AN ABSTRACT OF THE THESIS OF

Nael N. Abweh for the degree of Master of Science in Electrical and Computer Engineering presented on June 27, 1985.

Title: PRONTO Programs for Product Term Reduction

Abstract approved: Redacted for Privacy

V. M. Powers

PRONTO is a direct, one-pass heuristic method designed to shrink the size of reducible programmable logic arrays. Several people have contributed to the design, and translation to a computer program, of this attempt to produce a good solution quickly. This thesis discusses efforts to improve PRONTO's implementation in order to achieve a fast and efficient solution.

The principal feature examined is the time complexity of the algorithm by which the dominant time-growth factors are identified. The execution time of PRONTO was estimated from the PLA's tangible characteristics (i.e. the number of product terms, input bits and output bits) and the size of the resulting reduced PLA. It was found that, for a small set of examples, execution time varies almost linearly with the value [Solution size * Specification size * number of output bits].

The PRONTO algorithm is built of four major parts. The first selects a base product term that is most

favorable for expansion. The second finds a set of the most likely expansion directions. The third expands the base product term in those directions. The final part updates the solution and the specification arrays. Three alternative implementations were investigated for the tree search process of expanding a base product term. The tree expansion method chosen is a combination of a preorder and a level-order traversal method.

PRONTO Programs for Product Term Reduction

by

Nael Nabih Abweh

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Completed June 27, 1985

Commencement June 1986

APPROVED:

Redacted for Privacy

Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented June 27, 1985

Typed by Sadie's Word Processing for Nael N. Abweh

ACKNOWLEDGEMENT

I would like to present my gratitude to Professor Powers for the help and guidance he offered and the concern he showed. I would also like to thank my parents for their encouragement and patience. And last, but not least, many thanks to my committee members, Professor Kolodziej, Professor Nichols and Professor Stalley for their help.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

PRONTO Programs for Product Term Reduction

CHAPTER 1

INTRODUCTION

1.1 Thesis Topic

The major concern of this paper is to study PRONTO, an algorithm for PLA size reduction. This study has two main points:

a) Evaluating the time complexity of the overall PRONTO algorithm, in which the major dominant time-growth factors are identified.

b) Comparing different procedure implementations, and choosing the ones resulting in the fastest reduction.

1.2 Project Status

PRONTO is a heuristic approach for product term reduction of Programmable Logic Arrays that was developed by Martinez [MART1], [MART2].

Translating PRONTO algorithm into a program was the focus of two projects. At the end of the first project, that was worked on by E. Burns, and R. Stettler, the first two parts of PRONTO were coded in the Mainsail® Language.

Mainsail is a trademark of Xidak Corporation

Those parts chose a product term from a PLA and, by comparison to the other cubes, selected a set of possible expansion directions. The second project, worked on by D. Dagit, resulted in a correct PRONTO program that was tested for several PLAs.

As for my thesis, I started by testing some code alternatives and ended up with a version of PRONTO that was then evaluated for computational complexity. Knowing the dominant time-growth factors and some experimental time measurements, an attempt was pursued to find a formula relating execution time to a PLA's characteristics. With the knowledge of the most time consuming parts of PRONTO, an attempt was made to produce a faster PRONTO version by considering alternative implementations of some of those procedures.

CHAPTER 2

BASIC DEFINITIONS

In this chapter we introduce a collection of definitions, operations and theorems that are pertinent to PRONTO's implementation. The terminology presented here is to be used throughout this paper. Most of the definitions follow Dietmeyer [DIETM].

2.1 Definitions

2.1.1 Programmable Logic Array. A programmable Logic Array (PLA) is a representation of an "integrated circuit with n input terminals that drive inverters, m output terminals driven by OR (NOR) gates, and p AND gates. Also included are interconnections between any or all input terminals (or their inverters) and the AND gates, and between any or all AND gates and each output gate." [DIETM, p. 185].

2.1.2 Product Term. A product term is also referred to as a cube. It is an expression with an input and an output part which stands for a structure in a PLA. The expression-structure relation is given below.

1. Input bits,

   0 : The complement of the input variable is connected to the input of an AND gate.

1 : the input variable is connected to the input of an AND gate.

- : no connection to the AND gate's input.

2. Output bits,

0 : no connection to the input of OR gate.

1 : connection to the input of OR gate.

d : does not matter if connection is made or not.

For example a product term with 5 input variables and 4 output variables might look like: 1001- ; 10d0.

2.1.3 Specification. A specification of a PLA is a set of product terms.

2.1.4 Solution. A solution to the specification is a description of a PLA which implements the desired functions. (Solution might contain less product terms than specification but is still cover equivalent).

2.1.5 Direction. Direction is a single input variable (an input direction) or a set of output variables (an output direction).

2.1.6 Expansion. Expansion of a cube in an input direction replaces the '0' or '1' in that position with a '-'. Output expansion increases the number of 1's by forming the union of outputs with the given direction.

2.1.7 Projection. An input direction of a cube is projected by replacing the '-' bit with a '0' or '1'.

2.1.8 Adjacency. Two product terms are adjacent if they have the same input bits except at one position where one cube has a '0' and the other has a '1', while the output parts are equal. Two cubes with the same input parts are adjacent if they have different output parts.

2.1.9 Cube a is covered by cube b (a < = b) iff

a. for every '1' in the input part of b there exists a '1' in the corresponding bit of a. And

b. for every '0' in the input part of b there exists a '0' in the corresponding bit of a. And

c. for every '1' in the output of b there exists a '1' or 'd' in the corresponding bit in a.

2.1.10 If a and b are two product terms then,

a. (a = b) iff (a < = b) and (b < = a); and

b. (a < b) iff (a = b) is false and (a < = b) is true.

2.1.11 Common Cover. Product term c is a common cover of a and b if it covers both a and b.

2.1.12 Cube b is a single decomposition of cube a if b differs from a in only one column.

2.1.13 Cube b is an expansion of cube a, if for some input column where a_i = '0' or '1' the corresponding column b_i = '-', or for some output column where a_i = '0' there exists a '1' or a 'd' in the corresponding output column b_i.

2.2 Operations

2.2.1 Cube Intersection. The intersection of two cubes a and b (a Λ b) is a column by column operation comparing a and b as shown in Tables 2.1 and 2.2.

Table 2.1

Input Coordinate Cube Intersection

a Λ b	b_i		
a_i	0	1	-
0	0	ϕ	0
1	ϕ	1	1
-	0	1	-

If in the resulting cube c a ϕ occurs in the input part or all output bits c_i are 0's, then the result is said to be empty.

Table 2.2

Output Coordinate Cube Intersection

$a \wedge b$	b_i		
a_i	0	1	d
0	0	0	0
1	0	1	d
d	0	d	d

2.2.2 Sharp Product

2.2.2.1 The sharp operation of two cubes (a # b) results in

a) an empty cube iff $(a \wedge b) = a$;

b) cube a iff $(a \wedge b)$ is empty;

c) an array C iff $(a \wedge b)$ is nonempty and $(a \wedge b) < a$.

C consists of a set of cubes (maybe only one cube) determined by the following: For every a_i = '-' in the input where b_i = '0' or '1', there is a cube c where c_i = '1' or '0' respectively, with all other columns the same as a. If in the output there is some a_i = '1' where b_i = '0', then there is another cube c where all c_i in the input columns are the same as a_i, and for every c_i in each

output column, c_i = '1' iff a_i = '1' and b_i = '0', otherwise c_i = '0'.

For example let,

a = 101- / 1001

b = 10-0 / 01d1

Then C = a # b where

C = 1011 / 1001; due to the input parts,

101- / 1000; due to the output parts.

And C = b # a results in

C = 1000 / 01d1; due to the input parts,

10-0 / 0100; due to the output parts

2.2.2.2 The sharp product of an array of cubes A with a single cube b (A # b) is defined in terms of single-cube sharp operations as

A # b = (a_1 # b) U (a_2 # b) U...

A # b results in,

a) array A iff ($a_i \wedge$ b) is empty for every a in A.

b) an empty cube iff ($a_i \wedge$ b) = a_i for every a in A.

c) an array C if conditions a) and b) fail. C is the collection of cubes resulting from (a # b) for every a in A.

2.2.2.3 The sharp product of two arrays of cubes (A # B) is defined as follows:

(A # B) = ((A # b_1) # b_2) #... .

The above representation could be broken down by the array-cube sharp product presented in section 2.2.2.

$(A \# B) = \ldots((a_1 \# b_1) \cup (a_2 \# b_1) \cup \ldots) \# b_2) \# \ldots$

The operation results in

a) array A iff $(a_i \wedge b_j)$ is empty for every a in A and for every b in B.

b) an empty set if for every a_i in A, $(a_i \wedge b_j) = a_i$ for some b_j in B.

c) an array C if conditions a) and b) fail. C is the collection of cubes from a # B for every a in A.

2.2.2.4 A is covered by B iff (A # B) is empty.

2.2.2.5 If $(A \# B) \equiv (B \# A)$, then

a) $(A \# B) = (B \# A) = \emptyset$, and

b) A is <u>cover equivalent</u> to B, $A \equiv B$

2.2.2.6 Array A is a <u>solution set</u> of array B if they are cover equivalent.

2.2.3 <u>Cube Union</u>

The union of two cubes (a U b) is a column by column operation comparing both the input part and the output part of a and b resulting in a single term c according to the rules of Tables 2.3 and 2.4

Table 2.3

Input Coordinate Cube Union

a U b		b_i = 0	b_i = 1	b_i = -
	0	0	-	-
a_i	1	-	1	-
	-	-	-	-

Table 2.4

Output Coordinate Cube Union

a U b		b_i = 0	b_i = 1	b_i = d
	0	0	1	d
a_i	1	1	1	1
	d	d	1	d

2.3 Theorems

2.3.1 Equivalence. (a = b) iff there is a column by column equivalence in the input, and for every '1' in an output column in a or b, there must be a non zero entry in the corresponding output column of the other term.

2.3.2 If (a < b) then either there is a '1' or a '0' in a_i where there is a '-' in b_i on the input part, or there is a '0' in a_i where there is a '1' in b_i on the output part.

2.3.3 If (a < = b) then (a $\wedge$ b) = a.

2.3.4 If c = a U b, then c is called the <u>smallest common cover</u> of a and b, and c is always nonempty if either a or b or both are nonempty.

CHAPTER 3

DESCRIPTION OF ALGORITHM PRONTO

3.1 Introduction

PRONTO is a one-pass heuristically guided direct search method for the reduction of product terms of a given PLA function specification. PRONTO's main loop removes terms from the specification and places into the solution terms which cover them in the following four steps until the specification is exhausted.

a. It selects a most promising, yet uncovered product term from the current specification.

b. For this selected term, PRONTO locates a set of directions that would most likely provide expansions which attempt to cover most of the yet uncovered remaining terms.

c. PRONTO then expands the chosen product term in the previously found directions and checks the validity and usefulness of the expanded terms.

d. Finally, PRONTO updates the solution, deletes the fully covered product terms from the specification and modifies the partially covered ones.

These steps are looped until no product term is left in the specification.

PRONTO has three characteristics which enable it to produce practically fast, near-optimal results. First, it attempts to expand only in the direction most likely to be successful expandable directions. Second, PRONTO seeks to cover most uncovered product terms and does not require the primality of the expanded product terms. Third, its complexity depends linearly on the number of product terms in the solution, and thus makes PRONTO faster in producing better results. A PLA area is approximated by (see Figure 3.1),

$$A = (2 * ni + no) * nc,$$

where ni, no and nc are the numbers of inputs, outputs and product terms, respectively. Since the number of product terms (nc) is the dominant factor, PRONTO's main concern was to reduce this number.

PRONTO was also chosen to be a near-optimal reduction method because "It is very unlikely that an optimum reduction method for a PLA generator will be justified due to the prohibitively large requirements in memory and computation time." (page 548, [MART2]).

3.2 PRONTO

Here, PRONTO is introduced by a detailed presentation of its four main parts that are mentioned in the previous

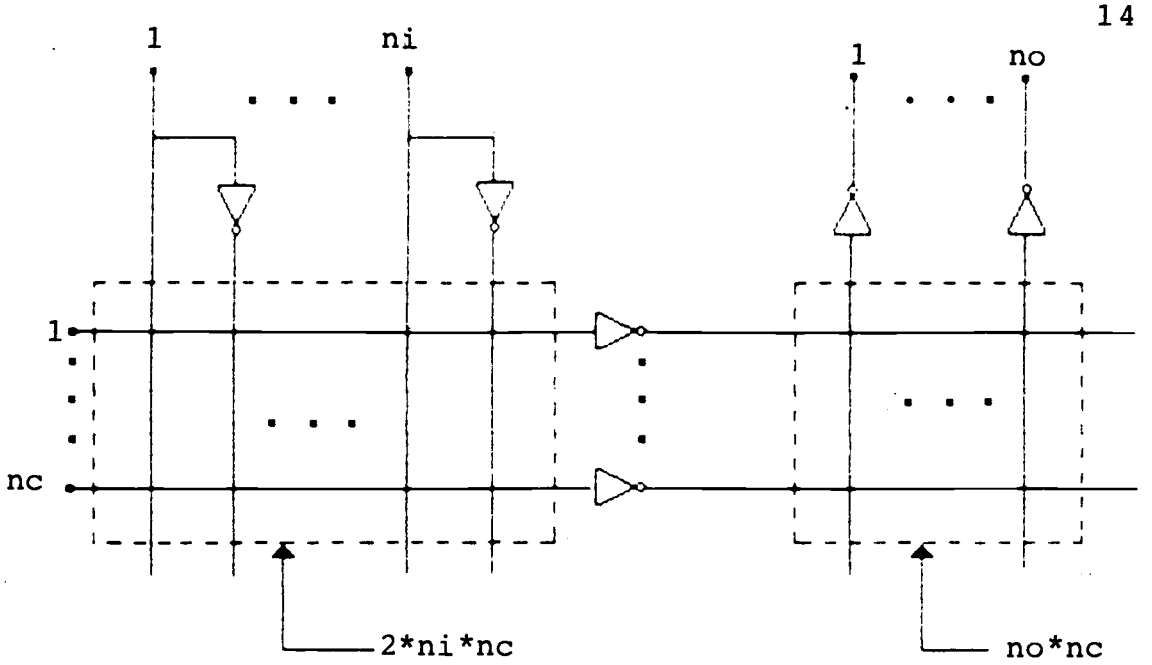

Figure 3.1 Area factors of a PLA description of ni-inputs, no-outputs and nc-product terms.

section. PRONTO's body could be presented simply by the following pseudo-algorithm, as given by Martinez [MART2].

For a given specification

Initially, the solution is empty;

REPEAT

Select a Base Product term;

Find a set of expansion directions;

Expand the Base Product term;

Update the solution and specification;

UNTIL the specification is empty.

3.2.1 Base Product Term Selection

A base product term, Pb, that is least likely to be covered by other terms is selected from the specification.

The selection process follows the following steps:

a) An output column is chosen with the minimum nonzero number of 1's in all the rows. A column with the minimum number of d's is picked in case of a tie in the number of 1's.

b) The preferred Base Product term must have a '1' in the selected column and the smallest total number of 1's in the output.

3.2.2 Expansion Directions

At this stage there are a base product term, Pb, and the set of product terms, $[T_i]$, of the rest of the specification. Comparing Pb with each term in $[T_i]$, a set of possible directions is found into which expanding Pb might prove successful in covering some product terms of the specification. The comparison process, outlined below, follows some heuristic criteria.

1. If T_i covers Pb, $T_i >= Pb$, by Definition 2.1.9 then Pb is replaced by T_i and the search process is repeated with the rest of $[T_i]$.
2. If Pb covers T_i, $Pb >= T_i$, then T_i is deleted because it is redundant.
3. If the intersection of Pb and T_i, $Pb \wedge T_i$, by Operation 2.2.1 results in a cube that is

a single projection of T_i then T_i is replaced by this projection.

4. If Pb and Ti are disjoint then PRONTO attempts to find those single expansion directions that might enable Pb to cover T_i. According to the following rules, the direction of adjacency is taken to be a direction for expansion when,
   a) T_i is adjacent to Pb; or
   b) A projection of T_i is adjacent to Pb; or
   c) A single expansion of T_i is adjacent to Pb; or
   d) A single expansion, in one direction, of a cube that is a projection of T_i in another direction, results in a cube adjacent to Pb. In other words, the intersection of T_i with a cube adjacent to Pb results in a cube which is a projection of each.

3.2.3 Expanding a Base Product Term

The Base Product term is to be expanded in order to cover most of the product terms left in the current specification. Given a set of possible expansion directions,

all different possible expansion combinations are to be applied. Each expanded term is cover checked for validity (i.e. for coverage by the original specification) and if successful then it is eligible to be considered for a solution cube. At the end of the expansion process a set of valid expanded terms might be produced that are then tested for usefulness in terms of covering any other product terms of the remaining specification.

3.2.4 Updating the Solution and Specification

For each valid expanded term "there was a corresponding term, or portion thereof, in the specification. The fully expanded base term is added to the developing solution, and all those terms, or portions thereof, found during cover checking to be covered by the expanded term are deleted from the specification." (page 550, [MART2]). When a portion of a product term is covered it is deleted only if deletion does not increase the number of terms remaining (i.e. the covered portion must be a single projection of the original product term).

CHAPTER 4

ALGORITHM IMPLEMENTATION

4.1 Programming Language

Mainsail™ is a highly structured programming language produced by Xidak Corporation. PRONTO was chosen to be coded in Mainsail, running on a Digital Equipment Corporation VAX™ 750 under AT&T's Unix™ 4.2 bsd, mostly because of Mainsail's appropriate data structure for cube and array representation, and because of the program module organization facilities. Some coding for array operations and input/output tasks were obtained for use as modules or incorporation as utilities [TEEL].

A PLA specification is represented by a Class (Record) structure as follows:

```
CLASS  Logic Array
(INTEGER nc, ni, no;
LONG BITS ARRAY (* To *) la);
```

The integer variables nc, ni and no stand for the numbers of cubes, input bits and output bits, respectively. The Long Bits' structure reserves two words in memory accounting for 32 bits. Array la is an array of Long Bits where the representation of the PLA bit structure is stored. Having three bit states (0, 1 and don't care) for both the input and output variables forces the use of a two-bit

coding to represent the state of a variable in the specification. Each product term of a specification is stored, in a predetermined number of words, starting with the input bits and followed by the output bits. Since each PLA variable representation requires two bits, therefore one word might represent up to 16 actual product term variables (bits). The bit representation is as follows:

a. Input part,

```
0 : 01
1 : 10
- : 11 ;
```

b. Output part

```
0 : 00
1 : 11
d : 10 ;
```

The different coding for the input and output bits is due to their different structural meaning and the choice was made in the best interest of the operations to be performed on each (i.e. seeking straight forwardness of the operations).

4.2 Algorithm Translation

The overall program consists of two modules: NEWFUN and PRONTO. The module NEWFUN contains the utility operations imported from [TEEL] to handle the Logic Array

data structure: MergeCube, NewLogicArray, DisposeLogicArray, Cube, InArray, SpamOutArray, CopyLogicArray, Compact and AddCube. NEWFUN also has some fundamental operations such as procedure Sharp. The module PRONTO contains the algorithm's body which performs the actual PLA reduction. Procedure Initial, of module PRONTO, performs most of the file manipulation, input and output tasks. It calls procedure Main which in turn calls several procedures including the four basic procedures: FindBaseProduct, Directions, Tree and Update, which stand for the basic parts of the reduction process of algorithm PRONTO presented in Section 3.2. Module PRONTO also contains several housekeeping procedures as well as secondary procedures that are used by the above four major procedures.

4.2.1 Housekeeping Procedures

The following is an introduction of the housekeeping procedures available in both modules, PRONTO and NEWFUN.

a. NewLogicArray (nc, ni, no)

   NewLogicArray creates a logic array structure and assigns the number of cubes, input bits and output bits as specified. Also created is an array of words necessary to store the cubes. The input and output bits are set to 1's. A pointer to this structure is returned.

b. DisposeLogicArray (A)

DisposeLogicArray disposes of the logic array structure A.

c. CopyLogicArray (A)

CopyLogicArray creates a copy of the logic array A and returns a pointer to this new structure.

d. Cube (A, I)

Cube accepts logic array A and integer I and creates a one-cube logic array in which the Ith cube of logic array A is returned.

e. Compact (A)

Compact looks for zeroed cubes in logic array A, copies the remainder of the cubes into them and updates the number of cubes A.nc.

f. AddCube (A, B, I)

AddCube simply appends the Ith cube of logic array B to the end of logic array A and returns a pointer to this new structure.

g. MergeCube (A, B, I)

MergeCube performs the same task as procedure AddCube but does some checks on the cube to be added with each cube in logic Array A. A cube with an all '0' output is

not appended. If the input parts of the Ith cube of B and some cube in A are equal then that cube is not appended and instead its output is Ored to expand the corresponding cube in A. If the Ith cube of B covers some cube in A then delete the latter cube and append the former to A. If the Ith cube of B is covered by some cube in A then discard the former cube.

h. InArray (TextFile)

InArray takes for an input the text file with the actual PLA specification and converts it into a logic array structure as introduced in Section 4.1. A pointer to this new structure is returned.

i. SpamOutArray (TextFile, A)

SpamOutArray accepts logic array A for an input, transfers the information into the regular representation of the PLA specification and outputs it to an output text file.

j. DisposeCoverings (C)

DisposeCoverings disposes of the array C of logic arrays of Class coverings.

k. Addla (C, A)

Addla appends a copy of logic array A to the array of logic arrays C. If C does not

exist, it creates C. A pointer to this new structures is returned.

l. Remove (C, I).

Remove discards the Ith logic array from the array C of logic arrays. A pointer to this new structure is returned.

m. Erase (A, I)

Erase zeroes the Ith cube in logic array A and calls procedure Compact to discard the zeroed cubes.

n. Equiv (A, B)

Equiv compares the size and contents of logic arrays A and B, and, if they are equal in all respects, it will return the boolean value 'True', otherwise a 'False' is returned.

o. Large (A)

Large turns all output don't cares in a copy of logic array A into 1's, and returns that new logic array.

p. Small (A)

Small turns all output don't cares in a copy of logic array A into 0's, and returns that new logic array.

4.2.2 Searching for a Base Product Term

This is a straight forward part of PRONTO which is implemented, as presented in Section 3.2.1, by the following two procedures.

a. SetUpTables (Spec).

   SetUpTables counts the number of 1's and the number of don't cares in the output part of the current specification, Spec, and prepares three global count arrays:

   1. Onesarray holds the number of 1's in each output column.
   2. Dontsarray holds the number of don't cares in each output column.
   3. Rowarray holds the number of 1's in the output part of each cube.

b. FindBaseProduct (Spec).

   FindBaseProduct starts by calling procedure SetUpTables then uses the information gathered by it, about the output part, in finding the most unlikely-to-be-covered term (i.e. the Base Product term, Pb) in the current specification, Spec. This term is taken out of Spec and is returned as a one-cube logic array.

4.2.3 Selection of Expansion Directions

The selection of expansion directions is done by procedure Directions which returns an array of possible expansion directions or an empty array if none exists. The expansion-direction rules laid out in Section 3.2.2 are the basic steps followed by this procedure. Procedure Directions takes for inputs Pb, the Base Product term, and Spec which stands for the remaining specification. "Directions" compares Pb with each cube in Spec, T_i, in order to end up with a set of expansion directions. First, the input parts of Pb and T_i are compared and if found to have more than one direction of opposition then the next T_i is pursued. This step is a time saver since product terms of a specification usually are not adjacent. Second, the output parts of Pb and T_i are compared. The output parts are said to intersect if a '1' in one cube's output part appears opposite to a '1' in the other. They are also said to be different if a '1' bit confronts a '0' bit. If Pb is found to be a single input projection of T_i and their outputs intersect then T_i is replaced by that cube that is still not covered by Pb, as presented by step 3 of Section 3.2.2. The current T_i is given up and the next T_i is pursued if either

a. The output parts of Pb and T_i do not intersect; or

b. Pb is a multiple projection of T_i, or T_i is a multiple projection of Pb.

At this stage the successful product term T_i compared to Pb might have

a. No input opposition direction and a different, non-intersecting output part; or

b. One input opposition direction and an equal, or intersecting, output part.

For the first case, the expansion direction is considered to be the union of the output parts of Pb and T_i. For the second case, the expansion direction is taken to be the input opposition direction. Moreover, none, one or both of the following statements must be true,

a. PB is a simple input projection of T_i;

b. T_i is a simple input projection of Pb.

The four different combinations of the truth value of the two statements represent the four cases that govern step 4 of Section 3.2.2

The expanded Base Product term is added to the array of expansion directions only if it is unique (i.e. it is not a duplicate of an existing possible expanded term). As mentioned before, each product term of the remaining specification, Spec, should be tested for possible expansion directions. An array of possible single-expansion direction terms is returned, which may be empty.

4.2.4 Expansion of a Base Product Term

Given a set of possible single-expansion directions, each possible term formed by expanding the Base Product term, into a different expansion combination, is tested for validity (i.e. for coverage by the original specification). At this stage, a set of promising expanded terms exists and we are left with testing if they are useful terms (i.e. if a term covers some product terms other than the Base Product term). Figure 4.1 shows an imaginary tree structure whose nodes represent the possible, different expansion terms formed from a Base Product term with three single-expansion directions.

The task of expanding the base term and testing its validity and usefulness is performed by procedure Tree as shown below.

Procedure Tree;

1. Procedure Treesearch: (Recursive)
   a. expands the Base Product term;
   b. Procedure Donts: checks the validity of the expanded term.
2. Procedure Covering: finds the terms covered by the valid expanded term.

Procedure Treesearch expands the Base Product term by pursuing the different expansion-direction combinations by traversing a tree structure similar to that of Figure 4.1.

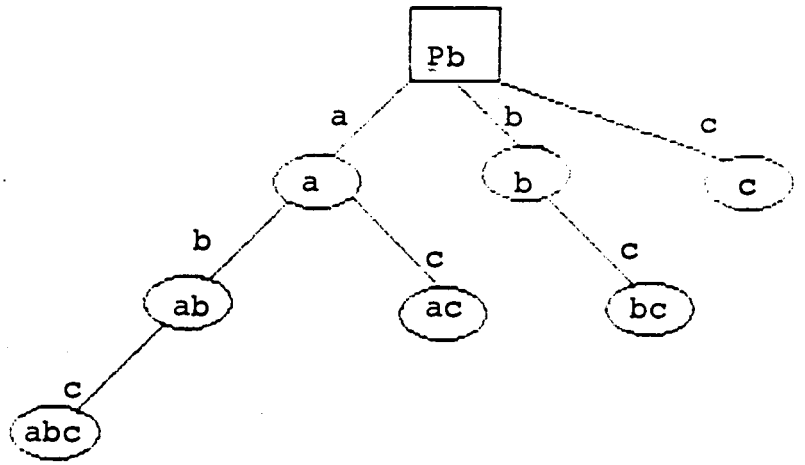

Figure 4.1 The tree structure of possible expanded terms

The order of traversal is the subject of Chapter 6 where three methods are compared. At each node the previously expanded term is expanded in that node's new direction to form the current base term. The current term is then tested for validity by procedure Donts. The term is returned if it was found valid, otherwise an empty cube is returned. The valid expanded terms are stored by procedure Treesearch in array "Coverers" after assuring that no redundancy occurs by keeping only the largest term in case it covers one or more valid terms.

At this point, each term in array "Coverers" is tested for usefulness by procedure Covering. Covering gathers the product terms, in both the current specification and the current solution, that are covered by each valid expanded term and stores them in an array called "Covered". The set of all resulting "Covered" arrays are stored in array "Covers".

4.2.4.1 Implementation of Procedure Donts

Procedure Donts accepts for input an expanded term, Pb, and Larger, the original specification with all output don't cares turned to 1's. "Donts" returns the output-modified expanded term if it is found to be covered by Larger, or an empty cube otherwise. To follow, are two different implementations of the validation process that depend on the number of output don't cares (d.c.'s) of the expanded term. The principal difference is in how don't care conditions in the output part of the expanded term are handled.

1. Procedure Donts (Pb, Larger);
    a. makes a copy of Pb with all output d.c.'s set to 0's;
    b. uses the Sharp operation to check if this copy is covered by Larger;
    c. if not covered, an empty cube is returned;
    d. if covered, then each output d.c. in Pb is iteratively set to '1' and the modified cube is again checked for cover by the Sharp operation. If not covered then the corresponding d.c. position is zeroed;

e. the modified expanded term, with the maximum covered output size, is returned.

This was the method used in PRONTO version that was evaluated in the following chapter. It turned out to have two drawbacks. The first, is that a successful expanded term with output d.c.'s was turned as large as possible by setting those d.c.s to 1's while restoring the term's validity. The second, is that if an expanded term had some "n" output d.c.'s then it might be sharped with Larger as many as n+1 times. This specific sharp operation is time consuming since Larger has the size of the original specification. These two problems could be avoided by implementing the following alternative method. The characteristics of this method is that the valid expanded term tries to keep as many as possible of its original d.c.'s.

2. Procedure Donts2 (Pb, Larger);
   a. checks if Pb is covered by Larger by performing the Sharp product operation;
   b. if more than one cube is produced then an empty cube is returned;
   c. if one cube resulted with a '1' in the output part then Pb is not

covered totally by Larger, and an empty cube is returned;

d. if a d.c. exists in the output part of the one cube then the corresponding d.c. bit is replaced by a zero in order to leave it covered by Larger. the modified Pb is returned.

This alternative procedure was implemented at a later stage in the process of improving PRONTO. It resulted in a considerable time saving, especially for large PLAs, as will be be shown in Chapter 7.

4.2.4.2 Implementation of Procedure Covering

Procedure Covering compares each term in the current specification array and the current solution array with the expanded valid product term P and stores those that are covered by P. The array of covered product terms, which might be empty, is returned. The cover checking process adopted here follows theorem 2.3.3 which could be stated as:

> Cube a is covered by cube b only if their intersection results in cube a.

Therefore, procedure Covering intersects each product term T_i of both logic arrays, one at a time, with the valid term P and if the resulting cube is equal to the former product term, T_i, then T_i is covered by P and is stored in

the array. Two procedures were considered for this equivalence check:

1. Procedure Equiv (a, c);

   (* c is the cube resulting from the intersection operation (a $\wedge$ P) *)

   If cubes a and c are equivalent in a word-by-word manner then the boolean value True is returned. A False is returned otherwise.

This procedure works efficiently for PLA examples with no output d.c.'s, but is incomplete for those which have d.c.'s. This incompleteness emerges from the fact that if an output d.c. of P is intersected with a '1' bit in cube a then the resulting bit is a d.c., and according to procedure Equiv cube a is not covered by P. In fact, cube a is covered by P as long as this d.c. bit in P is replaced with a '1'. This observation forces us to check the output part bit by bit according to the following alternative method:

2. Procedure CoverEquiv (a, c, P);

   a. if the input parts of cubes a and c differ then False is returned;

   b. compares the output bits, and for those that differ,

      - if a_i = '1' and c_i = '0' then False is returned (P does not cover a),

- if a_i = '1' and c_i = d.c. then the corresponding d.c. in P is replaced with a '1'.

c. Otherwise, returns True.

Procedure CoverEquiv, being the general one, is chosen for PRONTO. It is worth mentioning, here, that the majority of equivalence checkings stop at the first input bit mismatch in step (a) of procedure CoverEquiv. This is due to the fact that the expanded product term, P, usually does not intersect most of the remaining product terms.

By the end of procedure Tree there might exist a set of successfully expanded product terms. Each of these expanded terms, in turn, might have a set of product terms that are covered by it. The next step, which is the fourth part of PRONTO, is to choose the "Best" set of useful expanded terms to be appended to the solution, and, then to update the specification logic array.

4.2.5 Updating the Solution and Specification Arrays

Here, the decision should be made on which of the useful expanded terms is to be added to the solution. The simplest method is to pick the term that covers the largest number of cubes. Another, more involved method would place all expanded terms that cover two or more distinct product terms, that are different from the other covered sets, into the solution. The former method is the

one used, in the PRONTO version to be evaluated, because both produced the same size results when implemented on a set of PLA examples. However, the code for the latter method is included, commented out, in PRONTO. The major procedure Update is used to append the chosen expanded term to the solution array and to delete those terms, in the current solution and specification arrays, which are covered by that expanded term, as shown below.

Procedure Update (solution, coverer, maxcover, spec);
(* "coverer" is the expanded term to be added.*)
(* "maxcover" is the array of product terms that *)
(* are covered by "coverer". *)

a. Remove from the current specification and the current solution arrays any term covered by "coverer".

b. Append "coverer" to the solution array.

In the process of seeking covered terms in the current specification array, if a product term T_i was found not to be covered by "coverer" then that term is sharped with "coverer". If the result of the Sharp operation is a cube that is a single projection of T_i, then T_i is replaced by that cube. The point is that this swap shows no growth in the number of product terms, yet it allows for more future expansion directions.

4.3 Testing PRONTO's Time Consumption

Eleven PLA examples were used in testing PRONTO. Those examples were collected, by the people that worked on the project of constructing PRONTO, from previous literature. Table 4.1 shows reduction results. The second column is the number of original terms. The third column is the number of resulting product terms. The fourth and fifth are the number of input bits and output bits, respectively. The sixth and seventh columns give the processor execution time units and seconds, respectively, on a VAX UNIX timesharing system. One processor unit is 1/60 of a second.

In an attempt to find the relative time consumption by PRONTO's main parts, three PLA examples (4, 5 and 7) were subjected to a test where time was calculated for the major procedures. Figure 4.2 gives the averaged relative results. For example, procedure Donts consumes about 78% of the time spent in procedure Tree, which in turn consumes 53% of the time spent by Main.

The other 13% of time spent by the initial procedure represents time consumption of inputting the specification and outputting both the specification and solution arrays. Of course, time spent in the major procedures includes the time spent in some housekeeping and function procedures; Sharp, MergeCube, NewLogicArray, DisposeLogicArray, Cube

Table 4.1

Information on Some Examples Run on PRONTO

PLA Example	Number of Cubes in Specification	Number of Cubes in Solution	ni	no	Execution Time Units	Execution Time (sec.)
1	22	12	6	6	774	12.9
2	24	14	10	9	1100	18.3
3	33	18	9	8	1906	31.8
4	64	48	14	5	5586	93.1
5	38	24	9	14	2839	47.3
6	56	39	16	16	10374	172.9
7	91	35	15	23	12140	202.3
8	173	56	12	25	52911	881.9
9	30	13	5	4	1423	23.7
10	49	32	6	6	4670	77.8
11	37	32	12	3	1343	22.4

and some other minor ones. Those procedures are the most called ones. The last three, mentioned above, are straightforward procedures that do not show flexibility in terms of time consumption reduction. On the other hand, Sharp and MergeCube should be taken into consideration in pursuing a faster implementation of PRONTO, since Sharp was found to consume, on an average, about 13% of total execution time, while MergeCube consumed about 12%.

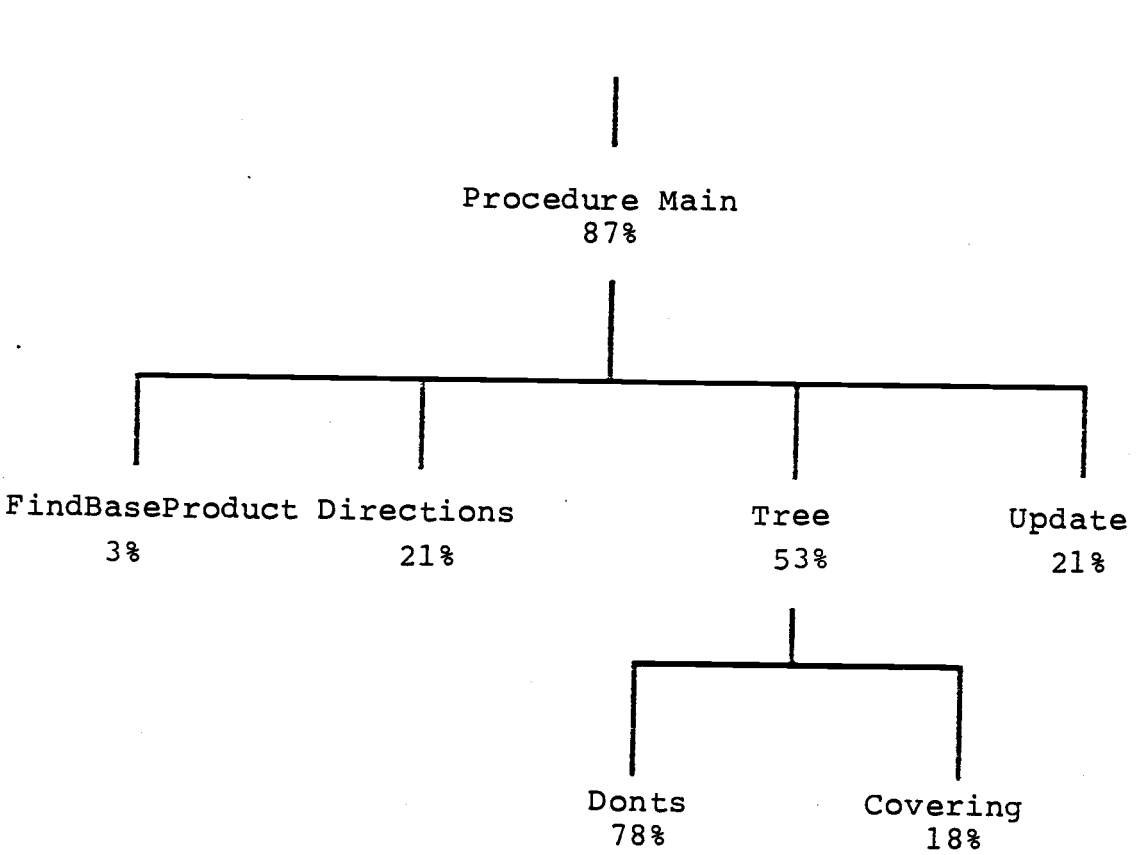

Figure 4.2 Distribution of time consumption among the major procedures.

This knowledge of the major time consuming procedures is of critical importance to the evaluation process of Chapter 5, attempting to generalize and relate time growth rate to PLA characteristics. On the other hand, in Chapter 6, based on the fact that procedure Tree is the major time consuming part of PRONTO, evaluates alternative implementations for Tree and ends up choosing the best expansion method.

CHAPTER 5

TIME COMPLEXITY EVALUATION

5.1 Introduction

Since PRONTO is a one-pass heuristically guided search method, it is not easy to find a general relation between the time spent in reduction and the characteristics of a PLA specification.

The time spent by PRONTO depends on the following:

a. The size of the PLA (number of product terms).

b. The size of the OR-plane (number of output bits per cube).

c. The size of the AND-plane (number of input bits per cube).

d. The success of the order of choosing the Base Product terms.

e. The extent to which the Base Product term is adjacent to other cubes.

f. The size of the reduced PLA.

The first three factors have the clearest effect on execution time since they are known constants. On the other hand, the last three are what makes this relation tough to generalize. This is because of the indefinite characteristics of the heuristic approach.

5.2 Purpose

In this chapter an estimate of the order of PRONTO's time complexity is found. Initially, each subroutine is to be evaluated by finding its major time growth factors. These results are to be accumulated to roughly find the dominant major factors of PRONTO's four main parts:

1. FindBaseProduct (finds the most favorable product term for expansion);
2. Directions (finds the most likely expansion directions);
3. Tree (checks the possible expanded terms for feasibility);
4. Update (adds the useful expanded term to the solution array and updates the current specification array).

Secondly, the cumulative dominant time growth factor is simplified to give an estimated order of complexity that would relate the amount of time spent in reduction to the specification array's characteristics (i.e. number of cubes, number of inputs and outputs). Finally, based on experimental results, an empirical relation is found to estimate execution time given the characteristics of a specification array.

5.3 Definitions

The following notations are to be used, throughout this chapter, as procedure parameters:

A, B, C, D and E are logic arrays of product terms.

Pb stands for a one-cube logic array.

A.nc = number of cubes in A;

A.ni = number of input variables in A;

A.no = number of output variables in A;

nwpc = number of words per cube

= (ni + no) Div 16 + 1;

nbpc = number of bits per cube = ni + no;

Spec = the current specification array;

Soln = the current solution array;

I = integer;

Cov is an array of logic arrays.

5.4 Subroutine Evaluation

The following is an individual evaluation of each of the subroutines comprising PRONTO's body.

1. NewLogicArray (A.nc, A.ni, A.no).

   NewLogicArray allocates storage for a new logic array. Although time spent here is a constant, regardless of the call, this procedure spends a considerable amount of time since it is frequently called.

2. DisposeLogicArray (A)

   DisposeLogicArray is called to dispose of an existing logic array. Processing time is a constant, but as is the case with "NewLogic-Array" it is called often and thus contributes in a considerable portion of total execution time.

3. CopyLogicArray (A)

   CopyLogicArray allocates storage for a copy of an existing array. Time spent in this subroutine is in direct proportion with the number of words in the logic array A. Therefore, the dominant factor is nwpc*A.nc.

4. Cube (A,I)

   Cube returns the Ith cube of array A. Time spent depends on the number of words per cube. Therefore it contributes in the factor nwpc.

5. Compact (A)

   Compact reduces array A by eliminating the zeroed cubes. The time spent varies according to the number of zeroed terms and to their position in the array. The array being a random access data structure complicates the elimination process. Thus a zeroed cube at the top of the array has to

be pushed down one cube at a time until it falls off. Therefore, the major time growth factor, in the worst case, would be nwpc*A.nc (the best case factor being nwpc).

6. Addcube (A, B, I).

   Addcube simply appends the Ith cube of array B to the end of array A. In this case the major time growth factor is nwpc.

7. MergeCube (A, B, I).

   MergeCube performs the same function done by "Addcube" plus some size-reducing cover checks in which the Ith cube in array B is compared to each cube in array A. The above cover checking process has A.nc*nwpc as a major time-growth factor. The factor nwpc contributed by appending the cube is seen already to be a part of the previous factor.

   "MergeCube" calls procedure "Compact" a few times whenever a covered cube needs to be deleted. "Compact" has, in a worst case, a major time-growth factor of nwpc*nc. Therefore, it is seen that the value A.nc*nwpc is the dominant factor for "MergeCube".

8. Sharp (A, B)

   Sharp performs the sharp product operation

A # B and returns the resulting cube(s) in array R. "Sharp" checks all two-cube combinations (A(k) and B(j)). First, the input parts are compared one bit at a time. Once the cubes are found not to intersect procedure MergeCube (R, A, I) stores A(I) in array R. This step turns out to be a valuable one, in terms of time saving, since usually cubes do not intersect. The major time-growth factor for the above process is A.nc * B.nc * ni * R.nc * nwpc. The factor R.nc * nwpc is the one contributed by "MergeCube".

If the cubes intersect then the actual sharp operation is performed where each input and output bit location is visited. During this process the actual sharp operation is performed whenever found needed. Then procedure MergeCube (R, C, 1) stores the sharp operation result C in array R. The process has the time-growth major factor of A.nc * B.nc * nbpc * (nwpc + R.nc * nwpc). "CopyLogicArray" is called once introducing the factor A.nc * nwpc.

Putting together the above factors the following overall major time-growth factor results, [A.nc * B.nc * ni * R.nc * nwpc] + [A.nc * B.nc * nbpc * (nwpc + R.nc * nwpc)] + [A.nc * nwpc].

The factor (nwpc + R.nc * nwpc) could be estimated by (R.nc * nwpc), then the factor (A.nc * B.nc * R.nc * nwpc) is pulled out as a common factor thus ending up with,
[A.nc * B.nc * R.nc * nwpc * (ni + nbpc)] + [A.nc * nwpc].

It is noticed that the factor A.nc * nwpc is a part of the larger factor and thus leads to the following simplified dominant factor, [A.nc * B.nc * R.nc * nwpc * (ni + nbpc)]. Considering the factor ni is a part of the larger factor nbpc, the following overall order of complexity results.

O(A.nc * B.nc * nwpc * R.nc * nbpc)

9. Addla (Cov, A)

Addla appends array A, which holds the cubes covered by a successful expansion term, to the array of covered arrays Cov. Its major time factor is (A.nc * nwpc) which is introduced by procedure "NewLogic Array".

10. Erase (A, I)

    Erase zeroes all words in the Ith cube, then calls procedure "Compact" to delete that cube. Therefore, the time consuming factor is (nwpc + (A.nc * nwpc)) which could be simplified to A.nc * nwpc.

11. SetUpTables (A)

    SetUpTables visits all output bits of the current specification A while constructing the counts arrays to be used by procedure "FindBaseProduct". Therefore, the major time-growth factor is represented by the OR-plane A.nc * no.

12. FindBaseProduct (A)

    FindBaseProduct is the first of PRONTO's four major parts in which the Base Product term is found based on an output part bit count, performed by procedure "SetUpTables". The major time spent is consumed by procedure "SetUpTables", thus resulting in A.nc * no as the major time-growth factor. "FindBaseProduct" consumes around 3% of the total execution time and about 3.3% of time spent by PRONTO's main body "Main".

13. CoverEquiv (A, B, Pb)

Equiv checks if arrays A and B are equivalent. Normally A and B are one-cube arrays. The major time factor is A.nc * nbpc.

14. Directions (Pb, A)

Directions is the second of PRONTO's four parts. It consumes almost 18.5% of total execution time and about 21% of procedure Main. It finds the possible expansion directions, that cube Pb has in comparison with the cubes of array A, according to the rules of subsection 4.2.3.

In the checking process each cube in A is visited and the input and output bits are tested. The major time-growth factor turns out to be A.nc * (2nbpc + no + 3nwpc).

It is seen that nwpc is negligible in comparison with the other factors. Therefore, the above factor may be simplified into the dominant factor of A.nc * (nbpc + no).

15. Covering (Pb, A, B)

Covering contributes about 18% of the time spent in procedure "Tree", and about 8.0% of the total execution time. It searches both Specification and Solution arrays (A and B)

for cubes covered by the successful expansion term Pb. Procedure CoverEquiv, used in the cover checking process, introduces the factor nbpc since it only checks single cubes. The overall dominant time-growth factor ends up to be (A.nc + B.nc) * nbpc.

16. Donts (Pb, $\tilde{A}$)

Donts checks the validity of an expansion term Pb by testing if Pb is covered by $\tilde{A}$ the original specification with all output dont-cares replaced by ones. This cover checking is performed by a call to procedure Sharp which might be repeated as many times as the number of d.c.'s in the output part of Pb. The major time-growth factor is

A.no * factor contributed by "Sharp",

thus ending up with the following factor,

A.no * $\tilde{A}$.nc * R.nc * nwpc * nbpc.

Procedure Donts contributes about 36.0% of the total execution time, and is the major part of procedure Tree contributing 78.0% of the time spent in it. It should be mentioned here that the factor A.no actually stands for the number of d.c. bits in the output part of Pb. Therefore if Pb has no

output dont-cares then "Sharp" is called just once, giving the factor Ã.nc * nbpc * nwpc * R.nc.

17. Tree (Pb, E, A, B, C, D, Cov)

Tree is the third and major part of PRONTO by which expansion terms are checked and the actual cover checking and reduction possibilities are sought. It spends about 46% of the total execution time and about 52% of the time spent in PRONTO's main part. The complexity of procedure Tree is evaluated here for a preorder tree expansion method. Chapter 6 will discuss some alternative methods.

Array E stands for the logic array of single-expansion directions. There are (2**E.nc) -1 different possible expanded terms, each representing one main loop in procedure Tree. In that main loop if Pb is found to be valid by "Donts" then it is stored in array B of valid expanded terms. After all the valid expanded terms are found, procedure Covering examines each valid term of array B in order to find those terms that are covered by it. Processing time has the following time-growth factors,

[(2**E.nc) * (factor by "Donts")] +

[factor by "Covering"],

which leads to

[(2**E.nc) * (A.no * Ã.nc * nbpc * nwpc * R.nc)]

+ [(A.nc + D.nc) * nbpc * B.nc].

The factor (A.nc + D.nc) represents the value of (Spec.nc + Soln.nc) which starts to be equal to the original specification array and ends up to be equal to the final solution array. Since the final solution array is always less (or equal) to the original specification size, therefore the part contributed by "Covering" can be simplified to (A.nc * nbpc * B.nc). It is seen that this factor could be considered as part of the factor from "Donts", since (A.nc * nbpc) already exists in the latter and the factor B.nc, the number of valid expanded terms, is always less than the value of (2**E.nc) representing the number of possible expanded terms. Therefore, the dominant time-growth factor could be simplified into,

(2**E.nc) * Ã.nc * no * nbpc * nwpc * R.nc.

It is found, from experimental results, that on an average the number of single expansion directions E.nc is equal to 2.

Now, let the above factor be represented by the following notation,

$$(2**E.nc) * F(A).$$

In the four examples which were studied in detail for the analysis, on the following pages, the number of expansion directions, E.nc, ranged from 1 to 8. Although the average value was close to 2, the occasional case of as many as eight directions can be costly. In the worst case, the main loop F(A) could occur as many as $(2^{E.nc}-1)$ times.

Fortunately, the above worst case has an almost zero probability, which leads to the second point to be considered - that is the majority of the possible expansion directions are not valid.

Since different numbers of expansion directions (E.nc) have different occurrence probabilities, therefore, to get an averaged result, the above dominant factor could be written as,

$$\sum_{E.nc=0}^{k} P(E.nc)*2^{E.nc} * F(A) \quad .$$

P(E.nc) stands for the probability of E.nc occurring which is taken to be the experimental relative frequency. Now looking at the factors within the main loop F (A) ($\tilde{A}$.nc, no, nbpc, nwpc and R.nc) it is seen that F (A) is independent of the number of expansion directions which permits writing the above factor

as

$$F(A) * \sum_{E.nc=0}^{k} P(E.nc) * 2^{E.nc} .$$

According to statistical results based on the examples run on PRONTO, where the upper limit k is equal to 8, the above summation added up to about 9.

The experimental data presented in Table 5.1 show the effect that the occurrence probability P(E.nc) has in defusing the drastic effect of the exponential value (2**E.nc). If P(E.nc) is exactly inversely related to the exponential value (i.e. is equal to (2**-E.nc)) then the value P(E.nc)*(2**E.nc) should be equal to unity for every E.nc. The value P(E.nc) * (2**E.nc) was found to range, experimentally, from 0 to 1.95, see

Table 5.1

Occurrence Probability of Expansion Directions

Number of Expansion Directions (E.nc)	0	1	2	3	4	5	6	7	8
Occurrence frequency P(E.nc)	0.184	0.252	0.238	0.170	0.075	0.061	0.014	0.0	0.007
$2^{-E.nc}$	1	0.5	0.25	0.125	0.063	0.031	0.016	0.008	0.004
$2^{E.nc} * P(E.nc)$	0.184	0.5	0.95	1.36	1.2	1.95	0.896	0	1.79

Table 5.1, with an average value of 0.98. The effect of P(E.nc) could be best shown by an example: for E.nc = 8, P(8) = 0.007 while (2**8) = 256 resulting in P(8)*(2**8) = 1.79. It is shown here that although for high E.nc the exponential value is devastating, the value P(E.nc) proved to neutralize that effect.

Again, taking into consideration that the majority of the expansion terms are not valid shows that, actually, a small fraction of the value 2**E.nc occurs more often than the original value.

The analysis of Table 5.1 concludes that, on an average, the exponential time growth effect by 2**E.nc is suppressed down to a constant level by its occurrence probability P(E.nc). Therefore the factor P(E.nc) * 2**E.nc is considered in the following to be a constant and is excluded from the overall time-growth major factor. Thus, the resulting simplified factor is,

$$\tilde{A}.nc * no * nbpc * nwpc * R.nc \quad .$$

18. Update (B, Pb, D, A)

Update is the fourth and last of PRONTO's main parts in which the successfully expanded term Pb is added to the Solution array B. The cubes that are already known to be covered by Pb (i.e. array D) are then deleted from the specification and solution arrays A and B. "Update" spends about 18% of the total execution time, which turns out to be about 21% of the time spent in PRONTO's main procedure "Main".

The call to procedure Sharp contributes in (nbpc + ni) * nwpc * R.nc for a time-growth factor. The overall major factor is A.nc * (D.nc * nwpc + nbpc * nwpc * R.nc) + B.nc * (D.nc * nwpc).

It was found, experimentally, that on an average the number of product terms in array D is almost equal to "1". Hence, the dominant time-growth factor could be simplified into (A.nc * nbpc * nwpc * R.nc) + (B.nc * nwpc).

As mentioned earlier, the Solution array B is of the same size order as the

specification array A, thus leading to the following dominant time factor,

A.nc * nbpc * nwpc * R.nc .

19. Large (A)

Large is called to replace all dont-care bits of the OR-plane with ones producing an as-large-as-possible specification array. The dominant factor is A.nc * no.

20. Small (A)

Small resembles procedure "Large" but instead replaces dont-cares with zeros. It also contributes the factor A.nc * no.

21. Main (A, Outfile)

Main is an interpretation of the PLA reduction algorithm PRONTO. It consumes about 87% of total execution time. "Main" calls four major procedures; "FindBaseProduct", "Directions", "Tree" and "Update", in a heuristic loop that has a size of the number of cubes in the Solution array.

The overall dominant time growth factor is Soln.nc [(A.nc * no) + (A.nc * (nbpc + no)) + (Ã.nc * no * nbpc * nwpc * R.nc) + (A.nc * nbpc * nwpc * R.nc)] .

tributed by procedure "Update" is already a part of the factor resulting from procedure "Tree". Second, the factors contributed by "Directions" and "FindBase Product" are also existant in the factor from "Tree". The above observations result in the order of time-growth complexity of

$$\mathcal{O}(\text{Soln.nc} * \tilde{A}\text{.nc} * \text{no} * \text{nwpc} * \text{nbpc}) \quad .$$

The factor R.nc depends only on the intrinsic features of the PLA (i.e. the internal structure and cube compactness). Since this factor is independent of the PLA's area constants (ni, no and nc) and due to the difficulty of evaluating its direct effect on time growth, I have chosen to drop it from the above dominant factor and considered its effect as one of the nondeterministic aspects of PRONTO's heuristic approach.

The above dominant factor stands for

$$\text{Soln.nc} * \tilde{S}\text{pec.nc} * \text{no} * \text{nwpc} * \text{nbpc} \quad .$$

5.5 Presentation

Introduced here is the idea behind the evaluation process in reaching a generalization relating execution

time to the specification's characteristics. Let T_m represent the time spent in procedure Main. Table 5.2 shows a summary of information about the four major procedures called by "Main".

Table 5.2

Summary of Results for the Four Major Procedures of PRONTO

Procedure	Percentage of T_m	Major Time-Growth Factor (one loop)
FindBaseProduct	3%	Spec.nc * no
Directions	21%	Spec.nc * (nbpc + no)
Tree	53%	Spec.nc * no * nwpc * nbpc
Update	21%	Spec.nc * nwpc * nbpc

To simplify the evaluation process let us consider the factor contributed by "FindBaseProduct" to be covered by the one contributed by "Tree". This is justified because of the smaller percentage contributed by "FindBaseProduct". The major effective factor of procedure "Tree" is Spec.nc * nbpc * no, where nwpc is discarded because of its low variability. It is also seen that procedures Directions and Update have equal percentages of execution time, which is due mainly to the major common factor Spec.nc * nbpc. Considering the above two observa-

tions, execution time, Tm, could be estimated by the following equation,

$$Tm = Soln.nc * [C_1(Spec.nc * nbpc) + C_2(Spec.nc * nbpc * no)]$$

or alternatively,

$$Tm = Soln.nc * Spec.nc * nbpc * [C_1 + C_2(no)].$$

Tm is, on an average, 87% of the total execution time T. However, taking into consideration that "InArray" and "SpamOutArray", the input/output handling procedures, have time complexities of the order of Spec.nc * nbpc shows that the time estimate equation could be modified to represent the total execution time by merely choosing different constants.

$$T = Soln.nc * Spec.nc * nbpc * [C_1 + C_2 (no)].$$

5.6 Experimental Evaluation

Execution time evaluation, here, will be based on the total execution time estimate equation reached at in the previous section, and on experimental results obtained by running '8' PLA examples. Table 5.3 shows the characteristics of the PLA examples. The estimate equation is written as

$$T/(Soln.nc * Spec.nc * nbpc) = C_1 + C_2 (no).$$

Comparison of columns (5) and (9) of Table 5.3 shows that the vlue of T/(Spec. nc * Soln.nc * nbpc) tended to decrease with an increase in the number of output bits.

Table 5.3

Characteristics of the Eight PLA Examples used in Evaluation

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Example No.	Spec. nc	Soln.nc	ni	no	nbpc	nwpc	Total Execution Time T	T + (Spec.nc * Soln.nc * nbpc)	$\frac{no}{nbpc}$
1	22	12	6	6	12	1	774	0.24	0.5
2	24	14	10	9	19	2	1,100	0.17	0.47
3	33	18	9	8	17	2	1,906	0.19	0.47
4	64	48	14	5	19	2	5,586	0.10	0.26
5	38	24	9	14	23	2	2,839	0.14	0.61
6	56	39	16	16	32	3	10,374	0.15	0.5
7(10)	49	33	6	6	12	1	4,670	0.24	0.5
8(11)	37	32	12	3	15	1	1,343	0.08	0.2

Examples '4' and '8' were not found to be consistent with the above generalization. Moreover, it turns out that both examples have a low (no/nbpc) ratio as shown in column (10) of Table 5.3. In an attempt to obtain a better linear fit relating the values of columns (5) and (9), the ratio (no/nbpc) is integrated into the original estimate equation to give,

$T/(Soln.nc * Spec.nc * nbpc) = (no/nbpc) * (C_1 + C_2 (no))$, which is simplified into

$$T/(Soln.nc * Spec.nc * no) = C_1 + C_2 (no).$$

Table 5.4 presents the results concerning the modified estimate equation. Comparing the value of T/(Spec.nc * Soln.nc * no) to (no), it is found that they have a correlation factor of (-0.70) with $C_1 = 0.49$ and $C_2 = -0.01$. Therefore we end up with the following time estimate equation,

$$T = Soln.nc * Spec.nc * no * (0.49 - 0.01 (no)).$$

Total execution time estimated by the above relation is shown in column (5) of Table 5.4 and by curve B of Figure 5.1. Curve A of Figure 5.1 represents the experimental total execution time.

In a less strict sense, since constant C_1 is the dominating one, the order of time complexity of PRONTO could be reduced to the major factor of Spec.nc * Soln.nc * no. Experimental comparison of this factor with (T)

Table 5.4

Experimental and Estimated
Total Execution Time Results

Example #	no	Experimental Execution Time (T)	T/ (Spec.nc * Soln.nc * no)	Estimated T (Linear)	Estimated T (power)
1	6	774	0.49	644	717
2	9	1100	0.36	1105	1246
3	8	1906	0.40	1802	1835
4	5	5586	0.36	6459	5010
5	14	2839	0.22	3783	4277
6	16	10,374	0.30	9389	10,125
7	6	4670	0.48	3946	3293
8	3	1343	0.38	1592	1430

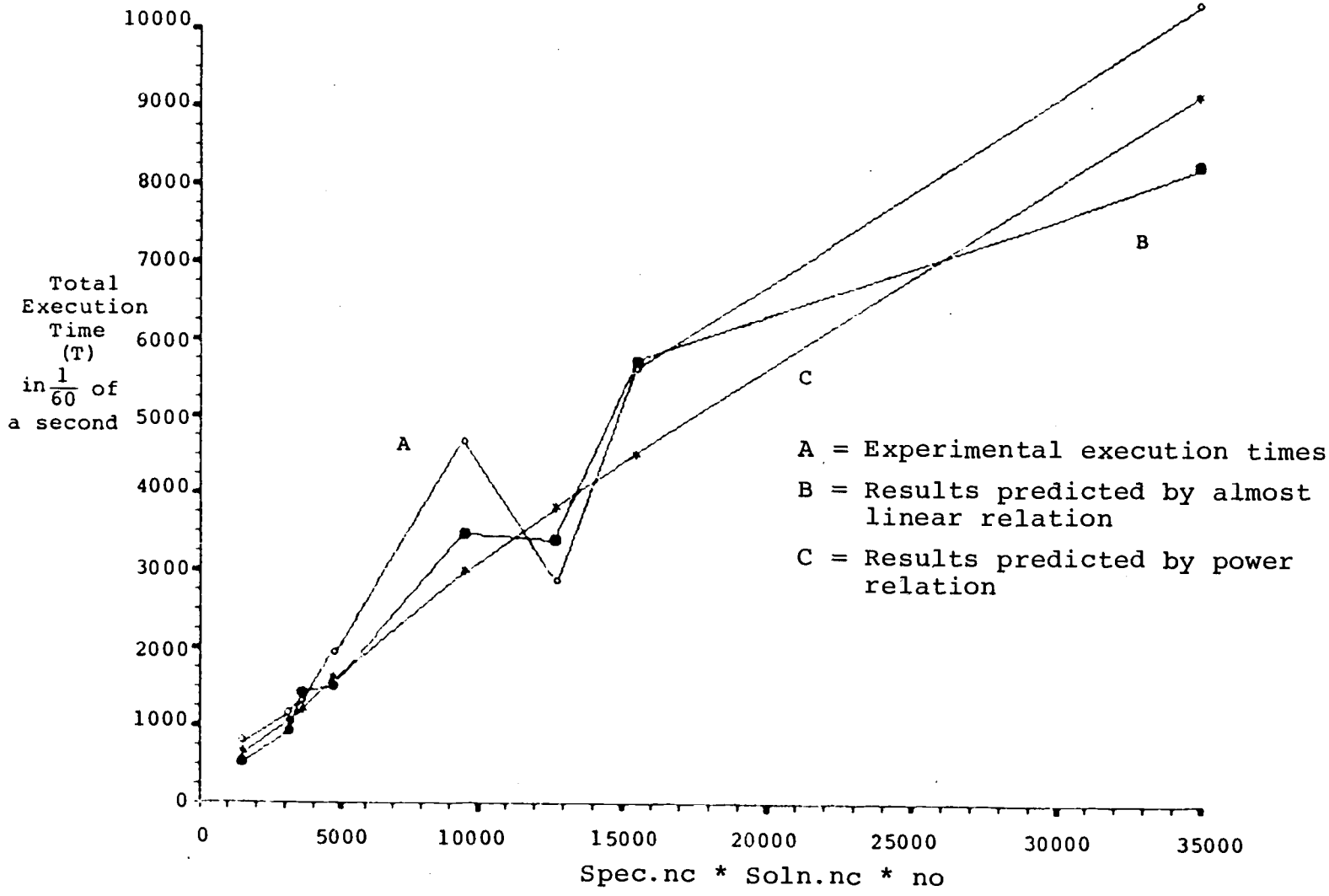

Figure 5.1. Comparison of experimental and estimated execution times.

shows that the best fit curve, with a correlation factor of 0.97, is a power relationship represented by

$$T = 1.307 * (Spec.nc * Soln.nc * no)^{0.856}$$

Column (6) of Table 5.4 and curve C of Figure 5.1 represent the total execution time estimated by the above power relationship. Curve C turns out to show an almost linear relation. This is because the power factor (0.856) is close to unity which is the case for a linear relationship. Having a power factor less than unity means the time increases in a slower manner on the long run, than it would in a linear relationship.

5.7 Conclusion

In the process of choosing the best execution time estimate relation, it was kept in mind that the power model predicts a smaller execution time, for large PLAs, than that predicted by the linear model.

Now what needs to be known is the average ratio of reduced size to original size, for large PLAs. As a rough measure, the results obtained by Brayton et al. [BRAYT], in running 56 PLAs on ESPRESSO-11 APL, are considered. The original size of these examples ranged from 10 to 1092 product terms. The average ratio of (Soln.nc/Spec.nc) was found to equal 0.67. Now, only PLAs with Spec.nc greater than 200 (i.e. 19 out of 56 PLAs) are considered and are found to have an average ratio (Soln.nc/Spec.nc) equal to

0.5. These results show that, on an average, for large PLAs a relatively greater reduction in size is obtained. This relative smaller solution array size is a direct measure of the number of main loops of PRONTO, and therefore is a measure of execution time. This result turns out to be in favor of the power model estimate.

Although the order of time complexity is determined by the algebraically higher-order terms (i.e. exponential over polynomial over linear), it is found that, for the few examples compared in Section 5.6, the actual computation time is comparable to the order of

$$O(\text{Soln.nc} * \text{Spec.nc} * \text{no}) .$$

CHAPTER 6

ALTERNATIVE APPROACHES FOR BASE PRODUCT TERM EXPANSION

6.1 Introduction

Experimental results have proven that procedure Tree is the most time consuming part of PRONTO, accounting for almost 46% of the total execution time.

In order to be able to evaluate the implementation alternatives for Tree, it is necessary to identify the major time consuming procedures that are extensively used in the process of expanding and cover checking a base Product term. There are two procedures that perform the major tasks in the expansion: The first, procedure Donts, tests the validity of a probable expansion term. The second, procedure Covering, finds the product terms that are covered by each term of the array of valid expanded terms. Comparing the orders of time complexity derived at in Chapter 5, it is found that procedure Donts contributes the major part of time consumption.

6.2 Purpose

No matter what method is used to expand a base product term, Pb, the number of cover checks performed by procedure Covering should be identical. This is due to the fact that any complete expansion method should result

in the same set of independent, valid expanded cubes (i.e. none of them is covered by any of the rest).

The intention, in this chapter, is to find an efficient (fast) implementation of expanding a base product term in order to cover other cubes. The major comparison factor of the alternative implementations would be the total expected number of calls of procedure Donts.

6.3 Expansion Methods

In this section three term-expansion procedures are considered. Each method starts with an array of possible expansion directions, based on the current base product term Pb. The number of validation checks by each method might differ since each approach has a different order of pursuing expansion terms and different termination rules. Expansion terms are represented by a tree structure with 2^n nodes, where n is the number of possible single-expansion directions. Figure 6.1 shows a tree structure for a case of three possible expansion directions (a, b, c) where the head node is the Base Product term and the rest representing the different possible expanded terms.

Node (ab) stands for the term Pb(ab) resulting from expanding the Base Product term in both directions a and b.

Two general rules, to be considered in the expansion validation process, are presented below.

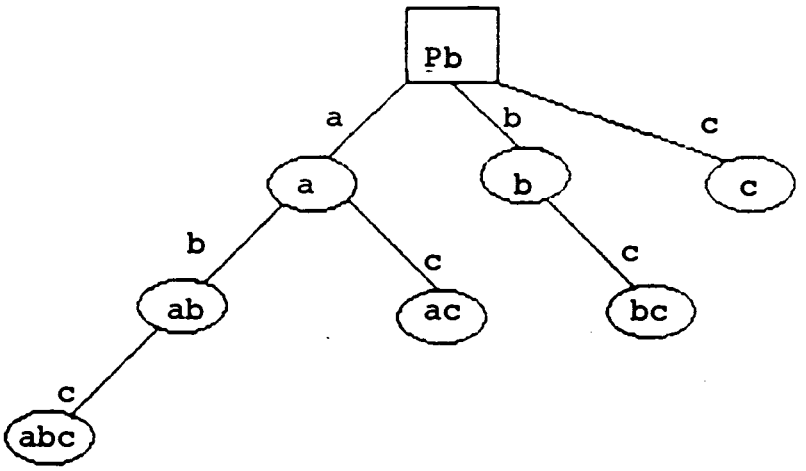

Figure 6.1 Tree representation of expansion terms formed from three expansion direction.

a. Redundancy rule. If an expanded term is proved to be valid then any smaller expanded term that is a subcube of it need not be tested for validity. For example if Pb(abc) is valid then it is also known that Pb(ac) is valid.

b. Dead-end rule. If an expanded term is found to be nonvalid then no further expansion is to be performed on that term. For example if Pb(ab) is nonvalid then Pb (abc) is also nonvalid.

6.3.1 Depth-First Expansion Method

The following pseudo-algorithm describes the process by which the Depth-first method proceeds in pursuing the expansion of a base product term. It represents a

recursive routine that follows a tree structure in carrying out the expansions. Starting with a base product term and a set of single-expansion directions, the method traverses the tree nodes (expanded terms) in a preorder fashion. The term "Base" stands for the current base term to be expanded, and it starts to be the Base Product term. The term "Node" is the current expanded term. Array "Expansions" holds the single-expansion direction. Procedure Treesearch returns an array of the current valid expanded terms.

```
Treesearch (Base, Expansions);
     While there are more directions do
          Dir←leftmost available direction in Expansions,
          Node←Base '+' Dir,
          delete Dir from Expansions,
          If Node is a valid expanded term
               Then - apply the Redundancy rule before
                      storing Node in the array of
                      valid expanded terms;
                    - Exp←Expansions;
                    - Treesearch (Node, Exp).
                End If,
          End While.
```

The Dead-end rule is taken care of internally by the "If" statement and the "While" iteration. For the Redun-

dancy rule, the valid term is compared with those terms already in the array of valid expanded terms. If it is covered by any of them then it is discarded. Otherwise, it is added to the array and those cubes, in that array, that are covered by it are deleted.

A noninterrupted preorder traversal by this method (refer to Figure 6.1), ends up with the following node order if all three directions are valid in every combination.

[a, ab, abc, ac, b, bc, c] .

In that particular case, the expansion process ends up with only Pb (abc) in the array of valid expanded terms since it covers all other combinations.

6.3.2 Level-Order Expansion Method

Level-order method traverses the expanded terms in a level-by-level manner. The expanded terms of some level are formed by using only the valid expanded terms of the previous level. The following pseudo-code describes the above recursive process. "Node" is the current base product. "Ni" is the newly expanded term. "Dirns" represents the current set of single expansion directions. "Queue" is a list of two-field records, where the first holds the current valid expanded term while the second holds an array of single expansion directions that are left to be tested with that term. "Di" is the first

available expansion direction from "Dirns". Procedure Treesearch returns an array of the current valid expanded terms.

```
Treesearch (Pb, Dirns);
    Queue ←—(Pb, Dirns),
    While Queue is not empty do
        (Node, Dirns)←—Queue
        While Dirns is not empty do
            Di←first available direction from Dirns
            delete Di from Dirns
            Ni ←—Node + Di
            If Ni is valid
                Then Queue ←—(Ni, Dirns)
                Else Do
                    - If at 1st level
                        Then delete Di from each
                        Dirns array in Queue.
            End If.
        End While Dirns
    End While Queue.
```

In this method, all small expanded terms are tested for validity before larger terms are reached thus making the Redundancy rule nonapplicable. On the other hand, the Dead-end rule is internally taken care of by the nature of the algorithm.

Figure 6.2 shows a case where only 3 directions (a, b, c) out of many are valid. Level-order expansion gives

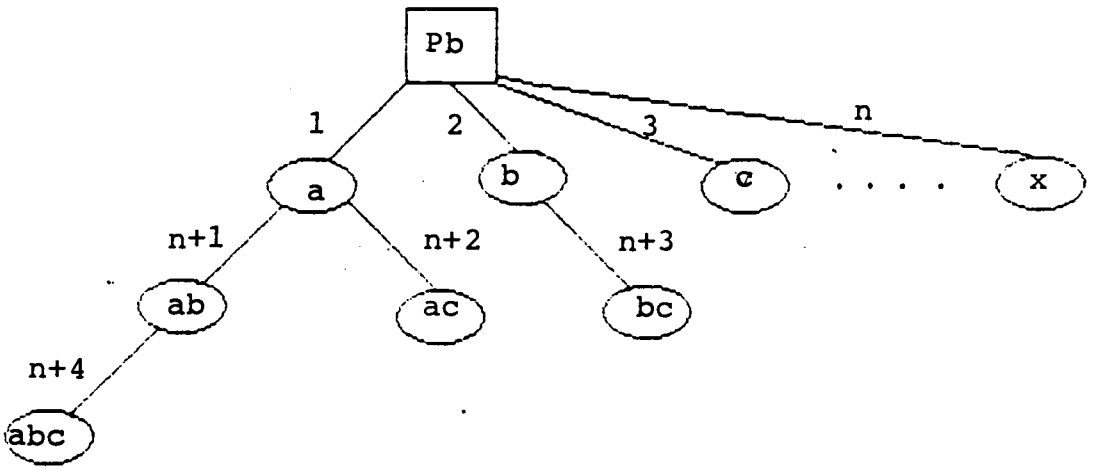

Figure 6.2 Order of traversal for the level-order expansion methods. (Case of three valid expansion directions out of n possible directions).

the following node order if all three directions are valid in every combination,

[a, b, c, ..., x, ab, ac, bc, abc] .

6.3.3 Compact-Depth Expansion Method

The Compact-depth method is a combination of the above two methods and follows the idea of the tree-search process presented by Rhyne et al. in their Direct Search Algorithm [RHYNE]. This method requires examining all single-expansion-direction terms, one at a time, to identify those valid ones. The Depth-first expansion process

is then applied to those valid single-expansion terms. Figure 6.3 shows the traversal, by this method,

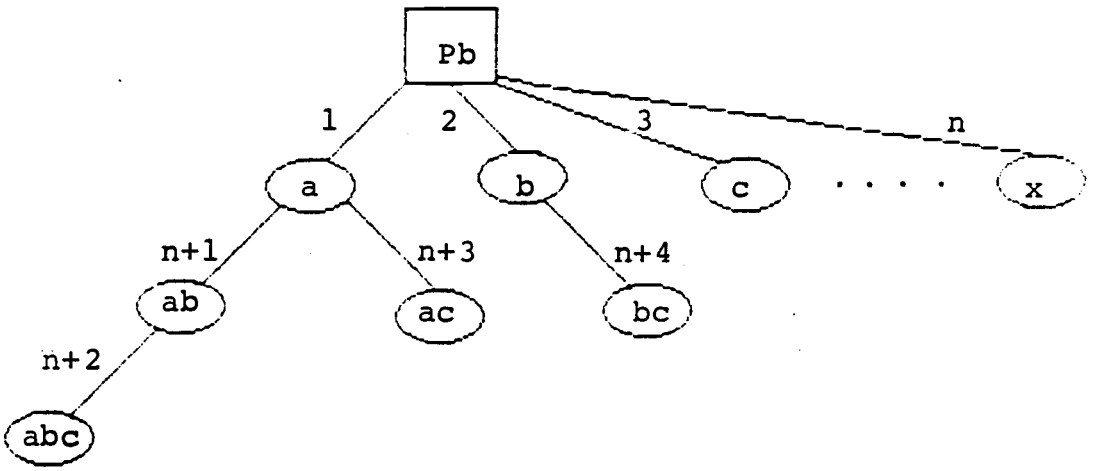

Figure 6.3 Order of traversal for the Compact-Depth expansion method. (Case of three valid expansion directions out of n possible directions).

for the same case presented in the previous section which results in the following node order,

[a, b, c, ..., x, ab, abc, ac, bc] .

6.4 Evaluation of Expansion Methods

To make the evaluation possible, four PLA examples (4, 5, 6 and 7) were run on PRONTO and monitored for the following:

a. The number of possible single-expansion directions for each base product term chosen for expansion. Results are shown in Table 6.1.

Table 6.1

Occurrence Frequency of n Possible Expansion Directions

Number of Possible Expansion Directions (n)	0	1	2	3	4	5	6	7	8
Number of Occurrences of n Possible Expansion Directions (O_n)	27	37	35	25	11	9	2	0	1
Relative Frequency ($O_n / \Sigma O_n$)	0.184	0.252	0.238	0.170	0.075	0.061	0.014	0.0	0.007
Maximum Possible Expanded Terms ($O_n * (2^n - 1)$)	0	37	105	175	165	279	126	0	255

Table 6.2

Occurrence Frequency of n Valid Expansion Directions

Number of Valid Expansion Directions (n)	0	1	2	3
Number of Occurrences of n Valid Expansion Directions (O_n)	57	66	20	4
Relative Frequency ($O_n / \Sigma O_n$)	0.392	0.446	0.135	0.027
Maximum Possible Expanded Terms ($O_n * (2^n-1)$)	0	66	60	28

b. The number of valid single-expansion directions for each base product term. Results are shown in Table 6.2.

The first rows of Tables 6.1 and 6.2 show the number of single-expansion directions (n); "possible" directions in Table 6.1 and verified valid directions in Table 6.2. The second rows show the number of occurrences of each n (O_n). The third rows represent the relative frequency of each n's occurrence ($O_n/\sum O_n$). The last rows show the maximum number of possible expanded terms for each n, ($O_n * (2^n-1)$), where 2^n-1 is the number of possible expanded terms for n expansion directions.

It was found that although the possible expansion directions ranged from zero to eight, there were never more than three valid ones. The evaluation to follow will revolve on the number of expansion terms to be tested for validity. One crucial consideration is that the majority of possible expansion directions are not valid. Table 6.1 shows that the four examples produced 291 possible expansion directions ($\sum n\ O_n$), while Table 6.2 resulted in 118 valid single-expansion directions ($\sum n\ O_n$) which amounts to about 40% of the total possible expansion directions. In order for this observation to be meaningful, the number of possible expanded terms is to be considered here, to give a more direct measure of the number of validity checks.

The four examples might result in up to 1142 possible expansion terms ($\sum(O_n*(2^n-1))$) to be tested for validity (refer to Table 6.1). Now, considering only the valid single-expansion directions of Table 6.2 we end up with 445 possible expansion terms for an upper limit. Out of the possible 445 validation checks, 291 are the ones performed on possible single-expansion-direction terms ($\sum n\ O_n$) to identify those that are valid (n stands for the number of possible single expansion directions). The rest, which amounts to 154 validation checks ($\sum(O_n*(2^n-1))$), are the ones resulting from the possible number of expansion terms that might be produced by expanding only in the valid directions.

Theoretically, the above results indicate the possibility of saving up to 60% (1142 - 445)/1142) of the time spent in validity checks. However, this is too high a percentage to be hoped for since in real application a small fraction of the number of possible expansion terms is to be checked for validity. This fraction depends on three points. The first is the number of valid single-expansion directions. The second is their relative position. The third is the validity of the combinations of valid expansion terms.

Anyhow, this possibility of time saving suggests the usefulness of pursuing the valid single-expansion direc-

tions for starting the expansion process, and therefore promoting the Level-order and Compact-depth methods. Both methods have the first-level validity checks in common. The numbers of nodes visited by the above two expansion methods differ when there are more than two valid single-expansion directions. This is shown in Table 6.3 which compares the possible numbers of validity checks, for both methods, after the first level checks. First level checks

Table 6.3

Number of Validity Checks for Level-order and Compact-depth Methods

No. of Valid Expansion Directions (n)	Range of Possible Validation Checks Not Including First Level Checks	
	Level-order	Compact-depth
1	0	0
2	1	1
3	3-4	2-4
4	6-11	3-11
5	10-35	4-35

are equal to the number of possible expansion directions for both methods. It is noticed that, for more than two valid directions, both methods have a range for the possible validity checks. For example, in the case of three

valid directions, the Level-order method's lower limit of checks (3) indicates that the Dead-end rule shows that the largest possible cube Pb(abc) is not valid and need not be checked for validity (refer to Figure 6.2). On the other hand, for the same case, the Compact-depth method's lower limit of validity checks (2) indicates that the Redundancy rule shows that the largest possible cube Pb(abc) is valid and no other smaller terms are to be tested for validity (refer to Figure 6.3).

The above observations suggest that the Compact-depth expansion method might be the faster method.

6.4.1 Quantitative Aspect of Evaluation

Table 6.4 shows the experimental relation of possible expansion directions to valid expansion directions. For example, the third row of data shows that for those base products with 3 possible expansion directions; 4 out of 25 base product terms (i.e. 0.16) have no valid directions, 17 (i.e. 0.68) have just one valid expansion direction, 2 (i.e. 0.08) have two valid directions and 2(i.e. 0.08) have three valid directions. The occurrence probability P_i (n) was estimated to be the experimental relative frequency.

The evaluations to follow are based on base product terms with one to four possible expansion directions. These terms constitute about 90% of the terms, with one or

Table 6.4

Possible and Valid Expansion-Direction Results Obtained from Four Examples

Number of Original Expansion Directions (i)	Number of Occurrences of the Following Number of Valid Single-Expansion Directions							
	0		1		2		3	
	#	$P_i(0)$	#	$P_i(1)$	#	$P_i(2)$	#	$P_i(3)$
1	15	0.417	21	0.583				
2	8	0.229	18	0.514	9	0.257		
3	4	0.16	17	0.68	2	0.08	2	0.08
4	2	0.182	5	0.455	4	0.364	0	0.0
5	1	0.111	3	0.333	3	0.333	2	0.222
6	0	0.0	1	0.5	1	0.5	0	0.0
7	0	0.0	0	0.0	0	0.0	0	0.0
8	0	0.0	0	0.0	1	1.0	0	0.0

more possible expansion directions, that are obtained by the previous four examples. Table 6.5 shows the expected number of validity checks, for the different expansion methods, in best and worst cases. The first column gives the number of possible expansion directions. The second column gives the different combinations of single direction validity. For example, for 2 possible expansion directions the combination (00) indicated that none is valid, (10 or 01) indicates one of the two directions to be valid and (11) shows that both directions are valid. The rest of the columns give the possible number of validity checks by each expansion method. In some cases a range appears for the number of checks due to the nature of the corresponding method. For example, in the case of 2 possible directions that are valid (11), the Depth-first method results in 2 validity checks if the expanded term in both directions, at the same time, is valid or in 3 validity checks if that term is not valid.

In the evaluation, the following points were taken into consideration:

1. For those combinations of valid expansion directions that have a range for the number of validity checks, the average number of checks is considered in the calculation (refer to Table 6.5).
2. For a given number of possible expansion directions in Table 6.4, the different combinations of single

Table 6.5

Validation Checks by the Three Expansion Methods for 1, 2, 3 and 4 Possible Directions

Number of Expansion Directions	Combinations of Validity	Number of Validation Checks for the Following Expansion Methods		
		Depth-Order	Level-Order	Compact-Depth
1	0	1	1	1
	1	1	1	1
2	00	2	2	2
	01	2	2	2
	10	3	2	2
	11	2-3	3	3
3	000	3	3	3
	001	3	3	3
	010	4	3	3
	011	3-4	4	4
	100	5	3	3
	101	4-5	4	4
	110	6-7	4	4
	111	3-7	6-7	5-7
4	0000	4	4	4
	0001	4	4	4
	0010	5	4	4
	0011	4-5	5	5
	0100	6	4	4
	0101	5-6	5	5
	0110	7-8	5	5
	0111	4-8	7-8	6-8
	1000	7	4	4
	1001	6-7	5	5
	1010	8-9	5	5
	1011	7-9	7-8	6-8
	1100	9-11	5	5
	1101	9-11	7-8	6-8
	1110	10-15	7-8	6-8
	1111	4-15	10-15	7-15

direction validity are not considered equally likely but are given probabilities that are estimated by the experimental relative frequency. For example, in the case of 3 possible expansion directions the occurrence of single valid-direction combinations (i.e. 001, 010, 100) has a probability of 0.68 and not 0.375 (i.e. 3/8) if compared to the eight different truth combinations.

3. The relative frequencies of Table 6.1 are taken to represent the probabilities of the different possible numbers of expansion directions. These probabilities are incorporated in the final evaluation step to reach a close-to-reality efficient estimate.

6.4.2. Calculations

The following are sample calculations of the average number of validation checks required for the case of three possible expansions directions. Refer to Tables 6.4 and 6.5 for values.

The average number of validation checks for the level-order and the Compact-depth expansion methods turns out to be the same for some numbers of possible expansion directions. The reason is that, for those cases no valid expanded term existed with more than two expansion directions. Table 6.5 clarifies this point by showing those

two methods to have the same results for terms based on two or less valid expansion directions.

6.4.2.1 Depth-first expansion evaluation

AVC = Average number of Validity Checks

$$AVC = \sum_{i=0}^{3} [Val.Checks(i) * P_3(i)]$$

$$= VC(o) * P_3(o) + \ldots + VC(3) * P_3(3)$$

$$= (3 * 0.16) + ((3 + 4 + 5)/3 * 0.68) +$$

$$((3.5 + 4.5 + 6.5)/3 * 0.08) + (5 * 0.08)$$

$$= 3.99$$

6.4.2.2 Level-Order Expansion Evaluation.

$$AVC = (3 * 0.16) + (3 * 0.68) + (4 * 0.08)$$

$$+ (6.5 * 0.08)$$

$$= 3.36$$

6.4.2.3 Compact-Depth Expansion Evaluation.

$$AVC = (3 * 0.16) + (3 * 0.68) + (4 * 0.08)$$

$$+ (6 * 0.08)$$

$$= 3.32$$

6.4.3 Results

Table 6.6 shows the average number of validity checks for the cases of 1, 2, 3, and 4 possible expansion directions. It is seen that the Depth-first expansion method is the worst of all three. The Level-order and Compact-

depth methods produce close results that would shift in favor of the latter method when the number of valid expansion directions is greater than two. This is shown in the three-expansion-direction case of Table 6.6.

The results of the Depth-first and the Compact-depth expansion methods of Table 6.6 are compared and the estimated precentage saving in time is calculated and shown in Table 6.7. For example the case of 4 possible directions might result in up to 24.8% (i.e. (5.81-4.37)/5.81) saving in time if the Compact-depth method is used instead of the Depth-first method. The savings are expected to increase for larger numbers of possible expansion directions.

6.5 Conclusion

As shown in Table 6.7 the Compact-depth expansion method produces appreciable savings in the number of validity checks for base product terms with more than one possible expansion direction.

Now, since terms with different numbers of possible expansion directions have different occurrence probabilities, a weighted overall saving in validity checks is calculated below. Refer to Tables 6.1 and 6.7. Remember, that the probabilities are estimated by the experimental relative frequencies.

$$\text{Overall Savings} = \sum_{i} [P_i * \% \text{ Saving}(i)]$$

Table 6.6

Average Number of Validation Checks Obtained by the Three Expansion Methods

Number of Possible Expansion Directions	Average Number of Validation Checks for the Following Expansion Methods:		
	Depth-First	Level-Order	Compact-Depth
1	1.00	1.00	1.00
2	2.39	2.26	2.26
3	3.99	3.36	3.32
4	5.81	4.37	4.37

Table 6.7

The Advantage of Compact-Depth over Depth-First

Number of Possible Expansion Directions	1	2	3	4
Savings in Time Consumption Using Compact-Depth Expansion Instead of Depth-first	0.0%	5.4%	16.8%	24.8%

$= (P_o * \%\ \text{Saving}\ (o) + (P_1 * \%\ \text{Saving}\ (1))$
$+ \ldots$
$= (0.0) + (0.0) + (0.238 * 5.4\%) +$
$+(0.170 * 16.8\%) + (0.075 * 24.8\%) +$
$> 6.0\%$ of validity checks

The above results show that the Compact-depth and the level-order expansion methods, compared to the Depth-first method, promise some saving in processing time even for the small problems tested here, which most of the time enter procedure Tree with few possible expansions directions. Larger PLA problems with product terms which are tightly packed, showing many adjacencies and many possibilities for expansion directions will benefit more.

6.6 Application

Both expansion methods, the Depth-first and Compact-depth, were implemented in PRONTO's code. Table 6.8 shows the results obtained by both versions of PRONTO applied to the eleven PLA examples. Time is expressed in CPU units where one unit is (1/60) of a second.

Table 6.9 gives the saving percentage in time consumption. Basically the Compact-depth expansion method proved to be better than the Depth-first method for all eleven PLA examples. However, savings ranging from 0% up to 4% were not considered as clear-cut savings since different runs of the same example might yield different

Table 6.8

Total Execution Time Obtained by the Depth-First and the Compact-Depth Versions

Example No.	Spec nc	Soln nc	Execution Time of PRONTO Version With Depth-first Method	Compact-depth Method
1	22	12	774	715
2	24	14	1,100	1,068
3	33	18	1,906	1,901
4	64	48	5,586	5,435
5	38	24	2,839	2,483
6	56	39	10,374	8,832
7	91	35	12,140	9,759
8	173	56	52,911	42,445
9	30	13	1,423	1,376
10	49	32	4,670	4,297
11	37	32	1,343	1,313

Table 6.9

Saving in Total Execution Time Resulting by Using the Compact-Depth Method over the Depth-First Method

Example No.	Saving in Total Execution Time	ni	no
1	7.6%	6	6
2	2.9%	10	9
3	0.3%	9	8
4	2.7%	14	5
5	12.5%	9	14
6	14.9%	16	16
7	19.6%	15	23
8	19.8%	12	25
9	3.3%	5	4
10	8.0%	6	6
11	2.2%	12	3

execution times but close within about 4%. Therefore, six out of the eleven examples were considered to have truly benefitted from the application of Compact-depth expansion, with savings greater than 6.0 percent.

On an average, the largest savings went to large PLA examples. Also it is noticed that for those examples with appreciable savings above 7.5% the number of output bits were equal to or greater than the number of input bits (refer to Table 6.9).

6.7 Summary

The previous results and observations show that the Compact-depth method is usually the fastest term-expansion method. Applying the Compact-depth method, instead of the Depth-first method, on eleven examples resulted in appreciable savings in execution time of up to 19.8%. Those PLA examples with considerable saving in execution time must have had many packed cubes that resulted in large numbers of possible expansion directions.

CHAPTER 7

SUMMARY AND CONCLUSION

7.1 Purpose of Thesis

The main goal was to improve PRONTO's implementation to achieve a faster solution. To start with, the PRONTO algorithm was evaluated and the major time-consuming procedures were identified. Moreover, the dominant time-growth factors (i.e. the PLA characteristics having a major effect on execution time) were identified. Then, the more flexible and most time-consuming part of PRONTO was subjected to different code implementations in an attempt to find the fastest alternative.

7.2 Major Achievements

In this section a summary of the basic work is presented in the first two subsections. The third subsection introduces some final modifications to PRONTO's code.

7.2.1 Processing Time Analysis

Execution time evaluation was the subject of Chapter 5. Each procedure was evaluated for its major time-growth factor. An overall time-growth factor dominating the execution of PRONTO was found. Based on that dominant factor and some experimental results, an estimate equation was

pursued to relate execution time to the characteristics of the original and the resulting PLA specifications. The result was the following execution time estimate equation.

$$T = 1.307 * (Spec.nc * Soln.nc * no)^{0.856}$$

where (Spec.nc) is the number of product terms of the original specification, (Soln.nc) is the number of product terms of the resulting solution and (no) is the number of output bits. This relation had a high correlation of 0.97 for the solution times of examples 1, 2, 3, 4, 5, 6, 10 and 11.

7.2.2 Base Product Expansion Alternatives

In Section 4.3 are described the results of monitoring PRONTO's major parts for time consumption. It was found that procedure Tree consumed about 46% of total execution time.

The main concern of Chapter 6 was to compare three different implementations of the Tree algorithm; The Depth-first expansion method, the Level-order method and the Compact-depth method. The fastest method, which was found to be the Compact-depth method, was chosen for the final implementation of PRONTO. Experimental results showed that the Compact-depth method, compared to the Depth-first method, resulted in considerable time saving which went up to about 19.5% for two PLA examples.

7.2.3 Final Modifications to PRONTO

During the course of evaluating PRONTO, some promising modifications for a faster implementation were detected as presented below. Table 7.1 gives the execution times for eleven PLA examples obtained by three different versions of PRONTO. Column 4 gives results of PRONTO with the Depth-first expansion method. column 5 gives results of PRONTO with the Compact-depth method. Column 6 represents the Compact-depth PRONTO version with the changes presented below.

7.2.3.1 Procedure MergeCube

MergeCube was found to consume around 12% of the total execution time. This considerable amount of time makes any attempt to modify the procedure's implementation worthwhile. As described in subsection 4.2.1, "MergeCube" first compares the input parts of two cubes, and if found equal the output parts are Ored together. Otherwise, the output parts are then compared. It is noticed here that if neither of the input parts cover the other, then the two cubes are different and there is no need to perform the output part comparison step. Therefore, a feasible modification is to test if both input parts are not covered by each other, by adding the following line:

Table 7.1

Execution Times by Three PRONTO Versions

Example #	Spec.nc	Soln.nc	Total Execution Time in CPU Units By			Time Saving by PRONTO.III Vs. PRONTO.I
			PRONTO.I (Depth-First Expansion)	PRONTO.II (Compact-Depth Expansion)	PRONTO.III (PRONTO.II + Modifications of Sec. 7.2.3)	
1	22	12	774	715	627	19%
2	24	14	1,100	1,068	680	38%
3	33	18	1,906	1,901	1,211	36%
4	64	48	5,586	5,435	4,822	14%
5	38	24	2,839	2,483	1,966 [Soln.nc=23]	31%
6	56	39	10,374	8,832	8,098	22%
7	91	35	12,140	9,759	8,067	34%
8	173	56	52,911	42,445	33,897	36%
9	30	13	1,423	1,376	1,219	14%
10	49	32	4,670	4,297	3,946	16%
11	37	32	1,343	1,313	1,207	10%

IF NOT (xley OR ylex) THEN Continue "JLOOP";

This change promises considerable time saving since cubes, most often, differ in their input parts. This modification was implemented in PRONTO's final version.

7.2.3.2 Procedure Update

As introduced in subsection 4.2.5, "Update" compares each product term in the current solution array with the terms that are covered by the chosen expanded term, in order to delete from the array those terms that are equal. Due to the low possibility of an expanded term covering a product term of the solution array, it pays to restrict the execution of this considerable part of Update. Since procedure Covering tests the solution array for terms covered by the expanded term, therefore it could flag whether the corresponding part of "Update" should be executed or not. This was done simply by setting a flag in "Covering" and testing for it in "Update".

7.2.3.3 Procedure Donts.

An alternative Donts implementation was discussed in subsection 4.2.4.1. It will be shown in the next section that using this alternative resulted in considerable saving in execution time.

The above three modifications were integrated in the

final PRONTO version with the Compact-depth expansion method. The results are shown in the 6th column of Table 7.1.

7.3 Conclusion

Implementations of optimal PLA reducers require large amounts of memory and computation time. Therefore near-optimal reducing methods are preferred for practical applications and are expected to give good, fast results using affordable memory space. PRONTO, following a one-pass, direct heuristic approach, is a suboptimal reducer that might prove to be very efficient for large, practical PLAs. It was shown that PRONTO could result in considerable size reduction. Moreover, PLA examples were reduced in an execution time that varied almost linearly with the value of Spec.nc * Soln.nc*no (i.e. the product result of the original number of product terms, the number of resulting terms and the number of output bits). The most important aspect of this observation is that actual execution time, for large-size PLAs, does not increase as drastically as would execution time of optimal reducers. For example, an optimal reducer generates all prime implicants and then extracts a minimum prime cover. "The number of prime implicants of a logic function with n input variables can be as large as $3^n/n$." (page 8, [BRAYT]).

Suboptimal solution is the price paid by heuristic methods for affordable usage of memory and computation time. There will be some possible reductions that PRONTO will not find, but most of the easy ones will be found quickly. Fast, good results for practical problems are encouraged by three PRONTO characteristics. First, PRONTO attempts to expand only in promising expandable directions. Second, PRONTO seeks to cover most cubes of the remaining specification, and does not require the primality of the expanded terms. Third, PRONTO's time complexity depends monotonically on the number of product terms in a solution, and therefore PLAs which yield the most reduction also yield the quickest solutions.

The most time consuming part of PRONTO is procedure Tree since, theoretically, it has an almost exponential major time-growth factor (2^n), where n represents the number of possible expansion directions. Nevertheless, this problem is kept under control by the restrictions imposed on selecting the directions for expansion, and by the nature of practical problems. Therefore, not many product terms, on an average, are expected to be adjacent to the base product term.

The 7th column of Table 7.1 shows that the cumulative effect of attempts to speed up PRONTO resulted in as much as 38% decrease in overall CPU time.

7.4 Suggested Future Work

A major step to be taken from here is comparing results obtained by PRONTO with results of other product-term reducing algorithms; MINI, developed by S. J. Hong, et al., [HONG], PRESTO, developed by A. Svoboda and reported by Brown [BROWN] and ESPRESSO-II, developed by R. K. Brayton, et al. [BRAYT]. Two things can be checked in the above comparison, the size of the reduced PLA and the execution time. The search objective might be to check whether PRONTO is efficiently fast for large PLAs since that was the idea behind its development.

The second suggestion would be to examine a new implementation of the process of testing each expanded term's validity. The current validation method sharps the expanded term with the original specification and bases the test on the result. This procedure might be very expensive since in the process of performing the sharp operation cube by cube the intermediate result might grow and further complicate the process. An alternative implementation might check if the expanded term intersects the complement array of the original specification. It is expected to be a better method since invalidity is proved at the first successful intersection, and the process is stopped. Another point in favor of the new procedure is that the majority of the expanded terms are nonvalid.

But, still to be considered is the price of computing the complement array.

BIBLIOGRAPHY

[BRAYT] Brayton, R. K. et al., "Logic Minimization Algorithms for VLSI Synthesis," Kluwer Academic Publishers, 1984.

[BROWN] Brown, D. W., "A State Machine Synthesizer - SMS," 18th Design Automation Conference, pp. 301-305, Jun. 81.

[DIETM] Dietmeyer, D. L., "Logic Design of Digital Systems," Allyn and Bacon, Inc., 1978.

[HONG] Hong, S. J., et al., "MINI: A Heuristic Approach for Logic Minimization," IBM J. Res. Develp., Vol. 18, No. 5, pp. 443-458, Sept. 1974.

[MART1] Martinez-Carballido, J. F., "PRONTO: A Product Term Reduction Approach," Ph.D. Dissertation, Oregon State University, 1983.

[MART2] Martinez-Carballido, J. F., and V. M. Powers "PRONTO: Quick PLA Product Reduction," 20th Design Automation Conference, pp. 545-552, Jun. 1983.

[RHYNE] Rhyne, V. T., et al., "A New Technique for the Fast Minimization of Switching Functions," IEEE Trans. Comput., Vol. C-26, pp. 757-764, Aug. 77.

[TEEL] Teel, B., Intel Corporation, private communication.