

1991

A practical parallel algorithm for the minimization of Krönecker Reed-Muller expansions

Paul John Gilliam
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds



Part of the [Electrical and Computer Engineering Commons](#)

Let us know how access to this document benefits you.

Recommended Citation


Gilliam, Paul John, "A practical parallel algorithm for the minimization of Krönecker Reed-Muller expansions" (1991). *Dissertations and Theses*. Paper 4178.
<https://doi.org/10.15760/etd.6062>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

AN ABSTRACT OF THE THESIS OF Paul John Gilliam for the Master of Science in Electrical and Computer Engineering presented August 23, 1991.

Title: A Practical Parallel Algorithm for the Minimization of Krönecker Reed-Muller Expansions.


APPROVED BY THE MEMBERS OF THE THESIS COMMITTEE:



W. Robert Daasch, Chair



Michael A. Driscoll



Laszlo Csanky

A number of recent developments has increased the desirability of using exclusive OR (XOR) gates in the synthesis of switching functions. This has, in turn, led naturally to an increased interest in algorithms for the minimization of Exclusive-Or Sum of Products (ESOP) forms. Although this is an active area of research, it is not nearly as developed as the traditional Sum of Products forms. Computer programs to find minimum ESOPs are not readily available and those that do exist are impractical to use as investigative tools

because they are too slow and/or require too much memory. A practical tool would be easy enough to use (faster/smaller) so that it could be run many times to explore the solution space of the minimization problem as well as to provide a baseline of comparison. This thesis develops and investigates such a tool.

Building on the work of Bioul and Davio, D.H. Green presents the so-called "fast" algorithm for finding minimum Krönecker Reed-Muller (KRM) type ESOP forms, basically an exhaustive search of the solution space. In this thesis, the "fast" algorithm is reformulated to take advantage of a shared memory, multiprocessor environment.

The reformulation of the "fast" algorithm is presented within a rigorous mathematical framework. Because ESOP forms are being manipulated, it is more natural to use the two-element Galois field, $GF(2)$, in place of the more traditional Boolean algebra. The Krönecker product, also known as the tensor product, is used to form a KRM by combining different vectors chosen from a set of basis vectors. This is formulated rigorously as a matrix algebra problem, over $GF(2)$, involving a bit vector and a Krönecker matrix, a matrix formed by repeated Krönecker products of some "seed" matrix. The parallelization of the algorithm stems directly from the recursive nature of the Krönecker matrices involved.

Several different versions of the algorithm were programmed and used to investigate the computer resource requirements of the algorithm. Three

results were found: 1) Using word-based logical instructions significantly increases performance over one-bit-at-a-time manipulations, with a word size of 32 bits being the best; 2) A recursive "fast" algorithm is much faster than the direct algorithm; and 3) The shared memory, parallel processor algorithm developed in this thesis is even faster. Even with the improved performance, however, the exhaustive search nature of the algorithm causes the resources required to grow exponentially with the problem size. Current technology imposes a practical upper limit on the size of the problem to 15 bits. A minimum for such a problem can be found in 12 minutes on a Sequent S81 using about 28 megabytes of memory and 9 processors.

The claim is made that "kromin", the tool developed in this thesis, is a good tool; that is to say it is easy to use and does its job well. Aside from user-interface issues, "kromin" is easy to use because it is fast enough for many problems to be run during a course of research. It does its job well because, as an exhaustive search, it provides a complete characterization of the solution space for a given problem.

Several possibilities are presented for future work. For the most part they represent improvements to "kromin", either by enlarging the class of problems that can be solved or by increasing the size of the problem that can be minimized. The real measure of usefulness of any tool, of course, is in its use. This thesis presents a tool intended to be useful in the development of algorithms for the minimization of ESOPs.

A PRACTICAL PARALLEL ALGORITHM FOR THE
MINIMIZATION OF KRÖNECKER REED-MULLER
EXPANSIONS

by

PAUL JOHN GILLIAM

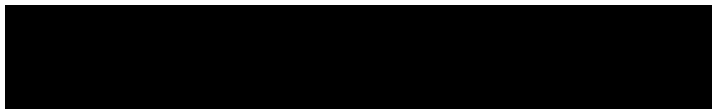
A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
ELECTRICAL AND COMPUTER ENGINEERING

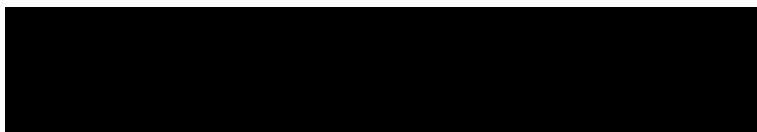
Portland State University
1991

TO THE OFFICE OF GRADUATE STUDIES:

The members of the Committee approve the thesis of Paul John Gilliam presented August 23, 1991.



W. Robert Daasch, Chair

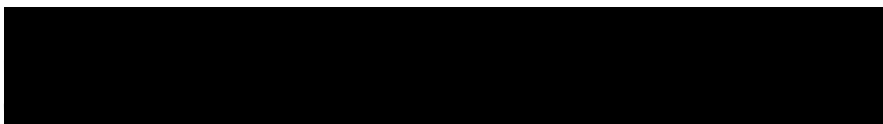


Michael A. Driscoll

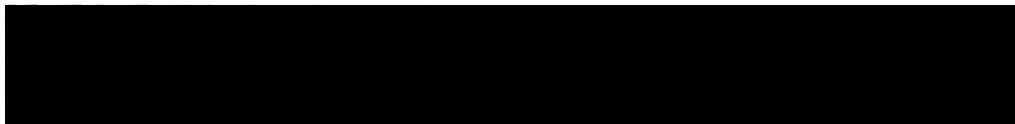


Laszlo Csanky

APPROVED:



Rolf Schaumann, Chair, Department of Electrical Engineering



C. William Savery, Vice Provost for Graduate Studies and Research

DEDICATION

This work is dedicated to Doreen Jaffee Gilliam. Not only was she supportive, she was supporting.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisor, Dr. W. Robert Daasch, and the other members of my committee: Dr. Michael A. Driscoll and Dr. Laszlo Csanky. My officemate, L. David Armbrust, was very helpful. I would also like to thank Dr. Malgorzata Chrzanowska-Jeske, who first got me interested in the general problem of ESOP minimization.

Special thanks to Sequent Computer Systems, Inc. for allowing me to use their computer system for the testing done in the second section of Chapter IV. Special thanks also to Argonne National Laboratory for the use of their Sequent "anagram".

This research was partially funded by the State of Oregon — Employment Division, Department of Human Resources, through the author's participation in the TRA training program. I would like to thank Dave Cleveland, the TRA administrator in the Hillsboro branch of the Employment Division.

Finally, I would like thank Ira Warren of Intel Corporation, whose offer of employment served as the final motivation I needed to finish the post-presentation modifications to this thesis.

TABLE OF CONTENTS

	PAGE
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vi
LIST OF FIGURES	vii
CHAPTER	
I Introduction and Motivation	1
An Example Easily Testable Realization	3
II Background Theory	8
Overview	8
The Two-Element Galois Field, $GF(2)$	8
Krönecker Products	10
Reed-Muller Canonical Forms	13
The Extended Truth and Weight Vectors	16
The 'Fast' Algorithm	19
III A Practical Algorithm	24
Overview	24
Chunky Bit Vectors	25
Recursive 'Fast' Algorithm	29
Practical Parallel Algorithm	36

IV	Experimental Results	43
	Overview	43
	Chunky Performance	44
	How Fast is ‘Fast’?	49
	Parallel Performance	55
V	Example Uses of Kromin	65
	Overview	65
	Cohn’s Conjecture and the Distribution of Weights	66
	Expansion on Work by Sasao and Besslich	71
	Incompletely Specified Functions	73
VI	Future Work and Conclusions	79
	Overview	79
	Incomplete Switching Functions	79
	ESOPs Are Not KRMs	81
	More Memory by Distributed Systems	83
	Less Memory by Close to Minimal Solutions	84
	Conclusions	85
	REFERENCES	87

LIST OF TABLES

TABLE		PAGE
I	Execution Times for the Direct Algorithm	45
II	Execution Times for the 'Fast' Algorithm	50
III	Execution Times for the Practical Parallel Algorithm . .	57
IV	Average Number of Products for All the Four-Variable Functions	75

LIST OF FIGURES

FIGURE		PAGE
1	An Example Easily Testable Circuit	4
2	Decomposition of e Through Recursion when Chunk Size is 27	33
3	Computation Tree for $n=3$	35
4	Execution Time of P Transform vs Chunk Size	47
5	Execution Time of P Transform vs $\log_3(\text{Chunk Size})$. . .	48
6	P Transform Time: Direct and 'Fast'	51
7	P Transform Time: Direct and 'Fast' (Log Scale)	52
8	Speed-up vs n	53
9	Speed-up vs n (Log Scale)	54
10	'Fast' P Transform Time vs n	55
11	'Fast' P Transform Time vs n (Log Scale)	56
12	Parallel P Transform Time vs n (Log Scale)	58
13	P Time vs $\log_3(\text{Processors})$	59
14	Predicted P Time vs n (Log Scale)	61
15	Predicted P Time vs $\log_3(\text{Processors})$	62
16	Unified Prediction: t vs n	63
17	Unified Prediction: t vs $\log_3(\text{Processors})$	64
18	Example Distribution of Weights for $n=5$	69

19	Example Distribution of Weights for $n=8$	68
20	Example Distribution of Weights for $n=11$	71
21	Example Distribution of Weights for $n=14$	72
22	For $n=14$, Distribution with Fewest Weights $\leq W$	73
23	For $n=14$, Distribution with Most Weights $\leq W$	74
24	Average Number of Products for All the Four-Variable Functions	76
25	Incomplete Functions Heuristic	77

CHAPTER I

INTRODUCTION AND MOTIVATION

A number of recent developments has increased the desirability of using exclusive OR (XOR) gates in the synthesis of switching functions. New technologies, such as Programmable Gate Arrays [1], make the cost, in terms of area and speed, equal for all types of gates. In older technologies, the cost of XOR gates, relative to the more traditional OR gates, needed to be balanced against the benefits. Currently, the major benefit is that circuits built from XOR gates can be easily tested [2]. This is demonstrated in the next section. It is also anticipated that for future technologies using optical switches, exclusive OR may be more natural than inclusive OR [3], because of the physics of these new devices.

One way to use XOR gates to realize an arbitrary switching function is to represent that function in exclusive sum-of-product (ESOP) form. Sasao [4] has shown that, on average, minimal ESOP forms require fewer product terms than the more traditional SOP forms. A family of ESOP representations, Reed-Muller canonical forms, has been used extensively, leading naturally to the problem of minimizing any such representation.

In his paper "Reed-Muller canonical forms with mixed polarity and their manipulations" [5], D.H. Green, building on the work of Bioul and

Davio [6], outlines an algorithm for finding expansions with minimal weight, i.e., a minimum number of AND terms. An algorithm is given in this thesis that incorporates the main idea of Green's paper, performing the arithmetic using word-based logical instructions of a digital computer to increase performance. A parallel version of the algorithm is also given, for an even greater increase in performance.

While these performance increases are substantial, as shown in the "Experimental Results" chapter of this thesis, they of course do not alter the brute force nature of the algorithm. Why then should we develop this practical algorithm when such brute force algorithms are usually superseded by more sophisticated techniques? The answer is simple: this is not meant to be the final answer to the problem of minimizing ESOP forms, but rather it is meant to be a tool to be used to study that problem. We call this tool "kromin".

Like any good tool, kromin should be easy to use and do its job well. For a computer program, ease of use is mostly a product of the user interface, which is outside the scope of this thesis. Ease of use is also affected by execution time, which is heavily influenced by algorithm design, which is a concern of this thesis.

The job of kromin is to help researchers develop better algorithms. One way to help is to serve as a "base line" against which other algorithms can be compared. This is a traditional role of a brute force algorithm.

Perhaps a more important way is to help provide insight. This insight would hopefully come from using kromin to explore the problem space. The larger the problems that the algorithm can handle, the larger the volume of the problem space that can be explored.

AN EXAMPLE EASILY TESTABLE REALIZATION¹

Testability, a driving force behind the use of ESOP forms, is shown in this section by presenting an example circuit that has the property of being easily testable, as defined by Readdy [2]. The example circuit realizes the following switching function:

$$f(x_1, x_2, x_3, x_4) = 1 \oplus \bar{x}_1 \oplus x_3 \bar{x}_1 \oplus x_3 x_2 \bar{x}_1 \oplus \bar{x}_4 x_2 \bar{x}_1 \oplus \bar{x}_4 x_3 x_2$$

Where the symbol \oplus is used to denote XOR.

The switching function is expressed in fixed-polarity Reed-Muller form where each literal is used in either complemented or uncomplemented form, but not both. Although Readdy [2] worked with zero-polarity Reed-Muller forms, where each literal is used only in uncomplemented form, this example shows the slight modifications needed to use Reed-Muller forms of any polarity. This same method can be used with Krönecker Reed-Muller forms (see below) with only slightly more complex modifications.

¹ Most of the material in this section is from reference [2].

The following logic diagram is a realization of the example switching function, in the form given above. The two "extra" outputs, g and g' , are included to reduce the number of vectors needed to test the circuit.

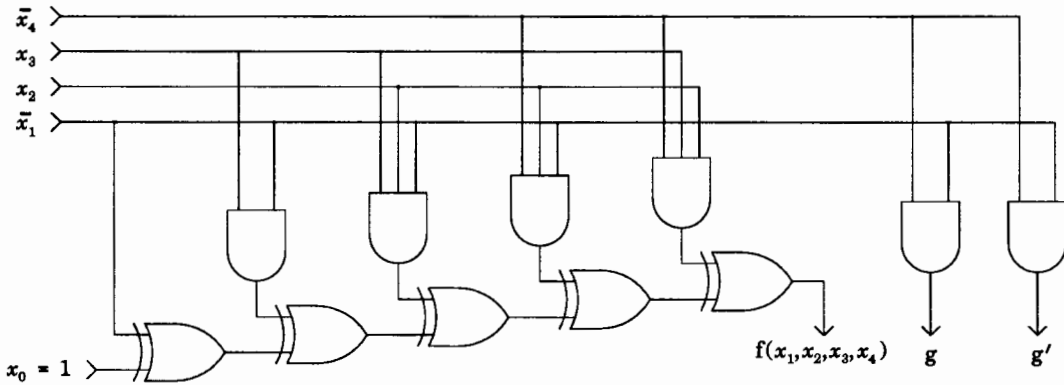


Figure 1. An example easily testable circuit.

To be testable, any single fault of the following types must be detectable in the circuit [2]:

- Stuck at 0 (s-a-0) faults in the input or output of an AND gate.
- Stuck at 1 (s-a-1) faults in the input or output of an AND gate.
- If an XOR gate is faulty, it may implement any other 2-input logic function.
- Faults in primary input leads.

From Readdy [2] we are given a set of four test vectors that will apply all possible input combinations to each XOR gate for any switching function in Reed-Muller form. When the polarity of the example switching function, in mixed-polarity Reed-Muller form, is taken into account, this set of four vectors is:

$$T_1 = \begin{array}{ccccc} x_0 & \bar{x}_1 & x_2 & x_3 & \bar{x}_4 \\ \left[\begin{array}{ccccc} 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{array} \right] & = & \begin{array}{ccccc} x_0 & x_1 & x_2 & x_3 & x_4 \\ \left[\begin{array}{ccccc} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \end{array} \right] \end{array}$$

If an XOR gate is faulty, its output may be any of the 15 two-input logic functions other than XOR. Because the XOR gates are cascaded in this circuit, the function's output is the parity of the outputs of the AND gates. This means that a change in the output of any single AND or XOR gate will cause the output of the function to change. In this way, these test vectors will detect a fault in any single XOR gate.

Either of the first two vectors in T_1 will force the outputs of the AND gates to 1. If a single s-a-0 fault occurs at any input or the output of any AND gate, it will be detected by either of these vectors. Either of the last two vectors in T_1 will force the outputs of the AND gates to 0. If a single s-a-1 fault occurs at the output of any AND gate, it will be detected by either of these vectors. If a single s-a-1 fault occurs at the input of any AND gate, this set of test vectors will not detect it because all of the inputs are 0.

The set of n test vectors that will detect a single s-a-1 fault at any of the inputs to the AND gates can be derived from those given in Readdy [2] by taking into account the polarity:

$$T_2 = \begin{array}{ccccc} x_0 & \bar{x}_1 & x_2 & x_3 & \bar{x}_4 \\ \left[\begin{array}{ccccc} d & 0 & 1 & 1 & 1 \\ d & 1 & 0 & 1 & 1 \\ d & 1 & 1 & 0 & 1 \\ d & 1 & 1 & 1 & 0 \end{array} \right] & = & \begin{array}{ccccc} x_0 & x_1 & x_2 & x_3 & x_4 \\ \left[\begin{array}{ccccc} d & 1 & 1 & 1 & 0 \\ d & 0 & 0 & 1 & 0 \\ d & 0 & 1 & 0 & 0 \\ d & 0 & 1 & 1 & 1 \end{array} \right] \end{array}$$

where d can be either a 0 or a 1. The i th vector in T_2 inputs a zero to any AND gate connected to \hat{x}_i , where $\hat{x}_i = x_i$ or $\hat{x}_i = \bar{x}_i$, depending on the polarity, and all other inputs of the AND gates are set to 1. If the input connected to \hat{x}_i on any single AND gate is s-a-1, then the output of that gate will change from 0 to 1, changing the output of the whole circuit, which is detectable as a fault.

To detect a single faulty primary input, we first assume that the rest of the circuit is fault free (we can make this assumption because only single faults are being detected). If the single faulty primary input is s-a-1, then the row of T_2 corresponding to that input will detect the fault if that input is connected to an odd number of AND gates. This is because an odd number of changes at the inputs of the XOR cascade will change the output of the cascade, because XOR gates are also modulo 2 adders. If the single faulty primary input is s-a-0, then either of the first two vectors in T_1 will detect the fault, again provided that the input is connected to an odd number of AND gates.

To detect a single faulty primary input that is connected to an even number of AND gates, we would need at most two new test vectors for each

one [2]. We can avoid the need for these extra vectors if we simply add an AND gate tied to each of these inputs, ensuring that every input connects to an odd number of test points. If we assume that this new AND gate is fault free, which can be checked, then any single fault in one of these primary inputs can be detected by the test vectors in $T_1 \cup T_2$, only now we monitor the output g instead of the function output f . To detect faults in this new AND gate, we can simply add another AND gate, connected to the same inputs, and compare the two outputs g and g' .

This section has shown how a realization of a 4-input example switching function, expressed in mixed-polarity Reed Muller form, can be tested for single faults using only 8 test vectors. Readdy [2] shows that any switching function of n inputs can be realized in a circuit that needs only $n+4$ test vectors to detect any single fault. While this scheme does require the use of two AND gates solely for testing, the cost of the "extra" gates should be compared to the cost of needing more test vectors. This can be compared to a time vs. space trade-off, where the number of test vectors takes the place of time and the number of gates added to increase the effectiveness of those test vectors takes the role of space. In this case, the trade-off is affected by n because as n increases, the relative cost of the "extra" gates decreases, making their "overhead" less and less of a factor.

CHAPTER II

BACKGROUND THEORY²

OVERVIEW

This chapter gives some of the theory behind the development of an algorithm, given by Green [5], for finding minimum Krönecker Reed-Muller (KRM) expansions. The first two sections cover algebra topics needed as background for the rest of the chapter. The remaining sections expand upon the development given in Green [5].

THE TWO-ELEMENT GALOIS FIELD, $GF(2)$

The traditional mathematical tool used in dealing with switching functions is the theory of Boolean algebras, specifically the two-element Boolean algebra. The operators of a Boolean algebra correspond well to the basic gates used in standard sum-of-product (SOP) implementations. When dealing with ESOP implementations, another tool must be found. Such a tool is the theory of Galois Fields, specifically $GF(2)$ [7]. Galois Fields are

² Most of the material in this chapter is from references [5] and [7].

named after their discoverer, the French mathematician Évariste Galois (1811-1831)³ [8].

In general, a field consists of a set, S , and two operators, \oplus and \circ (the usual shorthand of ab will be used for $a \circ b$), that have the following properties [9]:

1. S and \oplus form an abelian group.

- 1a. $\forall a, b \in S, a \oplus b \in S.$ (Closure)
- 1b. $\forall a, b, c \in S, (a \oplus b) \oplus c = a \oplus (b \oplus c).$ (Associative)
- 1c. $\exists 0 \in S \mid \forall a \in S, a \oplus 0 = 0 \oplus a = a.$ (Identity)
- 1d. $\forall a \in S, \exists -a \in S \mid a \oplus (-a) = (-a) \oplus a = 0.$ (Inverse)
- 1e. $\forall a, b \in S, a \oplus b = b \oplus a.$ (Commutative)

2. Let $S' = S \setminus \{0\}$, then S' and \circ form an abelian group.

- 2a. $\forall a, b \in S', a \circ b \in S'.$ (Closure)
- 2b. $\forall a, b, c \in S', (ab)c = a(bc).$ (Associative)
- 2c. $\exists 1 \in S' \mid \forall a \in S', a \circ 1 = 1 \circ a = a.$ (Identity)
- 2d. $\forall a \in S', \exists a^{-1} \in S' \mid a(a^{-1}) = (a^{-1})a = 1.$ (Inverse)
- 2e. $\forall a, b \in S, ab = ba.$ (Commutative)

3. $\forall a \in S, a \circ 0 = 0.$

4. The following distributive laws hold for any $a, b, c \in S$:

$$a(b \oplus c) = ab \oplus ac \quad \text{and} \quad (b \oplus c)a = ba \oplus ca.$$

³ Tragically, Galois was killed, at the age of 20, in a duel over a woman.

A simple example of a field is the field of real numbers, where \oplus is the ordinary addition operation and \odot is the ordinary multiplication operation. When a field, such as the field of real numbers, has an underlying set with an infinite number of members, it is an infinite field. If we let \oplus be modulo- q addition, \odot be modulo- q multiplication, and $S=\{0,1,\dots,q-1\}$, then if q is prime or an integer power of a prime, a field denoted $GF(q)$, is formed. This field is a finite field because S has a finite number of elements, namely q . Because we are working with switching functions where only two states are possible, we are only interested here in $GF(2)$.

Since $GF(2)$ is a field, many of the operators defined for the field of real numbers will be useful when performed over $GF(2)$. In particular, two matrix operations over $GF(2)$, with all additions modulo-2 and all multiplications modulo-2, are used throughout this paper. Besides the ordinary matrix product, which behaves as expected and is represented in the usual way, the Krönecker product is also used. The Krönecker product is covered in the next section.

KRÖNECKER PRODUCTS

The Krönecker product of two matrices, $A\otimes B$, is defined [10] as the partitioned matrix:

$$A_{i \times j} \otimes B_{k \times l} = C_{ij \times kl} = \begin{bmatrix} a_{0,0}B & a_{0,1}B & \dots & a_{0,j}B \\ a_{1,0}B & a_{1,1}B & \dots & a_{1,j}B \\ \vdots & \vdots & & \vdots \\ a_{i,0}B & a_{i,1}B & \dots & a_{i,j}B \end{bmatrix}$$

The Krönecker product can be defined over any field, in particular $GF(2)$. Some of the properties of Krönecker products are given below. In their descriptions (and in the definition above), it is important to remember that all the operations are over the same field.

Properties of Krönecker products:

1. If α is a scalar, then $A \otimes (\alpha B) = \alpha(A \otimes B)$.
2. $(A+B) \otimes C = (A \otimes C) + (B \otimes C)$ and $A \otimes (B+C) = (A \otimes B) + (A \otimes C)$.
3. $A \otimes (B \otimes C) = (A \otimes B) \otimes C$.
4. $(A \otimes B)^T = A^T \otimes B^T$.
5. $(A \otimes B)(C \otimes D) = AC \otimes BD$ (provided the dimensions of A , B , C , and D are such that the various matrix products exist). This is known as the mixed-product rule.
6. $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ (provided the inverses exist).

Krönecker products are used here, with binary matrices, to expand the right-hand matrix into a larger matrix by copying it into positions of the larger matrix using the left-hand matrix as a guide. We will call a matrix formed in this way a Krönecker matrix. The next section uses Krönecker

products over $GF(2)$ to define and discuss different types of Reed-Muller expansions.

As an example, consider the matrices A and B :

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

then

$$A \otimes B = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

and

$$B \otimes A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

REED-MULLER CANONICAL FORMS

Every switching function can be characterized by at least two binary vectors with 2^n elements, where n is the number of input variables. One such vector is called d , the truth vector, and is just the output column of the function's truth table. The function is expressed as the sum of 2^n Boolean minterms, each multiplied by a corresponding element of d . The elements of d serve as the (binary) coefficients of this "traditional" form.

A second vector is called a , the "function" vector, and consists of the coefficients of the Reed-Muller (RM) canonical form of the given switching function. This form consists of the XOR of 2^n product terms, each multiplied by a corresponding element of a . Together, these product terms comprise all possible combinations of the n literals. These product terms are used in a particular order, making this form canonical. The RM canonical form can then be expressed as:

$$f(x_1, x_2, \dots, x_n) = a_0 \oplus a_1 x_1 \oplus a_2 x_2 \oplus a_3 x_2 x_1 \oplus \dots \oplus a_{2^n-1} x_1 x_2 \dots x_n$$

which is expressed in the Krönecker product form as:

$$f(x_1, x_2, \dots, x_n) = ([1 \ x_n] \otimes [1 \ x_{n-1}] \otimes \dots \otimes [1 \ x_1]) a$$

The Krönecker product over n variables, each with the basis vector $[1 \ x_i]$, generates all the terms and, when multiplied by a , forms the RM canonical form. For example, when $n=2$:

$$\begin{aligned}
f(x_1, x_2) &= ([1 \ x_2] \oplus [1 \ x_1])a \\
&= [1 \ x_1 \ x_2 \ x_2x_1]a \\
&= a_0 \oplus a_1x_1 \oplus a_2x_2 \oplus a_3x_2x_1
\end{aligned}$$

The truth vector d can be related to the function vector a using a transform matrix T_n , recursively defined as:

$$\begin{aligned}
T_n &= \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \otimes T_{n-1}, \text{ for } n \geq 1 \\
T_0 &= [1], \text{ for } n = 0
\end{aligned}$$

since $T_n = T_n^{-1}$, $d = T_n a$, and $a = T_n d$.

If the vector $[1 \ \bar{x}_i]$ is added as a second choice for a basis vector, then the number of function vectors representing an n input function increases to 2^n . In other words, for an n input switching function, there are 2^n Fixed-Polarity RM (FPRM) forms in which each variable could occur complemented or uncomplemented but not both. For a given polarity $\langle p \rangle$, expressed as the binary number $\langle p_n, p_{n-1}, \dots, p_1 \rangle$, the terms of the corresponding fixed-polarity RM form can be expressed as

$$[x_n]^{p_n} \otimes [x_{n-1}]^{p_{n-1}} \otimes \dots \otimes [x_1]^{p_1}$$

where

$$\begin{aligned}
[x_i]^0 &\equiv [1 \ x_i] \\
[x_i]^1 &\equiv [1 \ \bar{x}_i]
\end{aligned}$$

The coefficients for the fixed-polarity RM function vector of polarity $\langle p \rangle$ can be obtained from the function vector a using the transform matrix $Z_{\langle p \rangle}$, defined as:

$$Z_{\langle p \rangle} = [Z]^{p_n} \otimes [Z]^{p_{n-1}} \otimes \dots \otimes [Z]^{p_1}$$

where

$$[Z]^0 \equiv \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$[Z]^1 \equiv \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

Combining the above expressions for the terms and coefficients of the RM of fixed-polarity $\langle p \rangle$ gives the following expression for the given switching function:

$$f(x_1, x_2, \dots, x_n) = ([x_n]^{p_n} \otimes [x_{n-1}]^{p_{n-1}} \otimes \dots \otimes [x_1]^{p_1}) Z_{\langle p \rangle} a$$

If a third basis vector, $[\bar{x}_i, x_i]$, were added to those given above, it would be possible to generate forms where the variable x_i occurs in both complemented and uncomplemented forms. For each of the n variables, there are now 3 basis vectors to choose from, giving 3^n possible expansions. Green calls these forms Krönecker Reed-Muller (KRM) expansions.

Each KRM expansion can be given a mixed-polarity number p , $0 \leq p \leq 3^{n-1}$. For a given polarity $\langle p \rangle$, expressed as the trinary number $\langle p_n, p_{n-1}, \dots, p_1 \rangle$, the terms of the corresponding KRM form can be expressed as:

$$[x_n]^{p_n} \otimes [x_{n-1}]^{p_{n-1}} \otimes \dots \otimes [x_1]^{p_1}$$

where

$$[x_i]^0 \equiv [1 \ x_i]$$

$$[x_i]^1 \equiv [1 \ \bar{x}_i]$$

$$[x_i]^2 \equiv [\bar{x}_i \ x_i]$$

The coefficients for the KRM expansion of mixed-polarity p can be derived from the function vector a using the transform matrix $K_{\langle p \rangle}$, defined as:

$$K_{\langle p \rangle} = [K]^{p_n} \otimes [K]^{p_{n-1}} \otimes \dots \otimes [K]^{p_1}$$

where

$$[K]^0 \equiv \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$[K]^1 \equiv \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

$$[K]^2 \equiv \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

If the vector containing the coefficients is called t (we will need this later), then $t = K_{\langle p \rangle} a$ and

$$f(x_1, x_2, \dots, x_n) = ([x_n]^{p_n} \otimes [x_{n-1}]^{p_{n-1}} \otimes \dots \otimes [x_1]^{p_1}) t$$

The next section develops a way to find the KRM expansion with the fewest terms for a given switching function, i.e. the minimum KRM.

THE EXTENDED TRUTH AND WEIGHT VECTORS

Bioul and Davio [6] introduced the concept of the extended truth vector which has 3^n components. Each component corresponds to a possible term in a sum-of-product expression of a function. There are 3^n such terms because for each of the n literals, there are three choices for a given term: the literal is absent from the term, the complement of the literal is present in the term, or the literal is present in the term.

Using the basis vector $[1 \bar{x}_i x_i]$, these terms can be represented by a Krönecker product over n variables. For $n = 3$:

$$[1 \bar{x}_3 x_3] \otimes [1 \bar{x}_2 x_2] \otimes [1 \bar{x}_1 x_1] = [1 \bar{x}_1 x_1 \bar{x}_2 \bar{x}_2 \bar{x}_1 \bar{x}_2 x_1 x_2 x_2 \bar{x}_1 x_2 x_1 \bar{x}_3 \bar{x}_3 \bar{x}_1 \bar{x}_3 x_1 \bar{x}_3 \bar{x}_2 \bar{x}_1 \bar{x}_3 x_2 x_1 \bar{x}_3 x_2 \bar{x}_1 \bar{x}_3 x_2 \bar{x}_1 \bar{x}_3 x_2 x_1 x_3 x_3 \bar{x}_1 x_3 x_1 x_3 \bar{x}_2 x_3 \bar{x}_2 \bar{x}_1 x_3 \bar{x}_2 x_1 x_3 x_2 x_3 \bar{x}_2 \bar{x}_1 x_3 x_2 x_1]$$

Any function of 3 literals will be some combination of these terms.

Each KRM form selects a different set of $2^n=8$ of these $3^n=27$ terms. For a specific function, some of the 2^n selected terms may correspond to zero coefficients. Finding the minimum KRM is equivalent to finding the KRM form with the maximum number of zero coefficients for the 2^n selected terms.

The extended truth vector e is comprised of all the components of the ordinary truth vector d and all their linear combinations over $GF(2)$. More precisely: $e = M_n d$ where

$$M_n = M_{n-1} \otimes M_1, \text{ for } n > 1$$

$$M_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix}$$

If we wish to express the extended truth vector e in terms of the function vector a , the transform matrix T_n can be used to find a matrix N_n such that $e = N_n a$ where

$$\begin{aligned}
N_n &= M_n T_n \\
(N_1 \otimes N_1 \otimes \dots \otimes N_1) &= (M_1 \otimes M_1 \otimes \dots \otimes M_1)(T_1 \otimes T_1 \otimes \dots \otimes T_1) \\
N_1 &= M_1 T_1 \\
&= \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}
\end{aligned}$$

Each polarity of the KRM expansion can be related to the extended truth vector by constructing an incidence matrix P_n . Each column corresponds to one of the 3^n possible terms and each of the 3^n polarities is represented by a row in P_n with 2^n ones and $3^n - 2^n$ zeros. Each row selects the 2^n coefficients of a function vector from the 3^n bits of the extended truth vector. Since all the possible expansions and all possible terms are both generated using the Krönecker product, it would seem reasonable to express P_n as $P_1 \otimes P_1 \otimes \dots \otimes P_1$, n times.

P_1 can be constructed by inspecting the three transformation matrices $[K]^0$, $[K]^1$, and $[K]^2$, in relation to the extended truth vector expressed in terms of the function vector:

$$e = N_1 \alpha = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \alpha = [\alpha_0 \ (\alpha_0 \oplus \alpha_1) \ \alpha_1]$$

$$[K]^0 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \Rightarrow \begin{matrix} t_0 = \alpha_0 = e_0 \\ t_1 = \alpha_1 = e_2 \end{matrix} \Rightarrow \text{row 0 of } P_1 = [1 \ 0 \ 1]$$

$$[K]^1 = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \Rightarrow \begin{matrix} t_0 = \alpha_0 \oplus \alpha_1 = e_1 \\ t_1 = \alpha_1 = e_2 \end{matrix} \Rightarrow \text{row 1 of } P_1 = [0 \ 1 \ 1]$$

$$[K]^2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \Rightarrow \begin{matrix} t_0 = \alpha_0 = e_1 \\ t_1 = \alpha_0 \oplus \alpha_1 = e_2 \end{matrix} \Rightarrow \text{row 2 of } P_1 = [1 \ 1 \ 0]$$

$$\therefore P_1 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

The weight of a particular KRM expansion is the number of terms with non-zero coefficients. The extended weight vector w is formed from the weights of all the KRM expansions for the given switching function. It can be computed by forming the real matrix product P_n and e : $w = P_n \times e$. Once computed, w can be used to identify the minimum weight KRM expansion. This is, essentially, Green's algorithm for finding minimum Kröneckner Reed-Muller expansions.

THE 'FAST' ALGORITHM

Although the direct algorithm described in the previous section will find the KRM with the fewest terms, it is not efficient in execution time. For this reason, we use the work of Zhang and Rayner [11], who introduced what they named the 'fast' algorithm. This is actually a family of

algorithms which computes the product of a Krönecker matrix and a vector more efficiently than a direct algorithm. The ‘fast’ algorithm is more efficient because it takes advantage of the recursive structure of the Krönecker matrix.

To develop the ‘fast’ algorithm, first let A_n be the Krönecker matrix formed by $(A_1 \otimes A_1 \otimes \dots \otimes A_1)$, n times, where A_1 has r rows and s columns. If v is a n -vector, then the product $A_n v$ can be factored as follows:

$$A_n v = (A_1 \otimes A_1 \otimes \dots \otimes A_1) v = (I_r I_r \dots A_1) \otimes (I_r I_r \dots A_1 I_s) \otimes (I_r I_r \dots A_1 I_s I_s) \otimes \dots \otimes (I_r I_r A_1 I_s I_s \dots) \otimes (I_r A_1 I_s I_s \dots) \otimes (A_1 I_s I_s \dots) v$$

where I_r and I_s are $r \times r$ and $s \times s$ unit matrices, respectively. Each term above is comprised of $n-1$ such unit matrices, along with a single A_1 . If we apply the mixed product rule (property 5, above, of Krönecker products) we get

$$A_n v = (I_r \otimes I_r \otimes \dots \otimes A_1) (I_r \otimes I_r \otimes \dots \otimes A_1 \otimes I_s) (I_r \otimes I_r \otimes \dots \otimes A_1 \otimes I_s I_s) \dots (I_r \otimes I_r \otimes A_1 \otimes I_s \otimes I_s \otimes \dots) (I_r \otimes A_1 \otimes I_s \otimes I_s \otimes \dots) (A_1 \otimes I_s \otimes I_s \otimes \dots) v$$

In the ‘fast’ algorithm, this product is computed from right to left. With each successive multiplication, "partial" sums are accumulated that represent progressively larger numbers of additions. Without the ‘fast’ algorithm, these partial sums would be recomputed each time they were needed. For example, consider the case of transforming d to a for $n = 3$:

$$\downarrow[(^L\alpha+^S\alpha) (^S\alpha+^T\alpha) ^S\alpha ^T\alpha (^S\alpha+^I\alpha) (^T\alpha+^0\alpha) ^I\alpha ^0\alpha] =$$

$$\begin{bmatrix} ^L\alpha \\ ^S\alpha \\ ^S\alpha \\ ^T\alpha \\ ^S\alpha \\ ^T\alpha \\ ^I\alpha \\ ^0\alpha \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$\alpha \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \oplus \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \oplus \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right) =$$

$$\alpha = \text{Let } \beta = (I^2 \otimes J \otimes I^2) \alpha$$

$$\downarrow[(^Lp+^Sp) (^Sp+^Tp) (^Sp+^Ip) (^Tp+^0p) ^Sp ^Tp ^Ip ^0p] =$$

$$\begin{bmatrix} ^Lp \\ ^Sp \\ ^Sp \\ ^Tp \\ ^Sp \\ ^Tp \\ ^Ip \\ ^0p \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} =$$

$$p \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \right) =$$

$$p = \text{Let } \alpha = (A^1 \otimes I^2 \otimes I^2) p$$

$$a = T^S p = (I^2 \otimes I^2 \otimes A^1)(I^2 \otimes A^1 \otimes A^1)(I^2 \otimes I^2 \otimes I^2) p$$

$$\begin{aligned}
\text{Now } a &= (I_2 \otimes I_2 \otimes T_1) \beta \\
&= \left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \oplus \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \oplus \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \right) \beta \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_3 \\ \beta_4 \\ \beta_5 \\ \beta_6 \\ \beta_7 \end{bmatrix} \\
&= [\beta_0 \ (\beta_0+\beta_1) \ \beta_2 \ (\beta_2+\beta_3) \ \beta_4 \ (\beta_4+\beta_5) \ \beta_6 \ (\beta_6+\beta_7)]^T
\end{aligned}$$

Now compare the 'fast' solution above to the direct solution below:

$$\begin{aligned}
a &= T_3 d = (T_1 \otimes T_1 \otimes T_1) d \\
&= \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \end{bmatrix} = \begin{bmatrix} d_0 \\ d_0+d_1 \\ d_0+d_2 \\ d_0+d_1+d_2+d_3 \\ d_0+d_4 \\ d_0+d_1+d_4+d_5 \\ d_0+d_2+d_4+d_6 \\ d_0+d_1+d_2+d_3+d_4+d_5+d_6+d_7 \end{bmatrix}
\end{aligned}$$

In the direct solution, the sum d_0+d_4 is computed 4 times. In the 'fast' solution, it is computed only once (as α_4). In the direct solution, the sum $(d_0+d_4)+(d_1+d_5)$ is computed twice, but only once in the 'fast' solution. In

general, the number of additions that are avoided by the 'fast' algorithm depends on the specific Kronecker matrix.

CHAPTER III

A PRACTICAL ALGORITHM

OVERVIEW

When implementing computer algorithms, many practical concerns need to be considered. For example, typical computers do not perform logical operations one bit at a time. They are usually performed a word at a time, where the number of bits per word is now usually 32. A given logical operation is applied to each bit of the argument words, in parallel, giving each bit of the resultant word. This implicit parallelism can be exploited to increase performance.

Another practical aspect to consider is that of space. The storage required for M , P , e , and w grows very fast with n . To reduce the amount of memory (and allow larger problems to be solved), the recursive structure of M and P can be exploited so that only small versions of them are needed. Even when careful attention is given to these practical concerns, current technology, in the form of memory limitations, limits sizes of problems to $n \leq 15$. For example, when $n=15$, the vector w will contain $3^n=3^{15}=14,348,907$ weights. The use of virtual memory eases the concern about space somewhat, but at the expense of the execution time, which is still a problem.

Another way to improve performance is to use multiple processors. The structure of Krönecker matrices makes them good choices for the application of parallel processing techniques.

For this algorithm to be the basis of a good tool, useful in the study of the problem of minimizing ESOP forms, it must be able to solve problems of a reasonable size in a reasonable amount of time on an accessible computer system. Of course, this is all highly subjective. The best we can do now is to provide a tool that is practical to use and wait for feedback from actual users.

The remaining sections in this chapter each address a different practical concern: The "Chunky Bit Vector" section examines how to use the "word-wide" operations. The "Recursive 'Fast' Algorithm" section reformulates the 'fast' algorithm into a form more easily implemented. Finally, the "Practical Parallel Algorithm" section increases performance of the algorithm by using multiple processors.

CHUNKY BIT VECTORS

Green's [5] algorithm manipulates two bit vectors (d and e), two matrices (M_n and P_n) and a vector of integers (w). The vectors are discussed next followed by the matrices.

The weight vector does not require special attention, except for its size and the size of its elements. The maximum weight is 2^n , so each

element must be at least n bits wide. The actual width chosen will be the upper limit on the size of problems the algorithm will allow. If the weight vector was implemented as an array of unsigned 16-bit words, then when $n=16$, w would have 43,046,721 (3^{16}) elements and take more than 80 megabytes of memory. This was chosen as a practical upper limit.

The vectors d and e are both arguments in matrix products over $GF(2)$. Rather than perform logical operations with the vectors one bit at a time, elements of the vectors can be grouped together so that the word oriented logical instructions available on most computers can be used. As a short-cut, such a group of bits will be referred to as a "chunk", and a vector divided into such chunks will be referred to as a "chunky" bit vector.

Since d and e are represented as chunky bit vectors, the matrices M_n and P_n must also be represented in some chunky way. Both d and e are column vectors that multiply their respective matrices on the right. This leads naturally to representing the rows of these matrices as chunky bit vectors.

The truth vector d is multiplied by the matrix M_n , so they must be divided into chunks of the same size. As will be seen in the next section, only a small version of M_n is stored in memory. Each chunk of d corresponds to one row of M_n . In this way, the choice of the chunk size for d determines how much memory is used by M_n . If chunks of d were 32 bits, then M_5 would be stored, requiring 972 bytes of memory. If the switching

function has fewer inputs than the number needed to "fill up" a chunk, then we can just put the whole vector in the low-order part of a chunk and ignore the rest.

The extended truth vector e is not quite so nice to deal with. Its chunks must match the chunks of the matrix P_n which, because of its structure, must have chunks that are a power of 3 bits wide. Word sizes on a typical computer are powers of 2, such as 8, 16 or 32. Corresponding chunk sizes for e would be 3, 9 and 27. The most efficient would be 27. P_n is just as hard to deal with, but because most of the product $P_n e$ is performed implicitly, only a small portion of P_n is stored. In the case of a 27 bit chunk size, only P_3 would be stored, requiring 108 bytes of memory.

The effect of chunky bit vectors is similar to the 'fast' algorithm. The 'fast' algorithm partitions a product into smaller pieces for the purpose of computing those pieces only once. Chunky bit vectors partition a bit32 vector into smaller pieces for the purpose of storage and computational efficiency. For example, consider the case of finding the extended truth vector when $n=4$ and the chunk size for d is 8 bits:

$$\begin{aligned}
 e &= M_4 d \\
 &= (M_1 \otimes M_3) d \\
 &= ((M_1 I_2) \otimes (I_9 M_3)) d \\
 &= (M_1 \otimes I_9) (I_2 \otimes M_3) d
 \end{aligned}$$

If we let d_{ci} denote the i th chunk of d , M_{3cj} the j th chunk (row) of M_3 , and

$\Sigma(M_{3cj}d_{ci})$ the sum, across $GF(2)$, of the bits of the logical word-product of M_{3cj} and d_{ci} , then let:

$$\alpha = (I_2 \otimes M_3)d$$

$$\alpha = \begin{bmatrix} M_{3c1} \\ \vdots \\ M_{3c9} \\ \\ M_{3c1} \\ 0 \\ \vdots \\ M_{3c9} \end{bmatrix} \begin{bmatrix} d_{c1}^T \\ d_{c2}^T \end{bmatrix}$$

$$= [\Sigma(M_{3c1}d_{c1}) \Sigma(M_{3c2}d_{c1}) \dots \Sigma(M_{3c9}d_{c1}) \Sigma(M_{3c1}d_{c2}) \Sigma(M_{3c2}d_{c2}) \dots \Sigma(M_{3c9}d_{c2})]$$

$$= [\alpha_{c1} \alpha_{c2} \alpha_{c3} \alpha_{c4} \alpha_{c5} \alpha_{c6}]^T$$

α is an intermediate chunky bit vector which must have the same chunk size as e because the chunks of α combined to form the chunks of e as follows:

$$e = (M_1 \otimes I_9)\alpha$$

$$e = \begin{bmatrix} I_9 & 0 \\ 0 & I_9 \\ I_9 & I_9 \end{bmatrix} \begin{bmatrix} \alpha_{c1} \\ \alpha_{c2} \\ \vdots \\ \alpha_{c6} \end{bmatrix}$$

$$= [\alpha_{c1} \alpha_{c2} \dots \alpha_{c6} (\alpha_{c1} + \alpha_{c4}) (\alpha_{c2} + \alpha_{c5}) (\alpha_{c3} + \alpha_{c6})]^T$$

$$= [e_{c1} e_{c2} e_{c3} e_{c4} e_{c5} e_{c6} e_{c7} e_{c8} e_{c9}]^T$$

RECURSIVE 'FAST' ALGORITHM

In the form given by Zhang and Rayner [11], the 'fast' algorithm does not lend itself well to a practical implementation. It is reformulated below into a recursive form more easily implemented. This is followed by a proof of the operation count equivalency, between the recursive reformulation and the original 'fast' algorithm. For this development, only the P transform is considered; the M transform is very similar.

First, as in the original 'fast' algorithm, the product $P_n e$ is factored as follows:

$$w = P_n e = (P_1 \otimes P_1 \otimes \dots \otimes P_1) e = (I_3 I_3 \dots P_1) \otimes (I_3 I_3 \dots P_1 I_3) \otimes (I_3 I_3 \dots P_1 I_3 I_3) \otimes \dots \otimes (I_3 I_3 P_1 I_3 I_3 \dots) \otimes (I_3 P_1 I_3 I_3 \dots) \otimes (P_1 I_3 I_3 \dots) e$$

Green [5] rearranged this as follows: Using the commutative property,

$$= (P_1 I_3 I_3 \dots) \otimes (I_3 P_1 I_3 I_3 \dots) \otimes (I_3 I_3 P_1 I_3 I_3 \dots) \otimes \dots \otimes (I_3 I_3 \dots P_1 I_3 I_3) \otimes (I_3 I_3 \dots P_1 I_3) \otimes (I_3 I_3 \dots P_1) e$$

Now isolate the left hand term and, using the distributive law, factor out an I_3 from the remaining terms:

$$= (P_1 I_3 I_3 \dots) \otimes I_3 [(P_1 I_3 I_3 \dots) \otimes (I_3 P_1 I_3 I_3 \dots) \otimes \dots \otimes (I_3 I_3 \dots P_1 I_3 I_3) \otimes (I_3 I_3 \dots P_1 I_3) \otimes (I_3 I_3 \dots P_1)] e$$

This is still the Krönecker product of a number of terms, each of which is a matrix product of several matrices. Using the mixed product rule, this becomes:

$$= (P_1 \otimes I_3 \otimes I_3 \otimes \dots) (I_3 \otimes [(P_1 \otimes I_3 \otimes I_3 \otimes \dots) (I_3 \otimes P_1 \otimes I_3 \otimes I_3 \otimes \dots) (I_3 \otimes I_3 \otimes P_1 \otimes I_3 \otimes I_3 \otimes \dots) \dots (I_3 \otimes I_3 \otimes \dots P_1 \otimes I_3 \otimes I_3) (I_3 \otimes I_3 \otimes \dots P_1 \otimes I_3) (I_3 \otimes I_3 \otimes \dots \otimes P_1)]) e$$

which is the matrix product of a number of terms, each of which is the Krönecker product of several matrices. The expression in square brackets will evaluate to simply P_{n-1} and there are $n-1$ identity matrices in the first term. This and the partitioning of e into three parts, each with 3^{n-1} elements, leads to:

$$\begin{aligned}
&= (P_1 \otimes I_{3^{n-1}}) (I_3 \otimes P_{n-1}) e \\
&= \begin{bmatrix} I_{3^{n-1}} & 0 & I_{3^{n-1}} \\ 0 & I_{3^{n-1}} & I_{3^{n-1}} \\ I_{3^{n-1}} & I_{3^{n-1}} & 0 \end{bmatrix} \begin{bmatrix} P_{n-1} & 0 & 0 \\ 0 & P_{n-1} & 0 \\ 0 & 0 & P_{n-1} \end{bmatrix} \begin{bmatrix} e_{[0]} \\ e_{[1]} \\ e_{[2]} \end{bmatrix}
\end{aligned}$$

Here the Krönecker products have been performed leaving only ordinary matrix products. The recursive nature of the algorithm is now clearly evident. If we developed the algorithm from this, it would need temporary memory to hold the results of the recursive products. Since the square of an elementary permutation is the identity matrix [12], we can rearrange things as follows:

$$\begin{aligned}
&= \begin{bmatrix} I_{3^{n-1}} & 0 & I_{3^{n-1}} \\ 0 & I_{3^{n-1}} & I_{3^{n-1}} \\ I_{3^{n-1}} & I_{3^{n-1}} & 0 \end{bmatrix} \begin{bmatrix} I_{3^{n-1}} & 0 & 0 \\ 0 & 0 & I_{3^{n-1}} \\ 0 & I_{3^{n-1}} & 0 \end{bmatrix}^2 \begin{bmatrix} P_{n-1} & 0 & 0 \\ 0 & P_{n-1} & 0 \\ 0 & 0 & P_{n-1} \end{bmatrix} \begin{bmatrix} e_{[0]} \\ e_{[1]} \\ e_{[2]} \end{bmatrix} \\
&= \begin{bmatrix} I_{3^{n-1}} & 0 & I_{3^{n-1}} \\ 0 & I_{3^{n-1}} & I_{3^{n-1}} \\ I_{3^{n-1}} & I_{3^{n-1}} & 0 \end{bmatrix} \begin{bmatrix} I_{3^{n-1}} & 0 & 0 \\ 0 & 0 & I_{3^{n-1}} \\ 0 & I_{3^{n-1}} & 0 \end{bmatrix} \begin{bmatrix} P_{n-1} & 0 & 0 \\ 0 & 0 & P_{n-1} \\ 0 & P_{n-1} & 0 \end{bmatrix} \begin{bmatrix} e_{[0]} \\ e_{[1]} \\ e_{[2]} \end{bmatrix} \\
&= \begin{bmatrix} I_{3^{n-1}} & I_{3^{n-1}} & 0 \\ 0 & I_{3^{n-1}} & I_{3^{n-1}} \\ I_{3^{n-1}} & 0 & I_{3^{n-1}} \end{bmatrix} \begin{bmatrix} P_{n-1} & 0 & 0 \\ 0 & 0 & P_{n-1} \\ 0 & P_{n-1} & 0 \end{bmatrix} \begin{bmatrix} e_{[0]} \\ e_{[1]} \\ e_{[2]} \end{bmatrix}
\end{aligned}$$

Now the recursive products can be computed "in place" without the need for temporary storage for those products.

This leads to the following recursive algorithm, given in a C-like pseudo code:

```

fast_p(lvl, *w_out, *e_in) {
1)   IF (lvl == 0) {*w_out = *e_in; RETURN;}
2)   c = pwrof3[lvl-1];
3)   fast_p(lvl-1, w_out, e_in);
      fast_p(lvl-1, w_out+c, e_in+2*c);
      fast_p(lvl-1, w_out+2*c, e_in+c);
4)   FOR (i=0; i<c; ++i) {
          temp = w_out[i];
          w_out[i] += w_out[i+c];
          w_out[i+c] += w_out[i+2*c];
          w_out[i+2*c] += temp; }
5)   RETURN; }

```

At the top level, *lvl* will have the value *n*, the number of literals.

When, through recursion, *lvl* reaches 0, we are down to the level of a single bit, and Step 1 will cut off the recursion. In the actual implementation, the level at which recursion stops depends on the size of the chunks of vector *e*.

If the algorithm does not return in Step 1, then e_in and w_out are divided into thirds, each having c elements. The value of c is found in Step 2 by a simple table look-up.

In Step 3, recursion is used to compute the three partial results, each in its own portion of w_out . These partial results correspond to $P_{n-1}e_{(i)}$, where i is either 0, 1, or 2, depending on the portion of w_out . For the M transform, this step would be comprised of only two recursive calls.

The final values for the elements of the w_out vector are computed in Step 4. The three partial results are combined as dictated by the placement of ones in the matrix P_1 . For the M transform, this step would add the two partial results together to form the middle third of the e vector.

The p_out vector is now complete and this level of recursion is terminated by the return in Step 5. When the top level invocation returns, the algorithm is complete.

Figure 2 graphically illustrates how the vector e is decomposed into smaller and smaller pieces through recursion. At the top level, the e_in vector is comprised of the entire 3^n bits of vector e , which is then divided into thirds, each passed to a recursive invocation of $fast_p$. Each level of recursion repeats this process until a third of the e_in vector can fit in a single chunk. At this point, a direct algorithm is used to compute w_out for this bottom level of recursion.

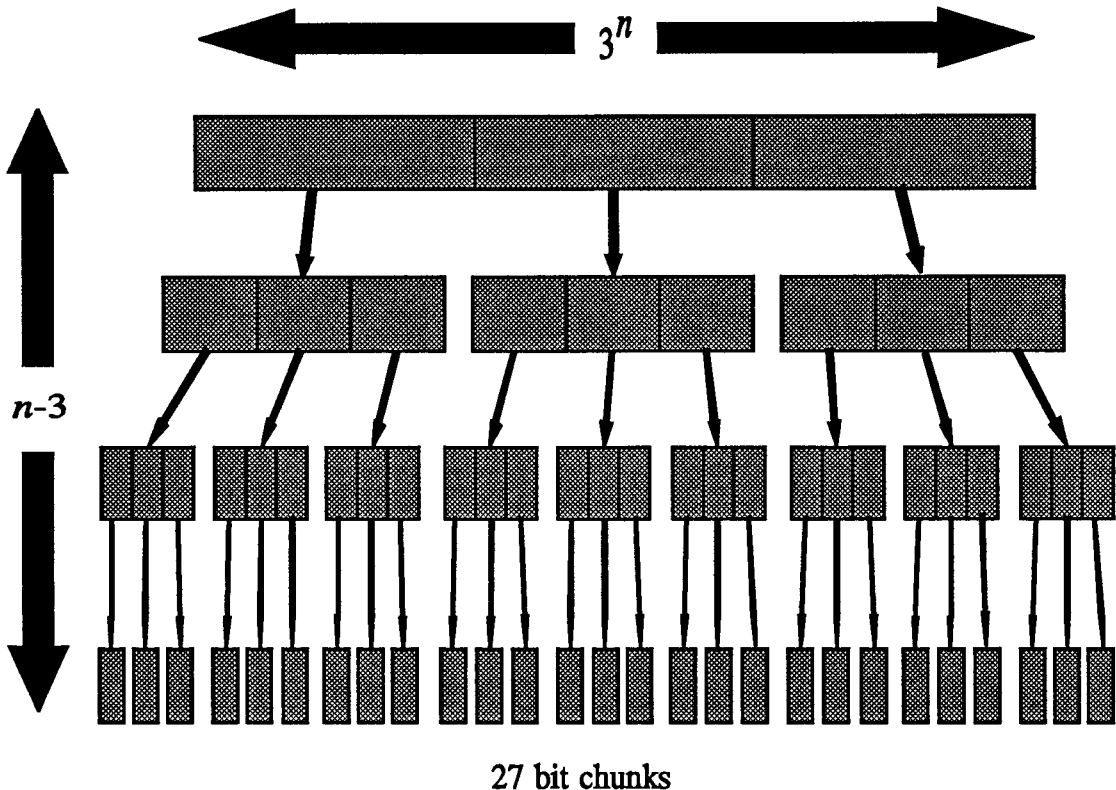


Figure 2. Decomposition of e through recursion when chunk size is 27.

To show that this reformulation performs the same number of operations as the original 'fast' algorithm, mathematical induction will be used to count operations. For the purpose of comparison with Green [5], only additions are counted. Since we are only concerned here with the equivalency of the two algorithms, this is sufficient. More practical measures of efficiency are presented in the next chapter.

THEOREM:

The number of operations performed by the original algorithm is the same as for the reformulated algorithm.

PROOF:For $n=0$:

For both algorithms, this is the trivial case and no operations are performed.

For $n=m+1$:

From Green [5], the original algorithm performs $n3^n$ operations. The reformulated algorithm first computes three partial results. By induction, we can assume that each partial result is computed in $m3^m$ operations. The three partial results are then added together in a loop whose body consists of three additions: the loop is repeated 3^m times. The total count of operations is then:

$$\begin{aligned}
 \text{count} &= 3(m3^m) + 3(3^m) \\
 &= 3(m3^m + 3^m) \\
 &= 3(m+1)3^m \\
 &= (m+1)3^{m+1} \\
 &= n3^n
 \end{aligned}$$

Q.E.D.

From this proof and from the careful derivation of the recursive 'fast' algorithm from the original, we know that the two algorithms perform the same computation. The two algorithms do not, however, perform

the same exact sequence of operations. Figure 3 shows the computation tree for the first element of the weight vector when n is 3.

The number beside an operator node in Figure 3 is the sequence

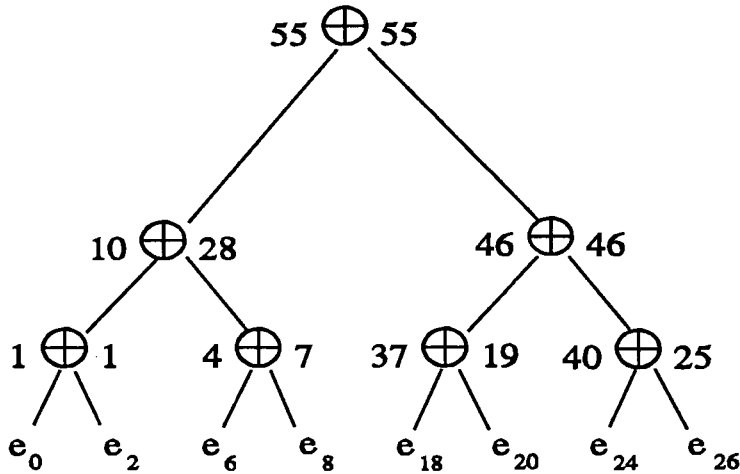


Figure 3. Computation tree for $n=3$.

number of that operation. Those on the left are for the recursive ‘fast’ algorithm and those on the right are for the original ‘fast’ algorithm. This figure shows that not only do the two versions perform these operations in a different order within the 81 total operations each performs, but also in a different order within the computation tree for an individual weight. The order of operations in the recursive ‘fast’ algorithm is based on the order of the recursive calls. These three calls can actually be made in any order; indeed, they can be made simultaneously, as shown in the next section. The order in which the results of the recursive calls are added together is also arbitrary. Note that the computation tree illustrated here is balanced only because there are two ones in each row of P_1 .

PRACTICAL PARALLEL ALGORITHM

As hinted at toward the end of the previous section, the recursive 'fast' algorithm not only lends itself to a practical implementation, it also lends itself to the application of parallel processing techniques. Not only can the three recursive calls be done in parallel, but so can adding together their results. The rest of this section explores these two areas and modifies the recursive 'fast' algorithm for parallelism. First, however, we must think about the model we wish to use for parallel processing.

When developing a parallel processing model, a very practical concern is for the types of computer systems expected to be able to support that model. This, of course, will determine the types of computer systems that will be able to run implementations of the given algorithm. This may seem to some like "putting the cart before the horse", but one goal of this research was to provide a tool for other researchers to use in developing better XOR minimization algorithms, ones that do not rely on brute force. To make the algorithm more portable, I have developed a simple model that is not targeted to any particular hardware, but rather uses two concepts common to several software environments running on many different computer systems: multiple processes and shared memory. The implementation detailed in the next chapter is for the DYNIX operating system running on computers manufactured by Sequent Computer Systems, Inc. This is the

technology on which the symmetric multiprocessor support of AT&T's UNIX System V.4 is based.

To show multiple processes in the C-like pseudo code used above, we will add a construct similar to the COBEGIN ... COEND mechanism used by Brinch Hansen [13] and first proposed by Dijkstra [14]. In this construct, each statement between the COBEGIN and the COEND is executed simultaneously, each by a different process. Since the C language does not support this mechanism directly, I have modified the mechanism to facilitate its implementation. In my scheme, both COBEGIN and COEND take as an argument the number of processes that are to execute the code between them. I have also added a statement prefix: `THREAD(i)`. This indicates that the prefixed statement is to be executed only by the *i*th process of the containing COBEGIN block. Statements not so prefixed will be executed by all processes executing the COBEGIN block. In addition, the variable `THREADNO`, which will be local to the COBEGIN block, will be set to the number of the "current" process within that block. The value of `THREADNO` will range from 0 to *i*.

In most operating systems, multiple processes do not automatically access the same memory. For example, when a process is created in the UNIX operating system by the `fork()` system call, the new process executes the same program code, but all the data of the original (parent) process is copied for use by the new (child) process. Many operating systems that

support multiple processes also support shared memory. Shared memory can exist in the address space of more than one process, each of which can access the shared memory as they would any other memory. This could lead to synchronization problems if two or more processes try to update the same memory at the same time. As shown later, this is not a problem here. Only the output vectors of the M and P transforms need to be allocated from shared memory. This will be indicated in the C-like pseudo code by preceding the appropriate formal parameter with the key-word SHARED. This leads to the following parallel algorithm for the P transform:

```

fast_p(lvl, SHARED *w_out, *e_in) {
1)   IF (lvl == 0) {*w_out = *e_in; RETURN;}
2)   c = pwrof3[lvl-1];
3)   COBEGIN(3);
      THREAD(0) fast_p(lvl-1, w_out, e_in);
      THREAD(1) fast_p(lvl-1, w_out+c, e_in+2*c);
      THREAD(2) fast_p(lvl-1, w_out+2*c, e_in+c);
      COEND(3);
4)   nproc = pwrof3[lvl];
5)   COBEGIN(nproc)
      FOR (i=THREADNO; i<c; ++nproc) {
          temp = w_out[i];
          w_out[i] += w_out[i+c];

```

```

w_out[i+c] += w_out[i+2*c];
w_out[i+2*c] += temp; }

```

```
COEND(nproc);
```

```
6) RETURN; }
```

This is still basically a recursive algorithm. Now, however, each recursive call is executed by its own process. For a switching function with n inputs, there will be 3^{n-m} such calls and processes, where there are 3^m bits per chunk. At each level of recursion, once the three recursive calls are made, their results must be added together. In theory, this could be done using a new process for each addition to be performed. At the bottom most level of recursion, performed by 3^{n-m} processes, no adding together is needed, so no additional processes are needed. At the next higher level, performed by 3^{n-m-1} processes, three adds are done to sum the results of the recursive calls, needing three processes. The total number of processes in use at this level is then $3 \times 3^{n-m-1}$ or 3^{n-m} . This argument can be applied at succeeding higher levels, up to the top level, showing that throughout the execution of the algorithm, 3^{n-m} processes are used.

To minimize execution time, each process should be executed by its own processor. In fact, if a process is not run on its own processor, then the overhead it takes to create and coordinate that process is wasted. A way is needed to limit the number of processes used so that it matches the number of processors available. The most practical way to do this is to limit the

number of recursion levels that use the parallel algorithm. After this limit has been reached, the recursive 'fast' algorithm is used. Step 4 would also have to be changed to $nproc = pwr of 3[\max(lvl-(n-limit),0)]$, where *limit* is the number of recursion levels which use the parallel 'fast' algorithm.

A concern central to the design of parallel algorithms is that of synchronization. A process must not try to use the value contained in some memory location before the value has been placed there by some other process. Many mechanisms have been developed for process synchronization [13], but only the simple synchronization provided by COEND is needed here. This is because, at each recursion level, the results of a single recursive call are not used until all three recursive calls are completed. Put another way, each recursive call, executing in its own process, writes to its own third of the portion of the w vector (the only shared data) that the given recursion level is responsible for computing.

Complexity analysis can be used to compare the algorithm presented in this section with the algorithm presented in the previous section. Again, the algorithms being compared are for the P transform; the analysis for the M transform would be similar.

From Green [5], the operation count for the 'fast' algorithm is $n3^n$. We must, however, take into account the chunky bit vectors. Once the recursion gets to the point where the part of e being worked on fits in a single chunk, then the direct algorithm is used for that part. Again from

Green [5], the operation count for the direct algorithm is $3^n(2^n-1)$. If 3^c is the number of bits per chunk, then the number of operations for our "chunked" 'fast' algorithm is $(n-c)3^{(n-c)}+3^{(n-c)}3^c(2^c-1)=((n-c)+3^c(2^c-1))3^{(n-c)}$. This is larger than the operation count for the "no chunk" 'fast' algorithm, so one may ask why we use chunky bit vectors at all. The answer is that they are used for two reasons: storage efficiency and computational efficiency. The latter reason seems to be in conflict with the operation counts only because they do not take into account the number of multiply (AND) operations, which are greatly reduced in the "chunky" case.

The parallel algorithm can be thought of as solving several smaller problems simultaneously and then adding the results together. If l is the number of recursion levels that use the parallel algorithm, then the total operation count for one of the smaller problems will be $((n-l-c)+3^c(2^c-1))3^{(n-l-c)}$. At the l th level of recursion, the three processors that were used to solve the three smaller problems are now available to add up the results. At the $l-1$ 'th level, there are 9 processors available to add up the results. And so on until at the top level, there are 3^l processors available. The operation count for recursion level i , $1 \leq i \leq l$, is $3^{n-i} / 3^{l-i+1} = 3^{n-l-1}$. Combining these counts, the total operation count for the parallel algorithm is $l3^{n-l-1}+((n-l-c)+3^c(2^c-1))3^{(n-l-c)}$.

By comparing the operation count for the algorithm presented in this section with that of the algorithm presented in the previous section we can

see that as n gets larger, both algorithms are equally bad. The parallel algorithm is, however, about 3^{l+1} times faster than the non-parallel algorithm. This doesn't take into account the overhead (processes creation, etc.) incurred by the parallel algorithm. This overhead would cause the parallel algorithm to be slower for values of n less than some small number, depending on l .

CHAPTER IV

EXPERIMENTAL RESULTS

OVERVIEW

The previous chapter developed a practical algorithm for finding minimum KRM, and fixed-polarity RM expansions. In this chapter, that algorithm is analyzed by timing the execution of programs based on the algorithm over a wide range of problem sizes. Because each is an exhaustive search of the solution space, all versions of the algorithm are data independent. This means that every switching function of a given number of input bits requires the same amount of time to minimize.

Each section in this chapter analyzes different versions of the algorithm as follows:

- The "Chunky Performance" section of this chapter analyzes the effect that the size of a chunk has on performance. The algorithm implemented for this analysis is the direct algorithm. Different versions were tested corresponding to three different chunk sizes: 8 bits, 16 bits, and 32 bits. These results are analyzed to predict performance for other chunk sizes.

- The "How Fast is 'Fast'" section of this chapter analyzes the performance of the 'fast' algorithm. The object of the analysis is to characterize the maximum performance of the algorithm. For that reason, a chunk size of 32 bits is used.
- The "Parallel Performance" section of this chapter analyzes the effect of using multiple processes. The practical parallel algorithm is implemented with a chunk size of 32 bits. A command line option is used to select the number of processors to use: one, three, or nine. The timing results of the single process are compared to the timing results for the 'fast' algorithm. Using an approach similar to that used in the first section of this chapter, the timing results using one, three, or nine processors are analyzed to predict performance of the algorithm with more processors.

CHUNKY PERFORMANCE

This section analyzes the effect of using chunky bit vectors on the performance of the direct algorithm. Three different versions of the program were compiled, each with a different word size: 8-bits, 16-bits and 32-bits with corresponding e vector chunk sizes of 3, 9, and 27 bits. Switching functions with sizes ranging from $n=3$ to $n=15$ literals were used as inputs. For each run, the M and P transformations were timed and the total CPU execution time (user and system) of the program was recorded.

These times were taken using the process profiling interval timer, which should not be affected by the number of users on the system. In each case, the system was a Sequent S81 computer with more processors than users when the test was performed and each test was performed only once. Table I gives the results, in seconds.

TABLE I
EXECUTION TIMES FOR THE DIRECT ALGORITHM

n	8-bit words			16-bit words			32-bit words		
	M	P	Total	M	P	Total	M	P	Total
3	0.00	0.00	0.02	0.001	0.001	0.001	0.001	0.001	0.001
4	0.00	0.01	0.07	0.01	0.00	0.04	0.001	0.001	0.001
5	0.01	0.02	0.04	0.01	0.03	0.06	0.00	0.00	0.02
6	0.01	0.10	0.17	0.00	0.03	0.09	0.01	0.02	0.06
7	0.02	0.58	0.66	0.01	0.16	0.20	0.03	0.09	0.15
8	0.03	3.44	3.52	0.08	1.54	1.68	0.04	0.41	0.50
9	0.08	22.68	23.04	0.06	6.68	6.94	0.10	2.11	2.41
10	0.33	198.67	199.38	0.18	41.66	42.25	0.21	11.41	11.89
11	1.13	1168.09	1169.96	0.50	280.79	282.19	0.45	65.34	66.38
12	9.78	5066.17	5077.78	1.75	1712.74	1716.99	1.08	383.18	385.95
13	16.79	34629.66	34655.36	9.63	10908.47	10926.38	2.91	2750.00	2759.42
14				23.39	48293.53	48342.19	8.80	17268.38	17293.00
15							29.83	109211.39	109302.26

The first thing one notices when looking at Table I is that not all the positions in the table have values. This is because the three computer runs associated with the empty table positions did not complete. For these runs,

the combination of their large demands for virtual memory and their CPU-intensive nature triggered a bug in the DYNIX operating system that caused these runs to become permanently "swapped out" [15].

Another thing apparent from Table I is that as n grows larger, the time taken for the P transform starts to dominate the program execution time. The program can be broken into roughly three areas: input/output, the M transform, and the P transform. The time needed for input/output is linear in n and therefore small compared to the other two areas. It will be ignored. The time needed for the M transform increases as $O(2^n)$. This does not increase nearly as fast as the time needed for the P transform, which increases as $O(3^n)$. For this reason, the M transform will generally be ignored for the rest of this chapter.

The main question to be answered in this section is: "How does chunk size affect performance?" A qualitative answer, based on Table I, is: "The larger the chunk size, the higher the performance." For example, for a problem of size $n=11$, performance was increased by about a factor of 4 each time the word size was doubled. This is deceptive because the next doubling of the word size, from 32 to 64, would not increase performance because the next appropriate chunk size for the P transform is 81 bits requiring a word size of 128 bits.

A quantitative analysis would not lead to confident answers because of the small number of samples: only three different chunk sizes were used.

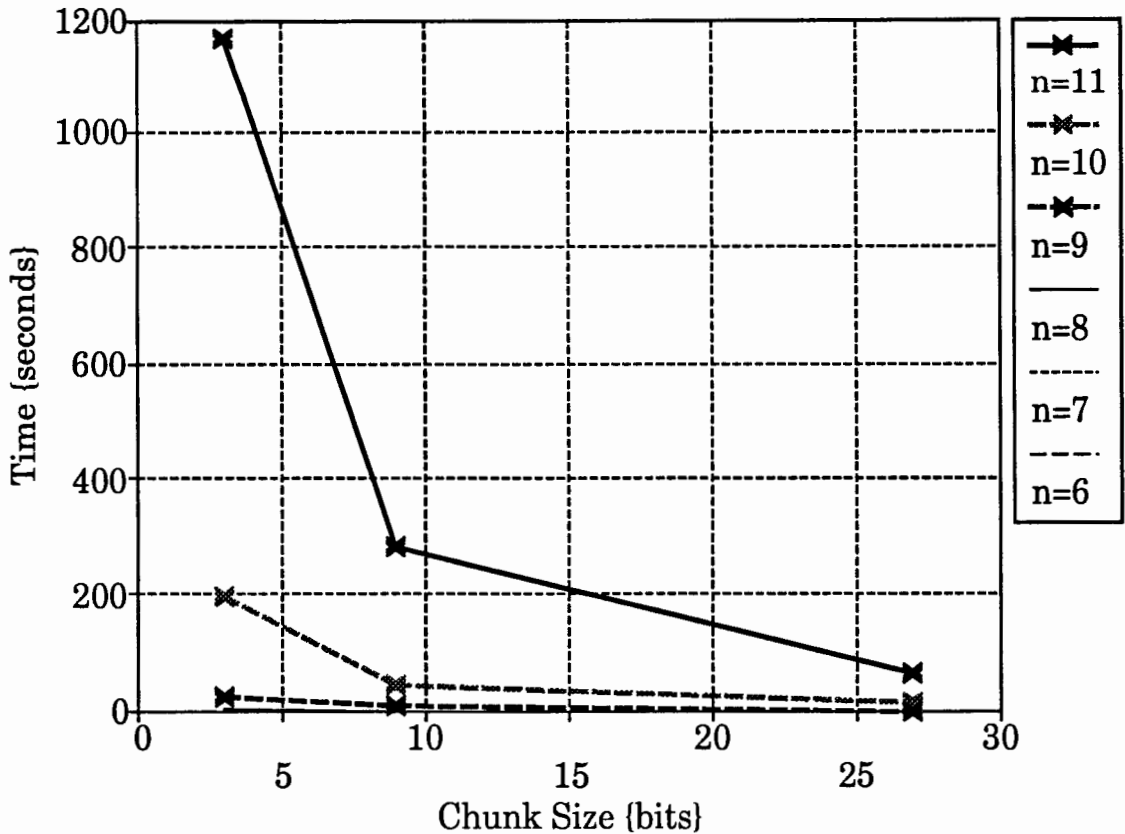


Figure 4. Execution time of P transform vs chunk size.

We can, however, use quantitative techniques to give us a qualitative "feel" for the data. For example, the graph in Figure 4 shows execution time of the P transform vs. the chunk size. The time is plotted for six different values of n ranging from 6 to 11. As can be seen, the curves for the largest three values of n are distinguishable. The curves for the smallest three are all muddled together at the bottom of the graph. This fact, along with the general shape of the curves, suggests that there might be some kind of exponential function involved.

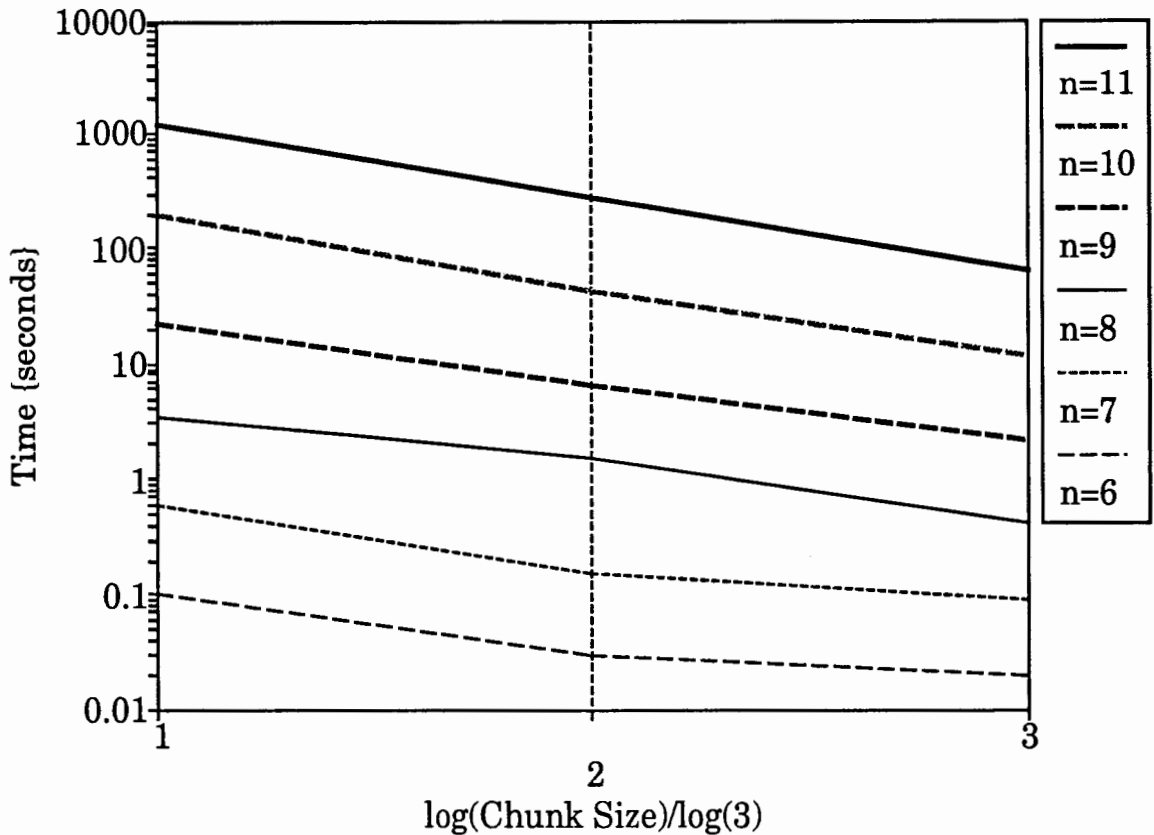


Figure 5. Execution time of P transform vs $\log_3(\text{chunk size})$.

The graph of Figure 5 again plots execution time on the y axis, but this time on a logarithmic scale. The \log_3 of the chunk size is plotted on the x axis. The first thing to notice about this graph is that the curves are nearly straight lines, especially for the larger values of n . This would suggest the following equation:

$$\text{time} = \exp(c + m \log_3(\text{chunksizes}))$$

where c is a constant and m is obviously negative. Another thing to notice about the graph is that the curves seem to be evenly spaced from each other. This suggests that the constant in the previous equation is a linear

function of the value of n , the number of literals. This would suggest the following equation:

$$\text{time} = \exp(c_0 + m_0 n + m_1 \log_3(\text{active bits}))$$

We could use curve fitting techniques to find values for c_0 , m_0 , and m_1 , but because we have so few points, we would not have much confidence in those values, especially for m_1 . In the next section, we will have enough points to use curve fitting techniques with more confidence.

HOW FAST IS 'FAST'?

This section analyzes the performance of the 'fast' algorithm. In place of the direct algorithm used in the previous section, the 'fast' algorithm was implemented on the Sequent S81. Another change that limited the size of problems to be minimized to $n \geq 5$, stems from the fact that a word size of 32 bits was used. A truth vector d , that fills a complete chunk, contains 2^5 bits. The algorithm was implemented in such a way that only whole chunks could be used. If a problem of size $n < 5$ is input to the program, the problem is solved by expanding it to a 5 bit problem.

The program was run with input problems ranging in size from $n=5$ to $n=14$. Larger problems ran into the same system bug as reported in the previous section. As before, the time taken by the M and P transformations for each run were recorded, along with the total execution time. The results are shown in Table II, again in seconds.

TABLE II
EXECUTION TIMES FOR THE 'FAST' ALGORITHM

	32-bit words		
n	M	P	Total
5	0.03	0.03	0.09
6	0.05	0.08	0.25
7	0.11	0.23	0.45
8	0.21	0.7	1.05
9	0.42	2.12	3.21
10	0.85	6.55	8.43
11	1.7	20.24	24.24
12	3.44	62.31	72.04
13	7.03	192.15	223.16
14	14.48	591.5	666.36

The first thing one notices when comparing Table I with Table II is that the 'fast' algorithm is actually slower than the direct algorithm for $n < 9$ but much faster for $n > 9$. This is best shown by plotting the execution times of the P transform, with a chunk size of 32 bits, for both algorithms. This is done in the graph in Figure 6. The graph in Figure 7 is of the same data, but with a logarithmic scale used for the y axis.

Why would the direct algorithm be faster than the 'fast' algorithm for $n < 9$? The answer is that it's not the algorithm that's faster, it's the implementation. For $n < 9$, the advantage of the 'fast' algorithm is lost to the extra overhead needed to implement it. For $n > 9$, the overhead be-

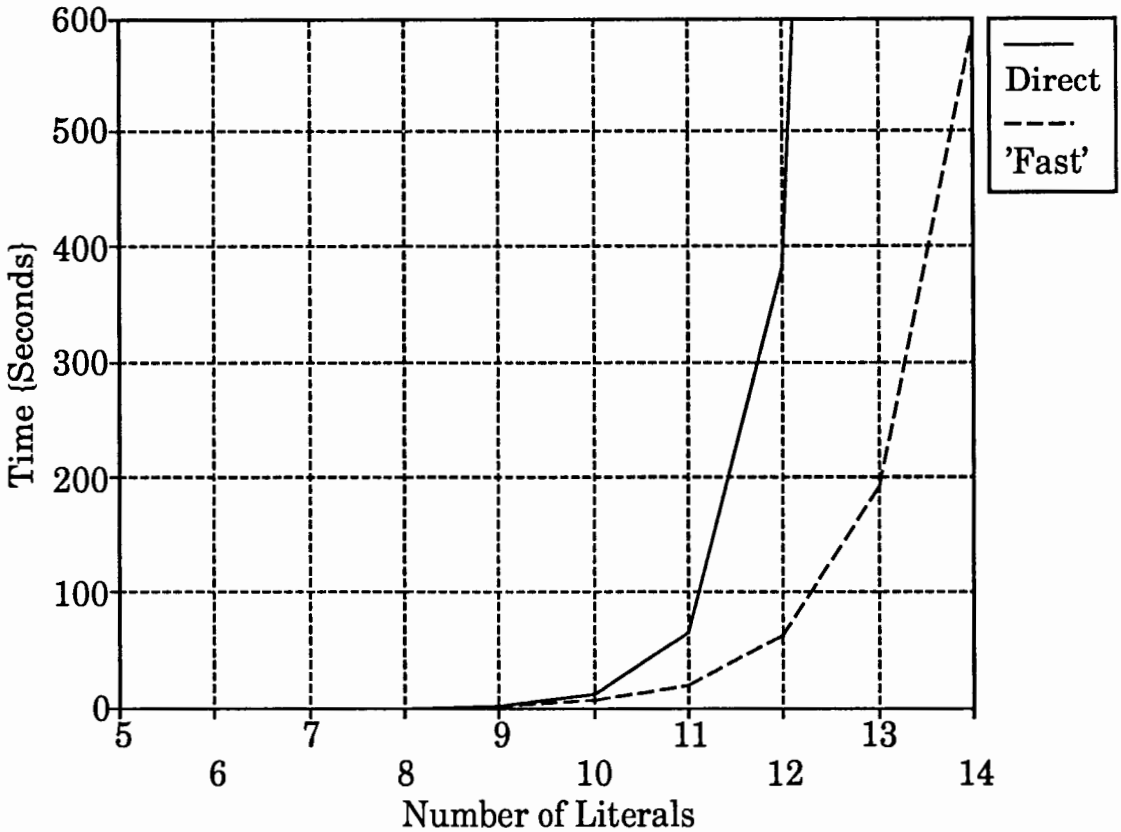


Figure 6. P transform time: direct and 'fast'.

comes smaller and smaller when compared to the computational advantage of the 'fast' algorithm. A program could be written that uses the direct algorithm for $n < 9$ and the 'fast' algorithm for $n > 9$, but the time saved would be small.

We can see that the 'fast' algorithm is indeed faster than the direct algorithm, but how much faster? If we define speed-up to be the ratio of the time needed by the direct algorithm to the time needed by the 'fast' algorithm, it is clear from Figure 7 that speed-up is a function of n . In fact, because the plots in Figure 7 are nearly straight lines, speed-up should be

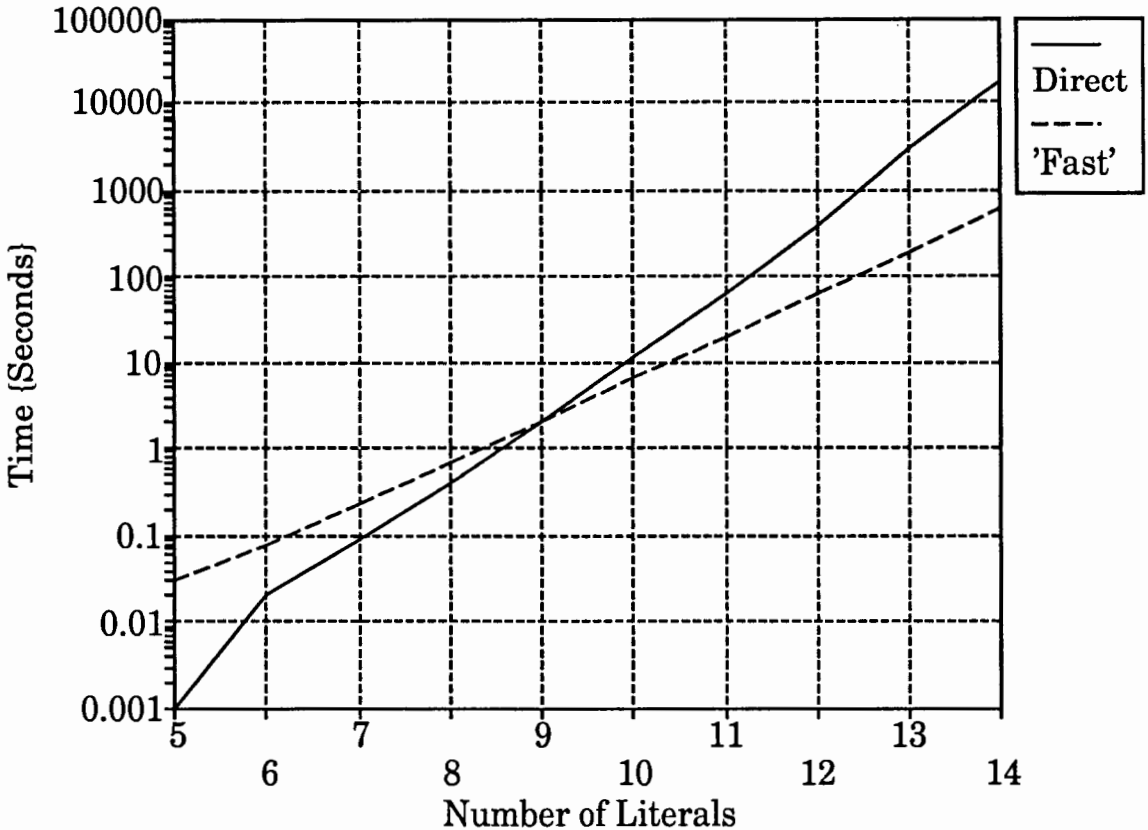


Figure 7. P transform time: direct and 'fast' (log scale).

"close" to an exponential function of n . Figures 8 and 9 plot speed-up vs n along with an exponential curve that tries to fit the data.

To find the "fit curve" in Figures 8 and 9, only the last 5 data points were used in the regression analysis. The first points were not used so that the resultant curve would fit the later points better. The equation for the fit curve is:

$$\text{speed up} = \exp(-6.64 + 0.71n)$$

We use this equation to predict speed-up outside the range of possible experiments. For example, we predict that for $n = 15$, the 'fast' algorithm

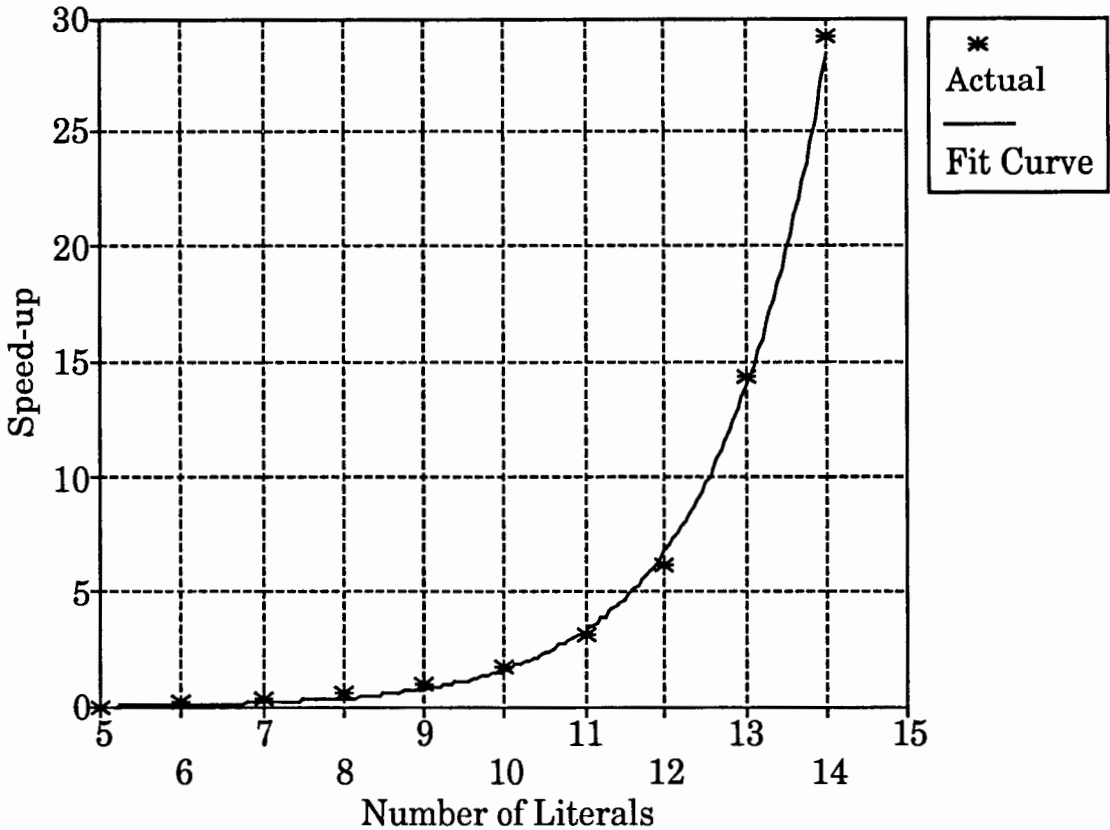


Figure 8. Speed-up vs n .

will be about 55 times faster than the direct algorithm.

While it might be useful to know this, it would be more useful to predict the execution time of the 'fast' algorithm itself. Figures 10 and 11 plot the P transform time vs n , the number of literals, along with an exponential curve fit to that data. For the regression analysis used to fit this curve, only times for $8 \leq n \leq 13$ were used. As seen in Figure 9, the overhead overshadows the execution time for the smaller values of n . $n = 14$ was not used because of a previously mentioned system bug for a problem of that size. The equation of this curve is:

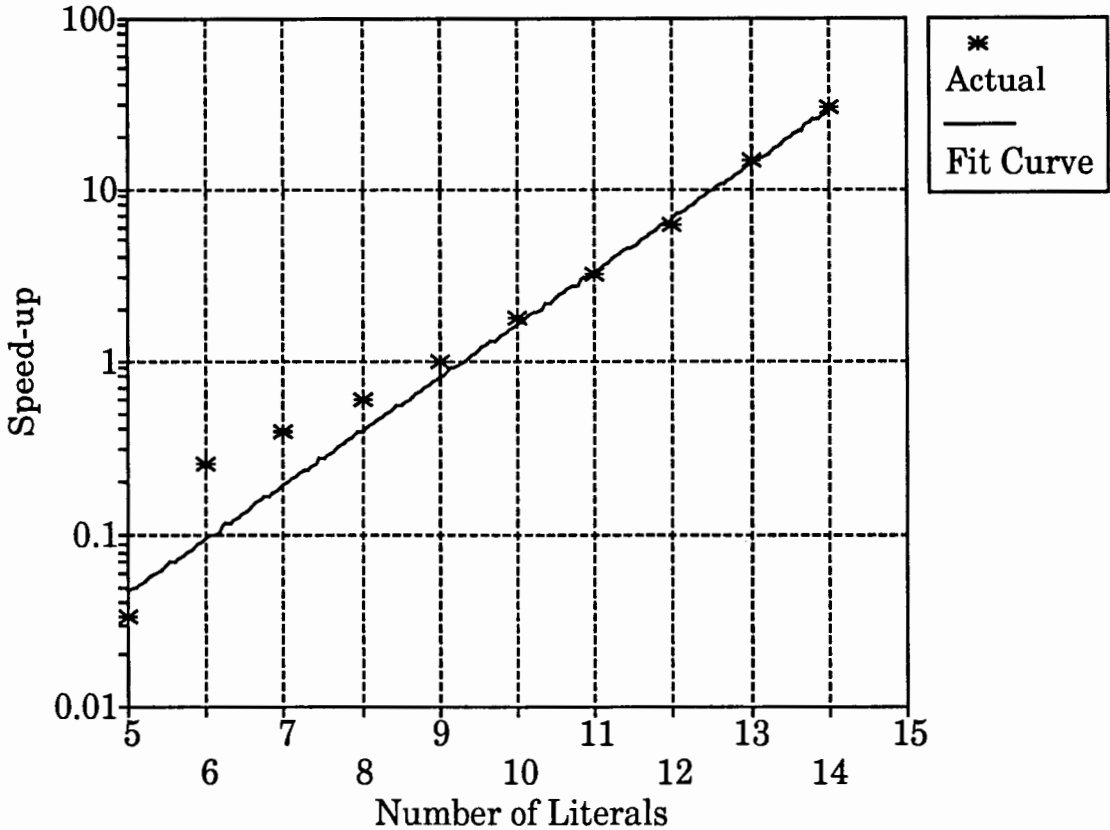


Figure 9. Speed-up vs n (log scale).

$$t = \exp(-9.358 + 1.124n)$$

Using the equation, we can predict that a 24 input switching function can be minimized in 518 days. Given the determination of the data used in this equation, it might be more reasonable to estimate a couple of years. This assumes, of course, that we can find a computer with more than 565 gigabytes of memory to run the program on. In the next section, we analyze execution time using multiple processors.

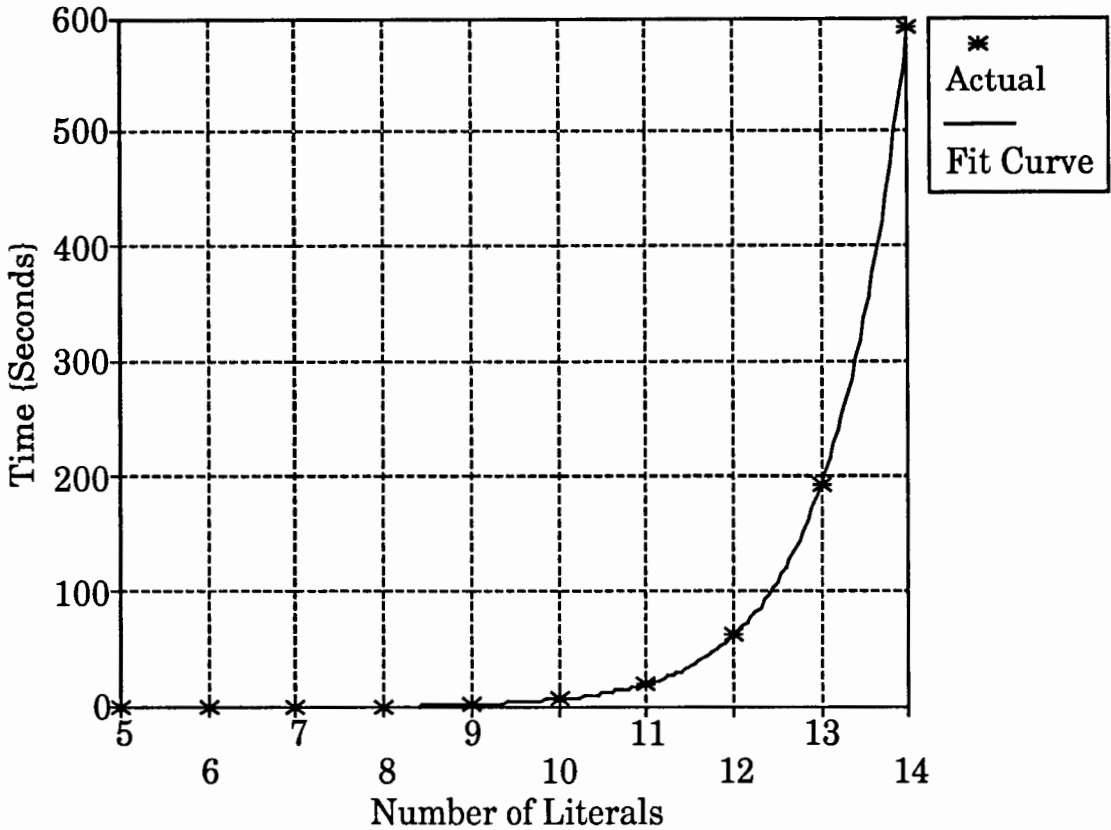


Figure 10. 'Fast' P transform time vs n .

PARALLEL PERFORMANCE

In this section we analyze the effect that the number of processors has on the performance of the practical parallel algorithm. This is a very similar analysis to that of the "Chunky Performance" section of this chapter. In that section, the effect of implicit parallelism was analyzed. In this section, explicit parallelism is analyzed.

The program used in the previous section was modified to support the use of multiple processors, as suggested in Chapter III. This program was

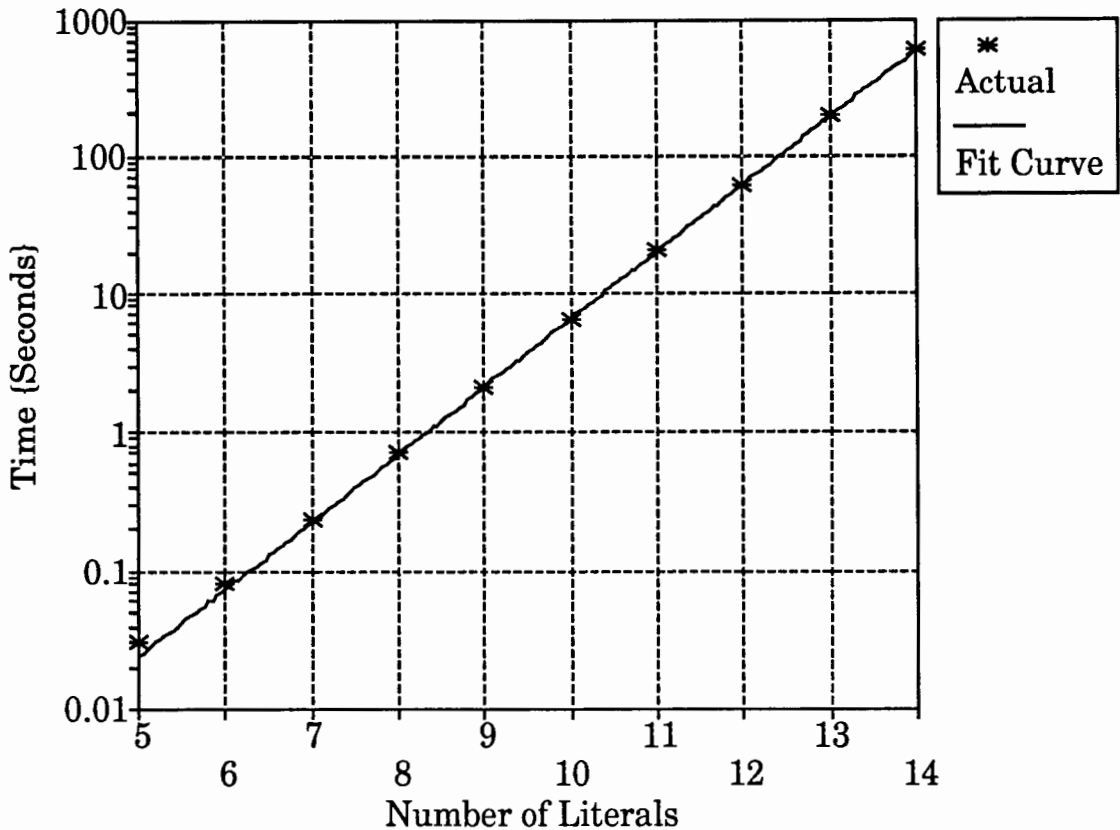


Figure 11. 'Fast' P transform time vs n (log scale).

then run using 1, 3, and 9 processors with input problems ranging in size from $n=5$ to $n=15$. The real elapsed time of the P transform was measured. These results, along with the equivalent times for the 'fast' algorithm, are given in Table III.

First, let's look at a graph of these times vs n , the size of the problem. This graph, with time on a log scale, is shown in Figure 12. There are several things that are apparent from this graph. The first is that for $n < 7$, the 'fast' algorithm is faster than the parallel one, no matter how many processors are used. This was predicted by the complexity analysis at the

TABLE III
EXECUTION TIMES FOR THE PRACTICAL PARALLEL ALGORITHM

n	fast	1	3	9
5	0.03	0.12	0.23	0.53
6	0.08	0.16	0.25	0.52
7	0.23	0.32	0.31	0.57
8	0.7	0.79	0.46	0.59
9	2.12	2.27	0.99	0.84
10	6.55	6.85	2.56	1.46
11	20.24	20.85	7.35	3.12
12	62.31	64.3	22.13	8.51
13	192.15	198.08	67.72	24.91
14	591.5	630.87	207.87	75.74
15		2495.65	872.79	518.47

end of chapter III and is due to the fact that for these smaller problems, the time spent in process management is significant. Another thing to notice is that for $n > 9$, the plot for a single processor overlays the plot for the 'fast' algorithm. This is expected because, except for the overhead of process management, the two algorithms are the same.

One last thing to notice about the graph in Figure 12 is that as n gets larger, the curves for the parallel algorithm start looking like straight lines that have the same slope, but different intercepts. This is not a surprise

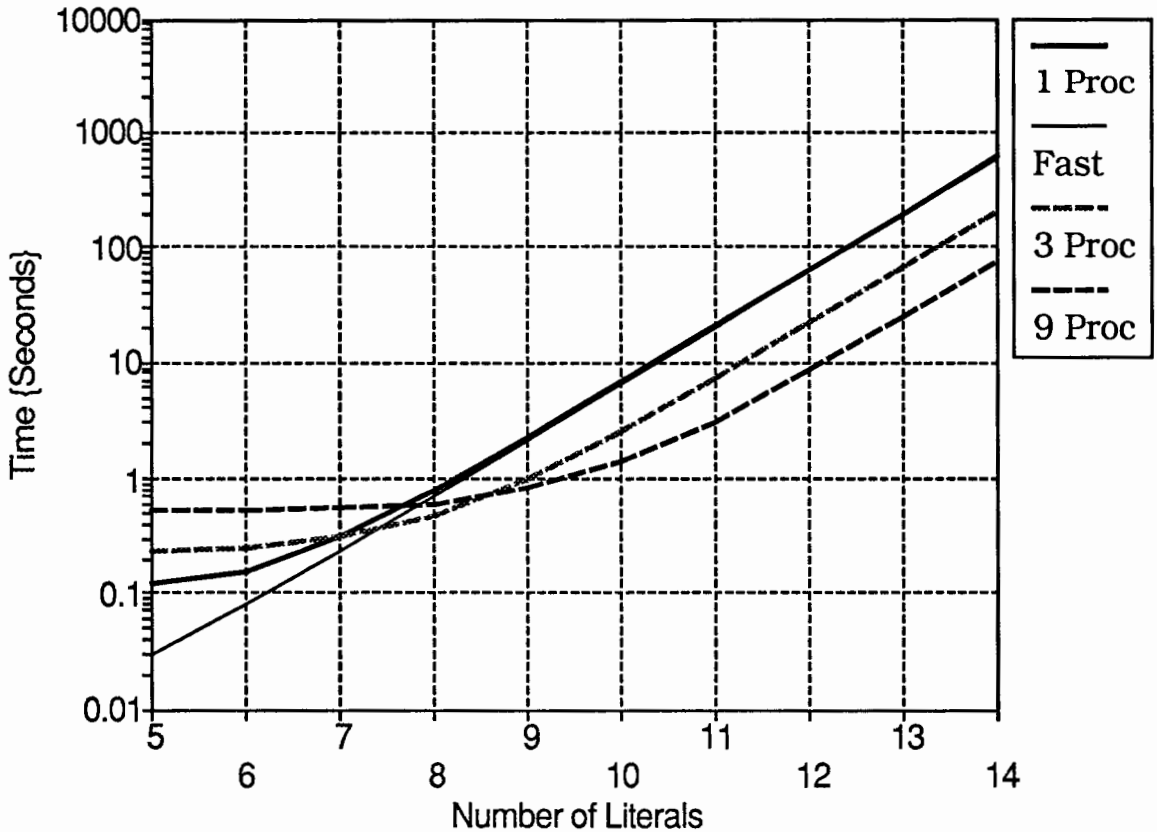


Figure 12. Parallel P transform time vs n (log scale).

because we saw in the previous section that the execution time of the 'fast' algorithm was an exponential function of n . What may be a little hard to see is that the distance between the intercepts is not constant, but decreases as the number of processors increases. This agrees with Amdahl's equation [16] which suggests that as the number of processors increase, the marginal increase in performance decreases.

In fact, based on the analysis in the "How Fast is 'Fast'" section of this chapter, we might expect execution time to be an exponential function of the $\lfloor \log_3 \rfloor$ of the number of processors. The floor of \log_3 is used because

performance is increased only when the number of processors is increased to the next higher integer power of three. If the number of processors is not an integer power of three, then at some level of recursion, the three partial results are computed using an unequal number of processors, some of which will block, waiting for the completion of the partial result with fewer processors. For this reason, tests were run using one, three, or nine processors. A Sequent system with 27 processors was not available. The graph in Figure 13 plots the execution time of the P transform vs the \log_3 of the number of processors for $9 \leq n \leq 14$.

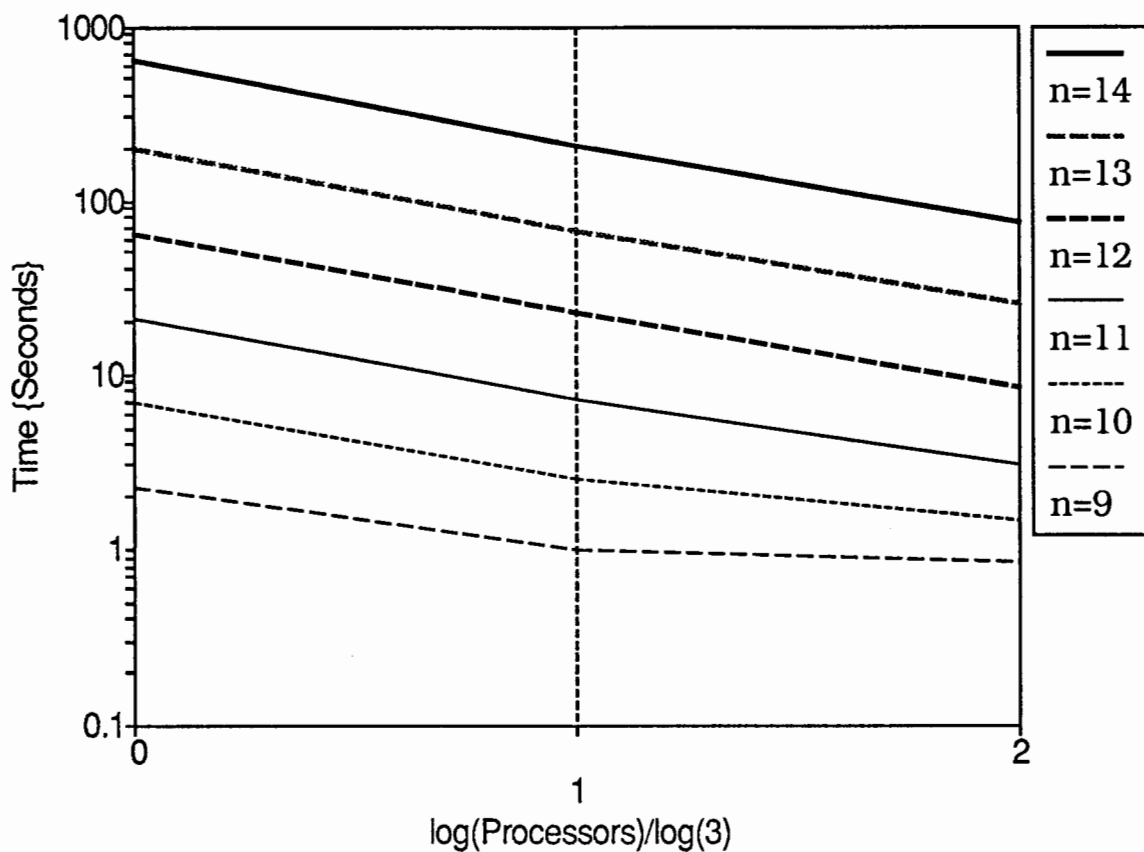


Figure 13. P Time vs $\log_3(\text{processors})$.

This graph looks very similar to the graph in Figure 5 from the "Chunky Performance" section of this chapter. Following the same reasoning as used in that section, we would expect the elapsed time for execution of the parallel algorithm for the P transform to be related to the number of processors by the following equation:

$$\text{time} = \exp(c_0 + m_0 n + m_1 \lfloor \log_3(\text{processors}) \rfloor)$$

As in the "Chunky Performance" section of this chapter, curve-fitting techniques can be used to find values for the constants, but there are not enough data points to have much confidence in those values. We could, however, use them to find "ballpark" estimates of the P transform times when using 27 processors.

First we fit exponential curves to the data plotted in Figure 13. This gives the following equations:

$$\text{time}_{n=14} = \exp(6.430 - 1.060 \lfloor \log_3(\text{processors}) \rfloor)$$

$$\text{time}_{n=13} = \exp(5.276 - 1.037 \lfloor \log_3(\text{processors}) \rfloor)$$

$$\text{time}_{n=12} = \exp(4.145 - 1.011 \lfloor \log_3(\text{processors}) \rfloor)$$

$$\text{time}_{n=11} = \exp(3.006 - 0.950 \lfloor \log_3(\text{processors}) \rfloor)$$

$$\text{time}_{n=10} = \exp(1.854 - 0.770 \lfloor \log_3(\text{processors}) \rfloor)$$

Figures 14 and 15 repeat the previous two graphs, but with the addition of the points predicted by the above equations. Notice in these equations that the slopes of the linear arguments to the exponential functions are all close to the same value. We assign their mean value -1.014 to be the constant m_1 . Now notice that the intercepts of the linear arguments to the exponen-

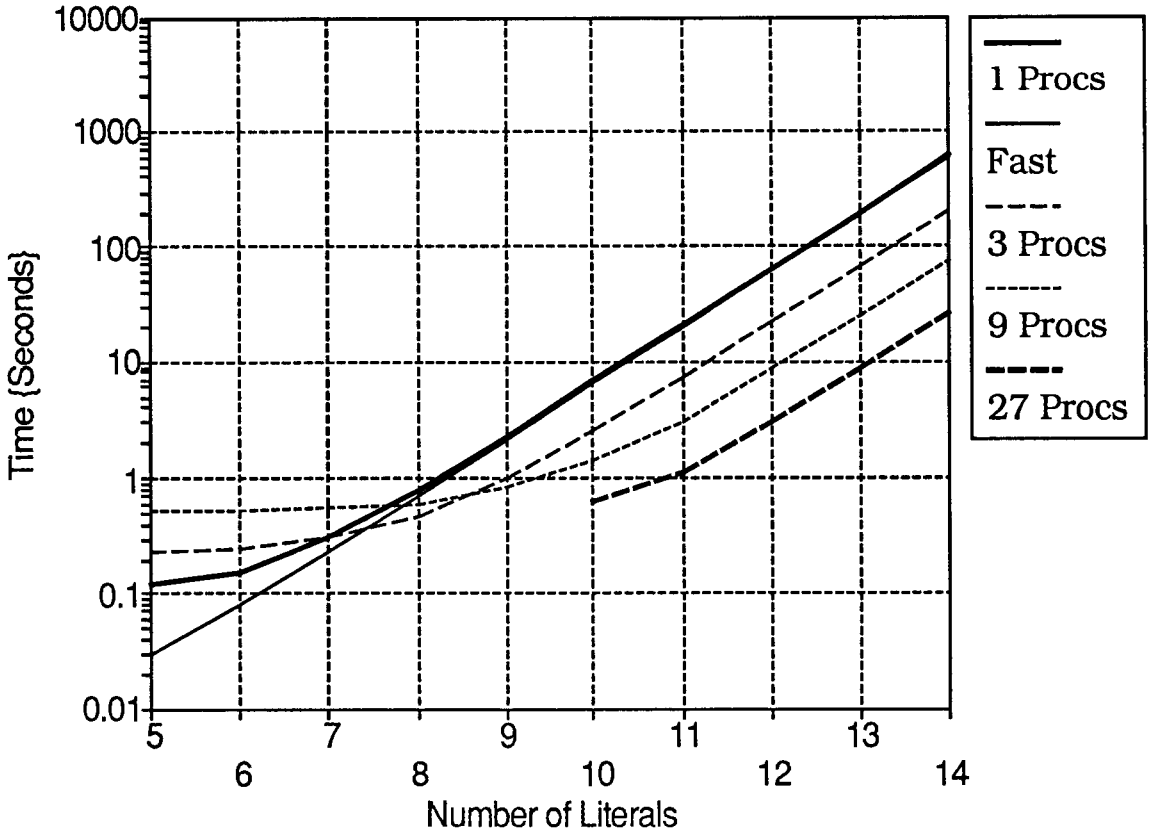


Figure 14. Predicted P time vs n (log scale).

tial functions seem to form a linear function of n . Using regression, we find the slope and intercept of that function as $m_0=1.142$ and $c_0=-9.565$ respectively. Filling these constants into the first equation in this section gives:

$$\text{time} = \exp(-9.565 + 1.142n - 1.014|\log_3(\text{processors})|)$$

We would like to use this equation to make "ballpark" predictions of the performance of the parallel algorithm using 27 processors. This is done in the graphs shown in Figures 16 and 17, which are the same as the previous two graphs, except that the single equation above is used for all the predict-

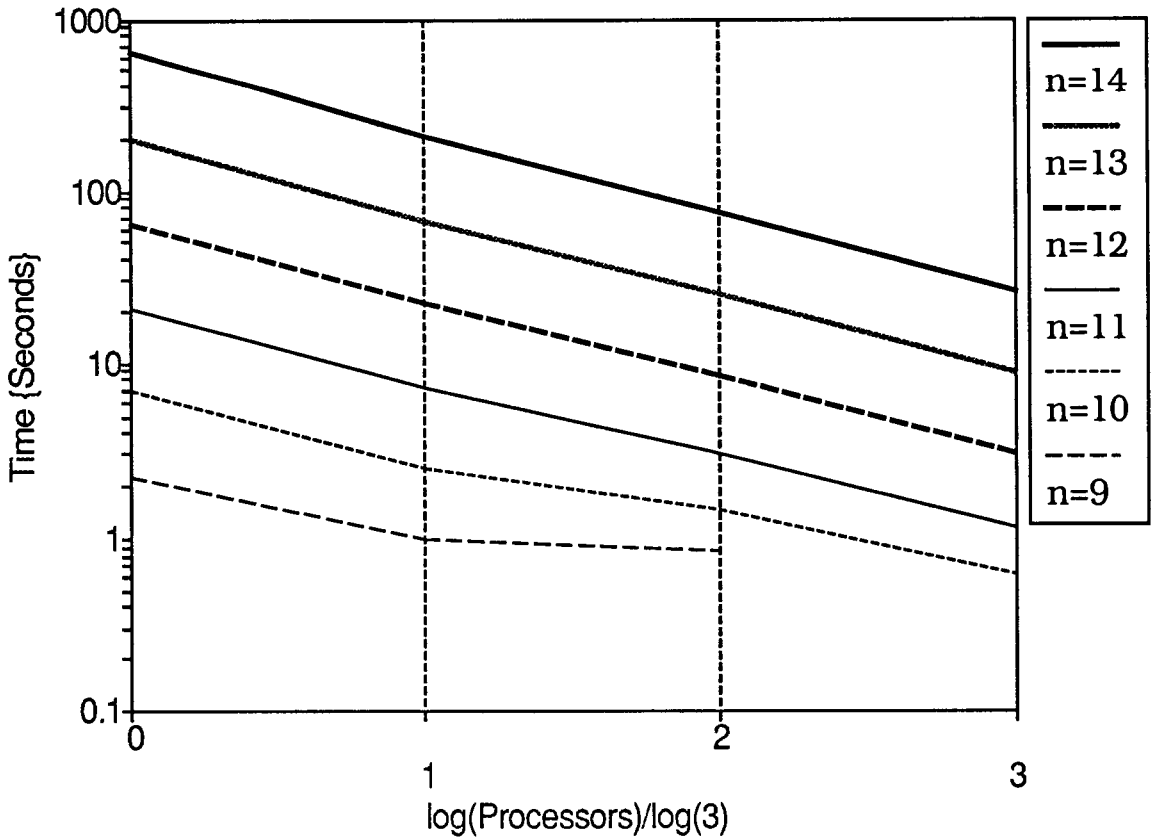


Figure 15. Predicted P time vs $\log_3(\text{processors})$.

ed points. The predicted values for 27 processors seem to fit well with the measured data, so we can use this equation to make really wild guesses, such as: a minimum KRM can be found for a switching function with 24 inputs in a couple of days using 729 processors. This is not a very reliable prediction, but it does suggest a line of future research, as discussed in the next chapter.

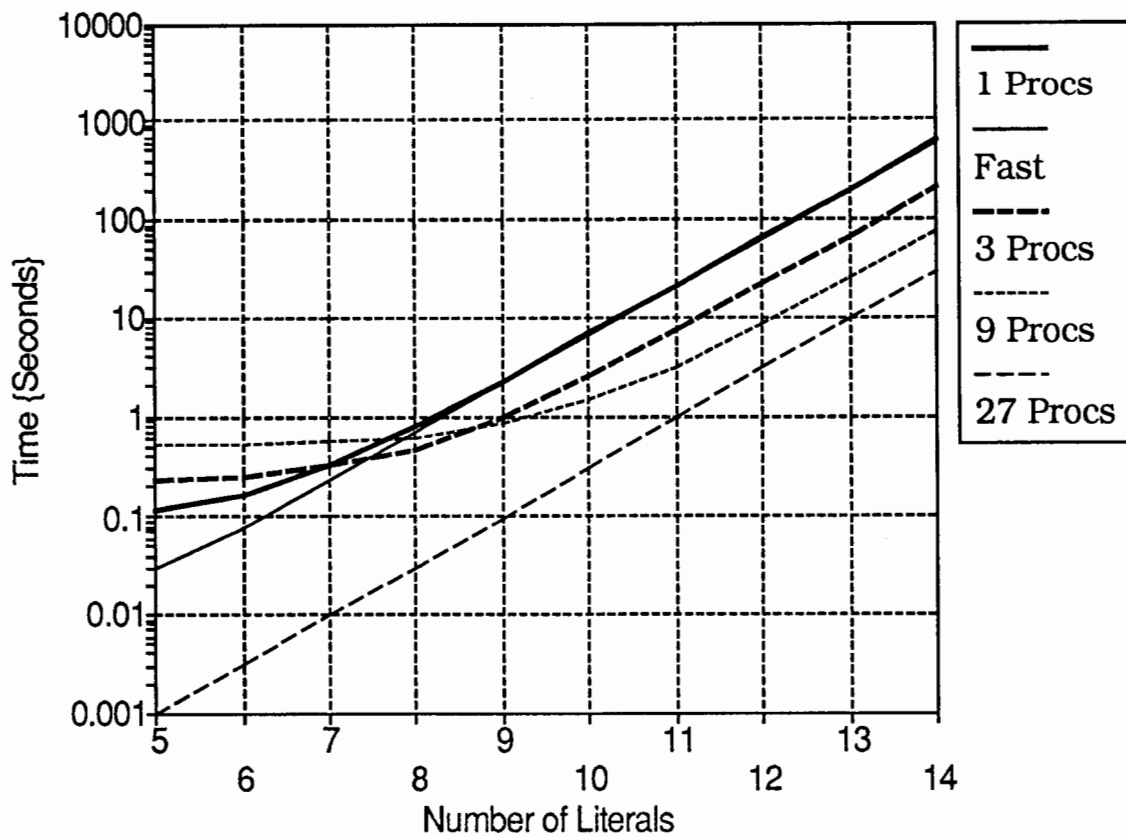


Figure 16. Unified prediction: t vs n .

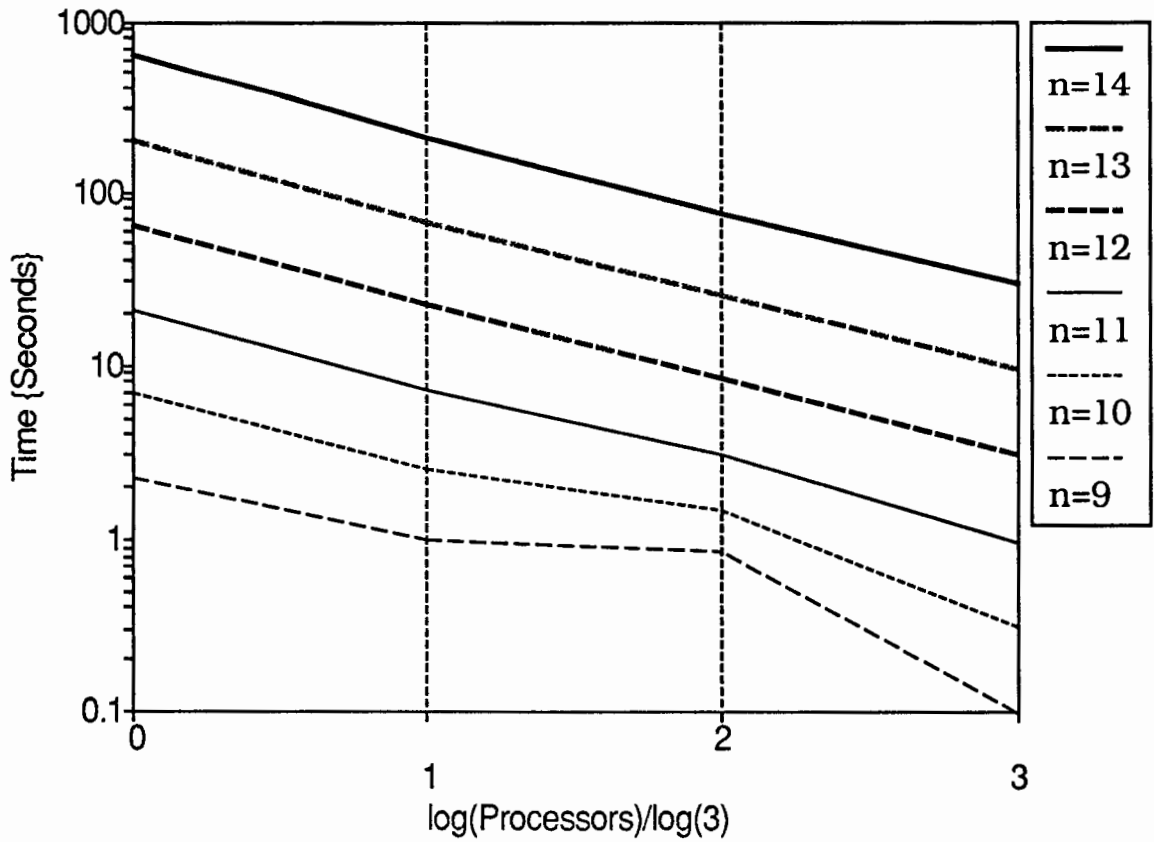


Figure 17. Unified prediction: t vs $\log_3(\text{processors})$.

CHAPTER V

EXAMPLE USES OF KROMIN

OVERVIEW

In the first chapter of this thesis, we stated that for kromin to be a good tool, it should be easy to use and do its job well. We went on to say that kromin's job was to help researchers develop better algorithms to minimize ESOPs and gave two ways it could be used in doing so: by serving as a "base line" for comparison and by helping to provide insight into the problem. The former use of kromin was developed in Chapter IV: the experimental results of that chapter can be used for comparison as newer algorithms are developed. Three examples of the latter use of kromin are presented in this chapter:

- In the "Cohn's Conjecture and the Distribution of Weights" section of this chapter, kromin is used to examine the distribution of weights within the context of a conjecture made by Cohn [17].
- In the "Expansion on Work by Sasao and Besslich" section of this chapter, kromin is used to add to work reported by Sasao [4].
- In the "Incompletely Specified Functions" section of this chapter, a modified version of kromin is used to compare two different ap-

proaches to finding minimum KRMs for incompletely specified switching functions.

These example uses of `kromin` are meant to demonstrate how the tool can be used and to add support to the conclusion presented in the next chapter that `kromin` is a useful tool for the development of ESOP minimization algorithms. Each of these examples represent an avenue of research that could prove fruitful. However, only initial steps are take here.

COHN'S CONJECTURE AND THE DISTRIBUTION OF WEIGHTS

In a letter to the IRE Transactions on Electronic Computers [17], Martin Cohn presents a conjecture made by himself and S. Even that for a switching function of n inputs, there exists a canonical form with at most W product terms, where

$$W = \binom{n}{\lfloor \frac{n}{2} \rfloor}$$

Because KRMs are canonical, this conjecture can be reformulated in terms of the vector w as:

$$\exists i \mid w_i \leq W$$

In a sense, Cohn's conjecture partitions w into two parts: weights less than or equal to W and weights that are greater. We can increase the number of

partitions and form a distribution. Such a distribution would, of course, be dependent on the particular switching function.

It is impossible to explore the problem space by examining all attributes of every possible switching function. We can, however, examine some attributes of some functions, hopefully those that will give us insight. One way to limit the number of functions to examine would be to only look at functions of a given number of inputs and then only at representative functions of the NP-equivalence classes [18]. This is one approach taken by Sasoa [4] and followed in the next section of this thesis. Another approach, taken in this section, is to look at functions of a particular type with several different values of n , the number of inputs.

In this section, we describe the results of an experiment in which the distribution of weights was found for 25 5-input functions, 25 8-input functions, 25 11-input functions, and 25 14-input functions. Each of these functions was generated at random according to the following rules:

- 1) The ON set of each function is comprised of 75% "cubes" and 25% single minterms.
- 2) A "cube" will have between $n/3$ and $n/2$ "don't care" inputs.
- 3) "Don't cares" will be contiguous 50% of the time. The rest of the time they will be split into two contiguous strings.
- 4) The truth vector for each function will have about 25% ones.

These rules were developed in the hope that random functions so generated will be good representations of "real world" functions. This hope has not been put to any rigorous scrutiny.

The program `randgen`, written to incorporate these rules, was used to generate the 100 functions used in this experiment. `Kromin` was then used to find the w vector for each function. Each time `kromin` was run, the program `wghtfreq`, written to accept a shared memory file produced by `kromin`, was used to convert the w vector to a distribution and plot it. `Wghtfreq` also indicates the "bin" of the distribution where Cohn's conjecture would fall.

Rather than include all 100 distribution plots here, I will show only a few to summarize the results of the experiment. Figures 18, 19, 20, and 21 represent typical weight distributions for $n=5$, 8, 11, and 14, respectively.

The first thing to notice is that as n gets larger, the shape of the distribution becomes more recognizably that of a poisson distribution. This observation leads to two questions: why does the "shape" of the distribution become more "smooth" as n increases and why does the shape match that of a poisson distribution? A possible answer to the former question is that as n increases, the distribution is formed from an increasing number of weights (samples). When samples are counted to form any distribution curve (as was done here), the more samples that are counted, the closer the match between the constructed distribution curve and the actual curve of

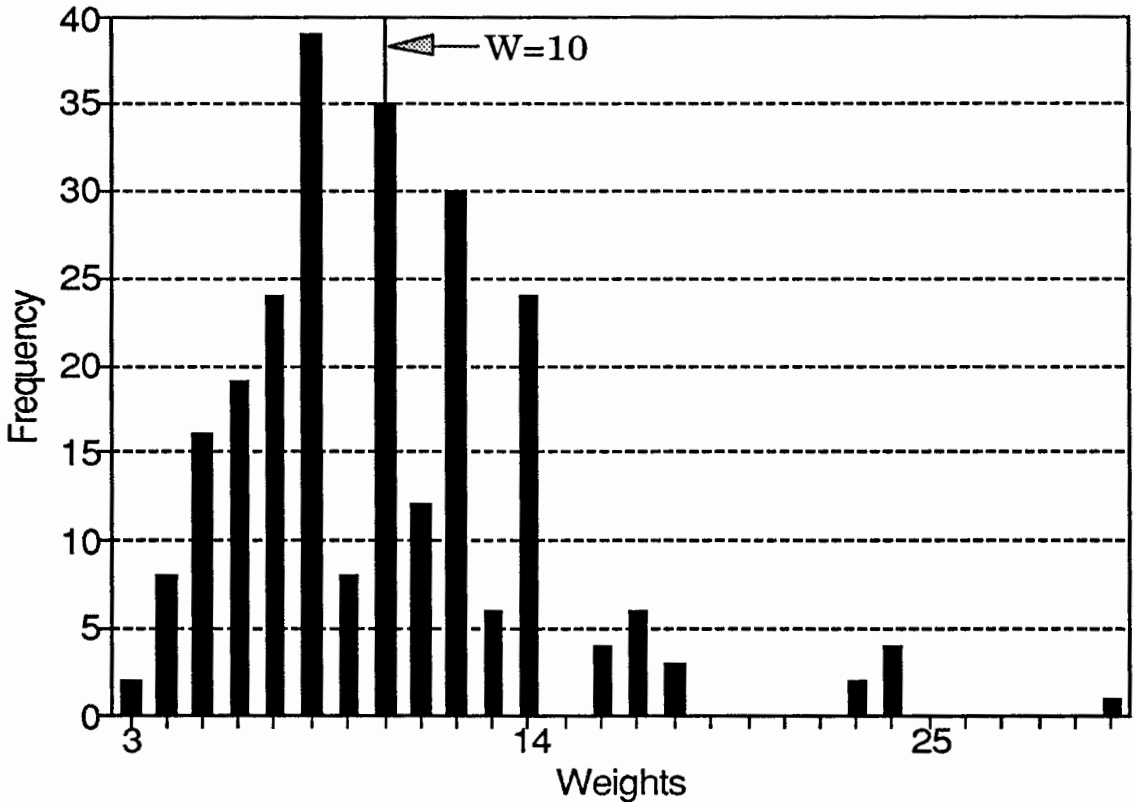


Figure 18. Example distribution of weights for $n=5$.

the underlying distribution. An answer to the latter question is not so easily found and is left to future research.

Where does Cohn's Conjecture fit into these distributions? The value of W is indicated in the distributions shown here with a line and a label indicating the value of W for the particular value of n . It is clear that for all the functions shown here (and, in fact, for all 100 functions used in this experiment), there are several weights less than or equal to W . Figures 22 and 23 show the distributions of weights for the 14-input functions with the least and most weights $\leq W$, respectively. In both cases, there are many

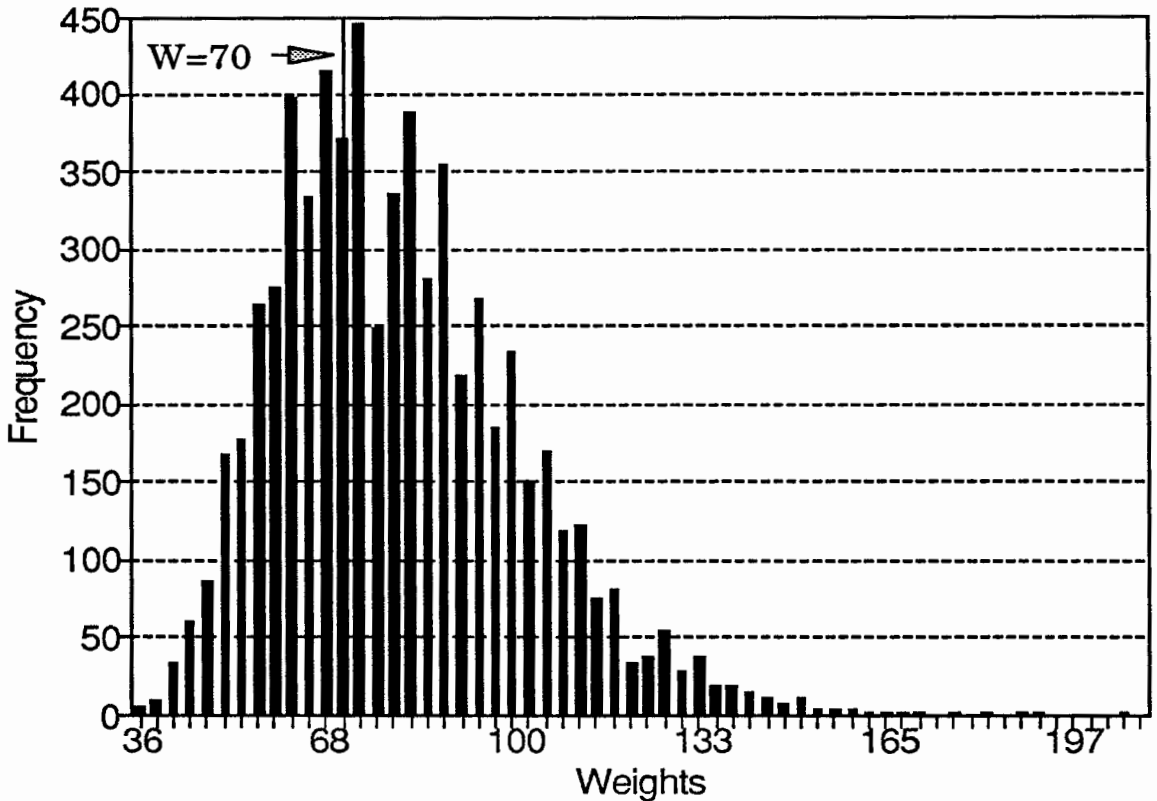


Figure 19. Example distribution of weights for $n=8$.

weights $\leq W$ but many more that are greater. This suggests that Cohn's Conjecture can be safely used as the basis of a heuristic that avoids computing weights larger than W . Such a heuristic could not be incorporated into any algorithm based on the recursive 'fast' algorithm presented in Chapter III because the weights are not computed one at a time. It could be possible to reformulate the original 'fast' algorithm into a "serial" 'fast' algorithm where the weights would be computed one at a time (perhaps in parallel). This new algorithm could then use the value of W as a "cut off" value. The development of this new algorithm is left to future research.

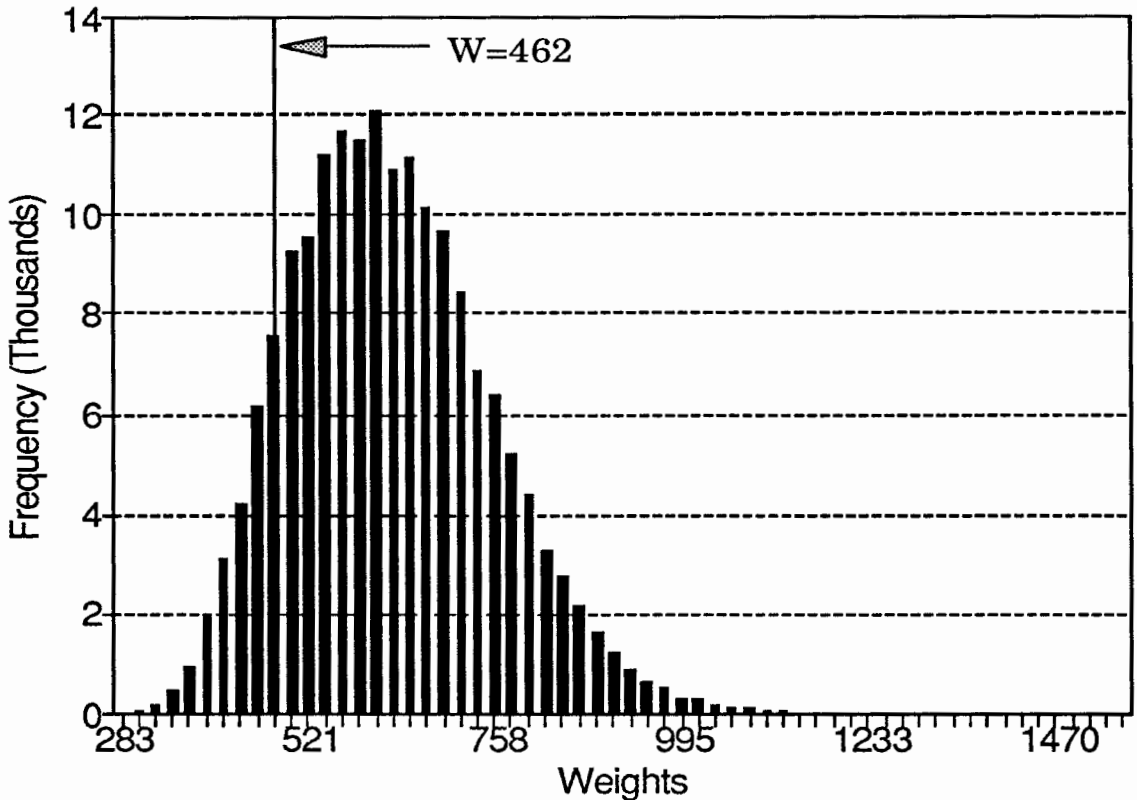


Figure 20. Example distribution of weights for $n=11$.

EXPANSION ON WORK BY SASAO AND BESSLICH

In the paper "On the Complexity of Mod-2 Sum PLA's" [4], Sasao and Besslich present a table titled "Average Number of Products for All the Four-Variable Functions". The first row of data in this table is for minimum ESOPs and the second row of data is for minimum SOPs. Kromin can be used to add rows for RMs FPRMs and KRMs. Table IV is a result of doing so, using the functions from Appendix 5 of [18]. Note that the first two rows of data are taken from [4]. The standard deviation for these

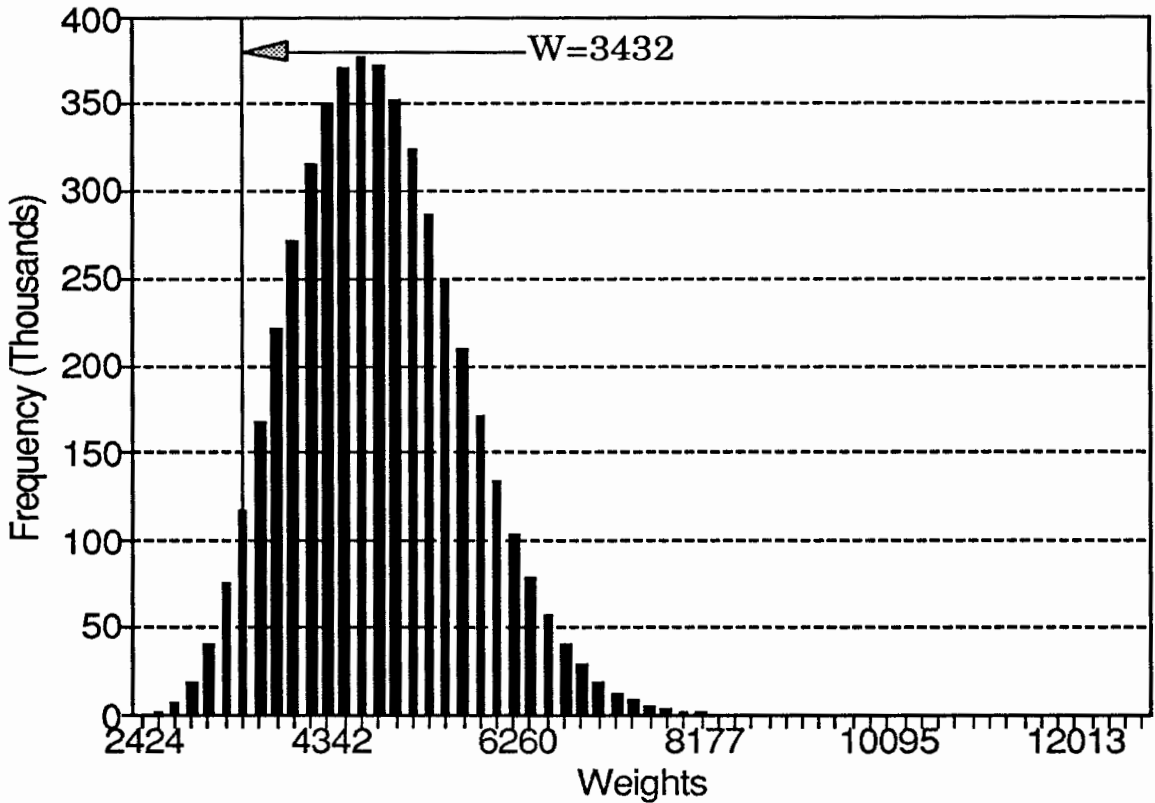


Figure 21. Example distribution of weights for $n=14$.

averages is also reported in the table, except for those from [4]. The averages are also shown graphically in Figure 24 (just for reference, the value of W in Cohn's Conjecture is indicated).

The main thing to notice from the new table is that as the form of expression gets more general (from RM to ESOP), the average number of products for a given number of minterms drops. KRMs seem to have about as many product terms as SOPs. In the next chapter, we speculate that there may be a way to transform KRMs to ESOPs, thereby reducing the average number of products.

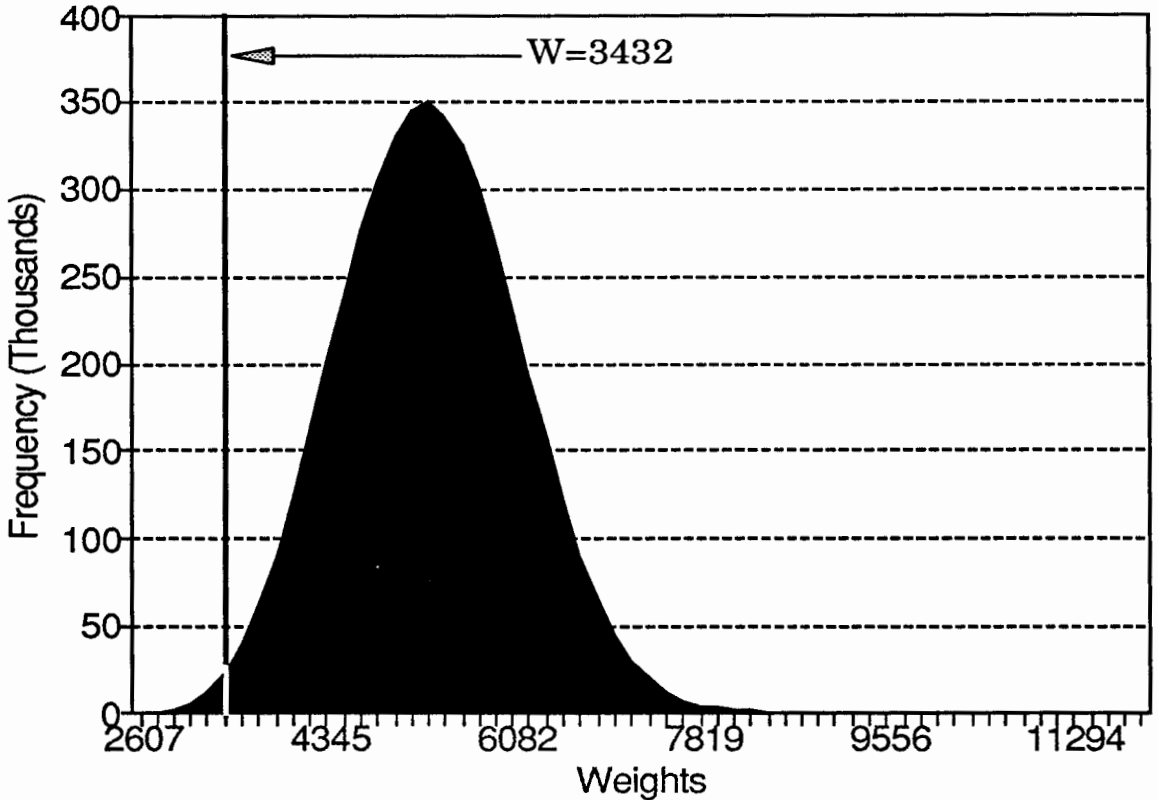


Figure 22. For $n=14$, distribution with fewest weights $\leq W$.

INCOMPLETELY SPECIFIED FUNCTIONS

If the truth vector is allowed to contain "don't care" values, then the switching function is said to be incompletely specified. As presented in Chapter III, the algorithm will only work with completely specified switching functions. This limits its usefulness because many switching functions are, by the nature of their application, incompletely specified. A good area for future research would be to modify the practical algorithm to work with incompletely specified functions.

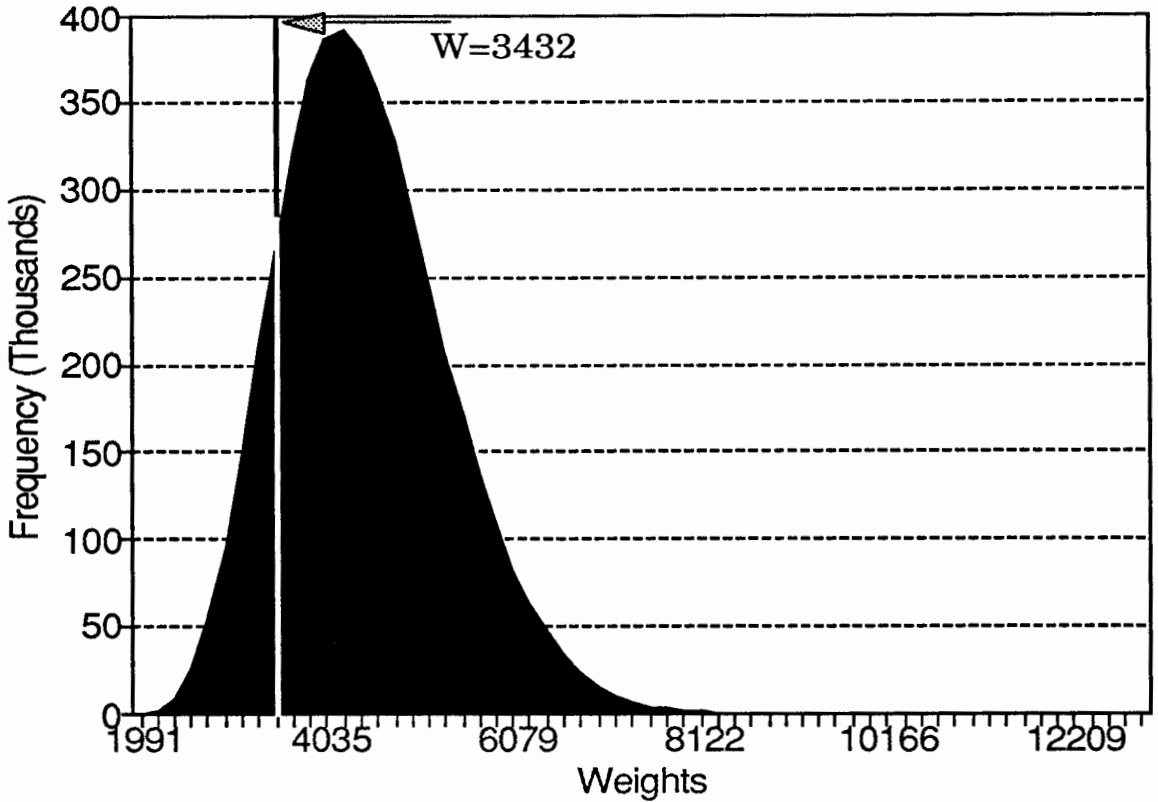


Figure 23. For $n=14$, distribution with most weights $\leq W$.

One way to deal with incompletely specified functions can be found in the paper "Reed-Muller Expansions of Incompletely Specified Functions" [19]. If there are k values of the d vector specified as "don't care", then simply solve 2^k minimization problems, each with different values substituted for the k "don't cares". Then pick values for the "don't cares" corresponding to the minimum of the 2^k minimums. For this section kromin was modified to allow incompletely specified functions and find their minimum as outlined above.

TABLE IV
AVERAGE NUMBER OF PRODUCTS FOR ALL THE FOUR-VARIABLE
FUNCTIONS

	1	2	3	4	5	6	7	8
ESOP	1.000	1.733	2.371	2.738	3.132	3.350	3.702	3.696
SOP	1.000	1.733	2.371	2.905	3.370	3.730	4.053	4.273
KRM	1.000	1.750	2.500	2.947	3.852	4.260	4.875	4.440
σ KRM	0.000	0.500	0.548	0.911	0.989	1.226	1.129	1.327
PRM	1.000	3.250	4.667	3.737	5.259	5.120	5.786	5.009
σ FPRM	0.000	2.217	1.633	1.195	1.347	1.649	1.371	1.623
RM	1.000	6.500	5.667	6.263	5.926	7.320	9.018	7.224
σ RM	0.000	6.191	2.160	3.380	2.480	2.945	2.378	2.337
	9	10	11	12	13	14	15	16
ESOP	3.912	3.864	4.088	3.732	3.371	2.733	2.000	1.000
SOP	4.457	4.537	4.546	4.426	4.200	3.733	4.000	1.000
KRM	5.268	5.140	5.407	4.684	4.833	4.250	2.000	1.000
σ KRM	0.981	1.309	1.338	1.204	1.169	2.217	0.000	0.000
FPRM	6.054	5.580	6.185	4.737	5.667	4.250	2.000	1.000
σ FPRM	1.197	1.372	1.331	1.195	1.633	2.217	0.000	0.000
RM	9.768	7.920	6.704	6.632	6.667	7.000	2.000	1.000
σ RM	2.071	2.546	1.938	2.629	2.160	5.292	0.000	0.000

While this solution is in keeping with the brute force nature of the practical algorithm, the performance gets exponentially worse as the number of "don't cares" is increased. That goes against intuition which suggests that we should be able to use the flexibility of having "don't cares",

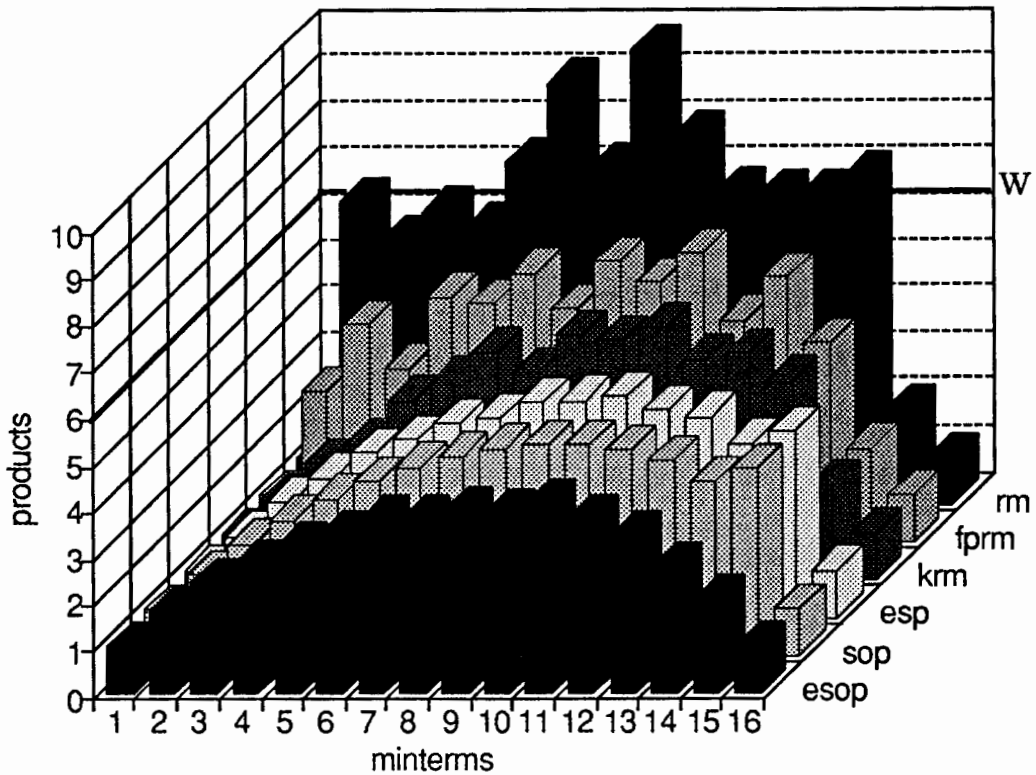


Figure 24. Average number of products for all the four-variable functions.

along with our knowledge of the M and P transformations, to avoid most of the work.

To take a first step in the development of such a heuristic, we first develop a very simple heuristic and see how well it performs against the brute force method. The heuristic tested in this section could not be much simpler: just assign random binary values to the "don't care" outputs. Do this some number of times and pick the "best" set.

To test this heuristic, the 100 random functions used above were modified so that two random ON-minterm had their output values set to

"don't care". The same was done for two random OFF-minterms. This gives a total of 16 different combinations of "don't care" output values. The functions were processed by kromin, which used both the brute force method and the simple heuristic, limited to 8 "don't care" combinations per run.

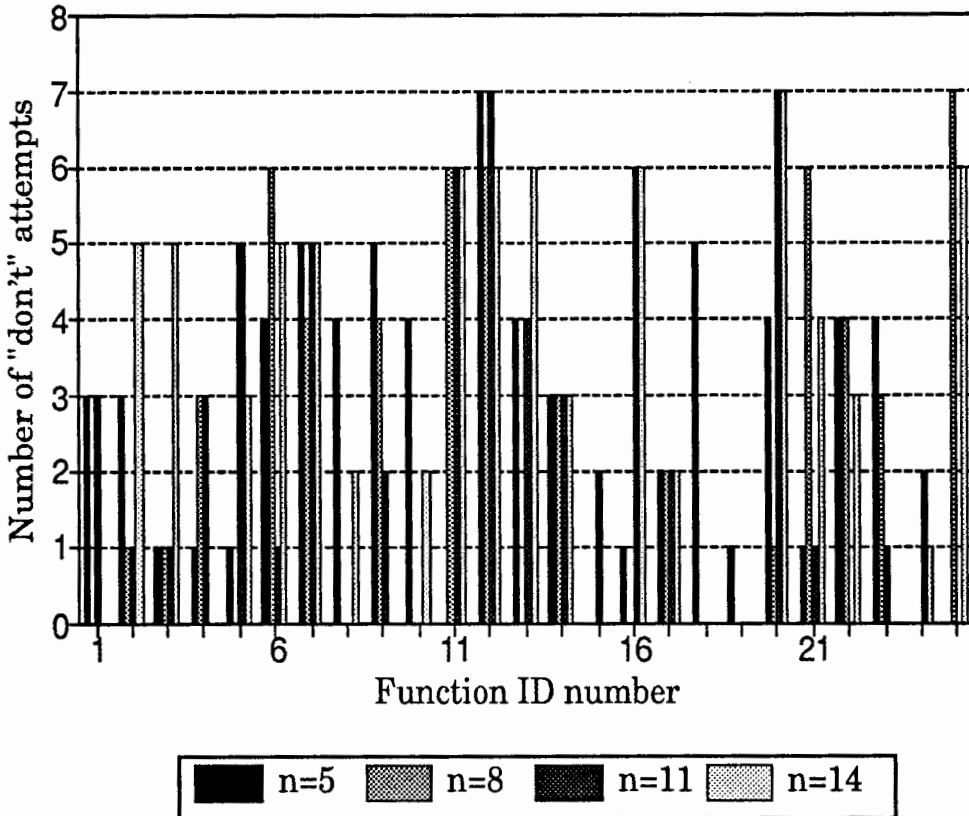


Figure 25. Incomplete functions heuristic.

Because the simple heuristic does half as many minimizations as the brute force method, it is twice as fast. But it doesn't always find the minimum. Figure 25 shows the number of attempts the heuristic needed to find the true minimum of each of the 100 random functions. If the true

minimum was not found in 8 attempts, then no bar is printed for that function. As can be seen from the figure, this simple heuristic doesn't do as badly as one would think. This experiment offers encouragement to the idea of finding a better heuristic for assigning values to "don't care" outputs. The section "Incomplete Switching Functions" in the next chapter offers an idea that could lead to a better heuristic.

CHAPTER VI

FUTURE WORK AND CONCLUSIONS

OVERVIEW

In the previous chapters we discussed the need for and then developed a practical algorithm for finding minimum KRM expansions. We then tested several implementations based on that algorithm. While the resulting software is valuable in its own right as a minimization tool, because of its brute force nature, it would probably be of more use as a research tool for building a better algorithm. This chapter discusses four different areas for further research: the first two would change the nature of the problem being solved and the last two would tinker with algorithm design.

INCOMPLETE SWITCHING FUNCTIONS

In the last section of the previous chapter we developed and tested a simple heuristic. The heuristic just tried a number of random values for "don't care" outputs and picked the best one. Although this approach didn't do too bad of a job, it didn't always find the minimum and doesn't use any of the information available about the function. As stated above, intuition suggests that we should be able to use the flexibility of having "don't cares",

along with our knowledge of the M and P transformations, to make better choices about the values assigned to "don't care" outputs.

Another approach would be the following: for each row in P_n , independently choose a "don't care" combination that forces as many of the selected elements of e as possible to the value zero. Note that changing an element of e from a one to a zero is guaranteed to reduce the weight if that element is selected by the corresponding row in P_n . A "provisional" minimum weight found in this way will actually be the true minimum only if the set of "don't care" assignments is consistent. That is to say, each time a value is assigned to a "don't care", that value is the same as any previously assigned to that "don't care".

The key to this future research would be to find a way to resolve any inconsistencies in "don't care" assignments in such a way as to: 1) increase the minimum weight as little as possible and 2) do so in polynomial time. Methods should be investigated that resolve inconsistencies on the fly and heuristics could be found that avoid inconsistencies in the first place. In either case, it should be possible to beat 2^k executions of the algorithm.

The completion of this research would give us a fast practical way to find minimum KRM expansions for incompletely specified switching functions. Some might say that this is not enough, because a minimum KRM expansion is not necessarily the minimum ESOP form of a given function. This problem is explored in the next section.

ESOPS ARE NOT KRMS

The purpose of the practical algorithm developed in this thesis is to find minimal KRM expansions. And while a KRM expansion is, of course an ESOP, the minimal KRM expansion of a given function may not be the minimum ESOP form of that function. For example, consider the following switching function, given in ESOP form:

$$f(x_1, x_2, x_3, x_4) = 1 \oplus x_2 x_1 \oplus \bar{x}_4 \bar{x}_3 \bar{x}_2 \oplus x_4 \bar{x}_3 x_2 x_1$$

There are several minimum KRM expansions for this function, but they have six terms, two more than the ESOP form above. The reason that this particular ESOP is not a KRM is that the variables x_2 and x_4 are used in three ways: for a given term, either the variable is used, or the complement of the variable is used, or the variable does not appear in the term. For a KRM form, each variable may be used in any two of these ways, but not all three. This is a consequence of the fact that three symbols are used in the basis vectors $(1, \bar{x}_i, x_i)$ for e , but each basis vector for an ESOP consists of only two of these symbols.

Future research would be to develop a way to find the minimum ESOP form of a switching function, given the minimum KRM expansions for that function. Perhaps some kind of intelligent factoring can be used. For example, consider the following minimum KRM expansion for the above function:

$$f(x_1, x_2, x_3, x_4) = x_2 \oplus \bar{x}_2 \oplus x_2 x_1 \oplus \bar{x}_3 \bar{x}_2 \oplus x_4 \bar{x}_3 \bar{x}_2 \oplus x_4 \bar{x}_3 x_2 x_1$$

Obviously, the first two terms can be combined into a single term, equal to

1. In addition, the following factorization can be performed:

$$\begin{aligned} \bar{x}_3 \bar{x}_2 \oplus x_4 \bar{x}_3 \bar{x}_2 &= (1 \oplus x_4) \bar{x}_3 \bar{x}_2 \\ &= \bar{x}_4 \bar{x}_3 \bar{x}_2 \end{aligned}$$

After these two straightforward simplifications, we are left with the original (minimal) ESOP. It might be possible to find a way to guide the factorizing based on the polarity numbers of the minimum KRM expansions. For example, the polarity number, in trinary, for the above KRM is <0211>. This alerts us to the fact that x_2 , either complemented or uncomplemented, appears in every term, making it a good candidate for factoring. An x_2 can be factored from the first and third terms and an \bar{x}_2 can be factored from the second and fourth terms as follows:

$$\begin{aligned} x_1 \oplus x_2 x_1 &= x_2 (1 \oplus x_1) \\ &= x_2 \bar{x}_1 \end{aligned}$$

$$\begin{aligned} \bar{x}_2 \oplus \bar{x}_3 \bar{x}_2 &= (1 \oplus \bar{x}_3) \bar{x}_2 \\ &= x_3 \bar{x}_2 \end{aligned}$$

After these simplifications, we are left with a new ESOP for the switching function that has the same number of terms as the original ESOP.

If this research were successful, we would be able to find minimal ESOPs for any switching function, provided we had a computer with enough memory, required by the exhaustive search nature of the minimization

algorithm. Even with virtual memory, this can be a problem because the need for memory grows exponentially with the size of the switching function. The next two sections deal with possible ways around this memory problem.

MORE MEMORY BY DISTRIBUTED SYSTEMS

As an example of how fast the need for memory grows, minimizing a 15-input switching function requires about 29 megabytes, but minimizing a 24-input switching function requires about 565 gigabytes of memory. It's not likely that any single computer, even a multiprocessor, will have that much memory any time soon. In fact, 565 gigabytes is 140 times the maximum addressability of a 32-bit processor! This section deals with a way to find enough memory to do larger problems. The next section discusses a way to reduce the amount of memory needed.

Let's say we had a network of engineering workstations, each with a 600 megabyte hard disk. It would take a network with just less than 1000 of these systems to have enough "collective" hard disk memory to cover the 565 gigabyte memory requirement. Of course, hard disk memory is not the same as main memory, and distributed processors are not the same as the shared memory multiprocessing of the Sequent. But because of the way the work is partitioned by the Krönecker product, it isn't difficult to see how to make the algorithm work in this kind of environment.

Another part of this future research would be to modify the algorithm to effectively use distributed processing. The key here, as in many distributed applications, would be in communication between processors. Perhaps the communications could be handled in a way that takes advantage of the topology of the network. In any event, we will need to use distributed systems if we want exact solutions to large problems.

LESS MEMORY BY CLOSE TO MINIMAL SOLUTIONS

Another way to get around the memory problem is to modify the algorithm to use less memory by not saving all of the weights. Methods should be investigated that allow only weights that have small values to be computed and stored. The problem is, how do we know which weights have small values without computing them? One possible approach would be to combine groups of bits in the extended truth vector as a way to estimate which weights need to be computed. Some memory-saving tricks we might try could interfere with the algorithm's ability to find the true minimum expansion. In this case, we want to at least find a solution that is close to the minimum.

If in the process of modifying the algorithm to use less memory we lose the guarantee of a minimum solution, we need some kind of assurance that the solution is still a good solution, in some sense. It might be that this research leads to a whole new approach, or it could be that a simple

modification to the existing algorithm could be made. Now that the practical algorithm has been developed, there are many different ways to use and extend it.

CONCLUSIONS

In this thesis we have discussed the need for and then developed a practical algorithm for finding minimum KRM expansions. We then tested several implementations based on that algorithm. Finally, in this chapter, several areas for future work were discussed.

In the first chapter, it was stated that the algorithm "should be easy to use and do its job well". We can take this to mean that the algorithm should be as fast as possible and should be able to minimize problems with as many inputs as possible.

From the experiments on a Sequent S81 with 32 megabytes of memory and nine processors, we found that a practical upper bound on the size of the problem is 15 bits and that a minimum for a problem of that size can be found in about 12 minutes. The maximum problem size is big enough so that minimum KRM expansions can be found for moderately complex switching functions and the speed is fast enough so that many problems of that size could be run during a course of research. Because the algorithm performs an exhaustive search, it is particularly useful as a "baseline" to compare against when evaluating other methods or heuristics.

Based on these findings, it could be argued that this thesis research has satisfied the goal of being a useful tool for the development of ESOP minimization algorithms. The proof of any tool, however, is in its use and only time will tell.

REFERENCES

- [1] Detjens, E. (1990, November 1). FPGA devices require FPGA-specific synthesis tools. *Computer Design*, p. 124.
- [2] Readdy, S.M. (1972). Easily testable realizations of logic functions. *IEEE Transactions on Computers*, C-21, 1183-1188.
- [3] Handschy, M.A., Johnson, K.M., Cathey, W.T., & Pagano-Stauffer, L.A. (1987). Polarization-based optical parallel logic gate utilizing ferroelectric liquid crystals. *Optics Letters*, 12, 611-613.
- [4] Sasao, T. & Besslich, P. (1990). On the complexity of mod-2 sum PLA's. *IEEE Transactions on Computers*, 39, 262-265.
- [5] Green, D.H. (1990). Reed-Muller canonical forms with mixed polarity and their manipulations. *IEE Proceedings*, 137(Pt. E), 103-113.
- [6] Bioul, G. & Davio, M. (1972). Taylor expansions of boolean functions and their derivatives. *Philips Research Reports*. 27(1), 1-6.
- [7] Green, D.H. (1986). *Modern logic design* (pp. 131-164). New York: Addison-Wesley.
- [8] Stewart, I. (1989). *Galois Theory* (2nd ed.). London: Chapman and Hill.
- [9] Moore, J.T. (1975). *Introduction to Abstract Algebra* (p. 139). New York: Academic Press.
- [10] Graham, A. (1981). *Kronecker Products and Matrix Calculus With Applications*. Chichester: Ellis Horwood Limited.
- [11] Zhang, Y.Z. & Rayner, P.J.W. (1984). Minimisation of Reed-Muller polynomials with fixed polarity. *IEE Proceedings*, 131(Pt. E), 177-186.
- [12] Stewart, G.W. (1973). *Introduction to Matrix Computations* (pp. 1-67). New York: Academic Press.
- [13] Brinch Hansen, P. (1973). *Operating System Principles* (pp. 57-58). Englewood Cliffs, NJ: Prentice-Hall.

- [14] Dijkstra, E. W. (1968). Cooperating sequential processes. In F. Genuys (Ed.), *Programming Languages*. New York: Academic Press.
- [15] Jayawardena, J. (personal communication, June, 1991)
- [16] Osterhaug, A. (Ed.) (1989). *Guide to Parallel Programming on Sequent Computer Systems* (pp. A-1 – A-2). Englewood Cliffs: Prentice Hall.
- [17] Cohn, M. (1962). Inconsistent canonical forms of switching functions. *IEE Transactions on Electronic Computers*, *EC-11*, 284-285.
- [18] Harrison, M.A. (1965). *Introduction to Switching and Automata Theory*. New York: McGraw-Hill.
- [19] Green, D.H. (1987). Reed-Muller expansions of incompletely specified functions. *IEE Proceedings*, *134*(Pt. E), 234.