tree, in terms of the one-cubes and zero-cubes in the decision tree. Since a decision tree corresponds to an implementation we can then relate this to the spatial entropy of an implementation as described in Chapter 3.

The spatial entropy of an implementation was defined in Chapter 3 as the dynamic communication effort required to compute a value (of 1 or 0) for a function. In the previous section we showed that the task of computing this (1 or 0) value is equivalent to the task of reducing the k-decomposition at the root of the decision tree to the monochromatic cube sets at the leaves. The paths from the root to each leaf with a monochromatic one-cube set describe the input onterms (1 values) that belong to this one-cube set. Similarly, the paths from the root to each leaf with a monochromatic zero-cube set describe the offterms (0 values) that belong to this zero-cube set. The effort required to generate all these monochromatic cube leaf sets from the root node is the *spatial entropy of the decision tree.*

How can we express this effort formally? For this we use the notion of information content defined in Definition 5. The spatial entropy of the decision tree is the *reduction in the information content* $I(f, D_k)$ of the function over all the levels in the decision tree. At the root node, $I(f_{root}, D_k) = I(f, D_k)$, which we reproduce below, expresses the information content over the entire function.

$$\begin{aligned} I(f, D_k) = \; & \left[\frac{|O|}{|O|+|Z|} * \log_2\left(\frac{|O|+|Z|}{|O|}\right) + \frac{|Z|}{|O|+|Z|} * \log_2\left(\frac{|O|+|Z|}{|Z|}\right)\right] * \\ & \left[\sum_{c^1 \in O} \frac{|c^1|}{2^n} * \log_2\left(\frac{2^n}{|c^1|}\right) + \sum_{c^0 \in Z} \frac{|c^0|}{2^n} * \log_2\left(\frac{2^n}{|c^0|}\right)\right] \end{aligned}$$

At the leaf node, since there are only one-cubes or only zero-cubes, the first entropy function is 0, because either $| O |$ or $| Z |$ is 0, or the log expression reducing to log(1) equals 0. As a result of this the information content at a leaf node v $I(f_v, D_k)$ is 0. At every level $i > 0$ in the decision tree the nodes $v \in V_i$ denote cofactors of the original function f with respect to all literal assignment(s) along the paths from the root to each of these nodes. We are only interested in those nodes whose cube sets C_v are elements of the maximal partition $\mathcal{P}^{max}_{i,x_j}$ at that level. This is because a cube set $C_v \in \mathcal{P}^{max}_{i,x_j}$ is a cube set whose one/zero-cube decomposition covers the one/zero-cube decomposition of other cube sets that might appear at this level. Hence any decisions made with respect to this cube set would be a superset of the decisions made with respect to all the others it covers. For example in Figure 5.6 (Page 5.6), the cube set $\{1, 2, 3, 4, 6*\} \in \mathcal{P}^{max}_{1,c}$ at the level corresponding to input variable c is a cube set that covers cube sets $\{1, 4, 6*\}$ and $\{2, 3, 6*\}$. This node represents the cofactors of the function $f = ab + cd$ with respect to 3 literal assignments, $[a = 0, b = 1]$, $[a = 0, b = 0]$, $[a = 1, b = 0]$, all of which yield cd.

Given that we are only interested in nodes $v \in \mathcal{P}^{max}_{i,x_j}$ we start by defining the information content with respect to such a node v as

$$\begin{aligned} I(f_v, D_k) = \; & \left[\frac{|O_v|}{|O_v|+|Z_v|} * \log_2\left(\frac{|O_v|+|Z_v|}{|O_v|}\right) + \frac{|Z_v|}{|O_v|+|Z_v|} * \log_2\left(\frac{|O_v|+|Z_v|}{|Z_v|}\right)\right] * \\ & \left[\sum_{c^1 \in O_v} \frac{|c^1|}{2^n} * \log_2\left(\frac{2^n}{|c^1|}\right) + \sum_{c^0 \in Z_v} \frac{|c^0|}{2^n} * \log_2\left(\frac{2^n}{|c^0|}\right)\right] \end{aligned}$$

where O_v is the one-cube set at v while Z_v is the zero-cube set at v. c^1 is a one-cube that belongs to O_v and c^0 is a zero-cube that belongs to Z_v. This defines the contribution of this node in making decisions to generate a monochromatic one/zero cube set at the leaves. It also estimates the cost of implementing the partial function at this node, its gate-count complexity.

We use this definition to define the cumulative information content at a level i in the decision tree.

Definition 11 *The cumulative information content $I(f_{l_i}, D_k)$ at a level i in a decision tree $T(D_k, \pi)$ is defined as*

$$I(f_i, D_k) = \sum_{C_v \in \mathcal{P}_{i,x_j}^{max}} I(f_v, D_k)$$

where $I(f_v, D_k)$ is the information content with respect to a node that belongs to the maximal partition set at that level.

The cumulative information content at a level defines the gate-count complexity of the function f with respect to the inputs in the set $\{x_1, \ldots, x_{j-1}\} \subset I$. All the elements of the set $\{x_1, \ldots, x_{j-1}\}$ precede x_j in π.

Going down the decision tree, the change in information content from level i to level $i-1$ with a literal assignment x_i or x_i' tells us the contribution of that literal in reducing the information content. This is expressed as

$$\Delta I(f_i, D_k) = I(f_i, D_k) - I(f_{i-1}, D_k)$$

We are now ready to define the spatial entropy of the decision tree.

Definition 12 *The spatial entropy of a decision tree $T(D_k, \pi)$ for a function f with k-decomposition D_k and ordering π is*

$$S(T(D_k, \pi)) = \sum_{\mathcal{P}_{i,x_j}^{max} \in \mathcal{S}} \Delta I(f_i, D_k)$$

where, $\mathcal{S}$ is the signature of the decision tree, $\mathcal{P}_{i,x_j}^{max}$ is a maximal partition set $\in \mathcal{S}$ at level i corresponding to variable x_j in π.

We have thus expressed the effort required to generate monochromatic one/zero cube sets at the leaves of a decision tree, as a cumulative sum of the change in information content over the maximal partitions at each level in the decision tree.

We now try and relate the spatial entropy of an implementation, as described in Chapter 3, to the spatial entropy of a decision tree. We use the fact that an implementation can be constructed bottom-up from the decision tree to establish this relationship. Since the decision tree representation does not capture wire length *we assume that we are computing spatial entropy in the implementation with unit wire lengths.* The monochromatic leaf cube sets correspond to spatially distributed input sources in the implementation. Every node in the implementation denotes a partial function computed by the implementation and corresponds to only those internal nodes in the decision tree whose cube sets belong to some maximal partition set. The root node, which is the output of the implementation, corresponds to the entire function. The effort required to reduce the k-decomposition at the root node to the monochromatic cube sets at the leaves can be expressed in reverse. On doing so we obtain the *spatial entropy of an implementation.* This is the effort required for the monochromatic leaf cube sets to combine with each other while constructing the k-decomposition of the function.

How can we express this effort formally? The spatial entropy of an implementation can also be expressed using information content similar to way we defined the spatial entropy of a decision tree. We define the spatial entropy of the implementation as the *incremental contribution to the information content* $I(f, D_k)$ of the function over all the nodes in the implementation. At the leaf nodes the information content is 0. The root node expresses the information content over the entire function. Every internal node v in the implementation denotes a partial function. The information content $I(f_v, D_k)$ at this node v is computed from the cube set C_v. It indicates the effort contributed by the partial function at this node in combining the one/zero cubes of the original function. The node's contribution is computed by the increase in information content from the previous level in the implementation.

Thus we can express the spatial entropy of an implementation $\mathcal{I}$ (constructed from a decision tree T) using the expression

$$S(\mathcal{I}(T)) = \sum_{v \in \mathcal{I}} \Delta I(f_v, D_k)$$

where $\Delta I(f_v, D_k)$ represents the increase in information content from one node in the implementation to its immediate fanout node.

Lemma 5 *The spatial entropy of an implementation constructed from a decision tree $S(\mathcal{I}(T))$ is equal to the spatial entropy of a decision tree $S(T(\pi, D_k))$ computed from the k-decomposition at the root.*

Proof This is true by construction. The only nodes that contribute to the spatial entropy of a decision tree are nodes whose cube sets belong to the maximal partition. By construction, every node in the implementation corresponds to a cube set belonging to some maximal partition. Hence the

contributions to the spatial entropy of the implementation and the decision tree come from the same sources.
□

It should be noted that while a broad class of implementations can be constructed from the decision tree, they are restrictive in that they do not have *logical* reconvergent fanout and technology mapping issues are ignored.

We now turn our attention to the spatial entropy as we have defined it in Chapter 3 to see how it relates to this definition. When a multi-level implementation is constructed for a function its k-decomposition is usually known. This is because the k-decomposition of the function corresponds to a minimal (irredundant, prime) two-level representation of a function; and multi-level logic minimization and synthesis often starts with such a minimized two-level expression. We have seen that while computing the spatial entropy of the decision tree $S(T(\pi, D_k))$ and while computing the spatial entropy of an implementation $S(\mathcal{I}(T))$ from the decision tree, we have knowledge of the one-cubes and zero-cubes and the function's k-decomposition.

On the other hand suppose we were interested in computing the spatial entropy of an implementation that was already constructed? In such a case its spatial entropy computation will have to be done without knowledge of the one/zero cube distribution and the k-decomposition of the function. This is precisely what happens in the gate-level spatial entropy computation procedure. In Chapter 3 we had defined the local spatial entropy at a node v as as:

$$\delta S_v = \sum_{v' \in V} H_v * l_{(v,v')}$$

where H_v is the information computed at the gate node v, and $l_{(v,v')}$ is the length of the fanout edge from node v to v'. With a unit length assumption $\delta S_v = H_v$. Now let us look closely at the expression

$$H_v = p_v^0 \log(\frac{1}{p_v^0}) + p_v^1 \log(\frac{1}{p_v^1})$$

p_v^0 is the 0-probability of a node v, and p_v^1 is the 1-probability of the node. The 1 (or 0) probability at the node is the fraction of onterms (or offterms), in the cube space of the partial function being computed at the node.

If we had used the expression for the information content at a node v in an implementation derived from the decision tree, we would have the expression

$I(f_v, D_k)$ defined earlier. The expression H_v is really an approximation to the expression $I(f_v, D_k)$. Let us see how. In a constructed implementation we do not know the k-decomposition of the function. As a result we do not know how the minterms of the partial function at each node contribute to the one/zero cubes of the original function. So a simplistic assumption would be to assume that all the onterms of the partial function at that node contribute to the ON set of the function f, and the all the off terms contribute to the OFF set. This yields one big one-cube and one big zero-cube at the node. This means that $\mid O_v \mid$ and $\mid Z_v \mid$ are 1, since there is only one-cube and one zero-cube. The expression $I(f_v, D_k)$ reduces to the following,

$$\begin{aligned} I(f_v, D_k) = \quad & \left(\tfrac{1}{1+1} * \log_2\left(\tfrac{1+1}{1}\right) + \tfrac{1}{1+1} * \log_2\left(\tfrac{1+1}{1}\right)\right) * \\ & \left(\tfrac{|c^1|}{2^n}\log_2\left(\tfrac{2^n}{|c^1|}\right) + \tfrac{|c^0|}{2^n}\log_2\left(\tfrac{2^n}{|c^0|}\right)\right) \\ & \left(\tfrac{1}{2} + \tfrac{1}{2}\right) * \left(\tfrac{|c^1|}{2^n}\log_2\left(\tfrac{2^n}{|c^1|}\right) + \tfrac{|c^0|}{2^n}\log_2\left(\tfrac{2^n}{|c^0|}\right)\right) \end{aligned}$$

The size of the cubes $\mid c^1 \mid$ and $\mid c^0 \mid$ is equal to the number of minterms in the cube. In $\mid c^1 \mid$ this is the number of onterms while in $\mid c^0 \mid$ this is the number of offterms. So the expression $\frac{|c^1|}{2^n}$ becomes the 1-probability of the node (p_v^1), and $I(f_v, D_k)$ reduced to H_v. Accumulating H_v over all the nodes $v \in V$ yields the spatial entropy of the implementation.

To summarize, we began by defining the gate-count complexity of a boolean function by its information content. We showed that the information content, which is computed on a minimum k-decomposition of the function, captures minimality in two-level function representations. We then showed how a class of multi-level implementations for a given function could be derived from the decision tree representation. Minimality of multi-level implementations was defined on the decision tree. We then defined the spatial entropy of an implementation, computed bottom-up from the decision tree, as the cumulative sum of the incremental information content over all the corresponding nodes in the implementation. This gives a cube space definition of spatial entropy that captures the communication between the minterms in the implementation. In the next section we verify empirically that the spatial entropy of an implementation tracks function behavior. We also show empirically that the information content and spatial entropy can be used as estimates of the gate-count complexity of a function.

5.6 Experiment

The first objective of this experiment is to study the effectiveness of spatial entropy in measuring information content. Since information content depends on the minterm characteristics of the function, this will tell us how effective spatial entropy is in capturing function behavior. As it may be recalled, this was one of our objectives in establishing a theoretical basis for using spatial entropy to generate BDD variable orders. Earlier in this chapter, we defined an estimate for the gate-count complexity of a boolean function, over a minimum k cube decomposition of the function. This estimate, called the information content $I(f, D_k)$ of the function, was defined over cube characteristics that captured the gate cost in an implementation of the function. We then showed that the spatial entropy of an implementation (derived from a decision tree) can be defined as a function of the information content. Computing the spatial entropy of an implementation this way is expensive because the size of a decision tree is exponential in the worst case. In the previous section we also motivated the idea that the polynomial-time gate level spatial entropy computation of Chapter 3 could be used as an approximation to the spatial entropy computation over a decision tree. We verify this empirically in this experiment by using this polynomial-time procedure to measure information content. Our objective is to determine if there is sufficient correlation to use spatial entropy as a discriminator (or measure) of information content. For instance, would we able to compare spatial entropies of a pair of functions and have the confidence that we are comparing their information content?

In order to test our hypothesis we use the following criteria. We compute the *sample correlation coefficient* r for the values of spatial entropy S and information content I for a sample of circuits. (For convenience, the information content expression $I(f, D_k)$ is denoted as I). The sample correlation coefficient r, which is a sample estimate of the population correlation coefficient, is given by [Sto84],

$$r = \frac{C_{IS}}{\sqrt{C_{II} C_{SS}}}$$

r measures the degree of correlation between information content I and spatial entropy S. Values of r near $+1$ indicate a strong positive correlation between S and I(S increases, as I increases); values of r near -1 indicate a strong negative correlation between S and $I(S$ decreases, as I increases); while if $r = 0$, then there is no correlation between S and I. We also determine the *lines of regression* [Sto84] of S on I and I on S. If there is perfect correlation $r = \mp 1$ then the two lines will coincide with each other, and all the experimental points will lie exactly on the common line of regression. For uncorrelated values the lines will be at right angles to each other.

The second objective of this experiment is to study the effectiveness of information content in estimating gate count complexity. As described earlier, our definition of information content indicates minimality in two-level functions. Since it captures cube characteristics that contribute to logic cost, we would like to use it to estimate the gate-count complexity in a multi-level implementation. To verify our hypothesis, we use the same criteria as above. We compute the correlation coefficient for information content I and gate count GC for a sample of circuits, where GC is the gate count in a technology mapped circuit implementation.

5.6.1 Assumptions

Our assumptions are along the lines of the quantities that we wish to compute - spatial entropy (S), information content (I) and gate-count (GC). With respect to spatial entropy computation, our assumptions involve the three factors discussed in Chapter 4: redundancy, wire length and reconvergent fanout. As mentioned in the previous section we would like to measure the spatial entropy on on a decision tree with the smallest signature size. Since this also corresponds to the spatial entropy of a minimized implementation we assume redundancy is removed in the implementation, by minimizing for literal count. We use a unit wire length assumption for spatial entropy computation. This is because the definition of information content that we wish to track does not capture wiring complexity. In addition we have not been able to explain the role of wire lengths in the decision tree representation. Finally we assume that errors in probabilities and spatial entropy due to reconvergent fanout are ignored.

With respect to information content I we assume the following. Our estimate is defined for single-output completely specified functions. We ignore the effect of *don't cares*, when functions are incompletely specified. Since we have not defined information content for multi-output functions we currently use a naive approach. The information content of a multi-output function is computed by computing the information content for a minimum k-decomposition of each of the outputs individually. So the information content $I(f)$ for an m-output function would be computed as

$$I(f) = \sum_{i=1,m} I(f(o_i), D_k(o_i))$$

This is actually an overestimate of the multi-output function's complexity because the k-decompositions for each output function overlap (due to sharing of cube spaces between individual output functions). Consequently the overall information content would in most cases be less than the sum of the information contents of the individual functions. In order to remain consistent in our comparisons, when we compute the spatial entropy of such multi-output functions we use the *cone-based* approach (discussed in Chapter 3) that computes the spatial entropy of each output function independently. Like the information content, this too will be an overestimate of the spatial entropy of the multi-output function since the spatial entropy contribution of shared logic is not discounted.

It was proved earlier in this chapter that the k-decomposition of a function is equivalent to its two-level prime irredundant representation. Since computing the information content requires a k-decomposition of one/zero-cubes we use the output of a two-level minimizer `espresso` [BHMSV84] to generate these one/zero-cubes. The program `espresso` consists of a collection of heuristic algorithms for two-level minimization, and does not always guarantee a minimum two-level form. Thus for some functions our information content estimate does not use a *minimum* k-decomposition but an *approximately minimal* k-decomposition.

The gate count GC in an implementation is obtained by counting the number of gates in a technology mapped multi-level implementation of the function. Like the assumption for spatial entropy, we assume a minimized implementation while computing gate count. Since we compute the information content for multi-output functions by computing the information content for each function separately it would be inappropriate to compare this with the gate count in multi-output implementations. This is because multi-output functions share logic and the gate count in their implementations would be a lot lesser than if they were implemented separately as single-output functions. So we compute gate count only only for single output implementations.

5.6.2 Data Set

The data set for the experiment is made up of two kinds of circuits, both described as boolean functions in the PLA format [BHMSV84]. The first set consists of several randomly generated single-output completely specified boolean functions of $4, 5, 6, 7, 8$ and 9 input variables. The number of minterms in each of these functions and their positions in cube space was randomly generated. This data set satisfies the requirement of a wide range of boolean functions from 4 input functions to 9 input functions with different cube sizes. Table 5.1 describes the characteristics of the $4, 5$ and 6 input randomly generated functions, while Table 5.2 does the same for $7, 8$ and 9 input functions.

The second set of data consists of two-level boolean functions from the MCNC Logic Synthesis 91 benchmarks. This data set consists of single-output and multi-output functions, some of which are incompletely specified. These functions are shown in Table 5.3. The functions here are larger, and more irregular in their distribution of input/output sizes, minterm count and cube sizes. With this data set we get an opportunity to estimate and measure the complexity of boolean functions that have (presumably) useful implementations, as against those of the randomly generated functions in the first data set set.

In the first part of the experiment we determine the spatial entropy measure S, information content estimate I, and the gate count GC for 3 sample sets of randomly generated functions, with varying input sizes, extracted from the data sets in Table 5.1 and Table 5.2. Each sample of 19 functions is made up of a uniform mix of $4, 5, 6, 7, 8$ and 9 input functions. This gives us 3 separate data sets on which to correlate spatial entropy and information content. In the second part of the experiment we generate spatial entropy, information content, and gate count for the the Logic Synthesis 91 benchmark circuits.

5.7 Experiment Outline

Figure 5.13 shows the software construction and the experiment outline for computing the information content I for each function and the spatial entropy S and gate count GC for a minimized implementation. Computing

Circuit Name	Inputs	Product Terms	One Cubes	Zero Cubes
f4_1	4	15	4	1
f4_2	4	4	3	5
f4_3	4	4	3	4
f4_4	4	14	4	2
f4_5	4	12	5	3
f4_6	4	5	3	4
f4_7	4	8	6	6
f4_8	4	10	5	4
f4_9	4	7	4	4
f4_10	4	9	6	5
f5_1	5	17	11	9
f5_2	5	18	6	7
f5_3	5	29	6	3
f5_4	5	9	5	7
f5_5	5	7	5	7
f5_6	5	16	7	7
f5_7	5	10	7	7
f5_8	5	19	8	6
f5_9	5	26	5	3
f5_10	5	21	10	8
f6_1	6	26	11	12
f6_2	6	47	16	12
f6_3	6	21	11	12
f6_4	6	36	15	13
f6_5	6	23	10	14
f6_6	6	16	10	12
f6_7	6	10	6	9
f6_8	6	13	8	12
f6_9	6	53	11	9
f6_10	6	20	12	13

Table 5.1: 4,5 and 6 input randomly generated functions

Circuit Name	Inputs	Product Terms	One Cubes	Zero Cubes
f7_1	7	97	20	16
f7_2	7	39	21	24
f7_3	7	86	25	22
f7_4	7	112	19	12
f7_5	7	48	23	25
f7_6	7	23	14	18
f7_7	7	70	27	24
f7_8	7	59	23	21
f7_9	7	29	16	20
f7_10	7	110	16	11
f8_1	8	59	34	38
f8_2	8	109	46	48
f8_3	8	40	29	34
f8_4	8	145	42	42
f8_5	8	133	48	46
f8_6	8	245	16	9
f8_7	8	146	49	49
f8_8	8	157	43	42
f8_9	8	90	45	46
f8_10	8	239	24	14
f9_1	9	273	90	84
f9_2	9	185	78	84
f9_3	9	303	94	89
f9_4	9	346	87	77
f9_5	9	465	48	39
f9_6	9	39	31	43
f9_7	9	406	68	60
f9_8	9	403	74	64
f9_9	9	169	76	81
f9_10	9	318	81	78

Table 5.2: 7,8 and 9 input randomly generated functions

Circuit Name	Inputs	Outputs	Product Terms
con1	7	2	9
xor5	5	1	16
misex1	8	7	32
rd53	5	3	32
sao2	10	4	58
m5xp1	7	10	75
m9sym	9	1	87
rd84	8	4	256
z9sym	9	1	420
t481	16	1	481
rd73	7	3	141
clip	9	5	167
misex2	25	18	29
duke2	22	29	87
bw	5	28	87

Table 5.3: Two-level Functions from the Logic Synthesis 91 Benchmarks

the information content estimate requires measuring the following variables: number of onterms/offterms, number of one/zero cubes and sizes of one/zero cubes. As the figure indicates, each PLA representation is minimized at the two-level for its ON set and OFF set to obtain the one/zero cube set that makes up the k-decomposition. This is then used to compute the information content measure. The spatial entropy S is computed along the lines of the experiment in Chapter 4. First the two-level PLA representation is minimized via SIS [BRSVW87] and a multi-level technology mapped implementation in VPNR [KB88] is generated. This is then used by the spatial entropy computation program `spent` to generate spatial entropy measures. The spatial entropy measures for both the ON and OFF set implementations of the PLA are obtained and the lesser of the two is selected. This is because we are interested in the spatial entropy of a minimized implementation, and there are several occasions where implementing a function as its OFF set (inverted) yields an implementation with lesser spatial entropy. Since the multi-level minimization program SIS does not always detect this scenario we perform this task explicitly.

The spatial entropy of an implementation is computed with unit wire length, and no reconvergent fanout removal. On encountering multiple fanout nodes the spatial entropy is divided equally amongst the fanout nodes. This ensures that the overall spatial entropy is preserved when fanout nodes reconverge. The gate count GC is computed from the VPNR implementations.

The *information content* for multi-output functions, is calculated by computing the on/off terms and the one/zero-cubes for each individual output functions, by factoring out the common cubes. These values of $I(f_{o_i})$ are then accumulated over all the output functions f_{o_i} The *spatial entropy* for multi-output functions is calculated in two steps. In the first step the logic cone for each output function f_{o_i} is computed by traversing backward from the output o_i. This determines the logic that contributes to the output function f_{o_i}. The spatial entropy $S_{f_{o_i}}$ with respect to this cone is then calculated. After performing this task over all output functions the spatial entropy values over all cones are accumulated to obtain the spatial entropy of the multi-output function S_f. For m outputs we have,

$$S_f = \sum_{i=1,m} S_{f_{o_i}}$$

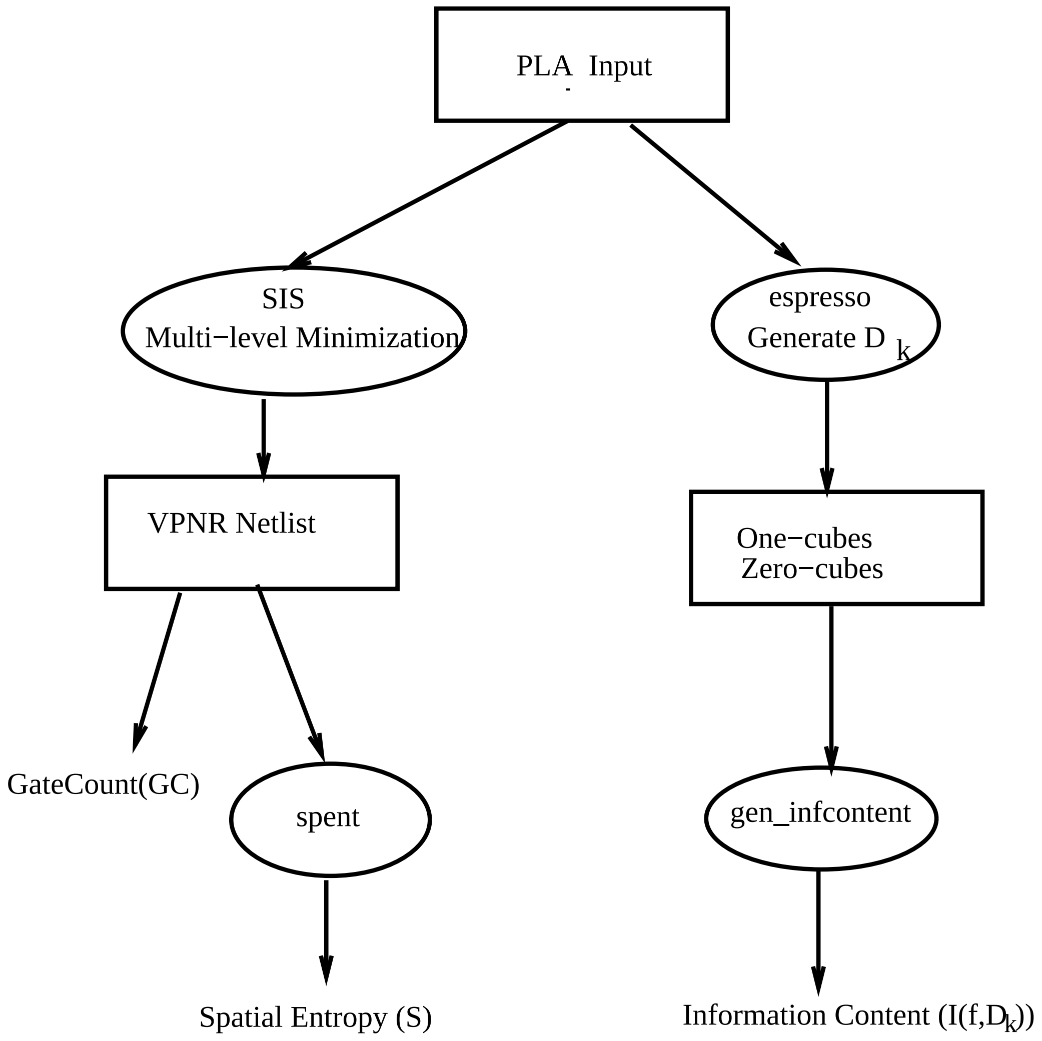


Figure 5.13: Experiment Outline

5.8 Results and Observations

5.8.1 Results

Tables 5.4, 5.5, and 5.6 illustrate the results of spatial entropy , gate count and information content for three sets of randomly generated functions. Table 5.7 tabulates spatial entropy and information content for the Logic Synthesis 91 benchmarks. As mentioned earlier, the gate count was not computed for these implementations since they are multi-output implementations that share logic, while the information content computation does not assume sharing. The spatial entropy measures shown here are obtained after computing the spatial entropy for minimized implementations of the ON-set and the OFF-set and then taking the lesser of the two. The gate count is obtained similarly.

5.8.2 Observations

Table 5.8 (page 168) shows the mean and variance of spatial entropy and information content values for the random function data sets and the Logic Synthesis 91 benchmarks. We observe that the variance of the random function (for both spatial entropy and information content) is not as high as it is for the benchmark circuits. This is because the benchmark circuits are far more diverse with a wider range of features. The individual functions are much further apart from each other in terms of their complexity and information content. There are functions with inputs ranging from 5 to 25, and outputs ranging from 1 to 29.

The other observation that we make with respect to the spatial entropy and information content is that spatial entropy values show a higher variance than the information content values for all data sets. The reason for this is that the implementations of two functions can differ in a lot more ways than the specifications of the same two functions. There are a lot more variables involved - gates, gate types, their connectivity and arrangement, *etc.*, probabilities, when spatial entropy is being computed at the implementation level. On the other hand the information content is computed on the function specification in terms a few fixed variables. So fluctuations are less drastic.

We study correlation between spatial entropy and information content

Circuit Name	Inputs	S	$I(f, D_K)$	Gate Count
f4_1	4	4.467	1.62	1
f4_5	4	9.58	3.34	9
f4_6	4	8.96	2.83	12
f5_2	5	11.72	3.735	12
f5_7	5	12.26	4.281	15
f5_9	5	10.04	3.01	8
f6_2	6	22.55	5.850	37
f6_5	6	16.76	5.389	29
f6_9	6	24.62	4.995	38
f7_4	7	33.08	6.387	61
f7_5	7	26.53	6.819	67
f7_1	7	34.52	6.202	71
f7_10	7	28.38	6.011	47
f8_1	8	35.09	7.811	102
f8_8	8	44.17	8.546	129
f8_10	8	38.69	7.514	69
f9_3	9	61.75	9.672	255
f9_6	9	28.14	8.871	103
f9_8	9	66.595	9.582	230

Table 5.4: S, $I(f, D_k)$ and GC for random circuits (Set 1)

Circuit Name	Inputs	S	$I(f, D_K)$	Gate Count
f4_2	4	8.697	3.10	8
f4_7	4	10.35	4.375	11
f4_4	4	5.206	2.296	3
f5_4	5	11.60	3.95	13
f5_5	5	10.96	4.195	12
f5_3	5	9.97	3.07	9
f6_1	6	16.53	5.243	27
f6_8	6	15.71	5.067	26
f6_6	6	16.66	5.156	28
f6_10	6	18.57	5.837	33
f7_3	7	34.29	7.127	79
f7_7	7	30.725	6.951	71
f7_8	7	25.54	6.459	58
f8_2	8	37.36	8.474	232
f8_4	8	43.36	7.852	129
f8_5	8	43.07	8.458	128
f9_7	9	60.32	9.950	740
f9_9	9	49.89	9.725	364
f9_10	9	63.20	9.646	243

Table 5.5: S, $I(f, D_k)$, and GC for random circuits (Set 2)

Circuit Name	Inputs	S	$I(f, D_K)$	Gate Count
f4_10	4	10.35	3.852	11
f4_3	4	8.806	3.08	10
f4_8	4	10.35	3.10	11
f5_1	5	15.72	5.18	23
f5_6	5	15.25	4.125	20
f5_8	5	14.02	3.97	17
f6_3	6	18.10	5.399	29
f6_4	6	22.505	5.604	40
f6_7	6	14.031	4.703	19
f7_2	7	23.47	6.720	59
f7_6	7	19.60	6.349	40
f7_9	7	20.02	7.007	46
f8_3	8	25.97	7.450	77
f8_6	8	36.95	5.848	61
f8_7	8	41.3	8.289	136
f8_9	8	37.10	7.851	125
f9_1	9	54.90	9.584	574
f9_2	9	53.44	9.409	408
f9_4	9	56.66	9.636	658

Table 5.6: S, $I(f, D_k)$ and GC for random circuits (Set 3)

Circuit Name	S	$I(f, D_K)$
con1	23.735	6.8135
xor5	17.32	5.0
misex1	89.401	26.324
rd53	49.93	15.884
sao2	110.834	27.819
m5xp1	164.607	43.308
m9sym	59.55	11.932
rd84	139.27	38.507
z9sym	53.466	11.932
t481	89.66	26.682
rd73	135.83	31.9375
clip	210.00	44.912
misex2	217.27	47.057
duke2	624.36	137.458
bw	368.33	107.351

Table 5.7: Spatial Entropy and Information Content for Logic Synthesis 91 benchmarks

Data Set	Spatial Entropy (S)		Information Content (I)	
	Mean	Variance	Mean	Variance
Random Set 1	27.26	298.51	5.92	5.67
Random Set 2	26.95	329.65	6.15	5.74
Random Set 3	26.24	247.53	6.17	4.54
LgSynt91 Benchmarks	156.90	25,038.064	38.86	1368.80

Table 5.8: Mean and Variance of Spatial Entropy (S) and Information Content (I)

Data Set	Correlation Coefficient(r)
Random Set 1	0.9111
Random Set 2	0.9648
Random Set 3	0.9207
LgSynt91 Benchmarks	0.9835

Table 5.9: Correlation Coefficients for (S vs I)

by computing the correlation coefficient r for each of the above 4 data sets. These are shown in Table 5.9 (page 169). The regression lines of spatial entropy (S) on information content (I) and information content on spatial entropy are calculated as follows [Sto84]. The correlation coefficient r is given by

$$r = \frac{C_{IS}}{\sqrt{C_{II}C_{SS}}}$$

For the data set D, the *regression of* S *on* I is plotted as

$$S = a + b * I$$

where $b = \frac{C_{IS}}{C_{II}}$, $a = \overline{S} - b * \overline{I}$, $\overline{S} = \frac{\sum_{i \in D} S_i}{|D|}$, and $\overline{I} = \frac{\sum_{i \in D} I_i}{|D|}$.

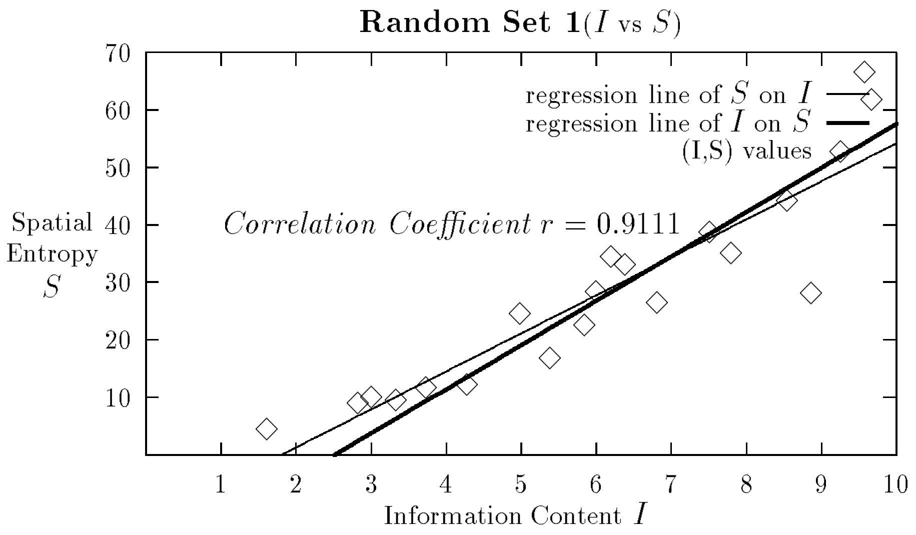
The *regression of* I *on* S is given by

$$I = a' + b' * X$$

where $b' = \frac{C_{IS}}{C_{SS}}$, and $a' = \overline{S} - b' * \overline{I}$.

Figure 5.14 and Figure 5.15 show plots of the regression lines and the data points (from the Table of results) for the 4 data sets.

From the correlation coefficient values and the plotted regression lines we observe that there is a strong positive correlation between spatial entropy S and information content I for all the data sets. The correlations appear particularly strong for the Logic Synthesis Benchmarks. This is because these circuits have a larger varaince of spatial entropy and information content. The inaccuracies in spatial entropy computation and the fluctuations in spatial entropy across implementations are not very significant relatively and are seen as minor perturbations. This yields a stronger correlation.



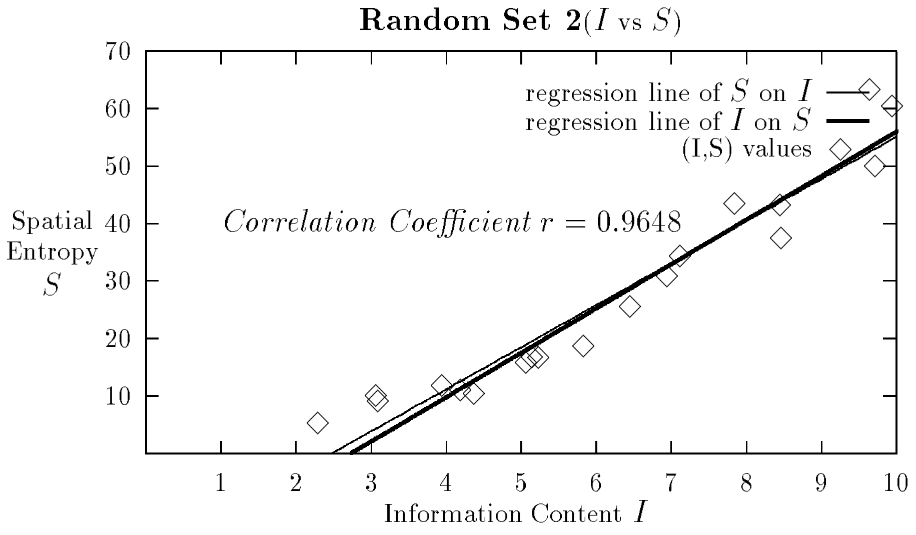
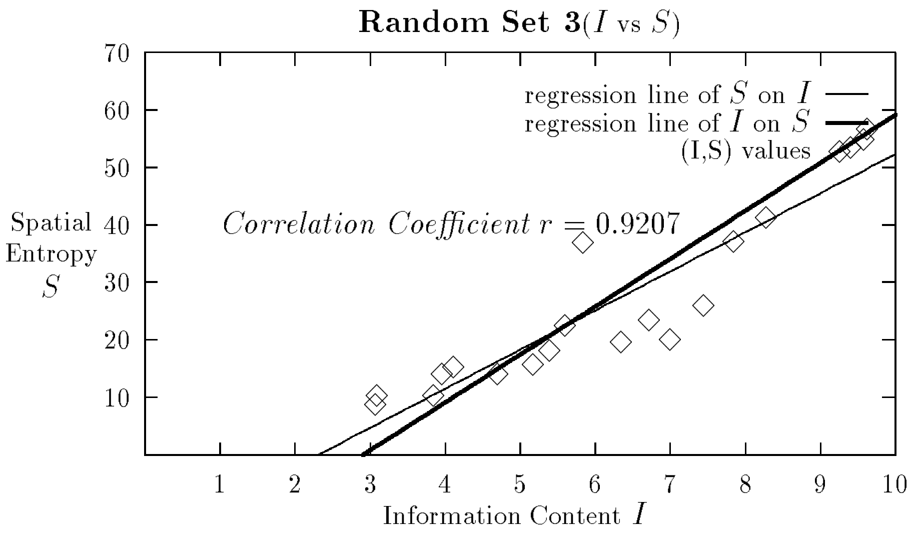


Figure 5.14: Regression Lines and Data plots (S vs I)



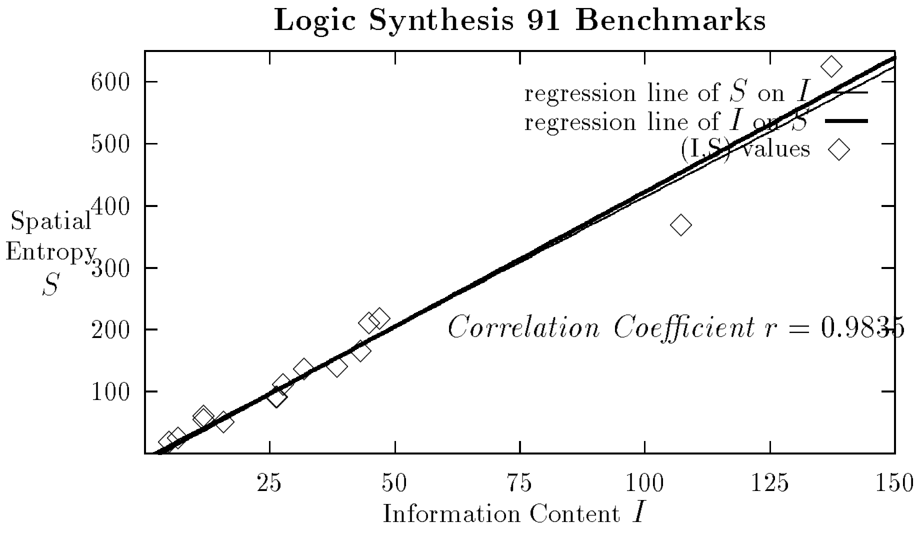


Figure 5.15: Regression Lines and Data plots (S vs I)

Data Set	Correlation Coefficient(r)
Random Set 1	0.871
Random Set 2	0.762
Random Set 3	0.794

Table 5.10: Correlation Coefficients for (GC vs I)

The average value of the correlation coefficient over the 3 random data sets is 0.9322, indicating a strong correlation. This is inspite of the fact that the random functions have a much lesser variance in information content. This lesser variance would normally make it more difficult to discriminate between information contents of different functions. But this correlation shows that spatial entropy is capable of measuring information content even in such functions.

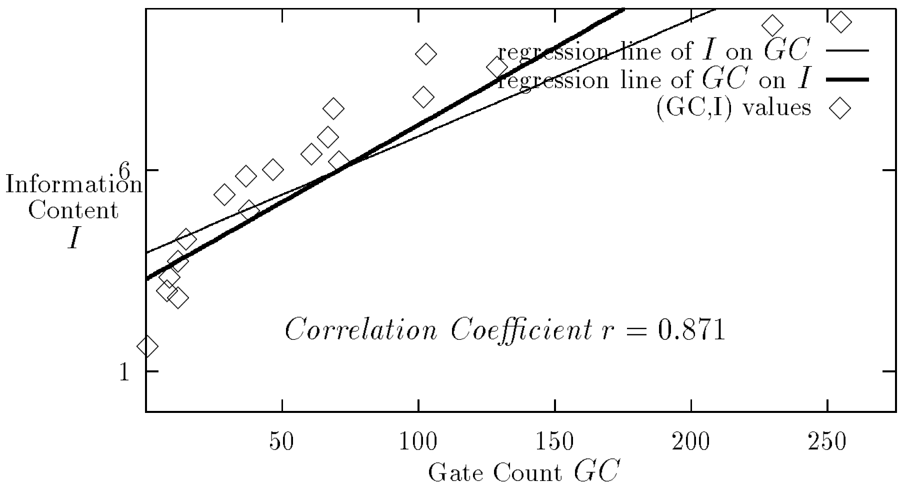
We now study the correlation between information content and gate count. Table 5.10 (page 172) shows the correlation coefficient r for the three random data sets. The plots are shown in Figure 5.16 (page 173) and Figure 5.17 (page 174).

The correlation coefficient values and the regression lines indicate a definite positive correlation between information content and gate count, though it is not as strong as the correlation between information content and spatial entropy. This tells us that information content is a good estimate of gate-count complexity in boolean functions. Some of the fluctuations can be explained by the fact that we have used a gate count measure obtained after technology mapping. A more accurate physical attribute like active area might have helped since it takes into account the difference in area occupied between complex gates and simpler gates.

The fact that there is a high correlation between information content I and spatial entropy S suggests that there might be a positive correlation between spatial entropy S and gate count GC in a multi-level implementation. We study this by computing the correlation coefficient r for spatial entropy and gate count for the three random data sets. These are shown in Table 5.11 (page 175). The regression line plots are shown in Figure 5.18 (page 175) and Figure 5.19(page 176).

We notice that there is high degree of correlation between gate count and

Random Set 1(GC vs I)



Random Set 2(GC vs I)

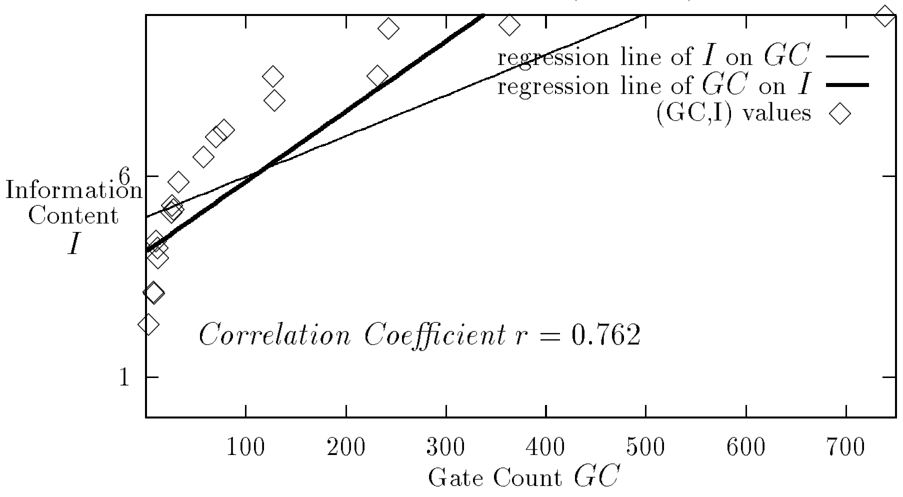


Figure 5.16: Regression Lines and Data plots (GC vs I)

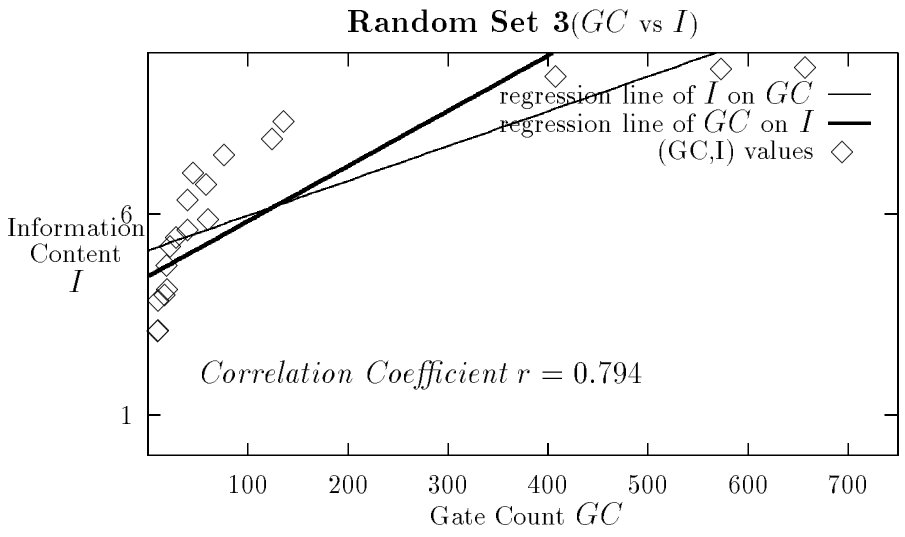
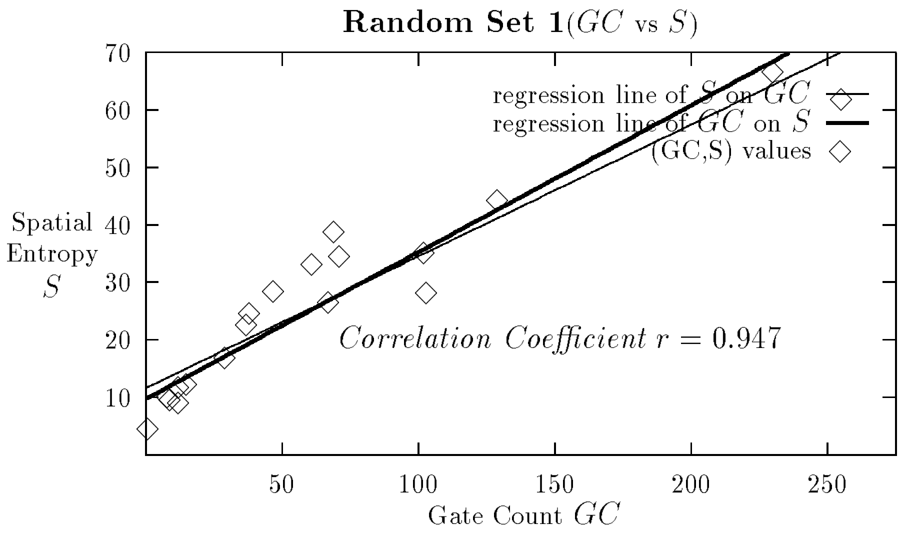


Figure 5.17: Regression Lines and Data plots (GC vs I)

Data Set	Correlation Coefficient(r)
Random Set 1	0.947
Random Set 2	0.80
Random Set 3	0.885

Table 5.11: Correlation Coefficients for 3 data sets (Gate Count vs S)



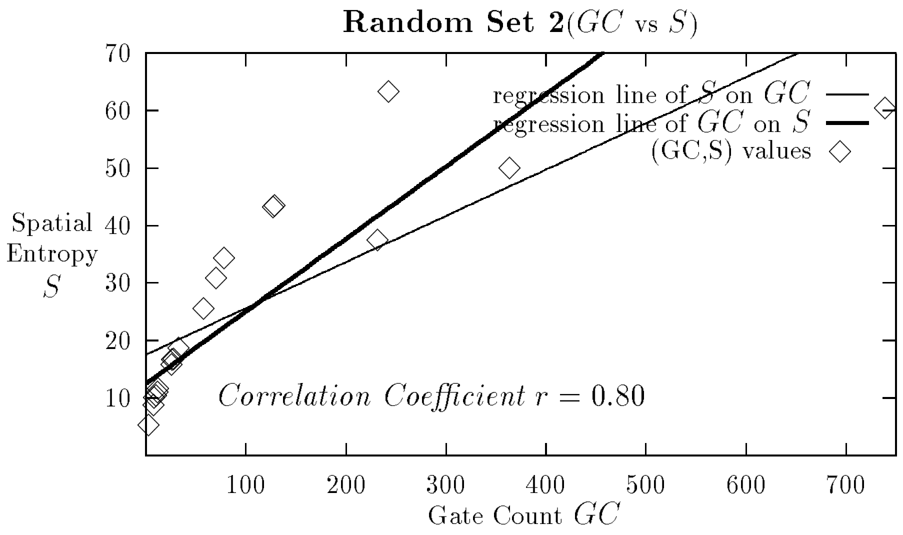


Figure 5.18: Regression Lines and Data plots (GC vs S)

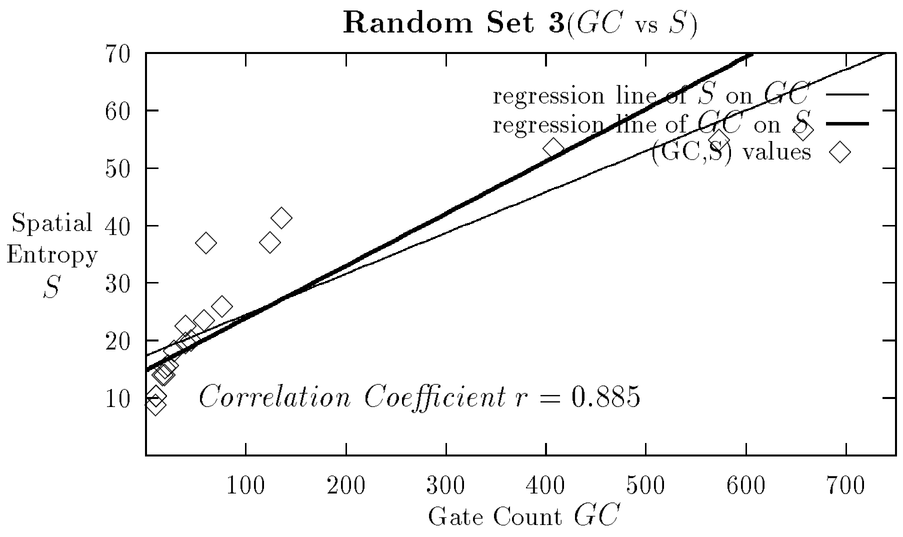


Figure 5.19: Regression Lines and Data plots (GC vs S)

spatial entropy for all the data sets. Thus spatial entropy gives us a measure of circuit area. In earlier chapters we explained how spatial entropy captures the switching energy in an implementation. How do these two relate? Given that we limit our scope to only combinational circuits, the switching energy in these circuits is a function of the switching of (1/0) states at the nodes and the switching delays at the nodes in the implementation. In most combinational circuits this switching energy is equivalent to the circuit area since almost the entire circuit switches while the circuit computes. This explains the high correlation between spatial entropy and circuit area measured by gate count.

5.9 Conclusions

From the experimental results of the previous section we can conclude that spatial entropy is a good measure for the information content of the function. Even when the functions are much closer in cube space as in the case of the random data sets, we observe that there is strong positive correlation (0.9322) between spatial entropy and information content. In Chapter 4 we studied the use of spatial entropy for variable ordering of BDDs. We observed that due to the sensitivity of the BDD construction process and the fine granularity of usage of the spatial entropy attribute (individual contributions in a spatial entropy vector), approximations and inaccuracies in the spatial entropy computation were quite significant. This in turn had an adverse effect on the resultant orderings and BDD sizes. In this chapter we observe that since we did not have to deal with sensitivity and spatial entropy values at such a fine level, the approximations in spatial entropy computation did not affect the quality of the solution. This is an important observation. It informs us that while the results of applying spatial entropy may not be as rewarding in one case, they might be a lot better in another.

The information content of a function is a characteristic of the function and its cube space distribution, independent of the myriad of implementations that might be generated to compute this function. The fact that the spatial entropy computed on a minimized implementation of the function is capable of measuring such a characteristic makes it a very useful measure to study functions over an entire class of implementations. This lends more credence to our work in Chapter 4, on using spatial entropy for BDD variable ordering and provides empirical evidence that there is some basis to using spatial entropy for BDD variable ordering. The decision tree for a given input ordering can be very easily transformed into a BDD for that ordering; it is quite likely that this might yield a more formal relationship between BDD sizes and spatial entropy. We intend to pursue work in this direction.

We can also conclude that information content, our definition of gate-count complexity, does estimate logic gate cost well. We observed a definite positive correlation between information content, an abstract attribute defined in cube space, and the gate count obtained from the implementation. The high correlation between spatial entropy and information content showed that it too can be used to estimate circuit area. Since we have not been able estimate wiring at the cube space level, wiring complexity is not captured in our results. Variances in cube sizes was not modeled in our definition of information content, neither did we use any kind of distance/length information while defining the spatial entropy of the decision tree or computing spatial entropy on a gate level implementation in the experiments. The variance in cube sizes affects the distance cubes need to travel before getting combined, and this might have an influence on the wiring factor. The recent work on using lexicographic ordering [PAS90, ASSP90] to minimize the routing factor in multilevel synthesis further supports this. We elaborate on this briefly.

In their research, Saucier *et.al* adopt a lexicographic factorization strategy to generate the multi-level factored form of a function, and show that this strategy reduces wiring complexity in the layout. A lexicographic factorization generates factors for a function, all of which are compatible with a reference order on the input variables. For example, the factors $ab(ce+d)$ and $ce(d+f)$ are compatible with the reference order $b \succ a \succ c \succ e \succ d \prec f$.

This idea can be related to the decision tree and its nodes. The factors and their sub-factors are partial functions at the cube sets that are members of maximal partition sets in a decision tree. The reference order determines the ordering of variables in the decision tree.

Suppose we had two decision trees built on two orderings, where one of the orderings is a reference order for a lexicographic factorization. The number of nodes in the decision tree built with lexicographic factorization will be lesser than the the number of nodes in the decision tree built without lexicographic factorization. The reference order for the lexicographic factorization generates compatible factors. These factors correspond to nodes or cube sets that make independent decisions about generating monochromatic one/zero-cubes. For example, in the factor $ab(ce+d)$ with the ordering $b \prec a \prec c \prec e \prec d$, the path $a=0, b=0$ will immediately yield a monochromatic zero-cube set at the level corresponding to c. Meanwhile, all the other a, b combinations will yield exactly one cube set belonging to a maximal partition set at the level corresponding to c. Now the factor $(ce+d)$ can act independently to generate its monochromatic zero cube set for $c=0, e=0$, at level c and below. Thus several cube sets in the decision tree get merged into maximal partition sets yielding decision trees with lesser nodes. An ordering like $b \prec e \prec d \prec a \prec c$ will not be able to achieve such similar sharing. This implies larger nodes due to more number of maximal partition sets. A formal treatment of this with the spatial entropy expressions at the nodes of the decision tree could yield some useful results with respect to wiring complexity. The fact that the lexicographic factorization strategy has also been used to generate BDD variable orders further indicates a possible fundamental relationship between spatial entropy and BDD sizes.

Chapter 6

Conclusions

In this dissertation we have introduced spatial entropy to the VLSI domain as an attribute that captures the dynamic communication effort in a circuit. Its quantitative definition unifies static structure and dynamic usage in a circuit. In a physical implementation (for CMOS circuits), this definition is a good model for measuring the switching energy in a circuit. We have shown how to compute spatial entropy at different levels of abstraction and have characterized it in extensive detail. We illustrated its use by applying it to a CAD problem that has traditionally relied on static attribute based solutions. We also showed that it can be used to capture function behavior through implementations.

6.1 Summary

There are several problems in CAD where function behavior needs to be captured through an implementation. Using static attributes computed from the topology of the implementation is not always helpful for this purpose. Firstly they do not provide sufficient usage information, and secondly they tend to exhibit variances with implementations which is not desirable while capturing function behavior. There is also a need for an attribute that can unify behavior and structure in order to answer questions about the function *and* the implementation.

We have introduced a circuit attribute capable of meeting some of these requirements using a concept called spatial entropy. Intuitively, it is the dynamic communication effort (or work done) in the circuit while computing the underlying function. We have defined the attribute quantitatively, using the entropy function from information theory. This definition is computable at different levels of abstraction and it unifies the static structure and dynamic usage in a circuit. On a physical (CMOS) implementation spatial entropy computes the switching energy in a circuit. This quantifies the intuitive notion of spatial entropy in the form of a physical attribute that is of relevance to designers.

It is difficult to compute spatial entropy accurately. The levels of abstraction hide information needed to compute it, and other factors like reconvergent fanout affect its accuracy. On applying spatial entropy to the problem of generating variable orders in BDDs, we found that it performs competitively with earlier static attribute based approaches. But as circuits get larger, the inaccuracies in computing spatial entropy have a significant effect on the quality of the solution.

We have shown that spatial entropy is capable of capturing function behavior through an implementation. An implementation with minimum spatial entropy has minimum switching energy and describes the minimum communication effort required to compute the function. Such an implementation is like a signature for the function since its measures of spatial entropy (and switching energy) provide a lower bound over all implementations of that function. Since it is difficult compute a minimal spatial entropy implementation we approximate it by using a minimal gate count implementation and show that the spatial entropy computed on such an implementation can capture function behavior. We characterize function behavior by defining *information content*, the gate-count complexity of a function in cube space. We then show that there is a strong correlation between the spatial entropy of a minimal gate count implementation and the information content of a function. Moreover inaccuracies in spatial entropy computation do not have the significant effect they had in BDD variable ordering. There is also empirical evidence that information content (in cube space) is a good estimator of actual gate-count complexity.

What can we conclude about spatial entropy's role in CAD from this research? The empirical evidence that spatial entropy tracks function behavior suggests that there is a fundamental basis for using spatial entropy to generate BDD variable orders. A BDD is strictly a characteristic of the function and its minterms. A decision tree for a function depends on its input variable ordering; and it is easy to transform one into a BDD for that ordering. The cube space definition of spatial entropy over a decision tree gives us a mechanism for distinguishing one tree (or BDD) from another and consequently one ordering from another. This can give us a theoretical basis for connecting spatial entropy and variable ordering.

The quantitative definition of spatial entropy provides a good model for switching energy in a physical implementation. Switching energy and dynamic power consumption are of concern to every designer; and spatial entropy provides a quantitative comparison of these measures between different circuit implementations. Generating a minimum spatial entropy implementation is a desirable goal for a designer, since minimum spatial entropy implies minimum switching energy. But existing synthesis and layout tools that generate physical implementations are incapable of generating implementations with minimum spatial entropy, since they are unable to minimize wires along with logic gates. Minimizing wires is also of concern in performance driven logic and layout synthesis, and minimum spatial entropy can become a useful metric of performance in these areas too.

Depending on the level of abstraction (boolean function, gate, or layout), at which the implementation is viewed there are different interpretations to spatial entropy; in all of them spatial entropy captures a notion of dynamic effort. For an implementation denoted by a multi-level function, it is the incremental contribution in cube space to compute the information content of the function. For an implementation denoted by a gate level netlist, it is the information flow computed as the information-distance product over all the nodes in the implementation. For an implementation at the physical layout level, it models the switching energy in the implementation. This indicates the generic applicability of the spatial entropy concept.

While trying to track function behavior from a gate level implementation we observed that spatial entropy can be efficiently computed, but computing it accurately is a difficult task. This inaccuracy in spatial entropy computation can affect some problems (BDD variable ordering) more than others (measuring gate-count complexity). The primary drawback here is poor estimates of wire length at the gate level. This lack of wiring information is also tied in to the goal of achieving minimum spatial entropy. An implementation with minimum spatial entropy is difficult to achieve currently, because existing logic synthesis tools only minimize for gate count ignoring wire length information. Since we could not capture wiring estimates in the form of variances in cubes sizes in our information content definition, we could not extend our estimate of gate-count complexity to include wiring complexity. Improved estimates of wire length will not only help us do this but also achieve minimal spatial entropy implementations and more accurate measures of spatial entropy at the gate level.

6.2 Future Research Directions

As discussed above, using decision trees to demonstrate that there is a theoretical basis for using spatial entropy for BDD variable ordering is one of the more immediate problems that has spawned from this research. The problem of estimating wiring complexity at the cube space level and using it to refine the information content definition $I(f, D_k)$ is another issue that has direct relevance.

In addition to these issues we see spatial entropy being applicable in other areas of CAD, primarily those where static structure and dynamic usage can together provide more information to solve a problem. A partial list of these areas is discussed below.

Spatial Entropy for Logic Synthesis: Logic synthesis involves several phases that repeatedly make area-time tradeoff decisions. At almost every stage in the synthesis process when data is needed in two different places at the same time, the logic computing it needs to be reused. The issue then is: should it be replicated (recomputed) at the desired location, or should it be factored out and transmitted (via a fanout or wire) to the desired location? Spatial entropy can capture the area-time tradeoff in this decision elegantly. We have seen that the wire lengths in spatial entropy computation can provide a measure of dynamic communication effort by modeling the distance traveled by information. At the same time they can also be used to model the wiring complexity in a circuit to study trade offs in circuit area minimization. Using spatial entropy (along with an active area measure) to control the extraction of kernel intersections during logic synthesis could lead to a new metric to measure the quality of synthesized circuits that captures active area and the wiring area.

Device-fitting: The logic-wire unifying property of spatial entropy and its ability to measure function complexity can also help address problems in device fitting software design for FPGA and PLDs. The software that maps two-level logic equations into complex cells in FPGAs and PLDs has to tackle problems of design partitioning via boolean function decomposition, minimization and technology mapping, followed by signal place and route. These are quite often iterative tasks, where literal complexity and wiring complexity trade off with each other. Spatial entropy could be used as a cost measure here to assist in global decisions.

Design Space Parsing: Spatial entropy is currently being used as a tool to perform high level design-space parsing and exploration [Tya91a]. The design-space of different area-time implementations of datapath functions can be characterized into primitive models on the basis of the communication pattern exhibited by these functions. Due to similar communication behavior the spatial entropy values for different area-time implementations of these functions will show a similar distribution. This structures the task of design space exploration.

State Transition Graph Discrimination: The distribution of spatial entropy values at the outputs of a sequential circuit helps discriminate the structure of state transition graphs. This is useful information for algorithms that perform state machine traversal during sequential circuit verification and test generation, since it gives an indication of the graph structure: whether it is deep or shallow, broad or narrow. This in turn can help state transition graph traversal algorithms.

Bibliography

[Agr81] V. Agrawal. An Information Theoretic Approach to Digital Fault Testing. *IEEE Transactions on Computers*, pages 582–587, August 1981.

[Ake78] S. B. Akers. Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, June 1978.

[AS87] K. J. Antreich and M. H. Schulz. Accelerated Fault Simulation and Fault Grading in Combinational Circuits. *IEEE Transactions on Computer-Aided-Design*, CAD-6(5):704–711, September 1987.

[ASSP90] P. Abouzeid, K. Sakouti, G. Saucier, and F. Poirot. Multilevel Synthesis minimizing the routing factor. In *27th ACM/IEEE Design Automation Conference*, pages 365–368, June 1990.

[BA92] H. Bouzouzou and P. Abouzeid. Personal Communication. MCNC Logic Synthesis 91 Benchmarks - BDD Orderings, May 1992.

[BBR89] K. S. Brace, R. E. Bryant, and R. L. Rudell. Efficient Implementation of a BDD Package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, June 1989.

[Ber91] C. Leonard Berman. Circuit Width, Register Allocation and Ordered Binary Decision Diagrams. *IEEE Transactions on Computer-Aided-Design*, CAD-10(8):1059–1065, August 1991.

[BHMSV84] R. K. Brayton, G. Hatchel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Press: Boston, 1984.

[BI86] D. Brand and V. S. Iyengar. Timing Analysis uusing Functional Relationship. In *IEEE ICCAD '86 Digest of Technical Papers*, pages 126–130, November 1986.

[BM82] R. K. Brayton and C. McMullen. The Decomposition and Factorization of Boolean Expressions. In *International Symposium on Circuits and Systems*, pages 49–54, May 1982.

[BMCM90] J. Benkoski, E. V. Meersch, L. J. M. Claesen, and H. D. Man. Timing Verification using Statically Sensitizable Paths. *IEEE Transactions on Computer-Aided-Design*, CAD-9(10):1073–1083, October 1990.

[BPH84] Franc. Brglez, P. Pownall, and R. Hum. Applications of Testability Analysis. In *Proceedings of the IEEE International Test Conference*, pages 705–712, 1984.

[Bra91] K. Brace. Personal Communication. BDD Orderings, June 1991.

[BRM91] K. M. Butler, D. E. Ross, and M. R. Mercer. Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 417–420, June 1991.

[BRSVW87] R. K. Brayton, R. Rudell, A. Sangiovani-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on CAD*, CAD-6:1062–1081, November 1987.

[Bry86] R. E. Bryant. Graph based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):667–691, August 1986.

[Bry91] R. E. Bryant. On the Complexity of VLSI implementations and Graph Representations of Boolean functions with applications to Integer Multiplication. *IEEE Transactions on Computers*, C-40(2):205–213, Feb 1991.

[CA90] K-T Cheng and V. D. Agrawal. An Entropy Measure for the Complexity of Multi-Output Boolean Functions. In *27th ACM/IEEE Design Automation Conference*, pages 302–305, June 1990.

[CF73] R. W. Cook and M. J. Flynn. Logical Network Cost and Entropy. *IEEE Transactions on Computers*, C-22:823–826, September 1973.

[Dus78] J. Dussault. A Testability Measure. In *Proceedings of the 1978 Semiconductor Test Conference*, pages 113–116, October 1978.

[DYG89] D. H. C. Du, S. H. C. Yen, and S. Ghanta. On the General False Path Problem in Timing Analysis. In *Proceedings of the 26th IEEE/ACM Design Automation Conference*, 1989.

[FFK88] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of Boolean comparison method based on binary decision diagrams. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1988.

[FMK91] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of Binary Decision Diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation*, pages 50–54, February 1991.

[FS90] S. J. Friedman and K. J. Supowit. Finding the Optimal Variable Ordering for Binary Decision Diagrams. *IEEE Transactions on Computers*, C-39(5):710–713, May 1990.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* Freeman Press: New York, 1979.

[Hel72] L. Hellerman. A Measure of Computational Work. *IEEE Transactions on Computers*, C-21:439–446, May 1972.

[HM86] K.S. Hwang and M. R. Mercer. Derivation and Refinement of Fan-out Constraints to Generate Tests in Combinational Logic Circuits. *IEEE Transactions on Computer-Aided Design*, CAD-5:564–572, October 1986.

[HOI89] T. Hwang, R. M. Owens, and M. J. Irwin. Communication Complexity Driven Logic Synthesis. In *Proceedings of the 1989 International Workshop on Logic Synthesis*, 1989.

[HOI92] T.T. Hwang, R. M. Owens, and M. J. Irwin. Efficiently Computing Communication Complexity for Multilevel Logic Synthesis. *IEEE Transactions on Computer-Aided Design*, 11:545–554, May 1992.

[ISY91] N. Ishiura, H. Sawada, and S. Yajima. Minimization of Binary Decision Diagrams based on Exchange of Variables. In *Proceedings of the International Conference of Computer-Aided Design*, pages 472–475, November 1991.

[JA84] S. K. Jain and V.D. Agrawal. STAFAN: An Alternative to Fault Simulation. In *Proceedings of the 21st Design Automation Conference*, pages 18–23, 1984.

[JPHS91] S. W. Jeong, B. Plessier, G. D. Hatchel, and F. Somenzi. Variable ordering for FSM Traversal. In *International Workshop on Logic Synthesis*, May 1991.

[Kap91] R. Kapur. Personal Communication. BDD Orderings for a collection of heuristics, July 1991.

[KB88] G. Kedem and F. Brglez. OASIS: Open Architecture Silicon Implementation System. Technical Report TR88-06, Microelectronics Center of North Carolina, Feb 1988.

[Kel68] E. Kellerman. A Formula for Logical Network Cost. *IEEE Transactions on Computers*, C-17:881–884, September 1968.

[Koh78] Z. Kohavi. *Switching and Finite A Auto Theory.* McGraw-Hill, 1978.

[Koo87] G. Koob. *An abstract Complexity Theory for Boolean Functions.* University of Illinois at Urbana-Champaign, 1987.

[Law64] E. J. Lawler. An Approach to Multilevel Boolean Minimization. *Journal of the ACM*, pages 283–295, 1964.

[LBdGG87] R. Lisanke, F. Brglez, Aaart J. de Geus, and David Gregory. Testability-Driven Random Test Pattern Generation. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):1082–1087, November 1987.

[LBK88] R. Lisanke, F. Brglez, and G. Kedem. McMAP: A Fast Technology Mapping Procedure for Multi-Level Logic Synthesis. In *Proceedings of the IEEE International Conference on Computer Design*, October 1988.

[Lee59] C. Y. Lee. Representation of switching circuits by Binary Decision Programs. *Bell Systems Technical Journal*, 38:985–999, July 1959.

[LKB87] R. Lisanke, G. Kedem, and F. Brglez. DECAF: Decomposition and Factoring for Multi-Level Logic Synthesis. Technical Report TR87-15, Microelectronics Center of North Carolina, August 1987.

[LP88] M. Lorenzetti and B. Preas. *Physical Design Automation of VLSI Systems.* Benjamin-Cummings Publishing Company, 1988.

[Mas78] K. Mase. Comments on "A Measure of Computational Work" and "Logical Network Cost and Entropy". *IEEE Transactions on Computers*, C-27:94–95, January 1978.

[MC79] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter Physics of Computational Systems, pages 333–381. Addison-Wesley Publishing Company, 1979.

[McC65] E.J. McCluskey. *Introduction to the Theory of Switching Circuits.* McGraw-Hill, 1965.

[Min92] S.-I Minato. Personal Communication. BDD Orderings using minimum width method for ISCAS85 and Logic Synthesis 91 Benchmarks, May 1992.

[MIY90] S.-I. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with Attributed Edges for efficient Boolean function manipulation. In *Proceedings of the IEEE/ACM Design Automation Conference*, pages 52–57, November 1990.

[MJ90] F. Maamari and J.Rajski. A Method of Fault Simulation based on Stem Regions. *IEEE Transactions on Computer-Aided Design*, CAD-9:212–220, February 1990.

[MK89] P. C. McGeer and R. K.Brayton. Efficient Algorithms for Computing the Longest Viable Path in a Combinational Network. In *Proceedings of the 26th Design Automation Conference*, June 1989.

[MKR92] M. Ray Mercer, R. Kapur, and D. E. Ross. Functional Approaches to Generate Orderings for Efficient Symbolic Representations. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 624–627, June 1992.

[Mul56] D. E. Muller. Complexity in Electronice Switching Circuits. *IRE Transactions on Electron. Comput.*, EC-5:15–19, Mar 1956.

[MWBV88] S. Malik, A. R. Wang, R. K. Brayton, and A. S. Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, November 1988.

[Ous83] J. K. Ousterhout. Crystal: A Timing Analyzer for nMOS VLSI Circuits. In *Proceedings of 3rd Caltech Conference on VLSI*. Computer Science Press, 1983.

[PAS90] F. Poirot, P. Abouzeid, and G. Saucier. Lexicographic Factorizations minimizing the critical path and the routing factor. In *IFIP Working Conference on Logic and Architecture Synthesis*, June 1990.

[PCD89] S. Perremans, L. Claesen, and H. DeMan. Static Timing Analysis of Dynamically Sensitizable Paths. In *Proceedings of the 26th Design Automation Conference*, June 1989.

[Pip77] N. Pippenger. Information Theory and the Complexity of Boolean Functions. *Mathematical Systems Theory*, 10:129–167, 1977.

[PM75] K. P. Parker and E. J. McCluskey. Probabilistic Treatment of General Combinational Networks. *IEEE Transactions on Computers*, C-24:668–673, June 1975.

[RBKM91] D. E. Ross, K. M. Butler, R. Kapur, and M. R. Mercer. Fast Functional Evaluation of Candidate OBDD Variable Orderings. In *Proceedings of the European Conference on Design Automation*, pages 4–10, February 1991.

[Ree91] D. Reeves. Personal Communication. vpnr2bdd information, July 1991.

[Ris82] R. H. Risch. Staggered Input Networks: An Approach to Automatic Logic Decomposition. In *Proceedings of the International Symposium on Circuits and Systems*, pages 55–57, 1982.

[RT90] A. Rajanala and A. Tyagi. A New Area-Based Figure of Merit for Layout Synthesis Systems. In *Proceedings of the 1990 International Workshop on Layout Synthesis*. MCNC/ACM SIGDA, May 1990.

[Sau92] G. Saucier. Analysis of the Trends in Logic Synthesis. In *Proceedings of the Synthesis and Simulation Meeting and International Interchange (SASIMI)*, pages 5–25, April 1992.

[SB87] M. Schulz and Franc Brglez. Accelerated Transition-Fault Simulation. In *Proceedings of the 24th Design Automation Conference*, pages 237–243, 1987.

[SDB84] J. Savir, G. S. Ditlow, and P. H. Bardell. Random Pattern Testability. *IEEE Transactions on Computers*, C-33:79–90, January 1984.

[Sea53] F. W. Sears. *Thermodynamics, the kinetic theory of gases, and statistical mechanics.* Addison-Wesley Publishing Company, Inc., 1953.

[Sha49] C. E. Shannon. The Synthesis of Two-terminal Switching Circuits. *Bell Systems Technical Journal*, pages 59–98, January 1949.

[SPA85] S. C. Seth, L. Pan, and V. D. Agrawal. PREDICT: Probabilistic Estimation of Digital Circuit Testability. In *Proceedings of the 15th Annual Symposium on Fault-Tolerant Computing*, pages 220–235, 1985.

[Sto84] K. Stoodley. *Applied and Computational Statistics.* Ellis-Horwood Limited, 1984.

[SW49] C. E. Shannon and W. Weaver. *The mathematical theory of communication.* The University of Illinois Press: Urbana, 1949.

[TA89] K. Thearling and J. Abraham. An easily computed Functional Level Testability Measure. In *Proceedings of the 1989 International Test Conference*, pages 381–389, 1989.

[Tho79] C. D. Thompson. Area-time Complexity for VLSI. In *Proceedings of the 11th Annual ACM Symposium on the Theory of Computing*, pages 81–88, 1979.

[TSB91] H. J. Touati, H. Savoj, and R. Brayton. Delay Optimization of Combinational Logic Circuits through Clustering and Partial Collapsing. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 188–191, November 1991.

[Tya91a] A. Tyagi. An Algebraic Model for Datapath design-space exploration. In *International Workshop on Formal Methods in VLSI Design*, January 1991.

[Tya91b] A. Tyagi. How Many Paths Classify a Function: A Study in Design Space Extraction. In *Proceedings of the 24th International Symposium on Circuits and Systems.* IEEE, June 1991.

[Yao79] A. C. Yao. Some Complexity Questions Related to Distributed Computing. In *Proceedings of ACM Symposium on Theory of Computing*, pages 209–213, 1979.