A New Approach to Partition Selection in Ashenhurst-Curtis Functional Decomposition

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Engineering

in the

College of Graduate Studies

University of Idaho

by

Craig Matthew Files

August 1995

Major Professor: James F. Frenzel, Ph.D.

Authorization to Submit

Thesis

This thesis of Craig M. Files, submitted for the degree of Master of Science with a major in Computer Engineering and titled "A New Approach to Partition Selection in Ashenhurst-Curtis Functional Decomposition", has been reviewed in final form, as indicated by the signatures and dates given below. Permission is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor	_____ Dr. James F. Frenzel	Date_____
Committee Members	_____ Dr. Gene Saghi	Date_____
	_____ Dr. Jim Alves-Foss	Date_____
Department Administrator	_____ Dr. Earl E. Gray	Date_____
College Dean	_____ Dr. Weldon R. Tovey	Date_____

Final Approval and Acceptance by the College of Graduate Studies

_____ Date_____

Jean'ne M. Shreeve

Authorization to Submit

Thesis

This thesis of Craig M. Files, submitted for the degree of Master of Science with a major in Computer Engineering and titled "A New Approach to Partition Selection in Ashenhurst-Curtis Functional Decomposition", has been reviewed in final form, as indicated by the signatures and dates given below. Permission is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor	_____ Dr. James F. Frenzel	Date_____
Committee Members	_____ Dr. Gene Saghi	Date_____
	_____ Dr. Jim Alves-Foss	Date_____
Department Administrator	_____ Dr. David P. Egolf	Date_____
College Dean	_____ Dr. David M. Woodall	Date_____

Final Approval and Acceptance by the College of Graduate Studies

_____ Date_____

Jean'ne M. Shreeve

Abstract

In the 1950's Ashenhurst created a multi-level logic minimization tool, called functional decomposition. Even though functional decomposition was first created to reduce the complexity of switching networks, it has been found to have many other applications, including pattern recognition, algorithm creation, data compression, machine learning and logic minimization. The difficulty is that many of the steps are exponential in complexity, limiting the size of functions which can be decomposed in a reasonable amount of time. This thesis reports on one particular step, partition selection, and presents the previous attempts to reduce the complexity. Three new search algorithms are presented that reduce the time complexity from exponential to a polynomial time complexity. The first algorithm presented is based on using different criteria than simply using column multiplicity to guide the search. The motivation for this algorithm is the similarities between functional decomposition and the construction of decision trees. This algorithm uses a search heuristic that evaluates $\frac{N*(N+1)}{2}-1$ partitions out of a total search space of 2^N possible partitions. The second algorithm presented finds decompositions with two inputs and one output. This algorithm is based on the smallest representation of a function of N inputs is N-1 two input, one output functions. The third algorithm is the combination of the other two algorithms. Results of this algorithm show that it is highly effective in producing minimized switching networks in a short amount of time.

Acknowledgments

I am grateful to many people for the assistance that I have received since I started my Master's degree. I especially wish to thank my major professor, Jim Frenzel, for showing me some of his research interests, and showing me his application of Genetic Algorithms as applied to partition selection. I realize that he would have liked it more if I had stayed with the Genetic Algorithm approach, maybe I will get back to it again. I would also like to thank Jim for getting me involved with the Air Force Office of Scientific Research. The summer that I spent working at Wright Patterson Air Force base was very enjoyable and a real learning experience. It was great being part of the Pattern Theory Group, and I really appreciate the help they gave me last summer. A special thanks to Tim Ross for putting up with me last summer and for replying to all my questions via e-mail. Thanks to Marek Perkowski, for being a great library of papers on functional decomposition. His knowledge on this subject made the presentation of this thesis much better. Last, thanks to the AbTech Corporation for giving me the research grants; I hope that the work that I have completed can be used in their continued research.

Contents

Preliminary

Authorization to Submit Thesis ii

Abstract iii

Acknowledgments iv

Table of Contents v

List of Figures viii

List of Tables ix

List of Examples xi

1. Introduction 1

1.1 Thesis Overview 3

2. Functional Decomposition 5

2.1 Ashenhurst Simple Disjunctive Decomposition 5

2.2 Ashenhurst Simple Non-Disjunctive Decomposition 9

2.3 Figure of Merit 13

2.4 DFC as a Figure of Merit 15

3. Previous Partitioning Techniques 20

3.1 Non-heuristic Techniques 20

3.1.1 Shen and McKellar Algorithm 21

3.1.2 Redundant Elements 26
3.1.3 Complex Disjunctive 28
3.2 Ashenhurst-Curtis Partition Heuristics 30
3.2.1 Genetic Algorithms 30
3.2.2 Increasing Row Column Ratio 31
3.2.3 Berkeley Logic Synthesis Tool: SIS 32
3.3 Partitioning Conclusions 33

4. Search Strategies 34

4.1 Best Bound Partition Selection (BBPS) 34
4.1.1 BBPS Example 43
4.1.2 BBPS as a Full Decomposer 45
4.2 Single Output Search 47
4.2.1 SOS as a Full Decomposer 53
4.3 Combination of the Two Algorithms 55

5. Results 57

5.1 Smallest SOCC Results 58
5.2 Analysis of BBPS, SOS, BBPS+SOS 61
5.3 Comparisons of BBPS+SOS and SIS 64
5.4 Execution Times 68

6. Conclusions 73

6.1 Comments on Results 73

6.2 Future Work 74

Appendices

A. Description of Test Functions 76

B. Partitions that Result in Minimum DFC 80

C. ASH Program Explanation 85

Bibliography 93

List of Figures

Figure 2.1: Decomposition blocks 8

Figure 2.2: Sum of Child Cardinality (SOCC) 15

Figure 2.3: Single output decomposition 17

Figure 2.4: Two output decomposition 18

Figure 2.5: Full decomposition of Φ 18

Figure 2.6: Decomposition of F and Φ 19

Figure 3.1: Decomposition compatibility graph 24

Figure 3.2: Genetic algorithm pseudo code 31

Figure 4.1: Simple decision tree with decomposition chart, X_0 as bottom node 36

Figure 4.2: Simple decision tree with decomposition chart, X_2 as bottom node 37

Figure 4.3: Converting decision tree to functional decomposition 38

Figure 4.4: Variable selection 42

Figure 5.1: 8-variable SOCC evaluation 59

Figure 5.2: 12-variable SOCC evaluation 60

Figure 5.3: Time versus number of inputs 70

Figure 5.4: 8-variable time vs. DFC plot 71

Figure 5.5: 12-variable time vs. DFC plot 71

List of Tables

Table 2-1: Decomposition chart with three bound variables 8

Table 2-2: Decomposition chart with two bound variables 9

Table 2-3: Shared variable decomposition charts 12

Table 3-1: Shen and McKellar Mod 2 map 22

Table 3-2: Shen and McKellar Mod 2 map with 3 bound variables 22

Table 3-3: Shen and McKellar constant tables 23

Table 3-4: Example simple disjunctive partition 28

Table 3-5: Example two non-disjoint partitions with XOR 28

Table 4-1: Decomposition chart of decision tree 39

Table 4-2: Decomposition chart, X_1 is bound variable 39

Table 4-3: Decomposition chart, X_2 is bound variable 40

Table 4-4: Decomposition chart, X_1 and X_2 as bound variables 40

Table 4-5: Decomposition chart, X_1 and X_0 as bound variables. 41

Table 4-6: Compatible variable partitions 49

Table 4-7: Partition of two compatible variables 49

Table 4-8: Compatible variable check 50

Table 4-9: Partition of three compatible variables 50

Table 4-10: Non-adjacent compatible variables 51

Table 4-11: Row type chart, three variables 51

Table 4-12: Partition of compatible variables, using row type chart 52

Table 4-13: Row type chart, five variables 53

Table 5-1: DFC results for BBPS, SOS, BBPS+SOS 62

Table 5-2: Comparison of SIS to BBPS+SOS on 8-variable test functions 65

Table 5-3: Comparison of SIS to BBPS+SOS on 12-variable test functions 67

Table 5-4: Summary of minimum DFC values found with each algorithm 68

Table 5-5: Comparison of SIS and BBPS+SOS execution times (seconds) 69

Table 5-6: ADD0 times (seconds) 69

List of Examples

Example 2.1: Simple disjunctive decomposition 7

Example 2.2: Nontrivial decompositions 11

Example 2.3: Simple non-disjoint decomposition 12

Example 2.4: SOCC 14

Example 2.5: Different forms of the same function 16

Example 2.6: Finding single output Φ and SOCC 17

Example 3.1: Shen and McKellar algorithm 21

Example 3.2: Changing the function by using a complex disjunctive decomposition 28

Example 4.1: Decision trees 35

Example 4.2: Decision tree to functional decomposition 38

Example 4.3: Compatible variables 48

Example 4.4: Multiple compatible variables 49

Example 4.5: Non-adjacent compatible variables 51

Example 4.6: Counting row types 53

Chapter 1

1. Introduction

Since the advent of combinational switching networks, researchers have attempted to develop techniques to make the networks more *economical*. In general, an economical network could be defined as a network with minimal size and delay. The majority of the research has focused on developing algorithms that minimize circuit size. Some of these algorithms incorporate a design with only two stages, meaning circuits that are in the form of a Sum of Products, or a Product of Sums. This type of network is fine when a small number of inputs is used, but when there is a large number of inputs a multilevel realization is more economical.

One approach to designing multilevel switching networks is the use of functional decomposition, a method of decomposing or breaking a function into smaller pieces. The concept of functional decomposition has been known as an effective method for logic minimization for years, and was first discovered by Ashenhurst [1]. The reason that this approach has not been incorporated into logic synthesis is because of the complexity issues that are involved with functional decomposition [29]. Thus, industry has used algebraic factorization and binary decision diagram methods instead. There has been, however, some recent increase of interest in functional decomposition for applications in FPGA mapping [27, 28, 31, 37, 38] and PLA decomposition [24-26]. It must also be noted that functional

decomposition has been recently formulated for multi-valued functions [24], real-valued functions [18] and for creating testable switching networks [38].

The reason for this increased interest stems from the advent of new technologies and the ability to compensate for some of the complexity issues of functional decomposition. By using heuristic search algorithms, it is possible to reduce the exponential complexity to a more reasonable problem. The drawback is that the resulting network is not always the most economical representation, but in most cases the differences between the two implementations are very minor.

Another application of Ashenhurst functional decomposition is in Pattern Theory, a concept for finding patterns in discrete and continuous mathematical functions. This approach has been developed at Wright Patterson Laboratories in an attempt to find a more formal notation of a pattern [16]. The Pattern Theory Group has picked functional decomposition because of its ability to simplify the design process and to lower the cost of the overall realization of a function. For this reason, functional decomposition can be considered a fundamental element in the development of all computing systems, including algorithm design and machine learning.

The basis of Pattern Theory is to gain an understanding of an object's "patterness", which can be done by measuring the complexity of a pattern. Techniques for measuring this complexity should be robust enough to handle many classes of applications. The applications considered by Wright Patterson Laboratories include pattern recognition,

algorithm creation, data compression, machine learning, and logic minimization. Their approach is to use the Ashenhurst functional decomposition for a binary function.

The rest of this thesis focuses on the application of functional decomposition to logic minimization, but it can be applied to the other applications such as those considered by Wright Patterson's Pattern Theory Group.

1.1 Thesis Overview

The major goal of this thesis is to present new heuristics that reduce the complexity of functional decomposition. The heuristics are applied to the concept of partition selection, one of the NP-complete problems in functional decomposition. The work that is presented here considers single output, completely specified Boolean functions. Some of the work that is referenced in this thesis may not be written for functions that fit within these constraints but is shown to represent other work that is in this area of functional decomposition.

Chapter 1 gives the reader a fundamental understanding of functional decomposition. It explains the NP-complete problem of picking and evaluating a partition. The evaluation process of partition selection is done by using a *figure of merit*, a way to rank the given partition for its ability to lead the decomposition process into finding the most economical representation. The figure of merit that is used in this thesis and its ability to find an economical representation of a network is also presented in Chapter 2.

Chapter 3 is a review of the previous techniques proposed to reduce the complexity of the partitioning problem. Because there are many different decomposition techniques, the concepts that are explored in this chapter do not apply strictly to Ashenhurst's

decomposition. The intent of this chapter is to give the reader a fundamental understanding of the partitioning problem, and the methods that have been developed. The concepts that are considered are partitioning with Reed-Muller forms, redundant logic blocks, complex functional decomposition techniques, and new techniques on evaluating basic Ashenhurst partitions.

Chapters 4 and 5 contain the descriptions and results of the three new algorithms that are presented in this thesis. The third algorithm is a combination of two new algorithms. The two algorithms are developed and explained in Chapter 4 by giving simple examples of how they work. The three algorithms are then compared to each other. The third algorithm is then compared to results found using the Berkeley synthesis tool SIS (a system for sequential circuit synthesis).

The conclusion of this thesis is presented in Chapter 6. This chapter concludes the work by reviewing the results that are discussed in Chapter 5 and then gives insight to future work that can be undertaken.

Chapter 2

2. Functional Decomposition

This chapter provides a review of the fundamentals of functional decomposition as defined by Ashenhurst and Curtis. Ashenhurst first defined the basic properties of functional decomposition in [1-3]. Curtis used the theorems of Ashenhurst to develop his own form of decomposition in [4-5, 10]. Ashenhurst decompositions are special cases of the Curtis decompositions. There have been many other proposed types of functional decompositions since the advent of Curtis decomposition. According to [6], there are only three types of decomposition in the literature that are essentially distinct and make use of the concept of partitioning input variables into bound and free sets (discussed later in this chapter). These three methods are Curtis Decomposition [5], Steinbach Decomposition [38], and Perkowski Decomposition [8, 9]. Although these three methods are distinct they are still based on the fundamentals of Ashenhurst and Curtis decomposition. The decompositions of Ashenhurst [1-3], Luba [24], Lai and Pedram [12], and many others are just special cases, or different realizations, of the Curtis decomposition [7]. Thus, the fundamentals of Ashenhurst and Curtis decomposition presented here will provide essential insight into a wide range of functional decomposition techniques.

2.1 Ashenhurst Simple Disjunctive Decomposition

The decomposition of a function is an expression in terms of a composition of other functions. For example, if $f(X_1,X_2,X_3,X_4)=F(\Phi(X_1,X_2),X_3,X_4)$ then the term to the right is a

set of compositions that are an equivalence of the function f. The above example is called "simple" because there is only one composite function Φ in the term to the right. Curtis took this one step further by defining complex decompositions, in which more than one composite function is used.

In general, an N input, single output Boolean function, f: $B^N \rightarrow B$, where $B=\{0,1\}$, has the set of input variables $\{X_1,X_2,\ldots,X_N\} \in X$. Ashenhurst's definition of simple disjunctive decomposition is to partition the input variables into a collection of two subsets whose union is the whole set and whose intersection is empty. The partition is then used to separate the input variables into the two composite functions F and Φ to form a decomposition. For example, if A is the set of inputs to Φ and B is the set of inputs to F, and $A \cup B = \{X_1, X_2, \ldots, X_N\}$ and $A \cap B = \emptyset$, then the sets are disjoint and a simple disjunctive partition can be created. The decomposition can then be written as $f(A,B) = F(\Phi(A),B)$, where Φ and F are the composite functions of f, with A defined as the bound set, and B defined as the free set.

A principal reason that Ashenhurst-Curtis decomposition has not been used in many practical engineering applications is the search complexity that it involves. For an N input function, there are 2^N different possible partitions under simple disjunctive decomposition. According to Curtis, a function is said to be distinctively decomposable if it possesses at least one *nontrivial* partition. In the cases when the free set or the bound set has no elements, or the bound set has only one element, the decomposition is trivial. Thus, there are 2^N-N-2 nontrivial partitions.

For a given function $f(X_1, X_2, \ldots, X_N)$, a *decomposition chart* may be represented by a partition $B \mid A$. A decomposition chart is defined as a rectangular array of the 2^N functional output values of f, with $2^{|B|}$ rows and $2^{|A|}$ columns. A decomposition chart is similar to a Karnaugh map, the only exception is that while a Karnaugh map uses gray scale values, the decomposition chart uses conventional binary values. Once the decomposition chart has been realized, the object is to minimize the chart by finding matching columns. Column multiplicity, denoted by $\nu(B \mid A)$, is the number of different columns in a decomposition chart. Column multiplicity is then used to calculate the number of outputs of Φ, which is equal to the next largest integer found by the equation $\log_2 \nu(B \mid A)$, or $\lceil \log_2 \nu(B \mid A) \rceil$. Calculation of column multiplicity and the corresponding number of outputs is demonstrated in Example 2.1.

Example 2.1: Simple disjunctive decomposition

> Consider the function f, which has minterms at 0, 6, 7, 10, 11, and 12. Thus, the function could be written as $f(X_3, X_2, X_1, X_0)$=1000001100111000. Because there are 16 possible input values, there are 16 different partitions that can be evaluated. For illustration purposes: $f(X_3, X_2, X_1, X_0)$= **0** 1 2 3 4 5 **6** **7** 8 9 **10** **11** **12** 13 14 15, where the bold face values denote the minterms of f. Thus, the partition $F(\Phi(X_2,X_1,X_0),X_3)$ could be denoted as the decomposition chart shown in Table 2-1.

Table 2-1: Decomposition chart with three bound variables

	$X_2X_1X_0$							
X_3	**0**	1	2	3	4	5	**6**	**7**
	8	9	**10**	**11**	**12**	13	14	15
	0	1	2	2	2	1	0	0

Column Labels

Because there are three distinctly different columns, the column multiplicity $\nu(X_3 \mid X_2X_1X_0)$ is equal to 3. Row multiplicity $\mu(X_3 \mid X_2X_1X_0)$, which is equivalent to $\nu(X_2X_1X_0 \mid X_3)$, is found by counting the number of different rows (in this example, μ=2). By using the column multiplicity results, the function f can be written as $F(\Phi_1(X_2,X_1,X_0), \Phi_2(X_2,X_1,X_0),X_3)$, since the number of output lines from $\Phi = \lceil \log_2 3 \rceil = 2$. Thus, the decomposition could be drawn as in Figure 2.1.

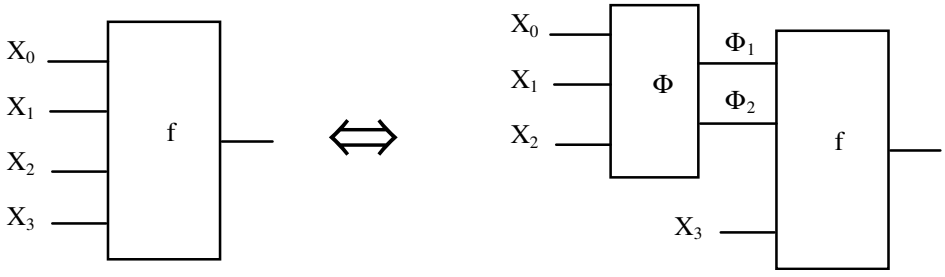


Figure 2.1: Decomposition blocks

The decomposition chart in Table 2-2 is the "best" (based on DFC which is discussed later in this chapter) partition for the above example.

Table 2-2: Decomposition chart with two bound variables

X_1X_0 \ X_3X_2	**0**	4	8	**12**
	1	5	9	13
	2	**6**	**10**	14
	3	**7**	**11**	15
	0	1	1	0

Column Labels

In the decomposition chart depicted in Table 2-2, the column multiplicity equals 2 and the number of outputs from Φ is 1. After finding the column multiplicity, the next step is to find the function Φ. In the partition $X_1X_0 \mid X_3X_2$, because the first and the fourth columns are equal and the second and third are equal, Φ can be realized as 0110, or X_3 XOR X_2, and $F(\Phi, X_1, X_0) = \overline{\Phi}\overline{X}_1\overline{X}_0 + \Phi X_1\overline{X}_0 + \Phi X_1 X_0$

$= \overline{\Phi}\overline{X}_1\overline{X}_0 + \Phi X_1$.

The example illustrates the processes that are involved with decomposing a function using simple disjunctive decomposition. The complex processes include partition selection, partition evaluation, column labeling, and determining Φ. Partition selection is the main emphasis in this thesis.

2.2 Ashenhurst Simple Non-Disjunctive Decomposition

The previous definition of simple disjunctive decomposition is the basis for more complex decomposition structures, i.e., more than one function Φ. Examples of complex structures are in the form of the following:

1. $f(A,B) = F(\Phi(A),\Im(B))$

2. $f(A,B,C) = F(\Phi(A),\Im(B), C)$

3. $f(A,B,C) = F(\Phi(\Im(A), B), C)$

However, it has been shown by Shannon [13] that the fraction of all functions of N variables possessing nontrivial simple disjunctive decompositions approaches zero as N approaches infinity. Thus, Ashenhurst used his disjunctive decomposition theory to build a non-disjunctive one. A non-disjunctive decomposition partitions the input variables into a collection of subsets whose union is the whole set and whose intersection is *not* empty. Thus, if $A \cup B=\{X_1,X_2,\ldots,X_N\}$ and $A \cap B \neq \emptyset$, then the sets are non-disjoint and a non-disjunctive decomposition can be formed.

Ashenhurst stated that a function f(A,B) of N variables can be expressed as f(X,Y,Z) where $X = A \cap \overline{B}$, $Y = B \cap \overline{A}$, and $Z = A \cap B$. The function can then be written in the expanded form:

$$f(X, Y, Z) = \sum_{j=0}^{2^{[z]}-1} f_j(X, Y) p_j(Z)$$

where f_j is a new function with inputs X and Y, and p_j is an associated elementary product. In relation to a partition matrix, f_j is the function corresponding to the row vectors, while p_j is the number of partition tables for a given value of the Z variables. Thus, a function f(X,Y,Z) of N variables can be represented by a partition $Y \mid X : Z$, where there is a set of $2^{|Z|}$ decomposition charts, each with $2^{|Y|}$ rows and $2^{|X|}$ columns. Notice that for an N input function there are 3^N different non-disjoint simple decompositions, compared to the 2^N-N-2 simple disjoint decompositions. Because simple non-disjoint decompositions are made up of

sets of simple disjoint decompositions, for each set there exists $2^{(X+Y)}$-(X+Y)-2 nontrivial partitions. In order to be nontrivial, there must be at least one shared variable and at least one variable in the free set and two variables in the bound set. Consequently, there are

$$\sum_{i=3}^{N-1}\binom{N}{i}*(2^i-i-2)$$

nontrivial, non-disjunctive decompositions. Also, the function being evaluated must have at least four inputs to have nontrivial, non-disjunctive decompositions. Example 2.2 shows the number of nontrivial partitions when the input function has seven inputs.

Example 2.2: Nontrivial decompositions

For a seven input function, there are $3^7 = 2187$ different partitions.

$$\begin{aligned}\sum_{i=3}^{6}\binom{7}{i}*(2^i-i-2) &= \binom{7}{6}*(2^6-6-2)+\\ &\quad \binom{7}{5}*(2^5-5-2)+\\ &\quad \binom{7}{4}*(2^4-4-2)+\\ &\quad \binom{7}{3}*(2^3-3-2)\\ &= 392+525+350+105\\ &= 1372\end{aligned}$$

Thus, for a seven input function, approximately half of the possible non-disjoint decompositions are nontrivial. Example 2.3 illustrates the construction of a simple non-disjoint decomposition.

Example 2.3: Simple non-disjoint decomposition

Consider a function f with minterms at 5, 6, 7, 10, 11, 13, 21, 22, 23, 24, 25, 26, 27 and 28. Because there are five input variables and 32 possible input combinations, there are $3^5 = 243$ non-disjoint partitions that can be evaluated. For illustration purposes: f= 0 1 2 3 4 **5 6 7** 8 9 **10 11** 12 **13** 14 15 16 17 18 19 20 **21 22 23 24 25 26 27 28** 29 30 31 32. One possible partition is $X_4X_3 \mid X_2X_0 : X_1$, as shown in Table 2-3.

Table 2-3: Shared variable decomposition charts

X_1=0

X_4X_3 \ X_2X_0	0	1	4	**5**
	8	9	12	**13**
	16	17	20	**21**
	24	**25**	**28**	29
	0	0	0	1

column multiplicity υ =2

X_1=1

X_4X_3 \ X_2X_0	2	3	**6**	**7**
	10	**11**	14	15
	18	19	**22**	**23**
	26	**27**	30	31
	0	0	1	1

column multiplicity υ = 2.

$\Phi_0 = X_2X_0$ or 0001

$\Phi_1 = X_2\overline{X}_0 + X_2X_0$ or 0011

$\Phi(X_1, X_2, X_0) = \Phi_0(X_2, X_0)\overline{X}_1 + \Phi_1(X_2, X_0)X_1$ or 00010011, and

$$F(X_1, \Phi(X_1, X_2, X_0), X_4, X_3) = \overline{X}_1\overline{\Phi}X_4X_3 + \overline{X}_1\Phi\overline{X}_4\overline{X}_3 + \overline{X}_1\Phi\overline{X}_4X_3 + \overline{X}_1\Phi X_4\overline{X}_3 + X_1\overline{\Phi}\overline{X}_4X_3 + X_1\overline{\Phi}X_4X_3 + X_1\Phi\overline{X}_4\overline{X}_3 + X_1\Phi X_4\overline{X}_3$$

or 0001111001011010.

2.3 Figure of Merit

Curtis pointed out that to evaluate each partition some sort of cost function must be used to determine the "effectiveness" of the partition. At the time of writing his book and his papers, the technology that existed placed constraints on the timing and area efficiency of certain Boolean gates. With current VLSI technology, we still must place some constraints on some devices, e.g. an XOR gate takes up more area than an AND gate or an OR gate. With FPGAs that use lookup table architectures there is not any difference between gates with equal numbers of inputs. In some cases, the use of ESOP (EXOR Sum of Products) forms will give smaller representations than a SOP or a POS implementation. Also, when using the concept of functional decomposition in pattern theory, the main concern is determining the "pattern" of the object. Thus, it is unnecessary to evaluate the functions differently as long as a "good" figure of merit is chosen to represent the pattern.

An appropriate figure of merit for representations that does not require distinction between the different gate types is Decomposed Function Cardinality (DFC). This idea was first proposed by Abu-Mostafa in 1988 [14]. DFC is defined as the sum of the cardinalities of the components of a combinational representation [15]. It is a good representation of the complexity or "patterness" of a function, since a function is a set of ordered pairs and, as with any set, has a definite property in its number of elements or cardinality. Thus, for a function that has multiple decompositions, DFC may be used to find the minimum combined component cardinality of all the decompositions of that function. Because of the simplistic representation of DFC, it is also a good figure of merit for evaluating function implementations in most of the current technologies.

The concept of the DFC measure is used by the Pattern Theory Group as a robust measure of pattern. Thus, DFC can be used in common pattern recognition applications, such as the machine learning approaches of geometric, syntactic and artificial intelligence. DFC has also been shown to be related to factors in data compression, time complexity and size complexity of a circuit [23].

The Pattern Theory Group found by experimentation that DFC is a good evaluation of a decomposed function [16, 19]. The basic approach of using DFC to evaluate a decomposed function is to find the dfc (dfc in lower case is used to show that only a section of the fully decomposed function is being evaluated) of a partition. In finding the dfc of a partition, the sum of the functional cardinality of the separated blocks of the decomposition is used. Instead of using dfc for a simple disjunctive decomposition, Sum of Child Cardinality (SOCC) can be used. Sum of Grand Child Cardinality (SOGCC) is found by decomposing a function into F and Φ, then adding the SOCC of F to the SOCC of Φ. Example 2.4 shows the fundamentals of finding the SOCC of a function.

Example 2.4: SOCC

Consider a six input function, which has cardinality = $2^6 = 64$. After decomposing the function, the netlist of decomposed blocks shown in Figure 2.2 is created.

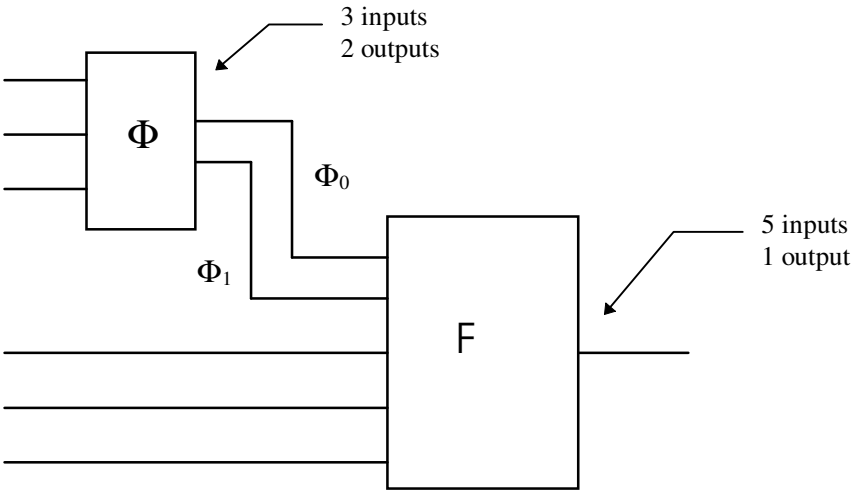


Figure 2.2: Sum of Child Cardinality (SOCC)

SOCC = cardinality(Φ_0)+ cardinality(Φ_1) + cardinality(F)

SOCC = $2^3 * 2 + 2^5 = 8 * 2 + 32 = 48$

The DFC of a decomposed function is then the sum of all the decomposed blocks. For example, the DFC for $F(\Phi(\Psi_0(X_1, X_2), \Psi_1(X_1, X_2), X_3), X_1, X_4, X_5)$ is equal to the Cardinality of $\Psi * 2$ + Cardinality of Φ + Cardinality of F, i.e. $2^2 * 2 + 2^3 + 2^4=32$. When calculating the DFC, the shared variable is not evaluated any differently from any of the other input variables.

2.4 DFC as a Figure of Merit

Having defined DFC as a measure of a function's complexity, this section will show how it can be used as a figure of merit. A function can be decomposed into many different

forms, each of which could have the same DFC. This also holds true for the SOCC. A function may have several simple decompositions that produce the same SOCC.

Example 2.5 shows how a function can have many different forms, but the same DFC. In most cases, the partition that results in the smallest SOCC can be decomposed to find the smallest possible DFC.

Example 2.5: Different forms of the same function

Given a three input function f that calculates parity, the following 3 decompositions exist:

1. $(X_2 \oplus X_3) \oplus X_1$
2. $(X_1 \oplus X_2) \oplus X_3$
3. $(X_1 \oplus X_3) \oplus X_2$

In each of the above decompositions, the resulting DFC is equal to $2^2+2^2=8$.

However, there are a few occasions that the decomposition with the smallest SOCC cannot be decomposed so that it produces the smallest DFC. Appendix B lists the example test functions for which the partition with the smallest SOCC does not decompose into the smallest DFC. In most of these, the partitions that decomposed into the smallest DFC had only one output from the function Φ. This occurs because using the concept of DFC as a figure of merit at the lower levels of the decomposition process cannot distinguish that a decomposition with one output is potentially a better solution. Example 2.6 shows how a

single output decomposition would not be selected to find lower level decompositions if an algorithm uses SOCC as a figure of merit.

Example 2.6: Finding single output Φ and SOCC

Given an eight variable input function with minterms (243, 244, 245, 246, 248, 249, 250), the following decomposition can be drawn either by selecting partition $X_7X_6X_5X_4X_3X_2 \mid X_1X_0$, or partition $X_7X_6X_5X_4X_1X_0 \mid X_3X_2$. Both partitions have a column multiplicity of 2, which is shown in Figure 2.3 with a single output line from **Φ**.

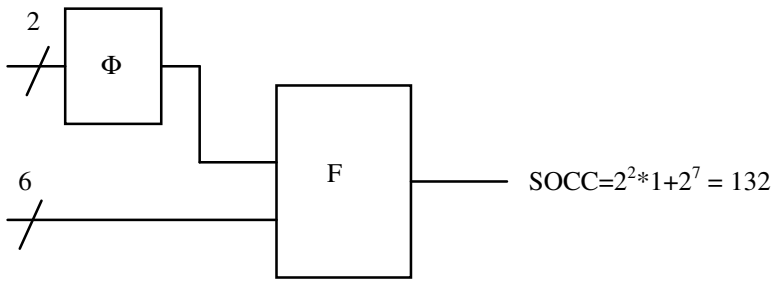


Figure 2.3: Single output decomposition

Now, by selecting $X_3X_2X_1X_0$ as bound variables, $\nu(X_7X_6X_5X_4 \mid X_3X_2X_1X_0) = 4$ as shown in Figure 2.4.

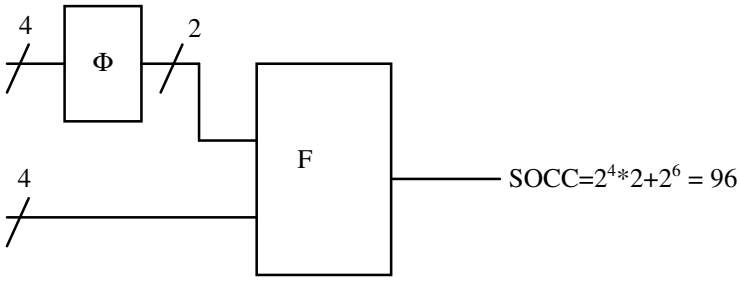


Figure 2.4: Two output decomposition

In choosing the partition to be decomposed at lower levels, the partition with the smallest SOCC is chosen, i.e., the second partition. But, continuing the decomposition process on these two partitions shows that the second partition is not the best pick. Figure 2.5 depicts the result of decomposing the bound set from the partition $X_7X_6X_5X_4 \mid X_3X_2X_1X_0$.

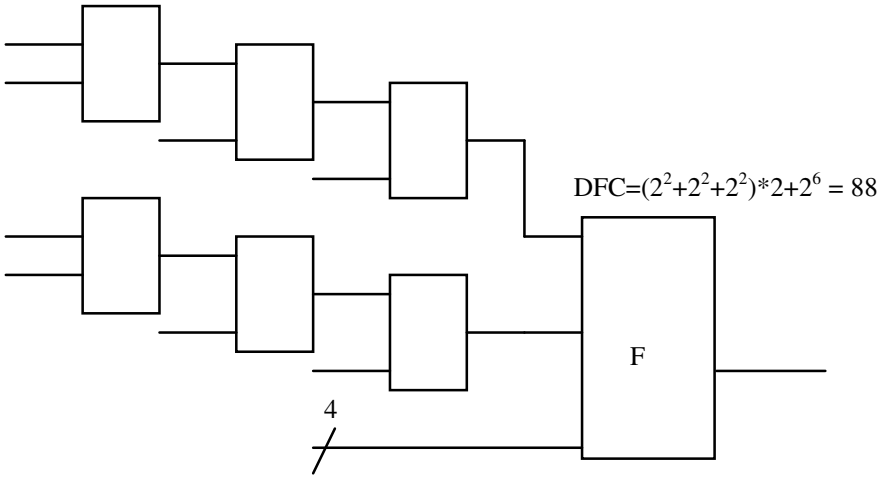


Figure 2.5: Full decomposition of Φ

Now, by decomposing the function F that resulted from the partition $X_7X_6X_5X_4X_1X_0 \mid X_3X_2$ the decomposition in Figure 2.6 is found.

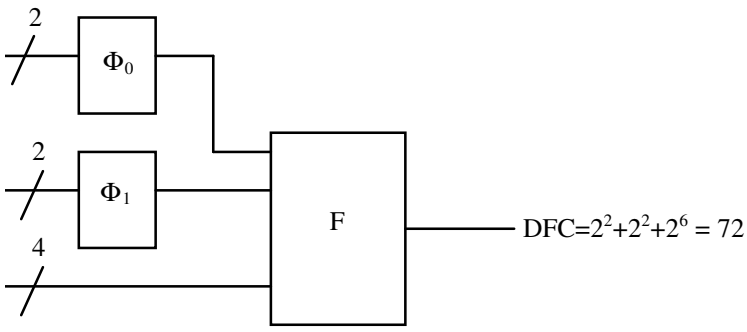


Figure 2.6: Decomposition of F and Φ

The reason that this works is that the original function decomposes easily and in different ways. Once a variable is made a bound variable, it is no longer related to the original function. This is the reason for the term "bound" set. Thus, there is no way of knowing how the function Φ will decompose, while F will decompose according to the attributes of the original function f. The conclusion is that minimum SOCC is a fairly good predictor to find the smallest DFC, but it would work much better if single output decompositions were found first.

Chapter 3

3. Previous Partitioning Techniques

Since the development of Ashenhurst-Curtis decomposition, the main emphasis of research has been in forming new decomposition techniques [32]. The research has developed many types of decompositions, but they are still based on Ashenhurst's ideas. Because many of these decompositions were designed for a specific type of problem, they have become obsolete, either by the advent of new technologies or new algorithms. Thus, there have been very few attempts to reduce the complexity of Ashenhurst-Curtis' partition problem. This chapter explores heuristic and non-heuristic partitioning techniques used in Ashenhurst-Curtis decomposition and other related decompositions.

3.1 Non-heuristic Techniques

The algorithms in this section find decompositions without using any search heuristics. The basic idea of these algorithms is to remove or not search all possible partitions for a decomposition. This is done by using different functional methods to choose which partitions will be evaluated. These methods select partitions through Reed-Muller expansions, Fourier transforms, binary difference equations, and technology-based mappings.

3.1.1 Shen and McKellar Algorithm

In 1970, Shen and McKellar [33] published an algorithm that provides a quick way to determine the bound and free sets so that the function will decompose. The algorithm uses the Reed-Muller representation of Boolean functions in the decomposition process. By modifying the tabular representation of Ashenhurst-Curtis into a mod 2 map, that fits the Reed-Muller mod 2 canonical form [35], they are able to determine if pairs of variables, when paired with the remaining variables in the bound set, will produce a decomposition. Example 3.1 is a basic example of their algorithm.

Example 3.1: Shen and McKellar algorithm

Because the Shen and McKellar algorithm needs to be expressed in a canonical form, the following function is given:

$$\text{Let } f(X_4, X_3, X_2, X_1, X_0) = 1 \oplus X_0 \oplus X_2X_0 \oplus X_1 \oplus X_3X_1 \oplus X_3X_2X_1 \oplus X_4X_3X_1 \oplus X_4$$

Pick the bound set to be $\{X_1, X_0\}$ and the free set to be $\{X_4, X_3, X_2\}$. The mod 2 map is shown in Table 3-1, where there is a 1 in a given row and column if there is a coefficient of 1 in the canonical expression. In this example, $X_3X_2X_1$ has a coefficient of 1. Therefore, a 1 is placed in row X_3X_2 and column X_1 of the table.

Table 3-1: Shen and McKellar Mod 2 map

	1	X_0	X_1	$X_1 X_0$	*Constant*
1	1	1	1	0	6
X_2	0	1	0	0	4
X_3	0	0	1	0	2
$X_3 X_2$	0	0	1	0	2
X_4	1	0	0	0	0
$X_4 X_2$	0	0	0	0	0
$X_4 X_3$	0	0	1	0	2
$X_4 X_3 X_2$	0	0	0	0	0

The column labeled *Constant* is Shen and McKellar's way to represent the value of the given row excluding the first column of the table. They proved that a function is decomposable with a bound set of two variables if and only if the nonzero *Constant,* with respect to the pair of variables, is identical. Thus, by adding any one of the remaining variables to the bound set, they can determine if the initial pair of variables will create a decomposition. For example, adding X_2 to the bound set is shown in Table 3-2.

Table 3-2: Shen and McKellar Mod 2 map with 3 bound variables

	1	X_0	X_1	$X_1 X_0$	X_2	$X_2 X_0$	$X_2 X_1$	$X_2 X_1 X_0$
1	1	1	1	0	0	1	0	0
X_3	0	0	1	0	0	0	1	0
X_4	1	0	0	0	0	0	0	0
$X_4 X_3$	0	0	1	0	0	0	0	0

After adding all the remaining variables to the initial pair, the *Constant* tables can be drawn as shown in Table 3-3. The *Constant* table $\{X_2,X_1,X_0\}$ is found by evaluating

each row, excluding the first column, in Table 3-2. The remaining two *Constant* tables are found by repeating the process that was used in finding the *Constant* table $\{X_2,X_1,X_0\}$.

Table 3-3: Shen and McKellar constant tables

$\{X_2,X_1,X_0\}$

6	4
2	2
0	0
2	0

$\{X_3,X_1,X_0\}$

6	2
4	2
0	2
0	0

$\{X_4,X_1,X_0\}$

6	0
4	0
2	2
2	0

The tables show that the nonzero rows are not equal. Thus, Shen and McKellar state that the function is not decomposable if the bound set has three variables and contains X_1 and X_0. For example, if the two columns in one of the above tables were equal, then using X_1 and X_0 in the bound set with any other variable would result in a decomposition.

Shen and McKellar's algorithm searches for a decomposition based on searching the available bound sets. There exists $\binom{N}{2} = \frac{N!}{(N-2)!2!}$ possible partitions to evaluate for bound sets of size N. The algorithm eliminates the bound sets that have nonzero rows that are not equal, the remaining sets become candidates for decomposition and further checking. After evaluating all possible pairings of a given input function, Shen and McKellar place the variables in a *decomposition compatibility graph*. Each node in the graph holds one variable

and each edge in the graph is the size of the bound set in which the two variables are compatible. Figure 3.1 illustrates this concept.

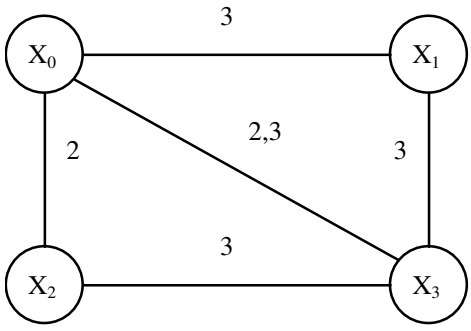


Figure 3.1: Decomposition compatibility graph

The *decomposition compatibility graph* shows that X_0 and X_1 are compatible with each other when there are three elements in the bound set, X_0 and X_3 are compatible when there are two or three elements in the bound set, and X_2 and X_1 are not compatible. From this graph, the bound sets left can be tested to see which one will best decompose the function.

Shen and McKellar's algorithm finds all of the simple disjunctive decompositions of a function. Thus, the algorithm performs well if the function is hard to decompose, i.e., when very few bound sets yield a decomposition. When every bound set results in a simple disjunctive decomposition, this approach will contain all 2^N possible partitions except those with two variables in the bound set. Another problem with this approach is that it requires the use of only AND and XOR gates to represent a function, due to the use of the Reed-Muller form. Although it is possible to represent every function with only AND and XOR gates, this might not result in the smallest representation of a function. Finally, the time that

it takes to convert a function to the Reed-Muller spectrum has an exponential time complexity.

At the very least, Shen and McKellar's algorithm will evaluate $\binom{N}{2}$ partitions to determine whether the function can be decomposed. In the worst case, this technique could find that all 2^N-N-2 nontrivial partitions result in a decomposition. Further, the algorithm was originally intended to find complex disjoint decompositions of all the possible simple disjunctive decompositions. Thus, in the worst case, the algorithm must search all possible partitions and does not reduce the partitioning problem at all.

Shen and McKellar extended their earlier research by using partial derivatives of the input function during bound set selection instead of using the *decomposition compatibility graph* [34]. The time complexity for the extended algorithm increases to N^3. Thus, the new algorithm executes much slower than the previous one and the execution time varies with the complexity of the expression that is given. Overall, the Shen and McKellar approaches were instrumental in leading researchers to look for heuristic methods instead of using an exhaustive search.

In 1989, Varma and Trachtenberg extended the work done by Shen and McKellar in [36]. Their approach differs in that it incorporates the necessary and sufficient conditions to test the existence of decompositions in one procedure. Again, the Reed-Muller form is used, but the authors use new parallel processing techniques on Fourier-like transforms to reduce the computation complexity of the Reed-Muller spectral procedures. The algorithm uses these spectral representations of a function to detect multiple disjoint decompositions

simultaneously. Again, the decomposition is composed of AND and XOR gates and the heuristics of the algorithm may end up testing all possible partitions if the function can be easily decomposed.

3.1.2 Redundant Elements

Another approach to selecting partitions during decomposition is to find redundant modules within the structure of a given function. There are two methods for applying this approach. The first is to look for redundant modules after decomposing a function and the second is to find redundant modules across many output functions.

In 1993, Schafer and Perkowski focused on reducing the size of multiplexer circuits for implementing completely specified, single output functions [37]. The first method they presented is based on a figure of merit derived from ratioing the parameters of a basic multiplexer with three inputs and one output. Thus, the search space is reduced from 2^N to $\binom{N}{3} = \frac{N!}{(N-3)!3!}$. The second method is based on the Rademacher-Walsh spectrum, which is complex in its transformation. However, the authors proposed a much simpler method they called a *local transformation.* The idea behind both methods is to reduce the multiplexer tree circuit by eliminating redundant elements level by level, starting at the output of the function. Thus, the function is decomposed into multiplexer blocks, from its output to its inputs, while the algorithm looks for possible input combinations that produce redundant multiplexer modules.

Tadeusz Luba has developed many different approaches to functional decomposition [32]. In the case of redundancy, he shows how decomposition of multiple-valued functions can benefit from looking for redundant modules [24]. The idea is to find decompositions that have redundancy across multiple outputs. This is done by looking for conditions that are based on the *r-admissibility* of a set of partitions. A partition is *r-admissible* if and only if there exists a decomposition of f for which F has r inputs and Φ has k outputs. This type of decomposition resulted from Luba's work on technology dependent algorithms where technologies like FPGAs and PLAs require a maximal number of inputs and outputs from a single block. Thus, the partitions that Luba evaluates must meet the constraints placed on the decomposition by the technology. In his redundant module algorithm, Luba searches all partitions with a certain number of variables in the bound set. If a partition with a selected set of bound variables is found such that each of the multiple-outputs of the function can use the bound set as a redundant element, then the partition is selected as the "best". For example, in the case of FPGAs that require five inputs to a logic block, the search complexity is reduced from 2^N to $\binom{N}{5} = \frac{N!}{(N-5)!5!}$. The algorithm is not looking for the best possible decomposition, it is looking for a decomposition that fits within the constraints on the number of inputs and outputs of a block. Once a decomposition is found, the algorithm continues by decomposing the Φ and F blocks that have just been found.

3.1.3 Complex Disjunctive

The goal of complex disjunctive partition selection is to force the function into a decomposition of the form f(A, B) = F(G(A), H(B)) or f(A,B,C)=F(G(A,C), H(B,C)). Thus, the decomposition results in all the variables being bounded. Complex disjunctive decomposition is illustrated in Example 3.2.

Example 3.2: Changing the function by using a complex disjunctive decomposition

Consider the following simple disjunctive decomposition chart for a five input function.

Table 3-4: Example simple disjunctive partition

1	1	0	0	1	0	0	0
1	0	0	1	1	1	0	1
0	1	1	0	0	0	1	0
0	0	0	0	0	0	0	0

ν=4

The partition can be changed into two separate partitions by using an XOR gate. This results in the partitions in Table 3-5.

Table 3-5: Example two non-disjoint partitions with XOR

1	1	0	0	1	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

ν=2

⊕

0	0	0	0	0	0	0	0
0	1	1	0	0	0	1	0
0	1	1	0	0	0	1	0
0	0	0	0	0	0	0	0

ν=2

While the example shows the use of a simple logic gate to transform a partition into two separate partitions, the resulting simple disjunctive decompositions result in the same DFC of 36. In fact, this type of decomposition will work only if the function is not decomposable and has a column multiplicity greater than 4. In [37] and [38] the authors propose the use of the above complex disjunctive decomposition technique coupled with simple search heuristics. This type of decomposition method and search works well for functions that are hard to decompose, but it is not a good method when the function can be decomposed using Ashenhurst's simple disjunctive decomposition process. The approach of this form of non-disjoint decomposition is to force variables in one of the partitions to have no relevant effect on the output so they can be thrown out. This reduces the DFC value that was found when building the non-disjoint decomposition. For example, if a three-input function uses this form of decomposition, the SOCC returned is $2^3+2^3=16$, which is much larger than the original DFC of a three input function, 8. If the variables can be forced to have no relevant effect on the output, the SOCC returned will be smaller than 16, and hopefully smaller than 8.

This section has discussed several previous methods of reducing the partition space. Each of these methods has been found to have prohibitive time complexities because each may search all possible partitions. Also, the complexity increases when the function is evaluated in other forms. These forms include the Reed-Muller, Fourier, and Rademacher-Walsh transformations, all of which result in an exponential time complexity.

3.2 Ashenhurst-Curtis Partition Heuristics

There has been very little success in algorithms that look directly at a Boolean function without any kind of transformation. In the past few years, there has been an increasing amount of research to find heuristics that only use the partitions found in Ashenhurst-Curtis decomposition. This section presents the three heuristic-based decomposition methods: genetic algorithms, increasing row column ratio, and Berkeley's sequential circuit synthesis.

3.2.1 Genetic Algorithms

The use of the Genetic Algorithm (GA) has received widespread attention recently (e.g., [21, 22]). GAs have been used to find approximate solutions to NP-complete optimization problems [17]. Thus, GAs would appear to be an appropriate approach to this problem. However, to the best of our knowledge, there have been few successful applications of genetic algorithms in the area of logic synthesis.

The GA is an evolutionary learning technique developed by Holland (1975) and based on genetic variation and natural selection. The GA is a simulation of the basic processes that are found in nature. Included in these processes are natural selection and reproduction. Highly fit individuals are allowed to reproduce with other individuals in the population, while other, unfit, individuals are removed from the population to make way for the newly formed individuals. By using this type of selection, it is hoped that the most promising areas of the search space will be explored. The GA should then converge towards an optimal solution of the problem. Figure 3.2 is the basic pseudo-code for a GA.

```
Initialize population

Repeat
        Evaluate fitness of each member
        Remove unfit members
        Reproduce based on fitness of remaining members
Until done
```

Figure 3.2: Genetic algorithm pseudo code

The initial population is probably the most important part of the GA. If the population is diverse then the probability of finding solutions will be much greater than if the population is localized in one area of the search space. This gives the GA the variance that it needs to complete the task. Another element in the implementation of the GA is the evaluation of the population. This is where the individuals of the population are given their fitness rating. SOCC has been used as a fitness measure in the GA work that has been conducted on functional decomposition.

In Chapter 5, two simple demonstrations of GAs are used for comparison with several new algorithms presented later in this thesis. The two GAs are very similar in the way they reproduce. The difference is that GA1 uses a randomly generated initial population and GA2 uses a set initial population that forces diversity between the members of the population.

3.2.2 Increasing Row Column Ratio

Another approach to finding the best partitions in Ashenhurst-Curtis decomposition is the method of Increasing Row-Column ratio (IRC) [11]. This search technique takes

advantage of the fact that the minimum SOCC is usually found when there are fewer elements in the free set than in the bound set. The IRC algorithm evaluates the partitions of a given function by starting with all the possibilities of one element in the free set. This is continued until all the possibilities of N-1 elements in the free set have been evaluated.

Both the GA and IRC approaches, on average, are faster at finding the minimum SOCC than random search techniques. The problem is that the algorithms do not guarantee when the minimum SOCC will be found. Neither approach has an early stopping condition. In some cases, all 2^N partitions, have to be searched before finding the minimum SOCC. Therefore, the time complexity of the algorithms is $O(2^N)$.

3.2.3 Berkeley Logic Synthesis Tool: SIS

SIS (a system for sequential circuit synthesis) is a program that can be used as a tool for automatic synthesis and optimization of sequential and combinational circuits [20]. The current implementation of SIS is built on top of the program MISII (a multiple-level logic optimization system). This program does not implement a decomposition-based algorithm but rather is a multiple-level optimization tool that is currently used in industry. SIS is one example of a method to find a multiple-level reduced function that is not based on functional decomposition. It represents a leading industrial algorithm and is used for comparison to the methods presented in the next chapter.

SIS can be used as a multi-level functional decomposer through the use of scripts. The scripts are text files which specify the sequence of algorithms to apply when reducing or minimizing function. Some of the algorithms that are included within SIS are timing

reduction, area reduction and testability. The synthesis proceeds by using state minimization and state assignment, global area minimization and performance optimization, local optimization, and technology mapping. The implementation of SIS makes it very difficult to always get the best results from the same script. Thus, the SIS tools come with several multi-level optimization scripts that are used by the program. The idea is to run all of these scripts on a function and then choose the representation that works the best. The different scripts that are included in the SIS package are Algebraic, Boolean, Delay, MCNC, MSU, OCT and Rugged.

3.3 Partitioning Conclusions

All of the partition selection techniques discussed in this chapter have the ability to reduce the complexity of the partitioning problem. Because the algorithms vary greatly in their approach and because their performance is dependent on the function being decomposed, it is very difficult to directly compare. Finding the conditions under which each algorithm excels is a topic of future work.

Each of the algorithms presented in this chapter may have to search all possible partitions at every level. Thus, each algorithm has an exponential time complexity in the worst case. However, in the best case most of the algorithms have a polynomial time complexity when finding simple decompositions. Thus, there is still the need to develop efficient algorithms that have an upper bound polynomial time complexity. Two new algorithms and a hybrid version of the two that all have polynomial worst case time complexities are presented in the next chapter.

Chapter 4

4. Search Strategies

This chapter is the culmination of the research presented in this thesis. The goal was to develop an algorithm that could reduce the exponential complexity of the partition selection problem to one with polynomial complexity. The success of the algorithms proposed is based on using figures of merit other than SOCC to guide the search to find the partition with the smallest SOCC. The partition with the smallest SOCC is then used to find lower level decompositions. The first algorithm is based on a best bound type search technique that was developed to find simple disjunctive decompositions with the smallest SOCC. The second algorithm is based on the techniques discussed in Section 2.4 (i.e., decompositions with single outputs). Both algorithms are restricted to disjunctive decompositions.

4.1 Best Bound Partition Selection (BBPS)

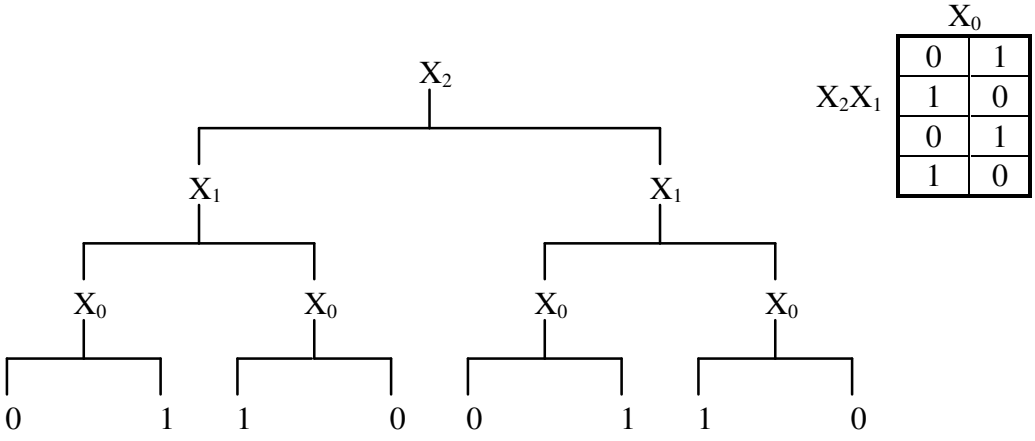
In Chapter 2, the use of a decomposition chart to find a decomposition for a function was discussed. It has been shown in [12] that it is possible to transform a decomposition chart into a Decision Tree (DT). A DT is a tree where every level in the tree is a specific variable and the leaf nodes in the tree are the outputs of the function (see Figure 4.1). The most general way of developing a DT is to start building it from the top, or the root. In the case of converting a decomposition chart into a DT, the bottom nodes in the tree are the column variables (bound set), and the top nodes in the tree are the row variables (free set).

Some point between the two sets of variables in the tree is selected to define the difference between free and bound sets.

The concept of Best Bound Partition Selection (BBPS) is based on the idea of representing functional decomposition as a DT. BBPS builds the DT from the bottom up, since the bottom nodes in the DT are the bound variables. Selecting the bottom node in the DT is accomplished by examining all N possible variable combinations. The variable or node that has the greatest number of folding (matching) elements is then chosen as the best bottom node. This approach is shown in Example 4.1.

Example 4.1: Decision trees

Consider the three input function $f(X_2, X_1, X_0)= 01100110$. If partition $X_2X_1 \mid X_0$ is selected, the following decision tree and corresponding decomposition chart exist. The decision tree branches to the left if the value of the variable is a zero, and to the right if the value is a one.



X_2X_1 \ X_0	0	1
1	0
0	1
1	0

Figure 4.1: Simple decision tree with decomposition chart, X_0 as bottom node

In this case, using X_0 as the bottom node in the tree results in four *mismatches* and the number of *mismatches* does not depend on the order of X_1 and X_2 in the tree. The term mismatch is used to denote the number of elements under X_0 that are not equal. The goal is to fold the elements under X_0, but this can only occur when the elements are equal. By switching the nodes in the tree such that X_2 is the bottom node, the DT and decomposition chart can be drawn as in Figure 4.2.

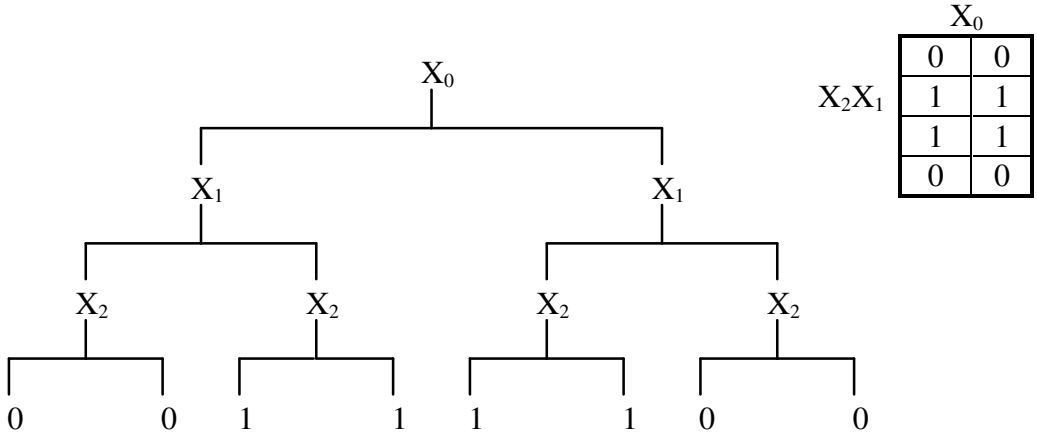


Figure 4.2: Simple decision tree with decomposition chart, X_2 as bottom node

In this configuration there are no mismatches. Thus, X_2 is the best candidate for being the bottom node of the tree. Because there are no mismatches, X_2 is found to be a vacuous variable, meaning that the variable has no relevant effect on the output of the function.

This process is then continued to select each successive level, until the root of the tree is chosen.

The process of choosing the bottom level of nodes in the tree is the same as selecting all the possible partitions that have only one bound variable. The bound variable that returns the best properties is then chosen as the best possible bound variable. The next set of partitions that are used will be the best bound variable individually paired with all the remaining variables (N-1 possible partitions). This process is continued until there is only one possible partition to evaluate.

For the remaining, N-1 variables, the best partition is selected using the following cost function. Cost is defined as the partition with the smallest column multiplicity, with a tie breaker being the smallest number of row mismatches. Example 4.2 shows the way BBPS uses this cost function to find the minimum SOCC.

Example 4.2: Decision tree to functional decomposition

Given the three input function $f(X_2, X_1, X_0) = 01011011$, the following decision tree exists:

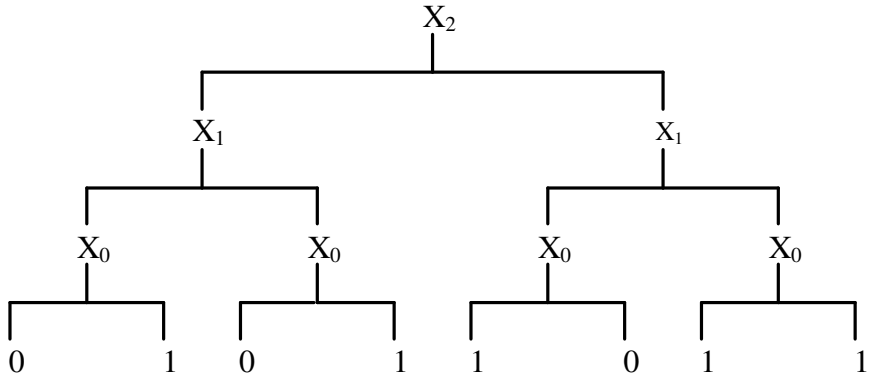


Figure 4.3: Converting decision tree to functional decomposition

The above tree can be represented as the following decomposition table.

Table 4-1: Decomposition chart of decision tree

	X_0		
X_2X_1	0	1	- Mismatch
	0	1	- Mismatch
	1	0	- Mismatch
	1	1	- Match

column multiplicity $\nu = 2$
number of row mismatches = 3

With X_1 as the bottom level of nodes, the decomposition table is:

Table 4-2: Decomposition chart, X_1 is bound variable

	X_1		
X_2X_0	0	0	- Match
	1	1	- Match
	1	1	- Match
	0	1	- Mismatch

column multiplicity $\nu = 2$
number of row mismatches = 1

With X_2 as the bottom level of nodes, the decomposition table is:

Table 4-3: Decomposition chart, X_2 is bound variable

	X_2		
X_2X_1	0	1	- Mismatch
	1	0	- Mismatch
	0	1	- Mismatch
	1	1	- Match

column multiplicity $\nu = 2$
number of row mismatches = 3

From the above three partitions, the partition with X_1 as the single bound variable is chosen to be the best, since it has the fewest mismatches. Continuing to the next level, X_1 is now paired with the other 2 variables as bound variables. The case where X_1 and X_2 are bound variables is shown in Table 4-4, while the case where X_1 and X_0 are bound variables is shown in Table 4-5.

Table 4-4: Decomposition chart, X_1 and X_2 as bound variables

	X_2X_1				
X_0	0	0	1	1	- Mismatch
	1	1	0	1	- Mismatch

column multiplicity $\nu = 3$
number of row mismatches = 2

Table 4-5: Decomposition chart, X_1 and X_0 as bound variables.

	X_1X_0				
X_2	0	1	0	1	- Mismatch
	1	0	1	1	- Mismatch

column multiplicity $\nu = 3$
number of row mismatches = 2

From the last two partitions, the partitions $X_2 \mid X_1X_0$ and $X_0 \mid X_2X_1$ have the same column multiplicity and the same number of mismatches. This is considered a *tie* and one of the partitions is selected the best. The BBPS process is finished at this point because the only partition that exists is the situation when all three variables are bound variables, which is a trivial decomposition.

To summarize, at each level of choosing the best partition, the variables in the best partition are then paired with the remaining individual variables. Figure 4.4 provides a graphical representation of the algorithm.

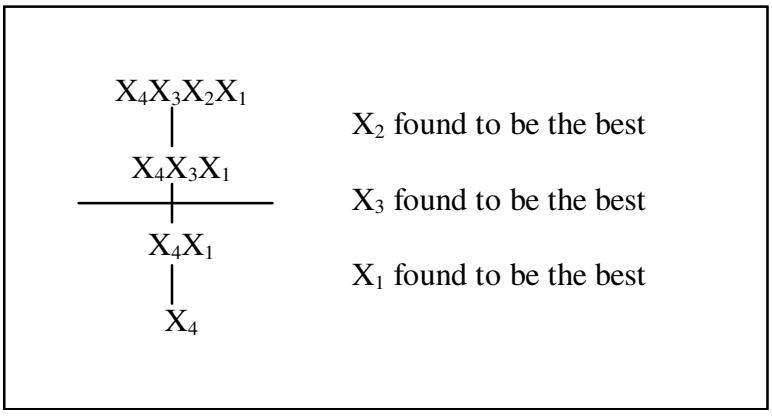


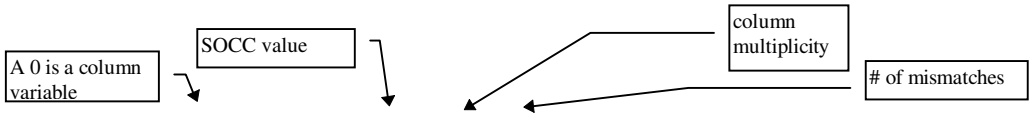
Figure 4.4: Variable selection

From Figure 4.4, there are four input variables to the function f. By considering each variable as an independent bound variable, the best input variable for the composite function is chosen. That leaves three more variables to examine. The process continues until there are no more variables left to consider. At each level a new variable is selected to be a bound variable: X_2 is chosen first, then X_3 and so on. This process results in the evaluation of 4+3+2+1=10 partitions, because there are four partitions to be evaluated with four input variables, three partitions to be evaluated with three variables, and so on. For an N input function, Euler's definition gives us $\frac{N*(N+1)}{2}-1=\frac{(N+2)(N-1)}{2}$ evaluated partitions. Since the number of partitions evaluated is polynomial, BBPS has the polynomial time complexity ($O(N^2)$). During the search, BBPS compares the current partition's SOCC to the best SOCC found, the one with the smallest SOCC becomes the current best. In Figure 4.4, the partition with the smallest SOCC is found at the location of the horizontal line. This is the point when all the bound variables chosen thus far will return the best SOCC. After this point any additional bound variables will make the SOCC grow in size. Because the value of

SOCC is not a monotonic increasing function during the search, the partition with the smallest SOCC will not be known until all $\frac{(N+2)(N-1)}{2}$ partitions have been evaluated.

4.1.1 BBPS Example

The following is an example of the BBPS algorithm finding the partition with the smallest SOCC of the eight variable function, ADD0 (described in Appendix A). The first seven partitions evaluated have only one bound variable. Partitions 11101111 and 11111110 have the fewest number of row mismatches. This is considered a *tie* and the partition 11101111 is selected as the best only because it was the first of the two partitions to be evaluated. Partitions with two bound variables are then evaluated. Because one of the bound variables must be the bound variable chosen during the first set of evaluations there are seven partitions to be evaluated. This process of finding the partition with the smallest column multiplicity and row mismatches is continued until all but one variable is a bound variable.




```
 0 partition:  01111111 SOCC 256   2  64  best part: 01111111 SOCC 256  2  64
 1 partition:  10111111 SOCC 256   2  32  best part: 10111111 SOCC 256  2  32
 2 partition:  11011111 SOCC 256   2  16  best part: 11011111 SOCC 256  2  16
 3 partition:  11101111 SOCC 256   2   8  best part: 11101111 SOCC 256  2   8
 4 partition:  11110111 SOCC 256   2  64  best part: 11101111 SOCC 256  2   8
 5 partition:  11111011 SOCC 256   2  32  best part: 11101111 SOCC 256  2   8
 6 partition:  11111101 SOCC 256   2  16  best part: 11101111 SOCC 256  2   8
 7 partition:  11111110 SOCC 256   2   8  best part: 11101111 SOCC 256  2   8
best ----------- partition 11101111 SOCC  256   2  8
 8 partition:  01101111 SOCC 256   4  36  best part: 11101111 SOCC 256  2   8
 9 partition:  10101111 SOCC 256   4  20  best part: 11101111 SOCC 256  2   8
10 partition: 11001111 SOCC 256   4  12  best part: 11101111 SOCC 256  2   8
11 partition: 11100111 SOCC 256   4  36  best part: 11101111 SOCC 256  2   8
12 partition: 11101011 SOCC 256   4  20  best part: 11101111 SOCC 256  2   8
13 partition: 11101101 SOCC 256   4  12  best part: 11101111 SOCC 256  2   8
14 partition: 11101110 SOCC 132   2   8  best part: 11101110 SOCC 132  2   8
best ----------- partition 11101110 SOCC  132   2  8
15 partition: 01101110 SOCC 144   4  20  best part: 11101110 SOCC 132  2   8
16 partition: 10101110 SOCC 144   4  12  best part: 11101110 SOCC 132  2   8
17 partition: 11001110 SOCC 144   3   8  best part: 11101110 SOCC 132  2   8
18 partition: 11100110 SOCC 144   4  20  best part: 11101110 SOCC 132  2   8
19 partition: 11101010 SOCC 144   4  12  best part: 11101110 SOCC 132  2   8
20 partition: 11101100 SOCC 144   3   8  best part: 11101110 SOCC 132  2   8
best ----------- partition 11001110 SOCC  144   3  8
21 partition: 01001110 SOCC 176   6  12  best part: 11101110 SOCC 132  2   8
22 partition: 10001110 SOCC 176   5   8  best part: 11101110 SOCC 132  2   8
23 partition: 11000110 SOCC 176   6  12  best part: 11101110 SOCC 132  2   8
24 partition: 11001010 SOCC 176   5   8  best part: 11101110 SOCC 132  2   8
25 partition: 11001100 SOCC  48   2   4  best part: 11001100 SOCC  48  2   4
best ----------- partition 11001100 SOCC   48   2  4
26 partition: 01001100 SOCC  96   4   6  best part: 11001100 SOCC  48  2   4
27 partition: 10001100 SOCC  96   3   4  best part: 11001100 SOCC  48  2   4
28 partition: 11000100 SOCC  96   4   6  best part: 11001100 SOCC  48  2   4
29 partition: 11001000 SOCC  96   3   4  best part: 11001100 SOCC  48  2   4
best ----------- partition 10001100 SOCC   96   3  4
30 partition: 00001100 SOCC 224   5   2  best part: 11001100 SOCC  48  2   4
31 partition: 10000100 SOCC 224   5   3  best part: 11001100 SOCC  48  2   4
32 partition: 10001000 SOCC  72   2   2  best part: 11001100 SOCC  48  2   4
best ----------- partition 10001000 SOCC  72   2  2
33 partition: 10000000 SOCC 256   3   1  best part: 11001100 SOCC  48  2   4
34 partition: 00001000 SOCC 256   3   1  best part: 11001100 SOCC  48  2   4
best ----------- partition 00000100 SOCC  256   3  1
BEST : partition 11001100 SOCC   48   4   0
```

The BBPS algorithm returns with the partition 11001100 since it is the partition with the smallest SOCC. Thus, Φ has inputs X_5, X_4, X_1, X_0 and F has inputs X_7, X_6, X_3, X_2 and the output from Φ. To fully decompose the function, the BBPS algorithm must proceed to decompose both F and Φ individually.

In the case of vacuous variables, the code evaluates all possible partitions with one bound variable. If a vacuous variable exists, then the variable is removed and the function is converted to an N-1 variable function. This procedure is continued until all vacuous variables are removed. Then, the BBPS algorithm is continued.

4.1.2 BBPS as a Full Decomposer

The following is a pseudo code description of the BBPS algorithm when it is used as a full decomposer. The pseudo code starts by calling the procedure Main with an input function. Main is a recursive procedure that calls the procedure BBPS to find the partition with the smallest SOCC.

```
Main (function)
N is the number of inputs to function
If N is less than or equal to 3 then
        return DFC=2^N
Call BBPS
If SOCC of partition returned from BBPS >=2^N
        return DFC=2^N
From the partition returned from BBPS create Φ and F
For each output i of Φ call Main (Φ_i)
Call Main (F)

If DFC of Φ plus DFC of F is < 2^N
        return DFC of Φ plus DFC of F
Else return DFC=2^N

BBPS (function)
N is the number of inputs to function
Bound set is set to contain no variables
Free set is set to contain all the variables
While the number of variables in the bound set < N do
        Repeat
                Select variable from free set and place in bound set
                Determine column multiplicity, row mismatches and SOCC for
                        current partition
```

If current partition has smaller column multiplicity or has the same
column multiplicity and fewer row mismatches than the best
then best = current partition
Remove current variable from bound set and place in free set
Until all variables in free set have been selected
Make a new free set and bound set from the best partition found
Compare and save the smallest SOCC found
Return partition with the smallest SOCC

From the above pseudo code the worse case number of partitions that must be evaluated to find the full decomposition can be determined. The worst case occurs when Φ has two inputs and one output, implying that the function F has N-1 inputs. There is no further decomposition process for Φ, but F can then be decomposed again into this worst case. This process continues until there are only three variables left to be evaluated. Because the BBPS algorithm evaluates $\frac{N(N+1)}{2}-1=\frac{(N+2)(N-1)}{2}$ partitions to find the smallest SOCC partition, the worst case number of partitions to be evaluated is given by:

$$\sum_{K=3}^{N}\frac{(K+2)(K-1)}{2}=\sum_{K=3}^{N}\frac{K^2+K-2}{2}$$

Which can be reduced by using the following identities:

$$\sum_{K=1}^{N}K=\frac{N(N+1)}{2}$$

$$\sum_{K=1}^{N}K^2=\frac{N(N+1)(2N+1)}{6}$$

Thus, producing the exact solution:

$$\sum_{K=3}^{N}\frac{(K+2)(K-1)}{2}=\frac{1}{2}\left(\frac{N(N+1)}{2}-2-1+\frac{N(N+1)(2N+1)}{6}-4-1-2N+2+2\right)$$

$$=\frac{1}{2}\left(\frac{N^2+N}{2}+\frac{2N^3+3N^2+N}{6}-2N-4\right)$$

$$=\frac{(3N^2+3N)+(2N^3+3N^2+N)-12N-24}{12}$$

$$=\frac{N^3+3N^2-4N-12}{6}$$

Thus, BBPS has a third order polynomial ($O(N^3)$) worst case time complexity. In the case where the function does not decompose, the number of partitions evaluated is $\frac{(N+2)(N-1)}{2}$, which is $O(N^2)$.

4.2 Single Output Search

The BBPS algorithm finds the simple disjunctive decomposition with the smallest SOCC in most cases. The problem is that the simple disjunctive decomposition with the smallest SOCC does not always decompose into the minimum DFC of a function. The reason for implementing this search strategy was to detect any single output decompositions that might exist. Fundamentally, the smallest implementation possible for any N input function, without vacuous variables, is N-1, two input, one output blocks. The Single Output Search (SOS) algorithm finds *compatible* input variables that have a single output such that a minimum DFC is more likely to be found. Compatible variables in this context means that by using a simple Boolean logic gate on the variables, a single output will result. Another

advantage is if two variables are found to be *compatible*, then the resulting function size is reduced by a factor of two.

Finding compatible variables has not been implemented in Ashenhurst-Curtis decomposition. Roth and Karp developed their own system of trying to find compatible variables for finding simple two input Boolean gates imbedded within a function [30]. In evaluating if two variables are compatible, their algorithm would look for implementations of basic Boolean gates plus other implementations of the basic gates. For example, other variations of the basic function X_1 AND X_2, can be written as $X_1 \cap \overline{X}_2$, $\overline{X}_1 \cap X_2$, $\overline{X_1 \cap X_2}$, etc. Also, in looking for two compatible variables in an N input function their algorithm would search $\binom{N}{2} = \frac{N!}{(N-2)!2!}$ different partitions.

The SOS strategy is based on the methods that were discussed in Section 4.1. There, the main goal was to reduce a Decision Tree by folding. Here, the main objective is to find input variables that, when placed at the bottom of the tree, fold together to result in a new reduced tree. Also, in keeping with Ashenhurst-Curtis fundamental ideas, any number of input variables greater than one and less than N are evaluated to see if there is only one output. Example 4.3 illustrates the SOS strategy.

Example 4.3: Compatible variables

Consider a three input function $f(X_2, X_1, X_0)=(X_2 * X_1) + X_0$. Evaluation of the partitions where there is only one element in the bound set results in the decomposition charts shown in Table 4-6.

Table 4-6: Compatible variable partitions

X_2X_1 \ X_0	0	1
0	0	1
0	0	1
0	0	1
1	1	1

X_2X_0 \ X_1	0	0
1	1	1
0	0	1
1	1	1

X_1X_0 \ X_2	0	0
1	1	1
0	0	1
1	1	1

The decomposition charts of X_1 and X_2 are the same. Therefore, X_1 and X_2 are compatible variables and the partition shown in Table 4-7 is evaluated $X_0 \mid X_2X_1$.

Table 4-7: Partition of two compatible variables

X_0 \ X_2X_1	0	0	0	1
1	1	1	1	1

Column multiplicity = 2
Number of outputs = 1

This technique also works for identifying more than two compatible variables. This is demonstrated in Example 4.4.

Example 4.4: Multiple compatible variables

Consider a four input function $f(X_3, X_2, X_1, X_0)=(X_3 * X_2 * X_1) + X_0$. Evaluation of the partitions where there is only one element in the bound set yields the decomposition charts in Table 4-8.

Table 4-8: Compatible variable check

$X_3X_2X_1$ \ X_0	0	1
	0	1
	0	1
	0	1
	0	1
	0	1
	0	1
	0	1
	1	1

$X_3X_2X_0$ \ X_1	0	0
	1	1
	0	0
	1	1
	0	0
	1	1
	0	1
	1	1

$X_3X_1X_0$ \ X_2	0	0
	1	1
	0	0
	1	1
	0	0
	1	1
	0	1
	1	1

$X_2X_1X_0$ \ X_3	0	0
	1	1
	0	0
	1	1
	0	0
	1	1
	0	1
	1	1

Again the decomposition charts of X_1, X_2 and X_3 are the same. Thus, X_3, X_2 and X_1 are compatible and the following partition in Table 4-9 shows the three input, one output function Φ that results.

Table 4-9: Partition of three compatible variables

X_0 \ $X_3X_2X_1$	0	0	0	0	0	0	0	1
	1	1	1	1	1	1	1	1

Column multiplicity = 2
Number of outputs = 1

This process of finding equal decomposition charts works well as long as the ordering of the free set variables is the same. This was easy in the previous two examples because the compatible variable charts were side by side (i.e., the variables were adjacent). In the case where the compatible variables are not side by side, the two decomposition charts should be examined to determine if they contain the same number of like elements. These elements, or

row types, are the values that can exist in a row, namely, 00, 01, 10, or 11. This is illustrated in Example 4.5.

Example 4.5: Non-adjacent compatible variables

Consider a three input function $f(X_2, X_1, X_0)=(X_2 + X_0) * X_1$. Evaluation of the partitions where there is only one element in the bound set produces the decomposition charts of Table 4-10.

Table 4-10: Non-adjacent compatible variables

X_2X_1 \ X_0	0	0
0	0	1
0	0	0
1	1	1

X_2X_0 \ X_1	0	0
0	0	1
0	0	1
0	1	1

X_1X_0 \ X_2	0	0
0	0	0
0	0	1
1	1	1

To generate a *row type chart* the number of each of the four row elements is found for each of the possible bound variables. When the counting is done it is placed in the form of Table 4-11.

Table 4-11: Row type chart, three variables

	X_0	X_1	X_2
00s	2	1	2
01s	1	3	1
10s	0	0	0
11s	1	0	1

Because X_0 and X_2 have the same number of row types, the next test is to determine if the two variables are compatible. This is done by checking the decomposition, $X_1 | X_2X_0$ as shown in Table 4-12.

Table 4-12: Partition of compatible variables, using row type chart

	X_2X_0			
X_1	0	0	0	0
	0	1	1	1

Column multiplicity=2
Number of outputs = 1

Therefore, the SOS algorithm finds compatible variables by counting the number of row types that were found by evaluating N partitions for an N input function. A problem with this method develops if two columns have the same number of individual elements and the resulting combining partition does not reduce. Thus, it is possible to check many different partitions without finding a decomposition with a single output. In general, this problem occurs rarely and a decomposition with a single output is found.

The idea of this algorithm is to search for partitions with a column multiplicity of 2. Because of this fact, if two variables have the same number of 00s, and 11s, but the number of 01s of the first variable equals the number of 10s of the second variable, and the number of 10s of the first equals the number of 01s of the second, then the two variables could still be compatible. This is illustrated in Example 4.6.

Example 4.6: Counting row types

Given a five input function with minterms 20, 22, 23 the *row type chart* of Table 4-13 exists.

Table 4-13: Row type chart, five variables

	X_0	X_1	X_2	X_3	X_4
00s	14	14	13	13	13
01s	0	1	3	0	3
10s	1	0	0	3	0
11s	1	1	0	0	0

Partition $\nu(X_4X_3X_2 \mid X_1X_0) = 2 \Rightarrow \Phi$ outputs = 1.

Partition $\nu(X_1X_0 \mid X_4X_3X_2) = 2 \Rightarrow \Phi$ outputs = 1.

4.2.1 SOS as a Full Decomposer

The following is a pseudo code description of the SOS algorithm when it is used as a full decomposer. The pseudo code is similar to that of the BBPS in that it starts by calling the procedure Main with an input function which calls SOS to find a partition with a Φ with one output.

```
Main (function)
N is the number of inputs to function
If N is less than or equal to 3 then
        return DFC=2^N
Call SOS with function
If SOS returns a NULL
        return DFC=2^N

From the partition returned from SOS create Φ and F
Since only one output of Φ call Main (Φ)
```

Call Main (F)
If DFC of Φ plus DFC of F is < 2^N
    return DFC of Φ plus DFC of F
Else return DFC=2^N

SOS (*function*)
N is the number of inputs to *function*
Find all possible partitions with one bound variable, N partitions
Construct row type charts for each of these partitions

Evaluate all partitions with two bound variables that have compatible row type charts
Stop evaluation if a partition has a Φ with one output
    return partition
If evaluated all possible partitions with compatible row types and a partition with Φ has not been found with a single output then return a NULL.

As in the case of the BBPS algorithm a worse case analysis can be performed to find the number of partitions evaluated by SOS to find the full decomposition. As with the BBPS algorithm, the worst case occurs when Φ has two inputs and one output, implying that the function F has N-1 inputs. Since the number of partitions that could be evaluated in the worst case of SOS to find a SOCC is

$$\binom{N}{1}+\binom{N}{2}=N+\frac{N!}{(N-2)!2!}=N+\frac{N*(N-1)}{2}=\frac{N^2+N}{2}$$

then the following analysis can be done:

$$\begin{aligned}\sum_{K=3}^{N}\frac{N^2+N}{2}&=\frac{1}{2}\left(\frac{N(N+1)}{2}-2-1+\frac{N(N+1)(2N+1)}{6}-4-1\right)\\&=\frac{1}{2}\left(\frac{N^2+N}{2}+\frac{2N^3+3N^2+N}{6}-8\right)\\&=\frac{(3N^2+3N)+(2N^3+3N^2+N)-48}{12}\\&=\frac{N^3+3N^2+2N-24}{6}\end{aligned}$$

Thus, SOS has a third order polynomial ($O(N^3)$) worst case time complexity. In the case that SOS only has to evaluate one partition to find a partition with a single output then the number of partitions that are evaluated in the worst case is (N-3)*N, $O(N^2)$. During the first decomposition process if none of the row type charts are compatible then only N partitions are evaluated.

4.3 Combination of the Two Algorithms

The goal of combining the two algorithms is to produce a new algorithm that finds a decomposed representation of a function very fast. The last section shows the speed that SOS can find a partition with two inputs and one output. The only problem is that most functions will not decompose in this form, thus BBPS can be used to find partitions with more than two variables in the bound set. The combination of the two algorithms in this chapter is a recursive procedure that calls SOS. If SOS does not find a decomposition then the BBPS algorithm is used. The recursive process is stopped if the BBPS cannot find a decomposition or the function has been decomposed down to a function with three or less input variables. The pseudo code for this process is shown as follows:

Main (*function*)
If N is less than or equal to 3 then
    return DFC=2^N
Call SOS with *function*
If SOS returns NULL
    call BBPS with *function*

If the partition has a SOCC >= 2^N return DFC=2^N
Calculate Φ and F from the partition
For each output i of Φ call Main (Φ_i)

Call Main (F)

If DFC of Φ plus DFC of F $< 2^N$
    return DFC of Φ plus DFC of F
Else return DFC=2^N

The worst case upper bound is found by adding the upper bounds of the BBPS and SOS algorithms, which gives an upper bound complexity of a third order polynomial ($O(N^3)$).

In summary, each of the algorithms presented in this chapter have the upper bound time complexity of a third order polynomial. This is better than any of the other algorithms that are presented in Chapter 3. The next issue is how well the algorithms perform compared to other algorithms and to the best possible complex disjunctive decompositions. This is presented in the next chapter.

Chapter 5

5. Results

The techniques presented in the previous chapter have been incorporated in a program named ASH (Ashenhurst Search Heuristics); a brief description is provided in Appendix C. ASH contains the BBPS and SOS algorithms. Although not used to present time results in this section, ASH has the ability to find complex non-disjoint decompositions. In the case of this thesis, ASH is only used to present complex disjoint decompositions using the algorithms BBPS, SOS, and BPPS+SOS and to search for all disjoint decompositions. The test functions that are used were developed by the Pattern Theory Group to test different applications of interest (see Appendix A for a brief description of the 53, 8-variable input functions and the 14, 12-variable input functions).

Verification of ASH was done by first checking simple disjoint decompositions to the results found by the Wright Laboratory program FLASH (Functional Synthesis and Test Bed). The verification of complex disjoint decomposition was done by confirming that ASH was returning the correct simple disjoint decompositions. Since the procedure to find complex disjoint decompositions is a recursive procedure that uses the simple disjoint decomposition, ASH was verified.

5.1 Smallest SOCC Results

To demonstrate the ability of BBPS to find a decomposition with the smallest SOCC, a comparison is made to the algorithms presented in Section 3.2. Figures 5.1 and 5.2 show the results of a comparison of BBPS with the IRC partitioning algorithm and two Genetic Algorithm searches that were described in Chapter 3. In determining the effectiveness of the algorithms, the cost for each function is 1.0 if the minimum partition has not been found, 0.0 if the minimum has been found. Average cost is found as follows:

$$\text{Average Cost} = \frac{\text{Sum of the costs for each function}}{\text{number of functions}}$$

Figure 5.1 and Figure 5.2 show the speed (in terms of the number of partitions evaluated) with which the algorithm finds the smallest SOCC for all the 8-variable and all the 12-variable test functions, respectively.

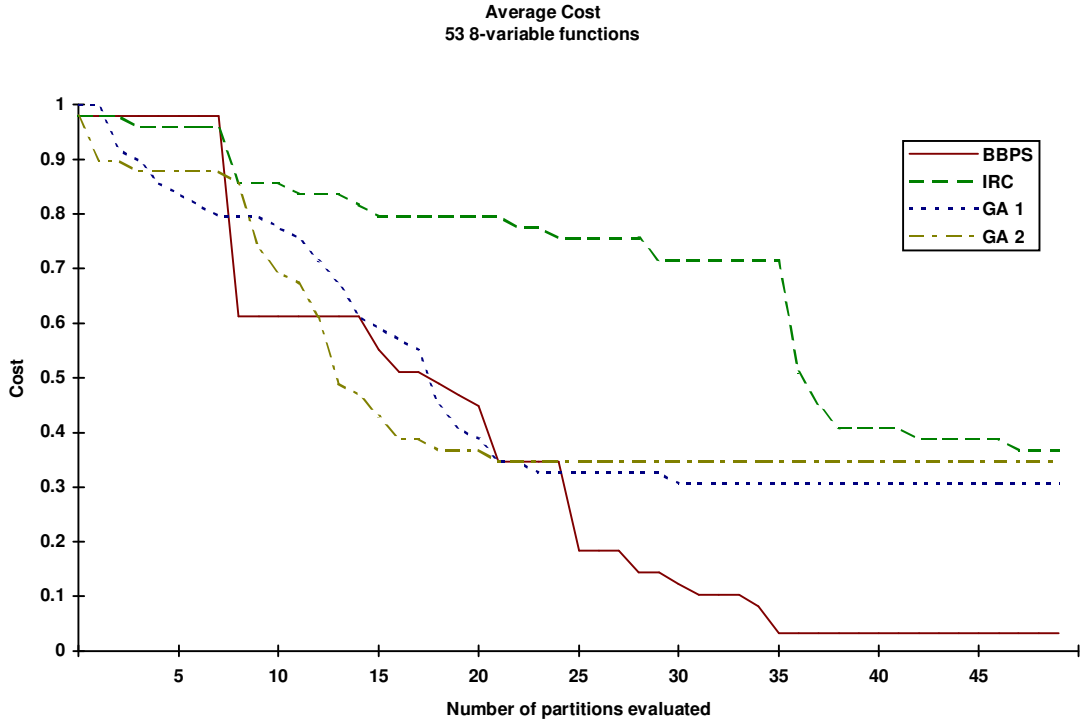


Figure 5.1: 8-variable SOCC evaluation

For an 8-variable input function BBPS evaluates $\frac{8*(8+1)}{2}-1=35$ partitions. As shown in the figure, after evaluating the 35 partitions the BBPS algorithm had found the best simple disjoint decomposition for all but four functions.

For a 12-variable input function BBPS evaluates $\frac{12*(12+1)}{2}-1=77$ partitions. In Figure 5.2, the 12-variable input functions are evaluated by the search algorithms. BBPS finds the smallest SOCC of all the functions before it has evaluated all 77 partitions. Again,

since the algorithm does not guarantee where it will find the partition with the smallest SOCC, all 77 partitions are evaluated.

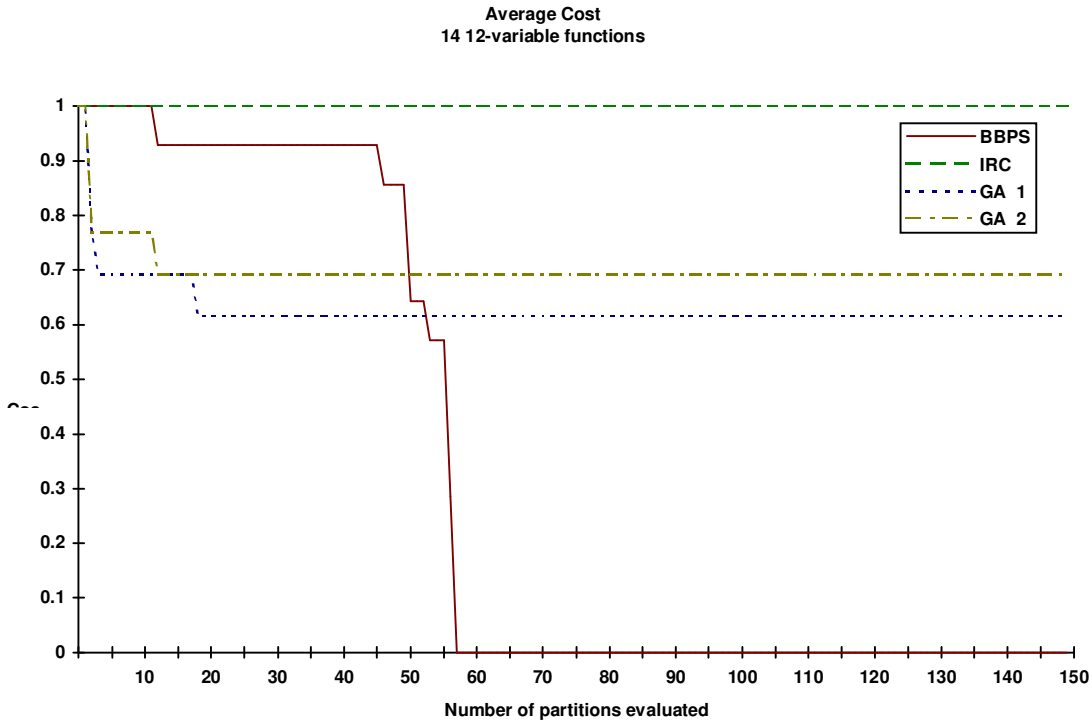


Figure 5.2: 12-variable SOCC evaluation

The above figures show how well BBPS finds the smallest SOCC partition for a function, as compared to other partitioning search techniques. In both cases, the BBPS algorithm finds a partition with the smallest SOCC, on average, with fewer partition evaluations than the other three functions. The reason that the other three algorithms do not find better results by the end of the table is that they cannot find the smallest SOCC partition of several functions. In several tests the algorithms would search all 2^N partitions in a level before finding the partition with the smallest SOCC.

The reason the BBPS algorithm does not find a partition with the smallest SOCC in every case is because it is a heuristic search that is based on probabilities. The algorithm assumes that the smallest SOCC will occur if a DT has the most matching (folding) elements. It is possible that the smallest SOCC will be found by folding branches of the tree, not by folding individual elements. Thus, there is always the possibility that the BBPS algorithm will not find a partition with the smallest SOCC.

5.2 Analysis of BBPS, SOS, BBPS+SOS

This section shows the ability of each of the algorithms to fully decompose a function. Table 5-1 shows the resulting DFC found when running ASH on the 8-variable test functions. The table shows the results of running Best Bound Partition Search (BBPS) by itself, Single Output Search (SOS) by itself, and a combination of the two (BBPS+SOS). The last two columns show the DFC that is found by looking through all possible decompositions in either disjoint mode, found by ASH, or in non-disjoint mode, found by the Wright Laboratory program FLASH. The reason for including the non-disjoint decompositions is to show that even though most functions contain non-disjoint decompositions, the disjoint decompositions will still find good representations for the functions. There are three cases where ASH was unable to find the best disjoint decomposition of a function due to the large amount of memory that is required to store multiple decompositions before comparing results and deciding which decomposition is the smallest. For these cases, indicated by “NF” in Table 5-1, a *memory fault* occurs.

Table 5-1: DFC results for BBPS, SOS, BBPS+SOS

Function	BBPS	SOS	BBPS+SOS	ASH Best Disjoint	FLASH Best Non-Disjoint
add0	28	28	28	28	28
add2	20	20	20	20	20
add4	4	4	4	4	4
and_or_chain8	28	28	28	28	28
ch15f0	64	64	64	64	60
ch176f0	8	8	8	8	8
ch177f0	4	4	4	4	4
ch22f0	20	20	20	20	20
ch30f0	44	64	44	36	28
ch47f0	64	64	64	64	60
ch52f4	224	256	224	180	180
ch70f3	60	132	40	40	40
ch74f1	128	256	128	76	72
ch83f2	136	256	136	120	116
ch8f0	40	64	36	NF	32
contains_4_ones	112	256	112	NF	76
greater_than	28	28	28	28	28
interval1	148	256	148	128	128
interval2	132	256	120	92	92
kdd1	8	8	8	8	8
kdd10	20	20	20	20	20
kdd2	16	16	16	16	16
kdd3	12	12	12	12	12
kdd4	2	2	2	2	2
kdd5	56	132	50	38	32
kdd6	12	12	12	12	12
kdd7	28	28	28	28	28
kdd8	8	8	8	8	8
kdd9	20	20	20	20	20
majority_gate	112	256	112	NF	64
monkish1	12	12	12	12	12
monkish2	60	256	60	60	60
monkish3	20	20	20	20	20
mux8	34	64	34	24	24
or_and_chain8	28	28	28	28	28
pal	28	28	28	28	28
pal_dbl_output	128	128	128	128	116

Continuation of Table 5-1: DFC results for BBPS, SOS, BBPS+SOS

Function	BBPS	SOS	BBPS+SOS	ASH Best Disjoint	FLASH Best Non-Disjoint
pal_output	256	256	256	256	256
parity	28	28	28	28	28
primes8	256	256	256	256	256
rnd1	256	256	256	256	256
rnd2	256	256	256	256	256
rnd3	256	256	256	256	256
rnd_m1	28	28	28	28	28
rnd_m10	148	256	144	88	80
rnd_m25	204	256	204	172	172
rnd_m5	68	256	64	64	64
rnd_m50	224	256	224	224	224
rndvv36	64	64	64	64	64
substr1	84	256	84	72	72
substr2	92	256	92	68	60
subtraction1	88	256	88	66	64
subtraction3	4	4	4	4	4

From Table 5-1, we observe that the combination of the BBPS and the SOS algorithms always produces a smaller DFC than either BBPS or SOS acting alone. Further, the SOS algorithm does not perform well by itself when the function is hard to decompose (i.e., has a large DFC). Out of the 53 test functions, the BBPS+SOS algorithm finds the minimum disjoint DFC 37 times. Thus, approximately 70% of the minimum disjoint DFCs are found, while in the cases that it did not find the minimum disjoint DFC, the DFC that was found was fairly close. One reason why the BBPS+SOS algorithm does not always find the minimum disjoint decomposition is that the partition with the smallest SOCC does not always result in the smallest DFC. Also, several partitions may have the smallest SOCC, while only one of these partitions may lead to the smallest DFC.

When running the program to search all possible partitions, the non-disjoint runs find DFCs that are less than or equal to those found by the disjoint runs. The difference between the non-disjoint and the disjoint DFCs is very small. Thus, disjoint decomposition techniques provide results similar to those of non-disjoint techniques. In fact, there are only 12 times that the non-disjoint DFC is smaller than the disjoint DFC and the largest DFC difference is 12.

5.3 Comparisons of BBPS+SOS and SIS

This section shows how the BBPS+SOS algorithm compares to the industry logic synthesis tool SIS (a system for sequential circuit synthesis). SIS was briefly described in Subsection 3.2.3. SIS can be used as a multi-level decomposer using different scripts, or methods to decompose a function. The different scripts that are included in the SIS package are Algebraic, Boolean, Delay, MCNC, MSU, OCT, and Rugged. While running the scripts on the test functions, the MSU and OCT scripts had the same results, so the OCT column was deleted from the tables. Tables 5.2 and 5.3 show the comparison of the BBPS+SOS to the SIS program. In these tables, whenever one of the scripts has a better DFC than BBPS+SOS, the value is in boldface.

Table 5-2: Comparison of SIS to BBPS+SOS on 8-variable test functions

Function	Algebraic	Boolean	Delay	MCNC	MSU	Rugged	BBPS+ SOS
add0	180	256	70	56	60	84	28
add2	44	24	78	44	40	32	20
add4	4	4	20	4	4	4	4
and_or_chain8	28	256	36	28	28	256	28
ch15f0	104	256	192	124	156	132	64
ch176f0	8	8	20	8	8	8	8
ch177f0	4	4	20	4	4	4	4
ch22f0	20	64	64	52	132	64	20
ch30f0	88	144	106	84	128	256	44
ch47f0	88	208	164	116	144	132	64
ch52f4	256	256	256	**212**	236	256	224
ch70f3	144	256	78	68	68	256	40
ch74f1	256	256	162	128	168	256	128
ch83f2	256	256	256	220	256	256	136
ch8f0	84	64	78	64	40	64	36
contains_4_ones	256	256	256	212	244	256	112
greater_than	180	256	64	64	64	84	28
interval1	256	256	256	212	256	256	148
interval2	256	256	256	188	232	256	120
kdd1	8	8	8	8	8	8	8
kdd10	24	24	48	20	20	20	20
kdd2	20	32	26	16	20	32	16
kdd3	12	16	12	12	12	16	12
kdd4	2	2	2	2	2	2	2
kdd5	56	256	76	**48**	**48**	256	50
kdd6	16	16	16	16	16	16	12
kdd7	256	256	36	36	256	256	28
kdd8	8	8	22	8	8	8	8
kdd9	20	24	48	32	20	24	20
majority_gate	256	256	256	188	164	256	112
monkish1	16	16	28	24	24	16	12
monkish2	100	256	164	160	180	256	60
monkish3	68	32	36	28	28	32	20
mux8	**24**	64	54	48	48	64	34
or_and_chain8	28	72	34	28	28	256	28
pal	28	48	92	32	32	72	28
pal_dbl_output	256	256	256	256	256	256	128
pal_output	256	256	256	256	256	256	256

Continuation of Table 5-2: Comparison of SIS to BBPS+SOS on 8-variable test functions

Function	Algebraic	Boolean	Delay	MCNC	MSU	Rugged	BBPS+SOS
parity	28	28	142	28	28	28	28
primes8	256	256	256	256	256	256	256
rnd1	256	256	256	256	256	256	256
rnd2	256	256	256	256	256	256	256
rnd3	256	256	256	256	256	256	256
rnd_m1	256	256	44	36	36	256	28
rnd_m10	256	256	208	200	256	256	144
rnd_m25	256	256	256	256	256	256	204
rnd_m5	256	256	228	120	148	256	64
rnd_m50	256	256	256	256	256	256	224
rndvv36	256	256	244	168	236	256	64
substr1	256	256	128	**72**	**72**	256	84
substr2	256	256	90	**72**	**72**	256	92
subtraction1	256	256	256	148	140	256	88
subtraction3	4	4	20	4	4	4	4

Table 5-2 shows that out of the 53 test functions, SIS found a smaller DFC on five functions while BBPS+SOS found a smaller DFC on 25 functions. This also shows that the MCNC script did beat BBPS+SOS four times, while BBPS beats MCNC on 30 functions. None of the SIS scripts individually performed better than all of the rest. This is the reason for comparing all the scripts to BBPS+SOS.

Table 5-3 compares SIS to BBPS+SOS on the 12-variable benchmark functions. In the cases where SIS was unable to complete the decomposition process in a reasonable amount of time, the value in the table is marked ‘NF’ since a decomposition was not found.

Table 5-3: Comparison of SIS to BBPS+SOS on 12-variable test functions

	Algebraic	Boolean	Delay	MCNC	MSU	Rugged	BBPS +SOS
add0	220	296	296	84	100	100	44
add2	84	40	40	92	88	288	36
add4	44	24	24	44	40	32	20
contains_4_ones	NF	NF	1024	452	448	1024	304
greater_than	220	296	296	104	104	100	44
interval1	1024	1024	1024	**104**	**104**	1024	116
interval2	1024	1024	1024	**296**	376	1024	308
majority_gate	NF	NF	1024	464	716	1024	264
pal	48	96	96	48	48	272	44
parity	NF	NF	NF	NF	NF	NF	44
sim12	NF	NF	192	56	56	192	44
substr1	736	1024	1024	**152**	**240**	1024	254
subtraction1	1024	1024	1024	292	292	1024	188
subtraction3	1024	1024	1024	248	252	1024	128

Table 5-3 shows that out of the 14 12-variable test functions, SIS found a smaller DFC on three functions while BBPS+SOS found a smaller DFC on 11 functions. From both of these tables it is obvious that the BBPS+SOS algorithm outperforms the SIS scripts. Timing becomes the next concern since all the SIS scripts must be run to find a reduced circuit implementation.

Table 5-4 is a summary of the results that have been presented so far in this chapter. The table illustrates the number of functions, out of the 8-variable set, that the minimum DFC was found and how well the three new algorithms compared to SIS. Notice that all three of the new algorithms have approximately the same number of functions that do not find the minimum DFC. In these cases the BBPS+SOS algorithm did not find the minimum DFC, but it found the same or a smaller DFC than the other two algorithms.

Table 5-4: Summary of minimum DFC values found with each algorithm

	BBPS	SOS	BBPS+SOS	SIS
best decomposition not found	18	20	16	28
SIS found better solution	5	8	5	

As shown in the table, the BBPS+SOS algorithm had 16 functions out of the 53 that it did not find the minimum DFC. In comparison, SIS did not find the minimum DFC on 28 functions and only beat BBPS+SOS five times. Overall, the BBPS+SOS algorithm performs the same or better than BBPS, SOS, and SIS when evaluating the programs for finding the smallest representation. The next section continues this analysis with a discussion of the execution times of the algorithms.

5.4 Execution Times

This section is used to show the timing requirements of BBPS+SOS and SIS. Table 5-5 shows the average time, in seconds, it takes to minimize a function using each of the minimization techniques. Because some of the 12-input functions could not be minimized in under an hour using the SIS tools, the program was stopped and times were not found. The *timex* command in UNIX was used as the method for determining the CPU execution times in this section. The times that are reported are the amount of CPU time spent executing the program, it is not the real time of running the program.

Table 5-5: Comparison of SIS and BBPS+SOS execution times (seconds)

	Algebraic	Boolean	Delay	MCNC	MSU	Rugged	BBPS+SOS
8 var.	2.3	2.3	11.4	4.4	5.7	2.1	0.04
12 var.	NF	NF	NF	NF	NF	NF	0.58

Notice that the average length of time to find a full decomposition by the BBPS+SOS algorithm is a magnitude larger for the 12-variable functions compared to the 8-variable functions. Because of the problems that were encountered when trying to evaluate the 12-variable times with SIS, Table 5-6 is used to compare the timing of SIS to BBPS+SOS, and to show the exponential time increase for each addition of two variables. The ADD0 function that is discussed in Appendix A has been implemented with different numbers of inputs, from six to 20, to show the exponential time increase. The data provided in Table 5-6 is represented in a logarithmic plot of time versus the number of inputs in Figure 5.3.

Table 5-6: ADD0 times (seconds)

ADD0	BBPS+SOS	SIS
6-input function	0.02	1.68
8-input function	0.02	1.96
10-input function	0.04	2.52
12-input function	0.15	4.60
14-input function	0.65	14.68
16-input function	2.78	103.34
18-input function	13.2	>3600
20-input function	63.2	>3600

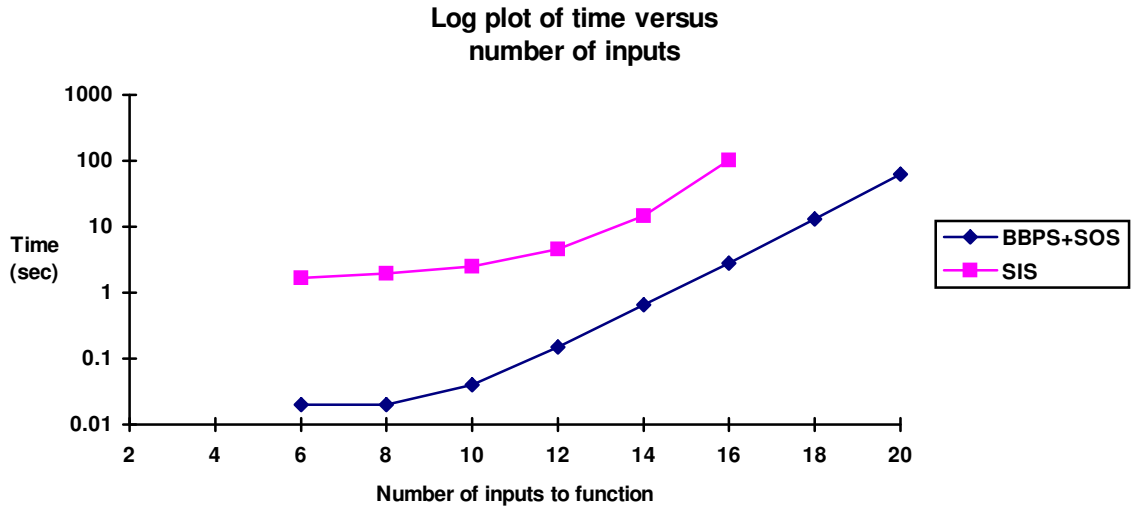


Figure 5.3: Time versus number of inputs

The linear logarithmic plot indicates an exponential time complexity. It has already been stated that the complexity of the BBPS algorithm is a third degree polynomial when selecting partitions. The problem is that evaluating a partition is an exponential time complexity.

The SIS run was completed with the script MCNC, the reason for this was because this script had the smallest DFC for the ADD0 functions in the previous runs. Overall, the BBPS+SOS algorithm is much faster than the SIS program at finding a minimized logic realization. Figures 5.4 and 5.5 are plots of the CPU time that BBPS+SOS took to decompose a function versus DFC. The reason for the zero times in the 8-variable chart is that the timing function only reports on values larger than one hundredth of a second. Thus, there are a few cases where the time to decompose a function is less than one hundredth of a second.

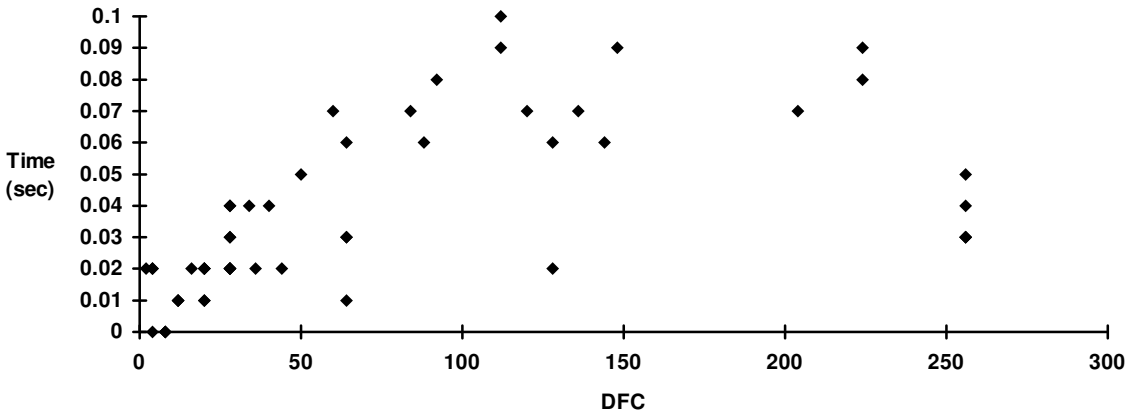


Figure 5.4: 8-variable time vs. DFC plot

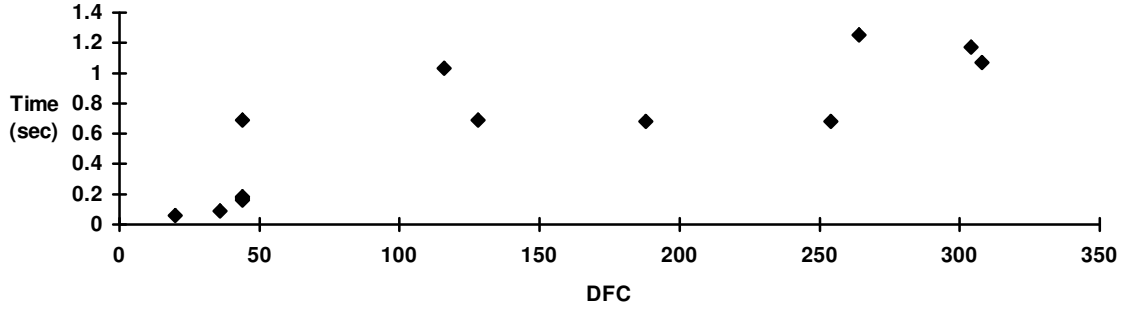


Figure 5.5: 12-variable time vs. DFC plot

The two figures show that for small functional DFCs the execution time is very small. However, notice that for functions with larger DFCs there is very little correlation between decomposition time and DFC. The reason that the times are small for functions with a small DFC is that the BBPS+SOS algorithm finds vacuous variables and removes them from the function. After vacuous variables are removed the search space is greatly reduced, thus the times reported are much smaller.

The results in this section show how well the BBPS+SOS algorithm compares to the SIS scripts. The BBPS+SOS algorithm, overall, beats the SIS scripts both in the amount of time to find a minimized function and in the size of the minimized circuits. It also compares very well to the actual minimum disjoint decomposition DFC of a function.

Chapter 6

6. Conclusions

This thesis demonstrates that a heuristic search algorithm can be applied to an NP-complete problem of logic synthesis. The results demonstrate that the BBPS algorithm is able to reduce the partition search complexity to a polynomial time complexity on the pattern theory benchmarks. Even though the algorithm may not find the minimum solution every time, as in the case of functions with high DFCs, we feel that this is a good start for actually understanding this area of functional decomposition. Ashenhurst-Curtis decomposition is a good method for minimizing timing, minimizing delay, and creating testable circuits. With more research into this area and other NP complete problems in decomposition, Ashenhurst-Curtis decomposition may be the best way of dealing with problems in machine learning and logic synthesis.

6.1 Comments on Results

Since the completion of [39], which first described the BBPS algorithm, the Pattern Theory Group has conducted their own set of experiments [40]. The experiments validated the results that were obtained for completely specified functions and compared them to running the BBPS algorithm on incompletely specified functions. The conclusions that they arrived at are the same as the ones presented here, including the fact that the partition with the smallest SOCC is not an indicator for finding the smallest DFC. In addition, they ran an experiment that used SOGCC as a figure of merit, and their results were quite exceptional.

Because of the way ASH evaluates partitions and the C++ classes included in ASH, I have not been able to conduct this test on my own, but I think that it is an area that needs further work. The only problem is that evaluating SOGCC partitions increases the number of partitions that are evaluated to find the simple disjoint decomposition that is then used to further decompose a function.

The Single Output Search algorithm works well when the function decomposes easily (i.e., has a small DFC), but it does not work well when there are no compatible variables. Future work in this area could be used to find compatible variables using non-disjoint simple decompositions. In fact, the research described in Subsection 3.1.3, shows complex disjunctive decompostions are actually a form of non-disjoint simple decompositions. Combining the two ideas may produce a method that finds minimum DFC much quicker in the case of functions that are hard to decompose.

The combination of the two algorithms performed better than executing the two algorithms alone. This demonstrates the effectiveness of looking for single-output decompositions while using BBPS to find decompositions with the smallest SOCC. This illustrates the methods that are described in Section 2.4.

6.2 Future Work

The goal of this research was to find algorithms that could reduce the complexity of the partition selection problem. Because all the work so far has been conducted with completely specified functions (except for the findings in [40]) it would be interesting to see how well the two algorithms perform on incompletely specified functions. This type of

problem is more complex than the issues in completely specified functions because the number of *don't cares* in a function's output has a dramatic impact on a function's decomposability. Another area of research is the selection of an indicator for finding a minimum DFC other than the partition with the smallest SOCC. The findings of the Pattern Theory Group tend to show the use of SOGCC, or even possibly Sum of Great Grand Child Cardinality, may be a better figure of merit. The problem with using levels of decompositions for a figure of merit is that the time to find a solution is exponential with respect to each new level, but on the other hand it could find the minimum DFC more readily. The results in Chapter 5 do show that the BBPS+SOS algorithm is not finding the minimum DFC for 16 functions. It is possible that the minimum DFC for these functions could be found by using SOGCC as the figure of merit, but again this means evaluating more partitions to obtain this result.

Even though the partition selection problem is distinct from the other NP-complete problems in functional decomposition they are still closely linked. Evaluation of a partition leads to a partition being selected which leads to column labeling and determining the two newly formed functions. In the future it may be possible to determine how all these pieces fit together which will resolve the NP-complete issues that are involved in functional decomposition. This could be accomplished by incorporating results from the Pattern Theory Group, Pedram, Steinbach, Luba, Sasao, Biblio, and Perkowski.

Appendix A

Description of Test Functions

Wright Lab
Pattern Theory Group Benchmark, 8-variable input function set:

This set of functions is intended to be representative of a wide variety of functions for testing machine learning systems. All functions are of the form: f:{0,1}^8 -> {0,1}

RANDOM
There are three randomly generated functions, from FLASH with seeds 1,2, and 3.
rnd1, rnd2, and rnd3.

RANDOM MINORITY ELEMENTS
There are five functions generated with a fixed number of minority elements placed at random. The seed for all was 1. Generated with FLASH.
rnd_m1, rnd_m5. rnd_m10, rnd_m25, rnd_m50.

RANDOM with vacuous variables
rndvv36: copied from Taylor set, was randomly generated except for x3 and x6 being vacuous.

BOOLEAN EXPRESSION
KDD 1, (x1 x3) + x2'
KDD 2, (x1 x2' x3)(x4 + x6')
KDD 3, NOT (x1 OR x2) + (x1' x4 x6)
KDD 4, x4'
KDD 5, (x1 x2 x4')+(x3 x5' x7 x8)+(x1 x2 x5 x6 x8)+(x3' x5')
KDD 6, x2 + x4 + x6 + x8
KDD 7, (x1 x2) + (x3 x4) + (x5 x6) + (x7 x8)
KDD 8, (x1 x2') XOR (x1 x5)
KDD 9, (x2 XOR x4)(x1' XOR (x5 x7 x8))
KDD 10, (x1 => x4) XOR (NOT (x7 x8))(x2 + x3)

MULTIPLEXERS

mux6, this is a 2-address bit, 4-data bit multiplexer with two vacuous variables (x0 and x1) to make 8-inputs. Generated mux6 with FLASH and then edited to make mux8.

and_or_chain8 = AND(OR(AND(OR(AND(OR(AND(x1,x2),x3),x4),x5),x6),x7),x8)

or_and_chain8 = OR(AND(OR(AND(OR(AND(OR(x1, x2), x3), x4),x5),x6),x7),x8)

MONKish Problems:

These are eight binary variable "approximations" to the Monk's problems

// x1: head shape (rnd, octagonal)
// x2: body shape (rnd, octagonal)
// x3: smiling (yes, no)
// x4,x5: holding (sword, balloon, flag, whistle)
// x6,x7: jacket color (red, yellow, green, blue)
// x8: has tie (yes, no)

monkish1: head shape equals body shape or jacket is red.

monkish2: exactly two of six attributes have first value.

monkish3: (jacket green and has sword) or (jacket not blue and body not octagonal)

STRING FUNCTIONS

pal, palindrome acceptor

pal_output, palindrome output, randomly generated 128 bits then mirror imaged them to create the outputs of an 8-variable function.

pal_dbl_output, palindromed output, 64 bits and flipped them twice.

interval1, interval acceptors, accepts strings with thre or fewer intervals (i.e. substrings of all zeros or all ones).

interval2, interval acceptors, accepts strings with four or fewer intervals two sub-string detectors.

substr1 accepts strings with the sub-string "101" substr2 accepts strings with the sub-string "1100"

IMAGES

chXfY means character X from font Y of the Borland font set. All were generated with the Pascal program charfn.exe.

ch8f0 - kind of a flat plus sign
ch15f0 - an Aztex looking design
ch22f0 - horizontal bar
ch30f0 - solid isosceles triangle
ch47f0 - slash
ch176f0 - every other column of a checker board
ch177f0 - checker board
ch74f1 - triplex J
ch83f2 - small S (thin strokes)
ch70f3 - sans serif F
ch52f4 - gothic 4

SYMMETRIC FUNCTIONS

parity.
contains_4_ones, (f(x)=1 iff the str x has 4 ones)
majority_gate, f(x)=1 iff x has more 1's than 0's

OTHER ACCEPTORS

primes8, output is 1 iff the input string, as a decimal number, is prime. 0 and 1 are not considered prime.

NUMERICAL FUNCTIONS

addition - outputs bits of a 4-bit adder, 0 is the most significant bit,

add0
add2
add4

greater_than: f(x1,x2)=1 iff x1 > x2

subtraction - output bits 1 and 3 of the absolute value of a 4 bit difference. 0 is most significant bit.

subtraction1
subtraction2

GEOMETRIC FUNCTIONS

Generated using nearest neighbor (Hamming distance):
- note that majority gate is effectively a nearest neighbor type function with samples:
00000000 0
11111111 1

- nnr1: kind of a four corners partitioning of the space.
  based on samples:
  00000000 0
  00001111 1
  11110000 1
  11111111 0
- nnr2: has two randomly selected points as templates seed 1.
  based on samples:
  00111010 0
  11011110 1
- nnr3: has eight randomly selected points as templates seed 3.
  based on samples:
  00010011 1
  00111010 1
  01000101 0
  01011001 0
  01110010 1

10001101	0
11000111	1
11011110	0

Appendix B

Partitions that Result in Minimum DFC

This appendix shows the partitions with the smallest SOCC and the partitions that result in the smallest DFC. The listings with the '**' are the partitions with the smallest DFC, while the listings without the '**' are the partitions with the smallest SOCC.

Decomposing: add0.bin
10001000 => SOCC 132 DFC 28 # Outputs 1 **
11001100 => SOCC 48 DFC 28 # Outputs 1 **
11101110 => SOCC 72 DFC 28 # Outputs 1 **

Decomposing: add2.bin
VACUOUS 128
0000001 => SOCC 64 DFC 20 # Outputs 1 **
0010010 => SOCC 68 DFC 20 # Outputs 1 **
1001000 => SOCC 68 DFC 20 # Outputs 1 **
1011010 => SOCC 64 DFC 20 # Outputs 2 **
1011011 => SOCC 80 DFC 20 # Outputs 2 **
1101100 => SOCC 32 DFC 20 # Outputs 1 **
1101101 => SOCC 40 DFC 20 # Outputs 1 **
1101111 => SOCC 68 DFC 20 # Outputs 1 **
1111101 => SOCC 68 DFC 20 # Outputs 1 **

Decomposing: add4.bin
VACUOUS 128
VACUOUS 64
VACUOUS 32
00001 => SOCC 16 DFC 4 # Outputs 1 **
00010 => SOCC 16 DFC 4 # Outputs 1 **
01000 => SOCC 16 DFC 4 # Outputs 1 **
01011 => SOCC 4 DFC 4 # Outputs 1 **

Decomposing: ch15f0.bin
VACUOUS 128
VACUOUS 64
function does not decompose past vacuous vars

Decomposing: ch176f0.bin
VACUOUS 128
VACUOUS 64
VACUOUS 32
00001 => SOCC 16 DFC 8 # Outputs 1 **
01000 => SOCC 16 DFC 8 # Outputs 1 **
01001 => SOCC 8 DFC 8 # Outputs 1 **
10010 => SOCC 20 DFC 8 # Outputs 1 **
11011 => SOCC 20 DFC 8 # Outputs 1 **

Decomposing: ch177f0.bin
VACUOUS 128
VACUOUS 64
VACUOUS 32
00001 => SOCC 16 DFC 4 # Outputs 1 **
00010 => SOCC 16 DFC 4 # Outputs 1 **
01000 => SOCC 16 DFC 4 # Outputs 1 **
01011 => SOCC 4 DFC 4 # Outputs 1 **

Decomposing: ch22f0.bin
VACUOUS 128
VACUOUS 64
000111 => SOCC 24 DFC 20 # Outputs 1 **
111000 => SOCC 24 DFC 20 # Outputs 1 **

Decomposing: ch30f0.bin
VACUOUS 128
VACUOUS 64
011011 => SOCC 48 DFC 44 # Outputs 2
100101 => SOCC 48 DFC 36 # Outputs 2 **
100111 => SOCC 48 DFC 44 # Outputs 2

Decomposing: ch47f0.bin
VACUOUS 128
VACUOUS 64
function does not decompose past vacuous vars

Decomposing: ch52f4.bin
01111110 => SOCC 224 DFC 184 # Outputs 3
10111110 => SOCC 224 DFC 188 # Outputs 3
11001111 => SOCC 224 DFC 196 # Outputs 3
11011110 => SOCC 224 DFC 204 # Outputs 3
11101110 => SOCC 224 DFC 180 # Outputs 3 **
11110110 => SOCC 224 DFC 204 # Outputs 3
11111010 => SOCC 224 DFC 204 # Outputs 3
11111100 => SOCC 224 DFC 184 # Outputs 3

Decomposing: ch70f3.bin
00001100 => SOCC 132 DFC 40 # Outputs 1 **
00001111 => SOCC 96 DFC 56 # Outputs 2
00101101 => SOCC 96 DFC 44 # Outputs 2
00111110 => SOCC 96 DFC 56 # Outputs 2
01110001 => SOCC 96 DFC 44 # Outputs 2
10001101 => SOCC 96 DFC 44 # Outputs 2
10110001 => SOCC 96 DFC 52 # Outputs 2
11110000 => SOCC 96 DFC 40 # Outputs 2 **
11110001 => SOCC 96 DFC

44 # Outputs 2
11110010 => SOCC 96 DFC 52 # Outputs 2
11111110 => SOCC 132 DFC 40 # Outputs 1 **

Decomposing: ch74f1.bin
11101110 => SOCC 144 DFC 76 # Outputs 2 **
11111110 => SOCC 132 DFC 88 # Outputs 1

Decomposing: ch83f2.bin
00001110 => SOCC 144 DFC 144 # Outputs 2
11110001 => SOCC 160 DFC 120 # Outputs 3 **
11111001 => SOCC 144 DFC 144 # Outputs 2

Decomposing: greater_than.bin
10001000 => SOCC 132 DFC 28 # Outputs 1 **
11001100 => SOCC 48 DFC 28 # Outputs 1 **
11101110 => SOCC 72 DFC 28 # Outputs 1 **

Decomposing: interval1.bin
00001111 => SOCC 176 DFC 128 # Outputs 3 **
00011110 => SOCC 176 DFC 128 # Outputs 3 **
00011111 => SOCC 160 DFC 148 # Outputs 3
00111100 => SOCC 176 DFC 128 # Outputs 3 **
00111110 => SOCC 160 DFC 148 # Outputs 3
01111000 => SOCC 176 DFC 128 # Outputs 3 **
01111100 => SOCC 160 DFC 148 # Outputs 3
10000111 => SOCC 176 DFC 128 # Outputs 3 **
10001111 => SOCC 160 DFC 148 # Outputs 3
11000111 => SOCC 160 DFC 160 # Outputs 3
11100011 => SOCC 160 DFC 148 # Outputs 3
11110000 => SOCC 176 DFC 128 # Outputs 3 **
11110001 => SOCC 160 DFC 148 # Outputs 3
11111000 => SOCC 160 DFC 148 # Outputs 3

Decomposing: interval2.bin
00001111 => SOCC 176 DFC 92 # Outputs 3 **
00011110 => SOCC 176 DFC 92 # Outputs 3 **
00011111 => SOCC 160 DFC 120 # Outputs 3
00111100 => SOCC 176 DFC 92 # Outputs 3 **
00111110 => SOCC 160 DFC 120 # Outputs 3
01111000 => SOCC 176 DFC 92 # Outputs 3 **
01111100 => SOCC 160 DFC 120 # Outputs 3
10001111 => SOCC 160 DFC 104 # Outputs 3
11000111 => SOCC 160 DFC 160 # Outputs 3
11100001 => SOCC 176 DFC 92 # Outputs 3 **
11100011 => SOCC 160 DFC 116 # Outputs 3
11110000 => SOCC 176 DFC 92 # Outputs 3 **
11110001 => SOCC 160 DFC 104 # Outputs 3
11111000 => SOCC 160 DFC 120 # Outputs 3

Decomposing: kdd2.bin
VACUOUS 128
VACUOUS 64
VACUOUS 32
00011 => SOCC 20 DFC 16 # Outputs 1 **
00101 => SOCC 20 DFC 16 # Outputs 1 **
00110 => SOCC 20 DFC 16 # Outputs 1 **
00111 => SOCC 16 DFC 16 # Outputs 1 **
11000 => SOCC 20 DFC 16 # Outputs 1 **
11001 => SOCC 16 DFC 16 # Outputs 1 **
11010 => SOCC 16 DFC 16 # Outputs 1 **
11011 => SOCC 20 DFC 16 # Outputs 1 **
11100 => SOCC 16 DFC 16 # Outputs 1 **
11101 => SOCC 20 DFC 16 # Outputs 1 **
11110 => SOCC 20 DFC 16 # Outputs 1 **

Decomposing: kdd4.bin
VACUOUS 128
VACUOUS 64
VACUOUS 32
VACUOUS 16
VACUOUS 8
VACUOUS 4
VACUOUS 2
function does not decompose past vacuous vars

Decomposing: kdd5.bin
00011011 => SOCC 96 DFC 52 # Outputs 2
00011111 => SOCC 96 DFC 64 # Outputs 2
00101011 => SOCC 96 DFC 56 # Outputs 2
00111011 => SOCC 96 DFC 60 # Outputs 2
01110100 => SOCC 96 DFC 56 # Outputs 2
10001011 => SOCC 96 DFC 60 # Outputs 2
10101011 => SOCC 96 DFC 60 # Outputs 2
10110100 => SOCC 96 DFC 56 # Outputs 2
11010100 => SOCC 96 DFC 46 # Outputs 2
11010101 => SOCC 96 DFC 80 # Outputs 2
11010110 => SOCC 96 DFC 80 # Outputs 2
11011100 => SOCC 96 DFC 68 # Outputs 2
11100100 => SOCC 96 DFC 38 # Outputs 2 **
11110000 => SOCC 96 DFC 38 # Outputs 2 **
11110100 => SOCC 96 DFC 60 # Outputs 2

Decomposing: kdd6.bin
VACUOUS 128
VACUOUS 64
VACUOUS 32
VACUOUS 16
0011 => SOCC 12 DFC 12 # Outputs 1 **
1100 => SOCC 12 DFC 12 # Outputs 1 **

Decomposing: kdd7.bin
00000011 => SOCC 132 DFC 28 # Outputs 1 **
00001100 => SOCC 132 DFC 28 # Outputs 1 **
00001111 => SOCC 48 DFC 28 # Outputs 1 **
00110000 => SOCC 132 DFC 28 # Outputs 1 **
00110011 => SOCC 48 DFC 28 # Outputs 1 **
00111100 => SOCC 48 DFC 28 # Outputs 1 **
00111111 => SOCC 72 DFC 28 # Outputs 1 **
11000000 => SOCC 132 DFC 28 # Outputs 1 **

```
11000011 => SOCC 48 DFC 28 # Outputs 1 **
11001100 => SOCC 48 DFC 28 # Outputs 1 **
11001111 => SOCC 72 DFC 28 # Outputs 1 **
11110000 => SOCC 48 DFC 28 # Outputs 1 **
11110011 => SOCC 72 DFC 28 # Outputs 1 **
11111100 => SOCC 72 DFC 28 # Outputs 1 **

Decomposing: kdd8.bin
VACUOUS 128
VACUOUS 64
VACUOUS 32
VACUOUS 16
0100 => SOCC 8 DFC 8 # Outputs 1 **

Decomposing: kdd9.bin
VACUOUS 128
VACUOUS 64
000100 => SOCC 32 DFC 8 # Outputs 1 **
001010 => SOCC 36 DFC 8 # Outputs 1 **
010000 => SOCC 32 DFC 8 # Outputs 1 **
010100 => SOCC 16 DFC 8 # Outputs 1 **
100000 => SOCC 32 DFC 8 # Outputs 1 **
100100 => SOCC 16 DFC 8 # Outputs 1 **
110000 => SOCC 16 DFC 8 # Outputs 1 **
110100 => SOCC 8 DFC 8 # Outputs 1 **
111110 => SOCC 36 DFC 8 # Outputs 1 **

Decomposing: monkish2.bin
00000111 => SOCC 144 DFC 56 # Outputs 2 **
00011000 => SOCC 132 DFC 56 # Outputs 1 **
00011011 => SOCC 96 DFC 60 # Outputs 2
00011101 => SOCC 96 DFC 60 # Outputs 2
00011110 => SOCC 96 DFC 60 # Outputs 2
00011111 => SOCC 96 DFC 60 # Outputs 2
01100000 => SOCC 132 DFC 56 # Outputs 1 **
01100011 => SOCC 96 DFC 60 # Outputs 2
01100101 => SOCC 96 DFC 60 # Outputs 2
01100110 => SOCC 96 DFC 60 # Outputs 2
01100111 => SOCC 96 DFC 60 # Outputs 2
01111000 => SOCC 96 DFC 72 # Outputs 2
01111001 => SOCC 96 DFC 64 # Outputs 2
01111010 => SOCC 96 DFC 64 # Outputs 2
01111100 => SOCC 96 DFC 64 # Outputs 2
10000011 => SOCC 144 DFC 56 # Outputs 2 **
10000101 => SOCC 144 DFC 56 # Outputs 2 **
10000110 => SOCC 144 DFC 56 # Outputs 2 **
10000111 => SOCC 96 DFC 56 # Outputs 2 **
10011001 => SOCC 96 DFC 60 # Outputs 2
10011010 => SOCC 96 DFC 60 # Outputs 2
10011011 => SOCC 96 DFC 60 # Outputs 2
10011100 => SOCC 96 DFC 60 # Outputs 2
10011101 => SOCC 96 DFC 60 # Outputs 2
10011110 => SOCC 96 DFC 60 # Outputs 2
11100001 => SOCC 96 DFC 60 # Outputs 2
11100010 => SOCC 96 DFC 60 # Outputs 2
11100011 => SOCC 96 DFC 60 # Outputs 2
11100100 => SOCC 96 DFC 60 # Outputs 2
11100101 => SOCC 96 DFC 60 # Outputs 2
11100110 => SOCC 96 DFC 60 # Outputs 2
11111000 => SOCC 96 DFC 64 # Outputs 2

Decomposing: mux8.bin
VACUOUS 128
0000001 => SOCC 64 DFC 32 # Outputs 1
0011101 => SOCC 64 DFC 26 # Outputs 2
0011110 => SOCC 64 DFC 34 # Outputs 2
0101011 => SOCC 64 DFC 26 # Outputs 2
0101110 => SOCC 64 DFC 48 # Outputs 2
0110011 => SOCC 64 DFC 42 # Outputs 2
0110101 => SOCC 64 DFC 42 # Outputs 2
0110110 => SOCC 64 DFC 48 # Outputs 2
1001011 => SOCC 64 DFC 42 # Outputs 2
1001101 => SOCC 64 DFC 42 # Outputs 2
1001110 => SOCC 64 DFC 36 # Outputs 2
1010011 => SOCC 64 DFC 26 # Outputs 2
1010110 => SOCC 64 DFC 34 # Outputs 2
1100101 => SOCC 64 DFC 26 # Outputs 2
1100110 => SOCC 64 DFC 34 # Outputs 2
1111010 => SOCC 80 DFC 24 # Outputs 2 **
1111100 => SOCC 80 DFC 24 # Outputs 2 **

Decomposing: pal_dbl_output.bin
00000001 => SOCC 128 DFC 128 # Outputs 1 **

Decomposing: pal_output.bin
function does not decompose

Decomposing: primes8.bin
00111111 => SOCC 224 DFC 212 # Outputs 3
01011111 => SOCC 224 DFC 192 # Outputs 3 **
01101111 => SOCC 224 DFC 224 # Outputs 3
01110111 => SOCC 224 DFC 192 # Outputs 3 **
01111011 => SOCC 224 DFC 224 # Outputs 3
01111101 => SOCC 224 DFC 200 # Outputs 3
01111110 => SOCC 224 DFC 204 # Outputs 3

Decomposing: rnd1.bin
function does not decompose

Decomposing: rnd2.bin
function does not decompose

Decomposing: rnd3.bin
function does not decompose

Decomposing: rnd_m10.bin
00111111 => SOCC 144 DFC 96 # Outputs 2
01110111 => SOCC 144 DFC 104 # Outputs 2
01111101 => SOCC 144 DFC 88 # Outputs 2 **
11101110 => SOCC 144 DFC 92 # Outputs 2
```

11110101 => SOCC 144 DFC 92 # Outputs 2

Decomposing: rnd_m25.bin
01101111 => SOCC 224 DFC 204 # Outputs 3
01110111 => SOCC 224 DFC 208 # Outputs 3
01111011 => SOCC 224 DFC 208 # Outputs 3
01111101 => SOCC 224 DFC 200 # Outputs 3
01111110 => SOCC 224 DFC 196 # Outputs 3
10011111 => SOCC 224 DFC 196 # Outputs 3
10101111 => SOCC 224 DFC 204 # Outputs 3
10110111 => SOCC 224 DFC 204 # Outputs 3
10111011 => SOCC 224 DFC 208 # Outputs 3
10111101 => SOCC 224 DFC 224 # Outputs 3
10111110 => SOCC 224 DFC 204 # Outputs 3
11001111 => SOCC 224 DFC 200 # Outputs 3
11010111 => SOCC 224 DFC 208 # Outputs 3
11011011 => SOCC 224 DFC 204 # Outputs 3
11011101 => SOCC 224 DFC 188 # Outputs 3
11011110 => SOCC 224 DFC 204 # Outputs 3
11100111 => SOCC 224 DFC 172 # Outputs 3 **
11101011 => SOCC 224 DFC 200 # Outputs 3
11101101 => SOCC 224 DFC 176 # Outputs 3
11101110 => SOCC 224 DFC 176 # Outputs 3
11110011 => SOCC 224 DFC 196 # Outputs 3
11110101 => SOCC 224 DFC 204 # Outputs 3
11110110 => SOCC 224 DFC 200 # Outputs 3
11111001 => SOCC 224 DFC 188 # Outputs 3
11111100 => SOCC 224 DFC 204 # Outputs 3

Decomposing: rnd_m50.bin
11101011 => SOCC 224 DFC 224 # Outputs 3 **

Decomposing: rndvv36.bin
00000100 => SOCC 128 DFC 64 # Outputs 1 **
00100000 => SOCC 128 DFC 64 # Outputs 1 **
00100100 => SOCC 64 DFC 64 # Outputs 1 **

Decomposing: substr1.bin
00000111 => SOCC 144 DFC 72 # Outputs 2 **
00001111 => SOCC 96 DFC 72 # Outputs 2 **
00011111 => SOCC 96 DFC 84 # Outputs 2
11100000 => SOCC 144 DFC 72 # Outputs 2 **
11110000 => SOCC 96 DFC 72 # Outputs 2 **
11111000 => SOCC 96 DFC 84 # Outputs 2

Decomposing: substr2.bin
00000111 => SOCC 144 DFC 84 # Outputs 2
00001011 => SOCC 144 DFC 84 # Outputs 2
00111111 => SOCC 144 DFC 88 # Outputs 2
11010000 => SOCC 144 DFC 84 # Outputs 2
11011101 => SOCC 224 DFC 68 # Outputs 3 **
11100000 => SOCC 144 DFC 80 # Outputs 2
11111100 => SOCC 144 DFC 80 # Outputs 2

Decomposing: subtraction1.bin
11000101 => SOCC 176 DFC 66 # Outputs 3 **
11001100 => SOCC 96 DFC 88 # Outputs 2

Decomposing: subtraction3.bin
VACUOUS 128
VACUOUS 64
VACUOUS 32
VACUOUS 16
VACUOUS 8
VACUOUS 4
function does not decompose

Appendix C

ASH Program Explanation

ASH can be broken up into two separate programs: a program that uses a function's output as an input, and a program that uses a function's minterms, in a cube format, as an input. The following are simple examples of input files to ASH.

Binary input file (identical to the format used by the Pattern Theory Group's FLASH program):

```
BIN                                  - Binary designation
5                                    - Number inputs
00000000000000000000111101100000     - Outputs of function
```

Cube input file:

```
CUBE                                 - Cube designation
8                                    - Number inputs
2                                    - Number of cubes in list
1010X0XX                             - Cube (cubes must be disjoint)
1011XXXX                             - Cube
```

Binary program description:

Because the input function is in the same form as the examples in this thesis, a decomposition chart is used to find if a partition can decompose. The idea here is to

take a partition string that designates the bound and free variables, such as 0011, which is a 4-input character string indicating X_3X_2 as free variables and X_1X_0 as bound variables. Notice that the least significant bit or right most bit is designated as X_0. A decomposition graph in memory is constructed with $2^{|\text{bound set}|}$ columns and $2^{|\text{free set}|}$. In the case of determining a non-disjoint decomposition, the shared variables are placed in the bound set. The next step is to find where each output should be placed in the decomposition chart. This is done by first finding the minterm values in the first row of the decomposition chart. For example, if given the partition string 01011, then the following first row can be found as:

0	1	2	3	8	9	10	11

This is done in a recursive fashion, the first column is always 0, the second column is found by finding the first bit in the partition string with a '1', then raising the position of the '1' to two. In the example, because the first one is in the X_0 position then 2^0=1, the second column is equal to 1. In the case of determining the '1' in position 3, 2^3=8, then by adding 8 to the previous columns gives, 8, 9, 10, 11. This also works for finding the first row values, the exception is that the '0's are evaluated instead. To find the remaining column and row values, add the first column value to the first row value. In continuing with the partition string 01011, the following decomposition chart can be drawn.

0	1	2	3	8	9	10	11
4	5	6	7	12	13	14	15
16	17	18	19	24	25	26	27
20	21	22	23	28	29	30	31

3+16=19

In the case of a non-disjoint partition, the decomposition chart first row is constructed without the shared variables, then the shared variables are used to complete the first row. Consider a partition string 0101S, where S is a shared variable. The following first row exists:

0	2	8	10	1	3	9	11

$x_0=0$ $x_0=1$

Column labeling to find the column multiplicity is the last step. This is an N! algorithm because the first column is compared to the N-1 remaining columns, the second column is compared to the N-2 remaining columns, and so on. To speed up this process, when comparing columns, two columns are not compared if one of them already has a label. Another method used here is to count the number of 1s in a column, if the number of 1s in two columns do not match then the columns are not compared. This is a very easy procedure to determine the number of 1s in a column, just by calculating the number while placing the values in the decomposition chart. In the case of the shared variable partition table, the column labeling is checked for each value of the shared variable.

For checking for vacuous variables, a decomposition chart does not need to be constructed, since there is only one bound variable. The object of searching for a vacuous variable is to see if all the row types are either a 00 or a 11. If they are, the variable is vacuous. While checking for vacuous variables and checking for 00s or 11s, it is very easy to start the evaluation of the Single Output function by counting the different row types. After the row types have been found, the next step to come up with a compatibility matrix. From the compatibility matrix the program calls the partition evaluation procedure to have it check for a column multiplicity of 2.

Cube program description:

The explanation of this is much simpler, even though the code segment is very complex. For checking a partition string the cubes are changed to fit the string. For example, given the following disjoint cubes:

```
00110011
11110000
11XX1100
```

and a partition string 11110000, the cubes are broken up into two halves producing the following:

```
0011      0011
1111      0000
11XX      1100
```

The column to the left contains bound variables and the column to the right contains the free variables. In the above, the cubes on the left overlap, or in this case the

11XX contains the 1111. New cubes must be calculated to remove the overlap producing the following cubes:

```
0011      0011
1111      0000
1111      1100
1110      1100
110X      1100
```

From this, a column multiplicity can be calculated, because the bound element 0011 (in the left column) has one free element 0011 (in the right column) that does not match any of the other free elements, then the bound element 0011 is a unique element. Thus, the column multiplicity is increased by one. The bound element 1111 contains the free elements 0000 and 1100, which does not match any other bound elements. The bound elements 1110 and 110X both contain the free element 1100 so they can be combined. Thus, the column multiplicity of the function is 3.

As for checking for vacuous variables, two things are done. First, if a given cube representation results in a variable position always having a X in that position then the variable is vacuous. For example,

```
111X
010X
```

In this case the position 0 variable is vacuous. The next test for vacuous variables is to see if the cubes have not been fully compressed leaving cubes in the following form.

```
1101
1111
```

This can be reduced to 11X1, which may possibly result in finding a vacuous variable.

The single output algorithm has not been implemented to take cube inputs.

The BBPS algorithm is a separate entity that involves checking the results of the partition evaluation from either the Binary procedures or the Cube procedures. This way the code can be changed to fit any other type of search technique very easily. The following is the .h file with the listing of the C++ classes.

```
/*************************************************************
** Filename:   ash.h
** Purpose:    Smaller version of flash, to test new algorithm
** Creater:    Craig Files
**             Oct 1, 1994
**             Feb 4, 1995 - ASH version 2
**
** Rev  Author    Date    Remarks
** .1   CF        4/05  Moved ash.h to top of .h file
**                      organization
** .2   CF        2/06  Changed procedures to incorporate
**                      mystring operations
** .3   CF        3/05  Ash version 3 started
** .4   CF        3/25  Implemented Class Evaluation
** .5   CF        3/31  Implemented Class First
** .6   CF        4/14  Added *next to Class Evaluation
** .7   CF        5/10  Created Function::decompose();
*************************************************************/
#include <math.h>
#include <stdlib.h>

typedef unsigned long int Huge;

class Partition {
  public:
  Huge       info[4];
  char      *array;
  int        num_ones;
  Huge       DFC, row_mismatches;
```

```
  int         column_mu;
  int         I_out;

// procedures
              Partition(int array_len);
  void        print();
  void        print(Partition *best);
  void        copy(Partition *first);
  void        cal_dfc(int inputs, Huge outputs);
};
class Cube {
  public: // see cube.C cube.h
  int    NUM_BITS;
  Huge   NUM_OUTPUTS;
  int    SIZE_CUBE;
  char **FUNCTION;
  int    first(int &dfc);
  void   eval(Partition *part);
  int    decompose(Cube *initial);
  void   getfunction(FILE *file);
  void   remove_var(int position, char value);
};
class Bin {
  public: // see bin.C bin.h
  int    NUM_BITS;
  int    NUM_OUTPUTS;
  int   *FUNCTION;
  void   print();
  void   getfunction(FILE *file);
  int    first(int &dfc);
  void   eval(Partition *part);
  void   evaluate(Partition *part,int **array,int col,int row)
  void   remove_var(int pos, char side);
  void   next_level(Partition *part,int **array,Bin
                    **col_block, Bin *row_block);
  int    decompose(Bin *intitial);
  int    next_level(char *part);
  int    full_next_level(Partition *part);
  int    full();
// new.C
  int    remove_vacuous(int &dfc);
  int    compare(int *array1, int *array2);
  void   shared_vars(Partition *part);
  void   compare_vars(Partition *part);
  int    backwards(Bin *inititial);
```

```
// vars
  int    LEVEL;
};
class Function {
  public:
  Partition *overall, *best;
  int        NUM_BITS;
  Huge       NUM_OUTPUTS;
  int        SIZE_CUBE;
// procedures
   Function(int num_bits, int size_cube, Huge num_outputs);
  void evaluate_partition(Partition *part, Cube *cube, Bin
                                  *bin);
  void check_overall(Partition *part);
  void init_part();
  void create_partitions(Cube *cube, Bin *bin, int location);
};
```

Bibliography

[1] R. L. Ashenhurst, "The Decomposition of Switching Functions," *Bell Laboratories Report*, No. BL-1(11), pp. 541-602, 1952.

[2] R. L. Ashenhurst, "Non-disjoint Decomposition," *Bell Laboratories Report*, No. 4, pp. 1V-1 - 1V-12, 1953.

[3] R. L. Ashenhurst, "The Decomposition of Switching Functions," *Proceedings of the International Symposium of Theory of Switching,* pp. 571-603, 1957.

[4] H. A. Curtis, "Non-Disjunctive Decompositions," *Theory of Switching Bell Labs Report*, No. 19, Section II, 1, 49, 1958.

[5] H. A. Curtis, "A New Approach to the Design of Switching Circuits," Princeton, N.J., Van Nostrand, 1962.

[6] M. Perkowski, T. Luba, S. Grygiel, R. Lisanke, N. Iliev, P. Burkey, M. Burns, R. Malvi, C. Stanley, Z. Wang, H. Wu, F. Yang, S. Zhou, and J.S. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," *Portland State University Technical Report*, Department of Electrical Engineering, March 1995.

[7] M. Perkowski, T. Luba, S. Grygiel, P. Burkey, M. Burns, N. Iliev, M. Kolsteren, R. Lisanke, R. Malvi, Z. Wang, H. Wu, F. Yang, S. Zhou, and J.S. Zhang, "Unified Approach to Functional Decompositions of Switching Functions," *Portland State University Technical Report*, Department of Electrical Engineering, May 22, 1995.

[8] M. Perkowski, H. Uong, "Generalized Decomposition of Incompletely Specified Multioutput, Multi-Valued Boolean Functions," *Unpublished manuscript,* Department of Electrical Engineering, Portland State University 1987.

[9] M. Perkowski, J. Brown, "A Unified Approach to Designs with Multiplexers and to the Decomposition of Boolean Functions," *Proceedings ASEE Annual Conference*, pp. 1610-1619, 1988.

[10] H.A. Curtis, "Generalized Tree Circuit-The Basic Building Block of an Extended Decomposition Theory," *J. ACM*, vol. 10, pp. 562-581, 1963.

[11] M.L. Axtell, "Partition Selection Algorithms: Row/Column Ratio Experiment," Technical Report, Veda Inc., c/o WL/AART-2, WPAFB, OH 45433-7408, April 1993.

[12] Y.T. Lai, M. Pedram, S. Sastry, "BDD-based Decomposition of Logic Functions With Application to FPGA Synthesis," *Proceedings of 30th DAC,* pp. 642-647, 1993.

[13] C.E. Shannon, "The Synthesis of Two-terminal Switching Circuits," *Bell System Technical Journal,* Vol. 28, pp. 59-98, Jan. 1949.

[14] Y.S. Abu-Mostafa, "Complexity in Information Theory," Springer-Verlag, New York, ISBN0-387-96600-5, 1988.

[15] M. Perkowski, T. Ross, D. Gadd, J.A. Golman, N. Song, "Application of ESOP Minimization in Machine Learning and Knowledge Discovery", *Pending submission to International Workshop on Applications of the Reed-Muller Expansion in Circuit Design,* 1995.

[16] T. D. Ross, M.L Axtell, M.J. Noviskey, "Logic Minimization as a Robust Pattern Finder," *International Workshop on Logic Synthesis*. Lake Tahoe, California. May 1993.

[17] S. Khuri. "An Evolutionary Approach to Combinatorial Optimization Problems," *Proceedings of CSC*, Phoenix, Arizona. March 1994.

[18] T. D. Ross, M.J. Noviskey, T.N. Taylor, D.A. Gadd, "On the Decomposition of Real-Valued Functions," *3rd International Workshop of Post-Binary VLSI Systems*, International Symposium on Multiple Valued Logic. May 24, 1994.

[19] T. D. Ross, M.J. Noviskey, T.N. Taylor, D.A. Gadd, "The Function Learning and Synthesis Hotbed," Wright Laboratory, WL/AART-2. WPAFB, OH 45433-6543. May 4, 1994.

[20] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," *Electronics Research Laboratory Memorandum,* no. UCB/ERL M92/41, Department of Electrical Engineering and Computer Science, University of California at Berkeley, May 1992.

[21] M.J. Noviskey, T.D. Ross, D.A. Gadd, M. Axtell, "Application of Genetic Algorithms to Functional Decomposition in Pattern Theory," *Report WL-TR-94-1015,* Wright Laboratory, WL/AART-2. WPAFB, OH 4533-5543. Jan 1994.

[22] J.F. Frenzel, "Application of Genetic Algorithms to Pattern Theory," *Final Report for Summer Faculty Research Program*, Wright Laboratory, Sponsored by Air Force

Office of Scientific Research, Bolling Air Force Base, DC and Wright Laboratory, July 1993, pp. 4-1 - 4-20.

[23] T.D. Ross, M.J. Noviskey, T.N. Taylor, D.A. Gadd, "Pattern Theory: An Engineering Paradigm for Algorithm Design," *Final Technical Report WL-TR-91-1060,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543. August 1991.

[24] T. Luba, R. Lasocki. "Decomposition of Multiple-Valued Boolean Functions," *Applied Mathematics and Computer Science,* Vol. 4 No.1 1994. pp 125-138.

[25] T. Sasao, "Multple-Valued Decomposition of Generalized Boolean Functions and the Complexity of Programmable Logic Arrays," *IEEE Transactions on Computers,* vol. 30. no. 9. pp. 635-643, 1981.

[26] T. Sasao, "Functional Decomposition of PLAs," *Proceedings of the International Workshop on Logic,* Research Triangle Park, North Carolina, May 1987.

[27] R. Murgai, Y. Hishizaki, N. Shenoy, R. K. Brayton, A. Sangiovanni-Vincentelli, "Logic Synthesis for Programmable Gate Arrays," *Proceedings of 27th ACM/IEEE Design Automation Conference*, pp. 620-625, 1990.

[28] R. Murgai, N. Shenoy, R. K. Brayton, A. Sangiovanni-Vincentelli, "Improved Logic Synthesis Algorithm for Table Look Up Architectures," *Proceedings IEEE International Conference on Computer Aided Design,* pp. 564-567, 1991.

[29] E. Boros, V. Gurvich, P.F. Hammer, T. Ibaracki, A. Kogan, "Decomposition of Partially Defined Boolean Functions," *DIMACS Technical Report 94-9*, March 1994.

[30] J.P. Roth, R.M. Karp, "Minimization Over Boolean Graphs," *IBM Journal of Research and Development*, April 1962, pp. 227-238.

[31] R. Murgai, R.K. Brayton, "Using Encoding for Functional Decomposition with Application to Look Up Table Architectures," *2nd International ACM/SIGDA Workshop on FPGAs*, 1994.

[32] M. Perkowski, "A Survey of Literature on Function Decomposition," *Final Report for Summer Faculty Research Program*, Wright Laboratory, Sponsored by Air Force Office of Scientific Research, Bolling Air Force Base, DC and Wright Laboratory, September 1994.

[33] V. Y. Shen, A. C. McKellar, "An Algorithm for the Disjunctive Decomposition of Switching Functions," *IEEE Transactions on Computers*, Vol. C-19, pp. 239-248, March 1970.

[34] V. Y. Shen, A. C. McKellar, P. Weiner, "A Fast Algorithm for the Disjunctive Decomposition of Switching Functions," *IEEE Transactions on Computers,* Vol C-20, pp. 304-309, March 1971.

[35] I. S. Reed, "A Class of Multiple-Error Correcting Codes and Decoding Scheme," *IRE Transactions on Information Theory*, Vol. IT-4, pp. 38-49, September 1954.

[36] D. Varma, E. A. Trachtenberg, "Design Automation Tools for Efficient Implementation of Logic Functions by Decomposition," *IEEE Transactions on Computer Aided Design,* Vol. 8, No. 8, pp. 901-917, August 1989.

[37] W. Wan, M. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Multi-Output Functions Based on Graph Coloring and Local Transformations and its Application to FPGA Mapping," P*roceedings of the IEEE European Design Automation Conference*, September 7-10, Hamburg, 1992.

[38] B. Steinbach, M. Stokert, "Design of Fully Testable Circuits by Functional Decomposition & Implicit Test Pattern Generation," *IEEE VLSI Test Symposium Proceedings*, pp. 22-27, 1994.

[39] C. M. Files, "Using a Search Heuristic in an NP-Complete Problem in Ashenhurst-Curtis Decomposition," *Final Report for Summer Graduate Student Research Program*, Wright Laboratory, Sponsored by Air Force Office of Scientific Research, Bolling Air Force Base, DC and Wright Laboratory, August 1994, pp. 10-1 - 10-19.

[40] T. D. Ross, "Variable Partition Search for Function Decomposition," *Technical Report,* Wright Laboratories, USAF, WL/AART/WPAFB, OH 45433-6543. November 1994.