Multiple-Valued Minimization for PLA Optimization

RICHARD L. RUDELL AND ALBERTO SANGIOVANNI-VINCENTELLI, FELLOW, IEEE

Abstract—**This paper describes both a heuristic algorithm, Espresso-MV, and an exact algorithm, Espresso-EXACT, for minimization of multiple-valued input, binary-valued output logic functions. Minimization of these functions is an important step in the optimization of programmable logic arrays (PLA's). In particular, the problems of two-level multiple-output minimization, minimization of PLA's with *input decoders* and solutions to the *input encoding problem* rely on efficient solutions to the multiple-valued minimization problem. Results are presented for a large class of PLA's taken from actual chip designs. These results show that the heuristic algorithm Espresso-MV comes very close to producing optimum solutions for most of the examples. Also, results from a chip design in progress at Berkeley show how important multiple-valued minimization can be for PLA optimization.**

I. Introduction

PROGRAMMABLE LOGIC ARRAYS (PLA's) are important subsystems in the design of digital integrated circuits [1], [2]. A PLA provides a simple and regular layout strategy for Boolean equations expressed in two-level canonical form, and is usually used to implement "random" logic (random in the sense that the designer sees no regular structure in the Boolean equations). Typical examples are the control logic for a reduced-instruction set computer, or the control logic for the microsequencer of a microcoded machine. With the addition of latches for feedback. PLA's are also often used for the combinational logic in a finite-state machine. The optimization of PLA's is a useful application of computer-aided design to the automatic synthesis of custom VLSI designs.

Techniques for optimizing the structure of a PLA are becoming well understood. The optimization goals are to minimize the area occupied by the PLA and to minimize the delay through the PLA. The regular structure of a PLA means that the area of the PLA is simply proportional to the number of product terms in the array, and to a first-order approximation, the delay through the PLA is also proportional to the number of product terms (i.e., independent of the structure of each product term). A complete strategy for the design of a PLA macrocell involves: (1) logic optimization of the PLA equations including input variable assignment and output phase assignment [3], [4]; (2) optimization of the PLA layout using simple folding [5] or multiple folding [6]; and (3) generation of the mask geometries implementing the PLA [7].

This paper is concerned with the logic optimization of PLA equations, and, in particular, with the use of multiple-valued minimization for PLA optimization. Multiple-valued minimization is an extension of the classical minimization of switching functions to variables which can assume more than two values. We will first motivate interest in multiple-valued functions by showing how they can be used for PLA optimization.

Boolean minimization [8], [9] is perhaps the most important and well-established optimization procedure for PLA's. Recent advances in heuristic multiple-output minimization of Boolean equations [3] have produced algorithms able to minimize large Boolean functions within a reasonable expenditure of resources. This is important for VLSI designs where a PLA can have more than 50 inputs and 50 outputs. However, there are other logic optimizations which can be used to effectively reduce the area required by a PLA. We will describe these in turn.

1) A PLA macrocell can use *input decoders* which group the input signals into pairs [1]. The four decodes of each pair are used in the core of the PLA rather than the signals and their complements. An example of a PLA with input decoders is shown in Fig. 1. A PLA which uses input decoders can always be built with no more rows than required by the normal PLA structure. However, optimization of the logic equations which result from the pairing can often significantly reduce the total number of product terms (hence, improving both the area and delay for the PLA). The most natural way to solve this optimization problem is to use *multiple-valued* minimization [4], [10]. Each pair of variables is viewed in the multiple-valued problem as a single variable which can assume one of four values.

2) There is often the possibility of changing the encoding of either the inputs or the outputs of the PLA. For example, if the PLA is implementing an instruction decode for a processor, then the codes for the instructions can be chosen to minimize the total area required by the PLA to decode the instructions. Another example would be a finite-state machine where the encoding for the state inputs (and outputs) can be chosen to reduce the size of the PLA. As an example of output encoding alone, consider the generation of the control signals for a data path. Each action in the data path needs to be encoded uniquely, but there is typically little reason to favor any particular encoding.

The problem of *input encoding* has been successfully

Manuscript received December 17, 1986. This work was supported by the IBM Corporation in the form of a Graduate Research Fellowship, by DARPA under Grant N00039-C-0107, and by SRC under Contract 82-11-008.

The authors are with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720.

IEEE Log Number 8715658.

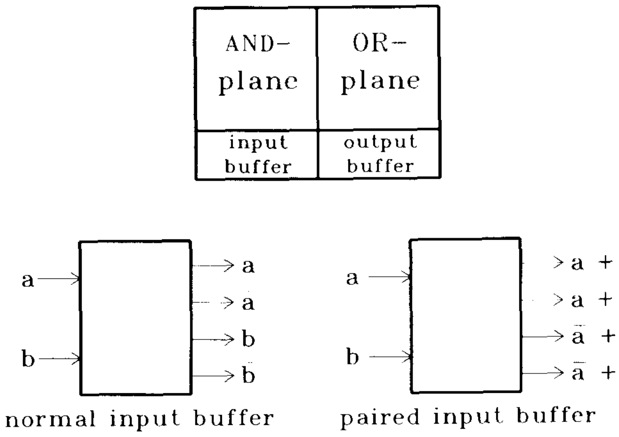


Fig. 1. PLA using input encoders.

solved using multiple-valued minimization followed by the solution of a constrained embedding problem [10]. A multiple-valued variable is used to represent the values of interest for a set of variables. This function is minimized, and then the result of the minimization is used to determine an optimal binary encoding for each value. This has been used as an approximation to the state-assignment problem [11], [12] and appears to be very successful for dense finite-state machines.

The problem of *output encoding* (and the simultaneous solution of the input and output encoding problems) is more difficult. Current techniques for solving this problem rely on the use of multiple-valued minimization [13].

3) Finally, there is the PLA optimization known as *output phase optimization* [1]. Each output of a PLA macrocell is buffered and each buffer can be made either logically inverting or noninverting. This choice, made independently for each output, can usually decrease the total size of the core of the PLA. Techniques for choosing an optimal assignment of phases for each output rely on multiple-valued minimization of the logic equations [4].

As we have shown here, there are many important optimizations which change the implementation of the logic equations in a PLA. The combination of all of these techniques can lead to a PLA implementation with significantly less area that a straightforward approach. To exploit these optimizations requires the use of an effective multiple-valued minimization algorithm. In this paper, we present the extension of the Espresso-II algorithms [3] for binary-valued minimization to the more general case of multiple-valued logic functions.

Espresso-II is a collection of algorithms for the minimization of two-level binary-valued switching functions. Some early ideas on the problem of minimizing multiple-valued Boolean functions were presented in [3]. With a simple transformation and the addition of an appropriate *don't-care set*, a multiple-valued minimization problem can be solved with any binary-valued minimizer. However, this technique fails to exploit any knowledge of the structure of the multiple-valued minimization problem, and hence can be inefficient. For example, the don't-care set can become very large, and the number of binary variables needed equals the sum of the number of values (for all variables) in the original problem. In one example we tried, the multiple-valued minimization problem for a dense 93-state machine resulted in a binary-valued minimization with more than 100 input variables, over 100 output functions, and more than 5000 don't-care terms. This is a very large problem, and it was hoped that a multiple-valued minimizer would be able to solve this problem efficiently.

Also, it is known that the multiple-output minimization problem for PLA optimization is a special case of multiple-valued minimization [14]. Therefore, it was hoped that a better understanding of the effect of the *output part* on the multiple-output minimization problem would result from working directly with multiple-valued variables. For these reasons, we became interested in extending the Espresso-II algorithms to the more general framework of multiple-valued logic functions.

In this paper we present the extension of Espresso-II to multiple-valued logic functions, and report our experience with the program **Espresso-MV** that implements these extensions. Espresso-MV was found to be more efficient than Espresso-IIC due to its more uniform treatment of the output part, and hence has replaced Espresso-IIC even for minimization of binary-valued multiple-output functions. We will also show how the Espresso-II algorithms can be extended to solve the Boolean minimization problem exactly. This exact algorithm relies on an algorithm for the *minimum cover* problem which has proven to be efficient for solving large, cyclic covering problems. We will present results from a large test set of PLA examples for several different minimization algorithms including the heuristic and exact modes of Espresso-MV. The PLA examples in the test set are graded with respect to difficulty to organize the comparisons among competing algorithms. Finally, we will report on a specific design example where input encoding has been used successfully to optimize several PLA's for custom VLSI circuits.

The paper is organized as follows. Section II contains the basic definitions of multiple-valued logic functions and the necessary extensions to the fundamental concepts of Espresso-II for manipulating multiple-valued logic functions. Section III contains a description of the basic Espresso-MV algorithm and describes how each step of Espresso is modified to deal with multiple-valued logic functions. In Section IV, we present an exact minimization algorithm which uses the basic Espresso algorithms to construct an algorithm for determining the minimum representation of a multiple-valued function. Section V will conclude with our experimental results.

II. Definitions

In this section we review the basic definitions for multiple-valued logic functions. There is a wealth of data in the literature regarding multiple-valued functions; in particular, we follow the notation and terminology of Sasao [4], [15], [16]. Chapters 2 and 3 of *Logic Minimization Algorithms for VLSI Synthesis* [3] are a valuable reference for these definitions in the special case of binary-valued multiple-output functions.

A. Multiple-Valued Functions

Let p_i for $i = 1 \cdots n$ be positive integers representing the number of values for each of n variables. Define the set $P_i \equiv \{0, \cdots, p_i - 1\}$ for $i = 1 \cdots n$ which represents the p_i values that variable i may assume, and define $B \equiv \{0, 1, *\}$ which represents the value of the function. A **multiple-valued input, binary-valued output function** f (hereafter known as a **multiple-valued function**) is a mapping

$$f: P_1 \times P_2 \times \cdots \times P_n \rightarrow B.$$

The function is said to have n multiple-valued inputs, and variable i is said to take on one of p_i possible values.

Each element in the domain of the function is called a **minterm** of the function.

An enumeration of all minterms with the value of the function is called a **truth table**.

A function which evaluates to 1 for all minterms is called a **tautology**.

The value $* \in B$ will represent a minterm for which the function value is allowed to be either 0 or 1. Hence, we allow functions which are *incompletely specified*.

An n-input, m-output switching function can be represented by a multiple-valued function of $n + 1$ variables where $p_i = 2$ for $i = 1 \cdots n$, and $p_{n+1} = m$. This special case is called a **multiple-output function**. It is easily proven that the Boolean minimization problem for multiple-output functions is equivalent to the minimization of a multiple-valued function of this form [14, theorem 4.1].

As an example of a multiple-valued function, we define a function of three variables with the first variable assuming three values ($p_1 = 3$), the second variable assuming two values ($p_2 = 2$), and the third variable assuming three values ($p_3 = 3$). The function is given by the following truth table:

X_1	X_2	X_3	value
0	0	0	1
0	0	1	1
0	0	2	0
0	1	0	1
0	1	1	0
0	1	2	1
1	0	0	0
1	0	1	1
1	0	2	1
1	1	0	1
1	1	1	1
1	1	2	0
2	0	0	*
2	0	1	*
2	0	2	0
2	1	0	1
2	1	1	*
2	1	2	0

Note that some of the function values are *, indicating that the function value may be either 0 or 1 for these minterms.

Let X_i be a variable taking a value from the set P_i, and let S_i be a subset of P_i. $X_i^{S_i}$ represents the Boolean function

$$X_i^{S_i} = \begin{cases} 0 & \text{if } X_i \notin S_i \\ 1 & \text{if } X_i \in S_i. \end{cases}$$

$X_i^{S_i}$ is called a **literal** of variable X_i. If $S_i \equiv \varnothing$, then the value of the literal is always 0, and the literal is called **empty**. If $S_i \equiv P_i$, then the value of the literal is always 1, and the literal is called **full**.

In the example, $P_1 = \{0, 1, 2\}$, and if $X_1 = 1$ then $X_1^{\{0,2\}} = 0$, and $X_1^{\{1\}} = 1$.

The **complement** of the literal $X_i^{S_i}$ (written $\overline{X_i^{S_i}}$) is the literal $X_i^{\bar{S}_i}$. The complement of a literal evaluates to 0 when the literal evaluates to 1, and vice-versa.

A **product term** (sometimes simply a **term**) is a Boolean product (or AND) of literals. If a product term evaluates to 1 for a given minterm, the product term is said to contain the minterm. If a literal in a product term is full, the product term does not depend on that variable. Without loss of generality, a product term consists of the Boolean AND of a literal for each variable.

If a literal in a product term is empty, the product term contains no minterms, and is called the **null product term** (written $\varnothing$). If all literals in a product term are full, the product term contains all minterms, and is called the **universal product term**.

A **sum-of-products** (also called a **cover**) is a Boolean sum (or OR) of product terms. If any product term in the sum-of-products evaluates to 1 for a given minterm, then the sum-of-products is said to contain the minterm.

The set X_{on} (called the **ON-set**) is the set of minterms for which the function value is 1 (i.e., $X_{\text{on}} \equiv f^{-1}(1)$). Likewise, the set X_{off} (called the **OFF-set**) is the set of minterms for which the function value is 0 (i.e., $X_{\text{off}} \equiv f^{-1}(0)$), and X_{dc} (called the **DC-set**) is the set of minterms for which the function value is unspecified (i.e., $X_{\text{dc}} \equiv f^{-1}(*)$).

An **algebraic expression** for f is a Boolean expression (written using Boolean sums and Boolean products of literals) which evaluates to 1 for all minterms of the ON-set, evaluates to 0 for all minterms of the OFF-set, and evaluates to either 0 or 1 for all minterms of the DC-set.

Proposition 1: An algebraic expression for f can always be written in sum-of-products form.

Likewise, it is possible to define a **sum term** as a Boolean sum of literals, and a **product-of-sums** as a Boolean product of sum terms. However, we restrict our attention to sum-of-product forms because of the next proposition.

Proposition 2: The minimal product-of-sums form for a function f can be derived from the minimal sum-of-products form for X_{off}.

An **implicant** of a function f is a product term which does not contain any minterm in the OFF-set of the function.

A **prime implicant** of a function f is an implicant which is contained by no other implicant of the function.

An **essential prime implicant** is a prime implicant which contains some minterm not contained by any other implicant.

In the example, $X_1^{\{0\}}X_2^{\{0,1\}}X_3^{\{0,1\}}$ is a product term (which is *not* an implicant of the function), and a sum-of-products expression for the function is

$$X_1^{\{0\}}X_2^{\{1\}}X_3^{\{0,2\}} \cup X_1^{\{1\}}X_2^{\{0\}}X_3^{\{1,2\}} \cup X_1^{\{0\}}X_2^{\{0\}}X_3^{\{0,1\}} \cup X_1^{\{1,2\}}X_2^{\{1\}}X_3^{\{0,1\}}.$$

B. Operations on Product Terms and Covers

In the definitions which follow, $S = X_1^{S_1}X_2^{S_2} \cdots X_n^{S_n}$ and $T = X_1^{T_1}X_2^{T_2} \cdots X_n^{T_n}$ represent product terms, and F and G will represent sum-of-product expressions.

A product term S is said to **contain** a product term T ($T \subseteq S$) if $T_i \subseteq S_i$ for all $i = 1 \cdots n$. If, in addition, $S \neq T$, then S is said to **strictly contain** T ($T \subset S$). S (strictly) contains T if S (strictly) contains all of the minterms that T contains.

The **complement** of a product term S ($\overline{S}$) (computed using De Morgan's law) is the sum-of-products $\cup_{i=1}^{n} \overline{X_i^{S_i}}$.

The **intersection** of product terms S and T ($S \cap T$) is the product term $X_1^{S_1 \cap T_1}X_2^{S_2 \cap T_2} \cdots X_n^{S_n \cap T_n}$ which is the largest product term contained in both S and T. If $S_i \cap T_i = \emptyset$ for some i, then $S \cap T = \emptyset$ and S and T are said to be **disjoint**. If $S \cap T$ are not disjoint, they are said to **intersect**. Likewise, the intersection of two covers F and G is defined as the union of the pairwise intersection of the cubes from each cover.

The **supercube** of S and T (*supercube*(S, T)) is the product term $X_1^{S_1 \cup T_1}X_1^{S_2 \cup T_2} \cdots X_n^{S_n \cup T_n}$ which is the smallest product term containing both S and T. Likewise, the supercube of a cover F is the smallest product term containing every product term of F.

The **distance** between S and T equals the number of empty literals in their intersection. If the distance between two cubes is 0 they intersect; otherwise they are disjoint.

The **sharp-product** of S and T ($S \# T$) equals S if S and T are disjoint, and is empty if $S \subseteq T$. Otherwise, it is the sum-of-products:

$$S \# T = S \cap \overline{T} = \bigcup_{i=1}^{n} X_1^{S_1} \cdots X_i^{S_i \cap \overline{T}_i} \cdots X_n^{S_n}.$$

$S \# T$ contains all of the minterms of S which are not contained by T.

The **consensus** of S and T (*consensus*(S, T)) is the sum-of-products:

$$\bigcup_{i=1}^{n} X_1^{S_1 \cap T_1} \cdots X_i^{S_i \cup T_i} \cdots X_n^{S_n \cap T_n}.$$

If *distance*$(S, T) \geq 2$ then *consensus*$(S, T) = \emptyset$. If *distance*$(S, T) = 1$ and $S_i \cap T_i = \emptyset$, then *consensus*(S, T) is the single product term $X_1^{S_1 \cap T_1} \cdots X_i^{S_i \cup T_i} \cdots X_n^{S_n \cap T_n}$. If *distance*$(S, T) = 0$ then *consensus*(S, T) is a cover of n terms. If the consensus of S and T is nonempty, it contains minterms of both S and T. Likewise, the consensus of two covers F and G is defined as the union of the pairwise consensus of the product terms from each cover.

The **cofactor** (or **cube restriction**) of S with repect to T (S_T) is empty if S and T are disjoint. Otherwise, it is the product term $X_1^{S_1 \cup \overline{T}_1}X_2^{S_2 \cup \overline{T}_2} \cdots X_n^{S_n \cup \overline{T}_n}$. Likewise the cofactor of a cover F with respect to a cube S (F_S) is the union of the cofactor of each cube of F with respect to S.

C. Positional Cube Notation

Let $X_1^{S_1}X_2^{S_2} \cdots X_n^{S_n}$ be a product term. This product term can be represented by a binary vector:

$$c_1^0c_1^1 \cdots c_1^{p_1-1} - c_2^0c_2^1 \cdots c_2^{p_2-1} - c_n^0c_n^1 \cdots c_n^{p_n-1}$$

where $c_i^j = 0$ if $j \notin S_i$, and $c_j^i = 1$ if $j \in S_i$. This is called the **positional cube notation** or more simply a **cube** [17]. A cube is a convenient representation for a product term, and the terms *cube* and *product term* will often be used interchangeably. (For example, a prime cube is a cube which represents a prime implicant.)

The notation c_i represents the binary vector $c_i^0c_i^1 \cdots c_i^{p_i-1}$, and $|c_i|$ represents the number of 1's in the binary vector. The notation $c_i \cup d_i$ refers to the bitwise OR of two binary vectors, $c_i \cap d_i$ refers to the bitwise AND of two binary vectors, and $\overline{c}_i$ refers to the bitwise complement of a binary vector.

A sum-of-products will be represented by a set of cubes, also called a cover. A cover also has a natural two-dimensional matrix representation, where each row of the matrix is a cube.

Continuing with the example, the following is a cover for the function:

X_1 012	X_2 01	X_3 012
100	01	101
010	10	011
100	10	110
011	01	110

The cube representation of a product term is useful because Boolean operations on the binary vectors correspond to the useful operations on the product terms. For example, one product term contains another if and only if their corresponding cubes contain each other as bit-vectors; the intersection of two cubes is the cube which results from componentwise Boolean AND of the two cubes; and the supercube of two cubes results from the componentwise Boolean OR of the two cubes.

For computer implementation of the algorithms, the cube provides a convenient data structure where one bit is used for each part of the cube. It is possible to perform operations on the cubes as word-wide operations (i.e., the bitwise Boolean AND of two 32-bit vectors on most 32-bit computers), which is more efficient than manipulating the binary vectors element by element.

D. Generalized Shannon Cofactor and Multiple-Valued Unate Functions

In [3], binary-valued unate functions were defined, and several important properties of unate functions were proven. In particular, it was shown that the problems of finding the smallest cube containing the complement of a function (an important step of REDUCE) and the problem of determining whether a function is a tautology (an important step of both IRREDUNDANT and ESSENTIAL) can be answered quickly for unate functions. When these results are combined with Shannon's theorem and the cofactor operation defined in Section II-B, efficient recursive algorithms can be devised for many basic operations. These algorithms split the function into smaller functions until each of the smaller functions becomes unate; then, a simple test is performed on the unate function to quickly determine the result. Finally, the results of each branch of the recursion are merged to produce the answer to the original problem.

The basic paradigm for manipulating multiple-valued functions is to use the multiple-valued extension of the Shannon cofactor which is called the *generalized Shannon cofactor* [18, lemma 3.2]. In Proposition 3, F is a cover of a multiple-valued function. Recall that F_{c^i} represents the cofactor of F with respect to the cube c^i.

Proposition 3: Let c^i, $i = 1 \cdots m$ be a set of cubes satisfying $\bigcup_{i=1}^{m} c^i \equiv 1$ and $c^i \cap c^j \equiv \emptyset$ for $i \neq j$. Then,

$$F = \bigcup_{i=1}^{m} c^i \cap F_{c^i}.$$

Remark: Using simple algebraic operations of Boolean algebra, it is easy to show that the operations of tautology, complementation, and computing the supercube of the complement of a cover can be computed using the properties

$$F \equiv 1 \Leftrightarrow F_{c^i} \equiv 1 \qquad \text{for } i = 1 \cdots m$$

$$\overline{F} = \bigcup_{i=1}^{m} c^i \cap \overline{F}_{c^i}$$

$$supercube(\overline{F}) = supercube\left(\bigcup_{i=1}^{m} c^i \cap supercube(\overline{F}_{c^i})\right).$$

It is not immediately obvious how to extend the definition for unate to multiple-valued functions. We will present here two extensions. The first, referred to as *weakly unate*, preserves the important property that tautology and computing the supercube of the complement are trivial operations for weakly unate functions. This will be the important extension for the Espresso algorithms. However, a weakly unate function does not satisfy other important properties which are satisfied by binary-valued unate functions. Hence, we will also define a *strongly unate* function (a stronger condition on the function than weakly unate) which preserves some of these properties. It is important to note that the definitions of weakly unate and strongly unate coincide for the special case of binary-valued functions.

1) Weakly Unate Functions:

Definition 1: A function is said to be *weakly unate* in variable X_i if there exists a j such that changing the value of X_i from value j to any other value causes the function value, if it changes, to change from 0 to 1. If a function is weakly unate in all of its variables, then the function is said to be weakly unate.

If a function is weakly unate in variable X_i, then changing the value of variable X_i to value j causes the value of the function, if it changes, to change from 1 to 0. Hence, there is no need to define both unate increasing and unate decreasing functions.

Definition 2: A cover F is said to be *weakly unate* in variable X_i if there exists a j such that all cubes *which depend on variable* X_i contain a 0 in the position j.

For example, the following cover is weakly unate because it is weakly unate in part 1 of variable 1, part 1 of variable 2, and part 5 of variable 3:

11111-00001-11110

01100-00011-01010

01010-00100-11111

00110-01001-11010

00001-11111-10110

Proposition 4: A weakly unate cover in variable X_i is a cover for a weakly unate function in variable X_i.

Proposition 5: A function f is weakly unate in variable X_i if and only if there exists a j such that each prime implicant of f which depends on variable X_i has a 0 in part j of variable X_i. Hence, a prime cover for a weakly unate function is also a weakly unate cover.

The proofs of these propositions are trivial extensions of the proof for the binary-valued case as in [3, propositions 3.3.1, 3.3.2, and 3.3.3].

A simple test for whether a cover is *weakly unate* in a variable X_i is to form the supercube of all cubes of F which do not have a full literal in variable X_i. This supercube has a 0 in any parts of X_i that are weakly unate.

The following result is useful for determining whether a weakly unate function is a tautology.

Proposition 6: Let F be a weakly unate cover in variable X_i. Let $G = \{ c \in F \mid c \text{ does not depend on } X_i \}$. Then $G \equiv 1 \leftrightarrow F \equiv 1$.

Proof: Clearly, if $G \equiv 1$, then $F \equiv 1$. Assume that j is the part required by Definition 2 for F to be weakly unate in variable X_i, and assuming $G \neq 1$. Then there exists a minterm $m \in \overline{G}$ with a 1 in value j of variable X_i. However, F is unate in X_i, and hence no terms of F have a 1 in value j of variable X_i. Therefore, it follows that $m \notin F$, and hence $F \neq 1$. ■

There is a special case when all variables are weakly unate.

Proposition 7: A weakly unate cover is a tautology if and only if one of the cubes in the cover is the universal cube.

Proof: By repeated application of Proposition 6, the function is a tautology if and only if $G = \{ c \in F \mid c$ does not depend on X_i for all $i \}$. Only the univeral cube can be in G, and hence $G \equiv 1$ if and only if the original function contains the universal cube. ■

A similar result is useful for determining the smallest cube containing the complement of a function.

Proposition 8: Let $F = \{ F^i \}$ be a weakly unate cover (F^i refers to the ith cube of F). Then,

$$supercube(\overline{F}) = \bigcap_{i=1}^{|F|} supercube(\overline{F^i}).$$

Propositions 6, 7, and 8 show that the problems of tautology and smallest cube containing the complement are trivial operations for weakly unate functions.

Weakly unate functions play the same role in multiple-valued minimization that binary-valued unate functions play in binary-valued minimization. However, they do not satisfy two properties that binary-valued unate functions satisfy: (1) all prime implicants of a binary-valued unate function are essential, and (2) the complement of a binary-valued unate function is also unate.

To understand the limitation of weakly unate, consider that, in the binary-valued case, if a cover F is unate, then the cover contains a cube c if and only if the cube is contained by some cube of the cover. This is true because F_c is unate if F is unate; hence $F_c \equiv 1$ if and only if F_c contains a universal cube. However, F_c contains a universal cube if and only if it contains a single cube which contains c. However, it is not true that F_c is weakly unate whenever F is weakly unate as the following example shows:

10-11-11-111

11-10-10-100

11-11-10-010

Cofactoring against c = 10-10-10-110 produces

11-11-11-111

11-11-11-101

11-11-11-011

which is not weakly unate in variable 4. Also note that the function F contains c, but that no single row of F contains c.

Also, in the binary-valued case, all primes of a unate function are essential, and the complement of a unate function is unate. However, the function presented earlier violates both of these properties:

11111-00001-11110

01100-00011-01010 (nonessential)

01010-00100-11111

00110-01001-11010 (nonessential)

00001-11111-10110

The complement of this function is

00110-01000-00101

11111-00001-00001

00001-11110-01001

01100-00010-10101

11000-11000-11111

10100-10100-11111

10010-10010-11111

which is weakly unate in variable 3, but not in variables 1 or 2.

It is interesting from a theoretical point of view to look for a condition stronger than weakly unate that preserves these properties.

2) Strongly Unate Functions:

Definition 3: A function is said to be *strongly unate* in variable X_i if the values of X_i can be totally ordered via $\leq$ such that changing the value of variable X_i from value j to value k (where $j \leq k$) causes the function value, if it changes, to change from 0 to 1. If all variables of a function are strongly unate, then the function is called strongly unate.

Clearly any function which is strongly unate is also weakly unate in the part of variable X_i which is less than (via $\leq$) all the remaining parts. A strongly unate function provides a total order for all of the parts, and a weakly unate function merely provides a single part which is less than all remaining parts.

Proposition 9: A strongly unate cover contains a cube if and only if the cube is contained in some cube in the cover.

Proof: If H is strongly unate, then H_c consists of those cubes of H which intersect with c, with the addition of full columns in the positions where c_i^j is 1. Hence, H_c is also strongly unate and is a tautology if, and only if, it contains a universal cube. But, H_c can contain a universal cube if, and only if, it contains a single cube which contains c. ■

Proposition 10: All primes of a strongly unate function are essential.

Proof: This proposition can be proved as [3, proposition 3.3.6] where Proposition 9 replaces [3, proposition 3.3.5]. ■

Proposition 11: The complement of a strongly unate function is strongly unate.

E. Choice of Partition

Once a cofactor F_{c^i} becomes weakly unate, it is trivial to determine if the function is a tautology, or it is trivial to compute the smallest cube containing the complement of the function. Hence, we wish to choose a partition c^i, $i = 1 \cdots m$ so that each cofactor F_{c^i} becomes a weakly unate function as quickly as possible.

The choice of partition is simplified by first choosing a *splitting* variable, followed by a choice of a partition of the splitting variable into a number of cubes which depend on only the splitting variable. Any cube in the cover which is independent of the splitting variable is duplicated in all branches of the recursion; hence this consideration enters into our choice of the splitting variable.

There is an important difference between the binary-valued case and the multiple-valued case. When the variable has only two values, the function is split with the cubes $c^1 = X_i^{\{0\}}$ and $c^2 = X_i^{\{1\}}$; the only choice is which variable X_i to use for splitting. But this choice is easy to make. The most *binate* variable [3], defined as the variable which has the most cubes in the cover which depend on it, leads to the minimum duplication of cubes after applying the Shannon cofactor. As a secondary consideration, it is desirable to keep the recursion balanced. Therefore, as a tiebreaker, Espresso chooses the variable which has the closest to an equal number of cubes with $X^{\{0\}}$ and $X^{\{1\}}$. These rules guarantee a minimum of duplication between F_{c^1} and F_{c^2} at the next level of the recursion.

When a variable has more than two values, however, we must also choose how to partition the parts of the variable into a number of different cubes. There are two possibilities.

1) Partition the values of the splitting variable into two disjoint sets: $l \subset P_i$ and $r \subset P_i$ (with $l \cap r = \emptyset$, and $l \cup r = P_i$). The function F is then split into two parts:

$$F = (X^l \cap F_{X^l}) \cup (X^r \cap F_{X^r}).$$

This enables us to maintain a binary recursive strategy. However, unlike the binary-valued case, this does not necessarily make each of the cofactors independent of the splitting variable.

2) Partition the values of the splitting variable X_i into the p_i cubes $X^{\{0\}}, X^{\{1\}}, \cdots X^{\{p_i-1\}}$. This effectively eliminates variable X_i at this level of the recursion, and forms a p_i-way splitting of the function

$$F = (X^{\{0\}} \cap F_{X^{\{0\}}}) \cup (X^{\{1\}} \cap F_{X^{\{1\}}}) \cup \cdots (X^{\{p_i-1\}} \cap F_{X^{\{p_i-1\}}}).$$

Strategy 1 is more desirable because it leaves more degrees of freedom at the next level of the recursion. For example, if a variable has eight values, splitting on all eight values (as suggested by strategy 2) gives us the eight-way tree shown in Fig. 2. Using the binary partition (as suggested by 1) and choosing the same variable for splitting at the next two levels, we get the binary tree shown in Fig. 3.

However, at either the second or third level there is more freedom in that a different variable may be chosen for splitting. Hence, strategy 1 reduces to strategy 2 in the case where the same variable is chosen at each level.

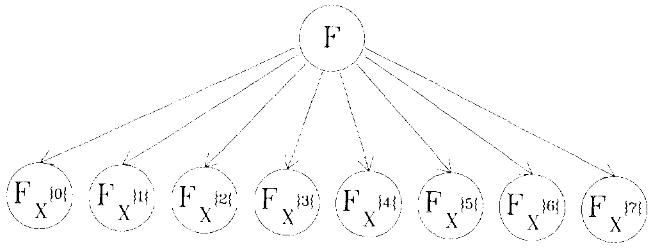


Fig. 2. Partition strategy 1, n-way split of single variable.

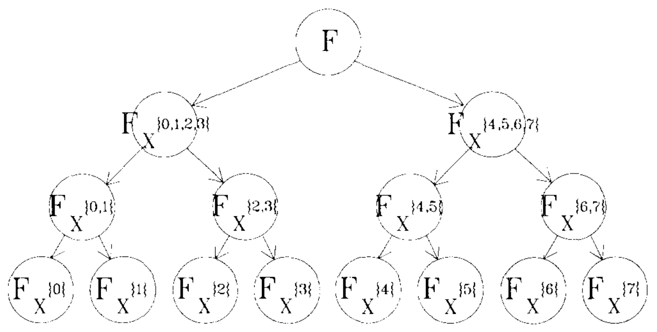


Fig. 3. Partition strategy 2, binary split of single variable.

Note, too, that strategy 1 also gives us a natural way to use a tree structure to perform the n-way merge which would be required by strategy 2.

In Espresso-MV, we use strategy 1. This reduces the problem to: (1) choosing the variable to be used as the splitting variable and (2) choosing a partition of the values of the variable. As always, there is a tradeoff between the time taken to determine what will be the next step in the recursion, and the depth and breadth of the tree to be searched. The heuristics used in Espresso-MV are kept very simple; yet, they still account for a significant percentage (on the order of 25 percent) of the execution time of the Espresso-MV algorithms.

1) Choice of Splitting Variable: As the cofactors F_{c_1} and F_{c_2} are formed, 1's are added to the cubes in each cofactor. In essence, we can view the 0's in the array for F as the "active" parts, some of which are eliminated as the cofactor is formed. The splitting variable is chosen in Espresso-MV using the following rules.

1) The variable with the largest number of active values (i.e., the multiple-valued variable with the most number of values).
2) In the case of a tie, choose the variable with the greatest number of 0's, summed over all of its values.
3) In the case of a further tie, choose the variable with the fewest 0's in the value with the greatest number of 0's.

These heuristics model the behavior of the *binate* heuristic of Espresso-II for binary-valued variables. There is no equivalent to the first rule for binary-valued variables. However, the first tiebreaker is equivalent to the most binate heuristic, and the second tiebreaker attempts to keep the recursion balanced.

2) Choice of Partition for the Splitting Variable: It was mentioned earlier that in the multiple-valued case it is more difficult to choose a partition of the values which yields a minimum of duplication of cubes during the recursion. This problem can be formulated as follows.

Problem: Find a set of values c^1 and c^2 such that the total number of cubes in $F_{c^1} \cup F_{c^2}$ is minimized.

Consider the submatrix of the cover F restricting our attention to only the columns associated with variable X_i. Consider finding a row and column permutation of the matrix into the form

$$\begin{array}{|c|c|}\hline A & 0 \\ \hline \multicolumn{2}{|c|}{B} \\ \hline 0 & C \\ \hline \end{array}$$

which minimizes the number of rows of B.

The columns of A are identified with the first half of the partition c^1, and the columns of C are identified with the second half of the partition c^2. The cubes of B are duplicated in both halves of the recursion.

This problem is a standard partitioning problem. Form a graph from the columns of the matrix by placing an edge between two columns that have 1's in the same row. The weight of this edge is equal to the number of different rows in which these columns share 1's. The problem is then to partition the nodes into two disjoint sets such that a minimum total edge weight connects the two sets.

Solving the preceding problem is potentially expensive, so we choose instead to partition the active parts to place the first $n/2$ into the set l and the remaining $n/2$ active parts into the set r. This heuristic is very fast to compute but it remains no worse than an initial n-way split on the function (as would have been provided by strategy 1).

Further work on the efficiency of these algorithms might concentrate on improving the heuristics for selecting the partitions of the function.

III. The Espresso-MV Minimization Algorithms

The Espresso-MV strategy for minimizing multiple-valued functions is similar to the strategy used by Espresso-II for multiple-output functions. In this section we describe each of the basic steps of the algorithm, concentrating on the extensions needed to handle multiple-valued functions. Pseudocode for the Espresso algorithm is shown in Fig. 4.

The first step performed by Espresso-MV is to read the function provided by the user and split the function into a cover of the ON-set, a cover of the OFF-set, and a cover of the DC-set. The user specifies a multiple-valued function by providing any two of these three covers, and Espresso-MV uses COMPLEMENT to compute the missing cover.

Processing begins by expanding each cube of the ON-set cover into a prime implicant (EXPAND). Then a minimal subset of this set of prime implicants is selected (IRREDUNDANT). Essential prime implicants are identified using ESSENTIAL and are removed from the solution before iterating over the cover.

```
/* Espresso-MV — minimize a multiple-valued Boolean function
 *      F is the ON-set of the function
 *      D is the DC-set of the function
 *      R is the OFF-set of the function
 *      cost ( F ) first considers the number of cubes in F,
 *      and then the number of literals to implement F.
 */
espresso(F, D)
{
        F_old ← F;                          /* Save original cover for verification */
        R ← COMPLEMENT (F + D);             /* Compute the complement */

        F ← EXPAND (F, R);                  /* Initial expansion */
        F ← IRREDUNDANT (F, D);             /* Initial irredundant */
        E ← ESSENTIAL (F, D);               /* Detect essential primes */
        F ← F - E;                          /* Remove essentials from F */
        D ← D + E;                          /* Add essentials to D */

        do {
                $\phi_2$ ← cost(F);

                /* Repeat inner loop until solution becomes stable */
                do {
                        $\phi_1$ ← |F|;
                        F ← REDUCE (F, D);
                        F ← EXPAND (F, R);
                        F ← IRREDUNDANT (F, D);
                } while (|F| < $\phi_1$);

                /* Perturb solution to see if we can continue to iterate */
                if (super_gasp_mode) {
                        F ← SUPER_GASP (F, D, R);
                } else {
                        F ← LAST_GASP (F, D, R);
                }

        } while (cost(F) < $\phi_2$);

        F ← F + E;                          /* Return essential to F */
        D ← D - E;
        F ← MAKE_SPARSE (F, D, R);          /* Make the solution sparse */
        if (! VERIFY (F, D, F_old)) exit("verify error");
        return F;
```

Fig. 4. The Espresso-MV algorithm.

The Espresso algorithm iterates over the cover until no improvement in the cost function is seen. The inner loop of the Espresso-MV strategy consists of reducing the implicants to nonprime cubes (REDUCE), expanding the cubes to prime implicants (EXPAND), and extracting a minimal subset of the prime implicants (IRREDUNDANT). The reduction step is important as it moves the solution away from the current local minimum without increasing the number of cubes in the cover.

When the solution stabilizes, the LAST_GASP routine performs the reduction and expansion in a different manner in an attempt to get past the current local minimum. If LAST_GASP is unable to decrease the number of cubes in the cover, there is the guarantee that no implicant can replace any implicant in the cover in order to reduce the cover cardinality. Finally, the MAKE_SPARSE routine attempts to reduce the number of transistors needed in a PLA for the function.

One interesting variant added in Espresso-MV is the routine SUPER_GASP. This procedure is used optionally instead of LAST_GASP to expend more effort in finding a better solution.

A. Tautology

Multiple-valued tautology is an important step in many heuristic minimization algorithms [18]. A well-known result [3], [18] is the following.

Proposition 12: A cover F contains a cube c if and only if F_c is a tautology.

Hence, multiple-valued tautology can be used to determine if a cover contains a cube (i.e., the cover contains all of the minterms of the cube). This is used in the IRREDUNDANT and ESSENTIAL algorithms.

The nontautology question for a multiple-valued function is NP-complete [19, p. 261], implying that there is little hope of finding a polynomial-time algorithm to solve the problem. However, in practice, the run time of the tautology algorithm accounts for a small fraction of the time for Espresso-MV. The generalized Shannon cofactor described in Section II is used to recursively divide the function into simpler functions which are examined for tautology:

Proposition 13: [18, lemma 3.3]. If a set of cubes c^i, $i = 1 \cdots m$ satisfies $\bigcup_{i=1}^{m} c^i \equiv 1$ and $c^i \cap c^j \equiv \varnothing$ for $i \neq j$, then F is a tautology if, and only if, each of F_{c^i} is a tautology for $i = 1 \cdots m$.

To reduce the complexity of answering the tautology question, we will use the properties of *weakly unate* functions given in Section II. Using Proposition 6, the size of the problem being examined for tautology can be reduced if there are any weakly unate variables.

Special Cases: Before the cubes c^1 and c^2 are chosen to decompose the function, a set of special cases are first examined.

1) If the cover has a row of all 1's (i.e., contains a universal cube), then the function is a tautology.
2) If the cover has a column of all 0's, then the function is not a tautology.
3) If the function is weakly unate, then the function is not a tautology because a row of 1's was not identified in case (1).
4) If there are any weakly unate variables, then cubes of F which are not full in the unate variable are discarded according to Proposition 6. At this point, return to case (1) to continue checking the reduced function.
5) If the cover H can be written as $A \cup B$, where A and B are defined over disjoint variable sets, then F is a tautology if and only if either A or B is a tautology. This case can be detected by finding a row and column permutation of F resulting in a matrix of the form:

$$\begin{array}{|c|c|}\hline A & 1 \\ \hline 1 & B \\ \hline\end{array}$$

where 1 represents an appropriately sized block of all 1's (and the division does not split a variable between the two halves). This partition can be easily detected with a greedy algorithm. However, in practice, such a decomposition may not occur often, and hence should only be checked for in the case that the matrix contains many 1's.

If none of these special cases apply, then two cubes c^1 and c^2 are chosen (as described in Section II-C) as a partition of a heuristically selected splitting varible, and then F_{c^1} and F_{c^2} are each checked recursively for tautology. The function is a tautology only if each of the two cofactors is a tautology.

B. COMPLEMENT

COMPLEMENT computes the complement of a multiple-valued function. In the Espresso-MV algorithms, the complement of a function is used by the EXPAND procedure. A procedure for generating the complement of a function is also a useful tool for manipulating multiple-valued functions.

The complement of a multiple-valued function is computed using the generalized Shannon expansion via the following proposition [15, lemma 3.2].

Proposition 14: Let c^i, $i = 1 \cdots m$, be a set of cubes satisfying $\bigcup_{i=1}^{m} c^i \equiv 1$ and $c^i \cap c^j \equiv \varnothing$ for $i \neq j$. Then,

$$\overline{F} = \bigcup_{i=1}^{m} c_i \cap \overline{F}_{c_i}.$$

In Espresso-MV, a splitting variable X_i and a partition the values of the variable into two halves c^1 and c^2 are selected. Half of the values of X_i are placed in c^1 and the remaining half are placed in c^2. The complement of the function is computed recursively for each of F_{c^1} and F_{c^2}, and the complement of F is $(c^1 \cap \overline{F}_{c^1}) \cup (c^2 \cap \overline{F}_{c^2})$.

1) Merging the Complement: Merging is the process of forming the union of $c^1 \cap \overline{F}_{c^1}$ and $c^2 \cap \overline{F}_{c^2}$ in such a way as to minimize the number of terms. The merge step can be viewed as a heuristic minimization algorithm that attempts to minimize the number of terms in the complement of the function while the complement is being computed.

If the same cube d appears in both $\overline{F}_{c^1}$ and $\overline{F}_{c^2}$, then the relation

$$(c^1 \cap d) \cup (c^2 \cap d) = (c^1 \cup c^2) \cap d = d$$

replaces the two cubes with the single cube d.

An expansion of the splitting variable is also attempted using one of two algorithms:

Algorithm 1:

Check, for each cube $d \in \overline{F}_{c^1}$, whether it is contained by $\overline{F}_{c^2}$. If so, use the relation

$$(c^1 \cap d) \cup (c^2 \cap \overline{F}_{c^2}) = ((c^1 \cup c^2) \cap d) \cup (c^2 \cap \overline{F}_{c^2})$$

to raise the values of c^2 in d (i.e., replace $c^1 \cap d$ with *supercube*$(c^1 \cap d, c^2)$.) The condition $d \subset \overline{F}_{c^2}$ can be checked in three ways.

a) Check if any single cube of $\overline{F}_{c^2}$ contains d: if so, $d \subset \overline{F}_{c^2}$.
b) Determine if $\overline{F}_{c^2_d}$ is a tautology.
c) Check if $(c^1 \cup c^2) \cap d$ does not intersect F; is this intersection is empty, then $d \subset \overline{F}_{c^2}$.

The conditions of Algorithm 1(b) and 1(c) are stronger then the single-cube containment of Algorithm 1(a) because they detect multiple-cube containment.

Algorithm 2:

Check, for each cube $d \in \overline{F}_{c^1}$, whether d is distance-1 from a cube $f \in F$. If so, the parts of f which are a 1 may not be raised in d (i.e., they must remain 0). Any parts of d which are not forced to be 0 by some cube $f \in F$ may be raised.

Both of these algorithms are symmetric in that the procedure is repeated for the cubes $d \in \overline{F}_{c^2}$.

Remark 1: Because the cubes have been sorted in order to remove the duplicates between the two lists, the complexity of Algorithm 1(a) can be reduced by roughly a factor of 2 by checking only the cubes of $\overline{F}_{c^2}$ which are larger than d to see if they contain d.

Remark 2: Algorithms 1(a) and 1(b) either raise all of the parts in the splitting variable or none of the parts. However, Algorithm 2 allows individual parts of a cube to be raised, and is able to determine precisely which parts can be raised and which cannot be raised. In fact, if the cubes of $c^1 \cap \overline{F}_{c^1}$ and $c^2 \cap \overline{F}_{c^2}$ are prime implicants, then the cover resulting from applying Algorithm 2 will consist of prime implicants. Because each leaf of the recursion in COMPLEMENT produces only prime implicants, we have, by induction, that the final cover returned by COMPLEMENT will consist of only prime implicants when using Algorithm 2.

Algorithm 2 is a more powerful merging algorithm and will, in general, yield a smaller representation of the complement than either Algorithm 1(a) or Algorithm 1(b). Assuming that the complexity of Algorithm 1(a) is approximately $0.5|\overline{F}_{c^1}||\overline{F}_{c^2}|$ and that of Algorithm 2 is approximately $(|\overline{F}_{c^1}| + |\overline{F}_{c^2}|)|F|$, the following heuristic is used. If

$$(|\overline{F}_{c^1}| + |\overline{F}_{c^2}|)|F| \leqslant (|\overline{F}_{c^1}||\overline{F}_{c^2}|)$$

use Algorithm 2 to raise the parts in the splitting variable; otherwise, use Algorithm 1(a). Algorithm 2 is favored (by a factor of two) because it has the possibility of generating a smaller representation of the complement (which improves the performance of the EXPAND procedure).

Note that, as mentioned in Section II-E, if the same variable is selected for splitting until all cubes in the cover are independent of that variable, then the leaves will be the functions $F_{X\{0\}}, F_{X\{1\}} \cdots F_{X\{p_i-1\}}$. Hence, in this case, the technique of splitting the parts in half provides a natural binary tree for performing the merge operation.

2) Special Cases: As usual, a set of special cases are checked before the function is split by the generalized Shannon cofactor. In the case of COMPLEMENT, the special cases are as follows.

1) If there are no cubes in the cover (i.e., the cover is empty), then the complement is the universe; if there is a row of all 1's in the cover (i.e., the cover contains a universal cube), then the complement is empty.
2) If there is only a single cube in the cover, compute the complement using De Morgan's law as described in Section II.
3) If the matrix of F contains a column of all 0's, form the cube c which has a 0 in a column which is all 0's, and a 1 in all other positions. Then, $F = c \cap F_c$, and $\overline{F} = \overline{c} \cup \overline{F}_c$. Hence, recursively compute the complement of F_c and return the union of $\overline{F}_c$ and the complement of the single cube c.
4) If all cubes of F depend on only a single variable, then the function is a tautology (because there were no columns of 0's detected in the previous step, the function must be a tautology if it depends on only a single variable) and hence the complement is empty.

If none of these special cases apply, the function is split into two pieces, and the complement is computed recursively.

C. EXPAND

The EXPAND procedure examines each cube $c \in F$ (where F is a cover of the ON-set of the binary function f) and replaces c with a prime implicant d with $c \subseteq d$. If c is not prime, then d covers more minterms of F than c does and hence it is said that c has *expanded* into a larger cube. (If c is known to be prime from a previous expansion, then c is not processed in EXPAND.) Note that each c is replaced with a single prime implicant d (out of all of the possible prime implicants which cover c) so that the number of cubes in the cover can never increase during the EXPAND step.

The goal for the minimization program is to minimize the number of cubes in F. There are several criteria that can be used in the EXPAND procedure to achieve this goal. For example, Espresso-II defines an *optimally expanded prime* as a prime d for which

a) d covers the largest number of cubes of F, and
b) among all cubes d which cover the same number of cubes of F, d covers the largest number of minterms of F.

Condition (a) is a local statement by the minimization objective, and condition (b) expresses the condition that ties be broken by covering as many minterms of F as possible.

Espresso-MV expends considerable effort in choosing a good set of parts to raise so as to achieve the minimization objective (which is to reduce the number of cubes in the cover). In particular, Espresso-MV first guarantees that if it is possible for the cardinality of F to decrease in a single EXPAND operation, that it will. In addition, the EXPAND operation is able to consider all of the prime implicants which cover a cube in order to select heuristically a single prime implicant.

1) EXPAND Cube Ordering: The order in which the cubes are expanded can affect the final result of the expansion. The cube-order dependency comes about in the heuristics which are used to expand a cube into a prime. The same ordering as used in MINI is used [20, ORDF1–ORDF3] in Espresso-MV (namely, to compute a weight for each cube as the inner product of the cube with the column sums of F, and then sort the cubes into ascending order based on the weights). This heuristic attempts to expand cubes first which are unlikely to be covered by other cubes.

Experiments were performed with a random choice of cube order, and the quality of the final results (from the complete run of Espresso-MV) were identical; however, the same results were achieved in fewer iterations using the heuristic cube ordering.

2) Blocking Matrix and Covering Matrix: Espresso-II [3] introduced the concepts of the *blocking matrix* and the *covering matrix*, and then used these matrices to guide the expansion of a cube into a prime. The blocking matrix is derived from the OFF-set by ensuring that each cube of the OFF-set has only a single 1 in the output part. (This operation is referred to as *unraveling* the output part.) The covering matrix is derived from the ON-set.

Espresso-MV views the problem differently and uses the ON-set and OFF-set directly to guide the expansion of a cube into a prime. The actual operations performed are very similar in the case of multiple-output functions. Thus, the technique used by Espresso-MV provides a different way of explaining the techniques used by Espresso-II.

The blocking matrix is less convenient for the case of multiple-valued functions because the size of the blocking matrix can become very large. A direct extension of the blocking matrix to multiple-valued functions requires unraveling each multiple-valued variable (i.e., each cube in the OFF-set which depends on variable X_i to have only a single 1 in the literal of X_i). The number of rows in the blocking matrix can become very large—a single cube r of the OFF-set of an n-variable function expands into

$$\prod_{\substack{i=1,\\ r_i \neq \text{full}}}^{n} |r_i|$$

rows in the blocking matrix (where $|r_i|$ equals the number of 1's in variable i of the cube r). This is clearly unacceptable, so we want to avoid forming the blocking matrix in this form if possible. We present here a new explanation of why it was necessary for Espresso-II to unravel the OFF-set to form the blocking matrix, and show how Espresso-MV can avoid doing so until the very last step of the expansion process (and, in many cases, completely avoid the unraveling of the multiple-valued variables).

3) Expansion of a Single Cube: Recall that the Boolean function being minimized is f, and a cover of the ON-set of the function is given by F. We assume we have access to a cover of the OFF-set of the function (which we call R), and that we are given a single cube $c \in F$ which we wish to expand. Initially, each part of the cube c which is not already a 1 belongs to the set of free parts which is denoted by *free*. As the algorithm progresses, parts are removed from *free*, and some of these parts are added to c. The algorithm terminates when *free* is empty, and at that point c is a prime cube. As a matter of terminology, when a part of c is changed from a 0 into a 1, the part is said to be *raised* or *expanded*.

Before proceeding, we first define two terms.

Definition 4: At each step of the algorithm, the *over-expanded* cube of c is the cube which results from raising simultaneously all parts of *free*. Initially, the over-expanded cube is the universe.

Definition 5: For any $f \in F$, the expansion of c which covers f is the smallest cube containing both f and c (i.e., *supercube*(c, f)); f is said to be *feasibly covered* if *supercube*(c, f) is an implicant of F.

Of course, all feasibly covered cubes of F are covered by the over-expanded cube of c, but it is possible that some cube which is covered by the over-expanded cube of c may not be feasibly covered (precisely because covering the cube would force c to intersect R). Also, initially, all parts are free so that the over-expanded cube of c is the universe. However, as parts are removed from *free*, the over-expanded cube changes, reflecting that only the parts of *free* can be raised.

Expansion Algorithm Overview:

1) Determination of essential parts: Determine which parts can never be raised and remove these from *free*; determine which parts can always be raised and raise these parts of c. Exactly how this is done will be explained later.

2) Detection of feasibly covered cubes: If there are feasibly covered cubes in F, expand c to cover one of the feasibly covered cubes by adding parts to c and removing these parts from *free*. After each such expansion, check again for parts which can never be raised, and parts which can always be raised. Repeat step 2 as long as there are feasibly covered cubes in F.

3) Expansion guided by the over-expanded cube: While there are cubes which are still covered by the over-expanded cube of c, expand c in a single part so as to overlap a maximum number of the cubes which are covered by the over-expanded cube. After expanding this part, again remove parts which can never be raised, and parts which can always be raised. Repeat step 3 as long as there are cubes of F covered by the over-expanded cube of c.

4) Finding the largest prime implicant covering the cube: When there are no cubes covered by the over-expanded cube of c, map the problem of maximal expansion of c into a covering problem whereby each minimal cover of the covering problem corresponds to a prime implicant which covers c. Choose, using some heuristic technique, a small (not necessarily minimum) cover for the covering problem. This minimal cover corresponds to a large (not necessarily maximally large) prime implicant.

1) Determination of Essential Parts: This step helps us identify parts which can always be raised and parts which can never be raised and helps us reduce F and R to just those cubes which will influence the expansion of c. The goal is to reduce the complexity of the subsequent steps.

Proposition 15: If any cube $e \in R$ is distance 1 from c, then all of the parts of the conflicting variable which are 1 in e may never be raised in c, and any part which does not appear in any cube $r \in R$ may always be raised in c.

Proposition 16: If any cube $r \in R$ is distance 1 or more from the over-expanded cube of c, then the cube r can be removed from R while still guaranteeing that the expansion of c is an implicant of f. If any cube $f \in F$ is not covered by the over-expanded cube of c, then f is not covered by any prime containing c; hence, F can be reduced.

Therefore, Proposition 15 is used to identify parts which can never be raised and Proposition 16 is used to reduce the number of cubes of F and R which have to be considered in subsequent steps. Note that any cube which is used by Proposition 15 to force parts out of the set *free* always satisfies the condition of Proposition 16 (after the parts are removed from the free set), and hence is immediately removed from further consideration.

After applying these two propositions, every cube of R is distance 2 or more from c, and every cube of R intersects the over-expanded cube of c. This is the equivalent to the statement that any single part of *free* can be raised in isolation without c intersecting R, and that it is not possible to raise simultaneously all the parts of *free*.

2) Detection of Feasibly Covered Cubes: A cube is feasibly covered if c can be expanded so as to cover the cube. A test to determine whether a cube can be feasibly covered is given by the next proposition.

Proposition 17: A cube $f \in F$ is feasibly covered if, and only if, *supercube*(f, c) is distance 1 or more from each cube of R.

Thus, each cube remaining in the cover F is tested for being feasibly covered (i.e., only the cubes of F covered by the over-expanded cube of c are checked for being feasibly covered.) To choose among the feasibly covered cubes, the feasibly covered cube which also covers the most other feasibly covered cubes is chosen. Hence, c is expanded so as to cover as many other feasibly covered cubes as possible.

After selecting a feasibly covered cube f to be covered, c is replaced with *supercube*(c, f), and parts of f are removed from the *free* set. Step 1 is repeated to find more essential parts, and then step 2 (this step) is repeated to detect any more feasibly covered cubes. The algorithm proceeds to step 3 when there are no more feasibly covered cubes.

This step allows us to guarantee that if it is possible for some expansion of a cube c to cover some other cube in F, then the expansion will be chosen and hence reduce the size of the cover.

3) Expansion Guided by the Over-Expanded Cube: When there are no more feasibly covered cubes and while there are still cubes covered by the over-expanded cube of c, then we select the single part of *free* which occurs in the most cubes which are covered by the over-expanded cube of c. We are allowed to expand c in this part because the distance between c and each cube of R is 2 or more. This has the goal of forcing c to overlap in as many parts as possible other cubes of F. After adding the part to c and removing it from *free*, repeat step 1 to detect essential parts and continue with step 3 if there are cubes still covered by the over-expanded cube of c.

This is similar to the static ordering used by MINI as the main heuristic for expanding a cube into a prime implicant. The difference is that after selecting a single part to add to c, Espresso-MV follows all consequences of that selection (by finding parts which can never be raised, and parts which can always be raised after raising the single part). Then the new set of cubes which are covered by the over-expanded cube are found, and another single part is selected. Thus, in some senses, Espresso-MV defines a dynamic ordering which is recomputed after each selection of a part to raise. Further, this heuristic is performed only while there are no cubes which can be completely covered, but while there are still cubes covered by the over-expanded cube of c.

One other important difference is that, with the strategy of MINI, it is not possible to reach all prime implicants containing c, even if all possible permutations of variables were to be considered. This is because MINI chooses to pick a single variable, and then expand maximally all of the parts in that variable before continuing to the next variable. Espresso-MV instead chooses a single part of a single variable to expand and is then free to choose another part of a different variable. Therefore, Espresso-MV is able to reach all possible primes which cover the original cube.

4) Expansion via the Minimum Covering Problem: In order for c to expand into an implicant of F, we must have that, after expanding, c be distance 1 or more from each $r^i \in R$. Let c_j^k be a Boolean variable representing the condition that part k of variable j of an expansion of c be set to 1. Also, let $(r^i)_j^k$ have the value of 1 if part k of variable j of the cube r^i is a 1. For any variable X_j, we express the condition that r^i and an expansion of c be disjoint in X_j as

$$G_{ij} = (r^i)_j^0 c_j^0 \cup (r^i)_j^1 c_j^1 \cup \cdots \cup (r^i)_j^{p_j-1} c_j^{p_j-1} = 0$$

or, equivalently,

$$G_{ij} = \bigcup_{k=0}^{p_j-1} (r^i)_j^k c_j^k = 0$$

or, using De Morgan's law, as

$$G_{ij} = \bigcap_{k=0}^{p_j-1} \left((\bar{r}^i)_j^k + \bar{c}_j^k\right) = 1.$$

We stress that the values of r^i written as $(r^i)_j^k$ are known values of either 0 or 1, and that the variables in the above equations are c_j^k.

The quantities r^i and c are disjoint if they are disjoint for some variable j. This condition is written as

$$H_i = \bigcup_{j=1}^{n} G_{ij} = 1.$$

Finally, the expansion of c is disjoint from R only if it is disjoint from all cubes $r^i \in R$, and we express this as

$$I = \bigcap_{i=1}^{|R|} H_i = 1.$$

We have a Boolean expression which expresses the condition that an assignment of $\{0, 1\}$ to the variables c_j^k results in an implicant of f. We write this in full as

$$I = \bigcap_{i=1}^{|R|} \bigcup_{j=1}^{n} \bigcap_{k=0}^{p_j-1} \left((\bar{r}^i)_j^k + \bar{c}_j^k \right).$$

An implicant of the function I corresponds to an assignment of $\{0, 1\}$ to the variables c_j^k which results in an implicant of f. Further, a prime implicant of I corresponds to an assignment of $(0, 1)$ to the variables c_j^k which is maximal in the sense that no other variable which is 0 can be made a 1; therefore, a prime implicant of I corresponds to a prime implicant of f.

Proposition 18: I is a binary-valued unate function in the variables c_j^k.

Proof: By construction, we see that I contains only the complements of the variables c_j^k, and is therefore unate.

Proposition 19: The prime implicants of I may be obtained by expanding the product-of-sum-of-product form into a sum-of-products form, and then performing single-cube containment on the resulting cover.

Proof: By [3, proposition 3.3.7], we know that a unate, single-cube contained minimal cover is in fact the set of all primes of the unate function defined by the cover.

Thus, if all c_j^k are considered variables, Proposition 19 outlines a procedure for generating all of the prime implicants of a function f given a cover for its complement. If, instead, we set the values of c_j^k to be 1 in those places where a cube c already has a 1 (and leave the variables for c_j^k where c has a 0). Proposition 19 outlines a procedure for generating all of the prime implicants which cover a cube c.

We can also modify the expression for I using De Morgan's theorem to get the equivalent form

$$\bar{I} = \bigcup_{i=1}^{|R|} \bigcap_{j=1}^{n} \bigcup_{k=0}^{p_j-1} \left((r^i)_j^k c_j^k \right).$$

Hence, we can directly write a sum-of-products expression for $\bar{I}$ and use COMPLEMENT to generate the sum-of-products form for I. We can identify the blocking matrix as proposed by Espresso-II as a representation of the Boolean function $\bar{I}$. The concept of unraveling the output part of each cube of the OFF-set in order to create the blocking matrix is equivalent to the expansion of the inner product-of-sums in the expression for $\bar{I}$ to yield a sum-of-product form for $\bar{I}$.

Thus, we have two techniques for generating all of the prime implicants of a function: one which involves repeated intersection of sum-of-products forms and one which involves the complementation of a sum-of-products form. We note here that the first formulation is equivalent to the technique outlined by Roth [21] for generating all of the prime implicants of a function.

We use the form of $\bar{I}$ to discuss now how to generate the largest prime implicant which covers a cube c. Take the cover R and unravel each variable for which there is more than 1 part in the variable. (As mentioned earlier, this is equivalent to multiplying out the product-of-sums subexpression in $\bar{I}$ to get a single sum-of-products representation of $\bar{I}$.) Let us call the resulting binary matrix R'. A binary row vector x is called a cover for R' if $R' \cdot x^T \geqslant (1, 1, \cdots, 1)^T$.

Proposition 20: Each minimal cover of R' corresponds to a prime cube in the complement of $\bar{I}$, and a minimum cover of R' corresponds to a maximum prime implicant in the complement of $\bar{I}$.

Hence, we can apply a heuristic technique (to be explained in more detail in Section IV) to compute from R' the largest prime implicant which contains c.

D. IRREDUNDANT

The IRREDUNDANT procedure extracts from a cover a minimal subset which is still sufficient to cover the same function. As before, we assume we have a cover F of the ON-set of the function f, and a cover D of the DC-set of the function f.

The cover F is first split into the *relatively essential* set E_r and the *relatively redundant* set R_r:

$$E_r \equiv \left\{ c \in F \mid c \not\subseteq (F \cup D - c) \right\}$$
$$R_r \equiv \left\{ c \in F \mid c \subseteq (F \cup D - c) \right\}.$$

The set E_r is relatively essential in the sense that all of the cubes of E_r must be retained in any subcover of F to maintain a cover of the same function.

The prime implicants of R_r are further divided into the *totally redundant* set R_t and the *partially redundant* set R_p:

$$R_t \equiv \left\{ c \in R_r \mid c \subseteq (E_r \cup D) \right\}$$
$$R_p \equiv \left\{ c \in R_r \mid c \not\subseteq (E_r \cup D) \right\}.$$

The cubes of R_t are totally redundant because they are completely covered by the relatively essential primes. The cubes of R_p are relatively redundant because, although any single cube of R_p can be removed, it is not possible to simultaneously remove all of the cubes of R_p while still maintaining a cover of f.

The cubes in R_p cause the most difficulty in trying to extract a minimum subcover of F. Consider the following simple irredundant algorithm used by many heuristic minimizers: for each cube $c \in F$ test whether $F \cup D - c$ contains c. If so, c is redundant and is removed from F. Any time a cube of E_r is tested, the cube cannot be removed. Any time a cube of R_t is tested, the cube can always be removed (regardless of the order in which the cubes are processed). However, when a cube of R_p is tested with this simple algorithm, the cube may or may not be removed, depending on the order in which the cubes are tested. With this simple algorithm, at least one member of R_p will be removed, but there is no guarantee that a maximum subset of R_p will be discarded.

The multiple-valued tautology algorithm described earlier is used to split F into E_r, R_t, and R_p.

The Espresso-MV techniques for extracting a minimal subset of primes from R_p is now described. The key in the algorithm is a simple modification of the multiple-valued tautology algorithm. Rather than testing whether the function is a tautology, the subsets of cubes which would have to be removed to prevent the function from becoming a tautology are determined.

Consider forming $H = E_r \cup R_p - c$, and using the multiple-valued tautology algorithm to determine if H_c is a tautology. H_c is a tautology because every cube of R_p is covered by the union of E_r and the remaining cubes of R_p. When we get to a leaf in the tautology algorithm (i.e., when we are able to determine that the function is a tautology), we examine the cubes which are in the cover at this leaf. If there is a cube from E_r (or D) which is the universe (in this leaf), then it is not possible to avoid the function being a tautology in this leaf. Otherwise, all of the cubes of R_p which are the universe (in this leaf) must be removed in order to prevent this leaf becoming a tautology. In terms of determining how a cover covers the cube, this is equivalent to saying the cover will fail to cover the cube if and only if all of the cubes of R_p which are universal in this leaf are discarded.

In this way, a binary matrix is formed with a cube of R_p associated with each column. At each leaf which is a tautology (and for which no cube from E_r is the universal cube), we add a row to our Boolean matrix with a 1 for each column where $(R_p)^i$ is universal. A minimal cover of this Boolean matrix corresponds to a minimal subset of the primes of R_p which must be retained in the cover for f. The heuristic covering algorithm outlined in Section IV will be used to select a good minimal cover of the covering matrix.

The algorithm proceeds by forming H_c for each $c \in R_p$, and calling a modified version of the TAUTOLOGY procedure called FIND_TAUTOLOGY. FIND_TAUTOLOGY returns the binary matrix described above. Note that after determining how c can be covered, c can be moved to the set E_r, thus improving the performance of the algorithm (because we now know how all of the minterms of c can be covered by selecting primes from R_p).

We can relate the binary matrix formed in this way to the prime implicant table of the Quine–McCluskey algorithm for Boolean minimization. By starting with the set of all prime implicants, the binary matrix created is a reduced form of the prime implicant table; rather than each row of the matrix corresponding to a minterm of the function, each row corresponds to a collection of minterms all of which are covered by the same set of prime implicants. The prime implicant table is further reduced because the set of essential primes (E_r) and the set of dominated prime implicants (R_t) have been detected and removed.

In practice, the set R_p has been observed to be small. Because the relatively essential and totally redundant sets are first identified, there is little overhead in this algorithm (compared to the simple IRREDUNDANT mentioned earlier). However, when there are partially redundant cubes, there is a much better chance of selecting a smaller subset of the partially redundant primes.

This formulation of the IRREDUNDANT algorithm, including the formation of the prime implicant table and the algorithm for finding a minimum cover for the prime implicant table, will be the basis for the exact minimization algorithm described in Section IV.

E. Essential

Essential primes are primes implicants that cover a minterm not covered by any other prime implicant, and hence must be in any cover for the function. There are efficient methods to detect those prime implicants in a cover which are essential. These essential prime implicants can be removed from the function before Espresso-MV iterates over the cover, thus providing fewer cubes which need to be processed in the inner loop.

The main theorem used for detecting which primes in a cover are essential is due to Sasao [4, Theorem A.1, 22]:

Theorem 1: Let F be written as $G \cup p$, where p is a prime implicant of the function f, and G and p are disjoint. Then, p is an essential prime implicant of f if, and only if, p is not covered by *consensus* (G, p).

The theorem can be understood by considering the following explanation: Given a $c \in G$, the distance between c and p is at least 1. If the distance is exactly 1, then the consensus of c and p is a cube with minterms in both c and p. Hence, every minterm of p covered by *consensus* (c, p) is covered by another prime implicant different from p. (That is, a prime implicant which covers *consensus* (c, p) covers all of the minterms of $p \cap$ *consensus* (c, p) and is different from p because it contains minterms of c.) Continuing in this manner for all cubes of G, every minterm of p is covered by two or more prime implicants if and only if every minterm is covered by some cube in *consensus* (G, p).

This theorem provides a simple test for detecting essential prime implicants in any cover:

Proposition 21: Given a cover F for ON-set, a cover D for the DC-set of a multiple-valued function, and a prime implicant $p \in F$, form

$$H = \textit{consensus}\left(\left((F \cup D)\#p\right), p\right).$$

Here, p is an essential prime implicant if and only if $p \not\subseteq H \cup D$.

Proof: p is to be tested as an essential prime of the function $F \cup D$. Set $G = (F \cup D)\#p$ and then $F \cup D = G \cup p$ with G and p disjoint. Hence, Theorem 1 applies and p is essential if, and only if, all of the *care* minterms of p are not covered by H. ■

Remark: The condition that all of the *care* minterms of p are not covered by H is tested by checking if $(H \cup D)_p$ is a tautology. Hence, p is an esential prime implicant if, and only if $(H \cup D)_p$ is not a tautology.

A potential problem with this procedure is that H may contain a large number of cubes (but no more than $n|F \cup D|$). In practice, the performance of the tautology algorithm depends strongly on the number of cubes in the function being tested for tautology.

For each cube of $c \in F \cup D$, we review here the procedure for generating the cubes of *consensus* $(c\#p, p)$.

1) If *distance* $(c, p) \geq 2$, then $c\#p$ is c and *consensus* $(c, p) \equiv \emptyset$. Hence, no cubes are generated for H. Likewise, if $c \subseteq p$, then $c\#p$ is empty, and no cubes are generated.

2) If *distance* $(c, p) = 1$, then $c\#p$ equals c, and a single cube results from *consensus* (c, p). Hence, a single cube is generated for H.

3) If *distance* $(c, p) = 0$, the sharp-product $c\#p$ generates one cube for every variable X_i satisfying $c_i \not\subseteq p_i$. The cube associated with such an X_i is

$$(c\#p)_j = \begin{cases} c_j \cap \bar{p}_j & \text{if } i = j \\ c_j & \text{if } i \neq j. \end{cases} \tag{1}$$

Each of these cubes is distance 1 from c, and hence generates a cube after the consensus operation according to

$$consensus\ ((c\#p), p)_j = \begin{cases} (c_j \cap \bar{p}_j) \cup p_j = c_j \cup p_j & \text{if } i = j \\ c_j \cap p_j & \text{if } i \neq j. \end{cases} \tag{2}$$

Thus, when p and c intersect, as many as n cubes may be generated for H (where n is the number of variables of c).

The number of cubes generated in the case where p and c intersect can be reduced by not generating extraneous cubes which result from the binary-valued variables (i.e., variables with two parts). Assume that $c \not\subseteq p$, and consider a cube $d \in H$ which results from a binary-valued variable X_i. This cube will necessarily have $d_i \equiv 11$, and $d_j = c_j \cap p_j$ for $j \neq i$. However, p_j cannot be 11 (it must either 10 or 01 to satisfy $c_i \not\subseteq p_i$). Hence $p \cup d$ and $\subseteq c \cap p$. Thus, with respect to Proposition 21, the single cube $c \cap p$ is sufficient to replace all of the cubes which result from considering each binary-valued variables.

This result can be improved by noticing that any cube which results from a multiple-valued variable (according to equation (2)) contains $c \cap p$; hence it is not necessary to consider the binary-value variables if any multiple-valued variable generates a cube for H.

Hence, to summarize, if c and p intersect (but $c \not\subseteq p$), a single cube is generated for each multiple-valued variable for which $c_i \not\subseteq p_i$. Then, if no cubes have been generated, the single cube $c \cap p$ is generated.

The TAUTOLOGY procedure outlined in the previous section is used to determine whether the resulting cover does indeed cover the cube c. If it does, then the prime c is nonessential. If it fails to cover the cube c, then the prime c is essential.

F. REDUCE

REDUCE transforms an irredundant cover of prime implicants into a new cover by replacing each prime implicant, where possible, with a smaller, nonprime implicant contained in the prime implicant. An irredundant, prime cover is a local minimum for the cost function, and REDUCE moves us away from the local minimum. The hope is that the subsequent EXPAND will determine a better set of prime implicants.

The main component of REDUCE (and both LAST__GASP__ and SUPER__GASP) involves the computation of the maximal reduction of a cube with respect to a cover:

Definition 6: The **maximal reduction** of a cube c with respect to a cover F is the smallest cube contained in c that can replace c in F without changing the function realized. The maximal reduction of a cube c is denoted as $\underline{c}$.

As described in MINI, the maximal reduction of a cube c with respect to a cover F and a don't-care cover D equals the supercube of $c\#(F \cup D - c)$. However, computing the reduction in this way is very inefficient.

Espresso-II uses the identify $\underline{c} = c \cap$ *supercube* $((F \cup D - c)_c)$ to compute the maximal reduction of a cube. Hence, the operation of finding the maximal reduction of a cube can be reduced to finding the smallest cube which contains the complement of a cover. This operation is readily computed recursively using the generalized Shannon cofactor.

1) REDUCE *Cube Ordering:* The reduction of a single cube depends on the form of the cover for the function. In particular, the order in which the cubes are processed for reduction affects the results of the REDUCE operation. The cubes which are reduced first will tend to reduce to smaller cubes, thus possibly preventing cubes which follow from reducing as much as they might have.

Experiments were performed with several ordering strategies (including the heuristic ordering of MINI [20] and random orderings). The solution returned by a single execution of REDUCE would vary, but did not favor any particular ordering. Just as in the case of cube ordering for EXPAND, the final result returned by Espresso-MV appeared to be largely independent of the actual ordering used. Therefore, the simple heuristic ordering of MINI was used to order the cubes. An important consideration is the LAST__GASP heuristic of Espresso which performs the reduction in a cube-independent manner.

2) Computing the Supercube of the Complement: The generalized Shannon cofactor is used to compute the supercube of the complement (i.e., the smallest cube containing the complement) of a function according to the two propositions that follow.

Proposition 22: If a set of cubes c^i, $i = 1 \cdots m$, satisfies $\bigcup_{i=1}^{m} c_i \equiv 1$ and $c^i \cap c^j \equiv \varnothing$ for $i \neq j$, then

$$supercube\ (\overline{F}) = supercube\left(\bigcup_{i=1}^{m} c^i \cap supercube(\overline{F}_{c^i})\right).$$

Proof: Using Proposition 3,

$$\overline{F} = \bigcup_{i=1}^{m} c^i \cap \overline{F}_{c^i}$$

which shows that

$$supercube(\overline{F}) = supercube\left(\bigcup_{i=1}^{m} c^i \cap \overline{F}_{c^i}\right).$$

Given that $supercube(c^i \cap \overline{F}_{c^i}) = c^i \cap supercube(\overline{F}_{c^i})$, the proposition is proven. ■

This recursion naturally terminates when F_{c^i} becomes a single cube where the following test is applied.

Proposition 23: Given a cube c,

$$supercube(\overline{c}) = \begin{cases} \varnothing & \text{if } c \text{ depends on no variables} \\ \overline{c} & \text{if } c \text{ depends on one variable} \\ \textbf{universe} & \text{if } c \text{ depends on two or more variables.} \end{cases}$$

Remark: If c depends only on the variable X_i, then $\overline{c}$ is a single cube resulting from the bitwise complement of c_i.

Proof: The proof is trivial if one considers computing the complement of a cube using De Morgan's law. If the cube depends on more than two variables, then the complement contains more than two cubes. Each of these cubes depends on only a single variable (with the remaining literals all full); hence the supercube of these cubes is the universe. If the cube depends on only a single variable, there is only one cube in the complement. Finally, if the cube is the universe, the complement is empty.

However, there is also the following, more powerful result.

Proposition 13: If F is a weakly unate cover and F^i represents the ith cube in the cover, then

$$supercube(\overline{F}) = \bigcap_{i=1}^{|F|} supercube(\overline{F^i}).$$

Thus, if the cover is weakly unate, this result is applied to determine the supercube of the complement of a cover. Further, only the cubes of the weakly unate cover which depend on a single variable need be considered (assuming the cover does not contain a universal cube), because the supercube of the complement of any cube which depends on two or more variables is the universe and hence does not affect the intersection.

There are two other results (easily derived from De Morgan's law) which can be useful in reducing the amount of work necessary to compute the supercube of the complement of a function.

Proposition 25: If the cover F contains a column of 0's, form the cube c which has a 0 in each position, where F has a column of all 0's, and 1 elsewhere. Then, from the identity $F = c \cap F_c$, it is seen that

$$supercube(\overline{F}) = supercube(supercube(F_{\overline{c}}), supercube(\overline{c})).$$

Hence, if there is a column of 0's in the matrix for F, this proposition is applied to compute $supercube(\overline{F})$. In particular, if F has a column of 0's in two separate variables, then it is immediately determined that $supercube(\overline{F}) \equiv$ **universe.**

Proposition 26: If F can be factored into the form $F = A \cup B$, where A and B are over disjoint variable sets, then

$$supercube(\overline{F}) = supercube(\overline{A}) \cap supercube(\overline{B}).$$

Detecting such a partition of H corresponds to finding a row and column permutation resulting in the form

A	1
1	*B*

where 1 represents an appropriately sized block of all 1's (and the division does not split a variable between the two halves). As in the case of tautology, such a partition is easily determined with a simple greedy strategy. In practice, such a decomposition may not be common, and should only be checked for when the matrix contains many 1's.

G. LAST_GASP *and* SUPER_GASP

The basic iteration of Espresso-II (REDUCE, EXPAND, IRREDUNDANT) faces the following obstacles: (1) The EXPAND step uses heuristics to choose one prime implicant (from all of the prime implicants which cover a cube) to replace each cube in the cover; and (2) the REDUCE algorithm is cube-order dependent so that cubes which are reduced first tend to reduce more than cubes which are reduced later. Espresso-II uses the LAST_GASP algorithm (and Espresso-MV adds the SUPER_GASP algorithm) for improving the basic minimization algorithm. These are described in this section.

1) LAST_GASP: This algorithm computes the maximal reduction of every cube of the ON-set cover F. (If a cube cannot be reduced it is ignored.) The EXPAND algorithm is used to expand each cube, even if it becomes covered by a previous cube expanding. As shown in [3], those cubes that succeed in covering some other reduced cube are potentially useful primes for reducing the cardinality of the cover. In fact, if no maximally reduced cube can expand to cover another maximally reduced cube, then there is no way to improve the current cover without removing or replacing two or more primes simultaneously. In this sense, the current solution is a *deep* local minimum. The new primes which do succeed in covering two maximally reduced cubes are added to the cover F, and the IRREDUNDANT procedure then extracts a minimal subcover. Because the number of reduced cubes which can expand to cover other reduced cubes tends to be very small, this technique is applicable to a wide range of problems.

2) SUPER_GASP*:* Espresso-MV also has an optional routine SUPER_GASP. This algorithm computes the maximal reduction of each cube of the cover F and then generates *all* of the prime implicants which cover the cube (rather than only a single prime implicant which covers the cube). In order to generate all of the prime implicants which cover a cube, the algorithm given in subsection III-C (EXPAND) is used. By sorting this set of prime implicants, duplicate prime implicants are easily detected. IRREDUNDANT then extracts a minimal subcover from the remaining set of prime implicants. Note that if IRREDUNDANT returns the minimum number of cubes necessary to implement the function, then no single iteration of REDUCE, EXPAND, and IRREDUNDANT can do any better from the same starting point.

Of course, the process of generating all of the primes which cover the maximally reduced cubes may greatly expand the size of the cover. (In particular, if the original cover were all minterms, the generation of all of the primes covering each minterm would be an inefficient way to generate all of the primes for the function.) The program Espresso-MV is careful to terminate the generation of all of the primes in the case where there are too many primes, in which case the LAST_GASP strategy is used instead. Experimental results show that this option is efficient for a wide range of problems, and occasionally results in a better solution.

H. MAKE_SPARSE

When the outer loop of the Espresso-MV algorithm terminates, the solution consists of an irredundant cover of prime implicants which represents the original function. However, depending on the final implementation of the multiple-valued function, we may desire a final cover which does not necessarily consist of prime implicants. One goal is to reduce the number of transistors needed to implement each literal of a cube. This depends on the number of 0's and 1's in the literal, but it also depends on the type of variable, as shown in Table I.

TABLE I
TRANSISTORS PER LITERAL IN A PLA

Variable Type	Number of Transistors	Comment
binary-valued variable	count number of 0's	dense
multiple-valued variable (for a two-bit decoder)	count number of 0's	dense
multiple-valued variable (for the output part)	count number of 1's	sparse
multiple-valued variable (for the input encoding problem)	count number of 1's (unless literal is full)	sparse

For example, if the function being minimized represents a two-level multiple-output PLA function, then each 0 in the cube for a binary-valued variable corresponds to a transistor in the AND plane of the PLA, but each 1 in the multiple-valued output variable corresponds to a transistor in the OR plane of the PLA. With these observations, we define, for each variable, whether the variable is to be a **sparse variable** or a **dense variable.** The MAKE_SPARSE procedure then attempts to satisfy these goals.

MAKE_SPARSE consists of two steps: LOWER_SPARSE removes redundant parts from the sparse variables, and RAISE_DENSE attempts to add parts to the dense variables (which may be possible following LOWER_SPARSE because the cubes are no longer prime implicants). These two algorithms are iterated until there is no more reduction of any sparse variable, or until there is no more expansion of any dense variable.

During the first iteration of LOWER_SPARSE and RAISE_DENSE, the cardinality of the cover cannot decrease (because the cover is an irredundant, and consists of prime implicants). However, in extreme cases, it is possible for the cardinality to decrease in subsequent iterations. In fact, the procedure MAKE_SPARSE can be viewed as a complete minimization algorithm. (The **pop** program from Berkeley [23] uses essentially this simple algorithm, but without the powerful techniques for each of the basic steps as in MAKE_SPARSE; however, this minimization algorithm is restricted in the size of the set of prime implicants which it can explore.)

In the discussion that follows, we assume, as usual, that F is a cover for the ON-set, D is a cover for the DC-set, and R is a cover for the OFF-set.

1) LOWER_SPARSE*—Reduce the Sparse Variables:* The goal of LOWER_SPARSE is to remove parts from the sparse variables so as to reduce (if possible) the number of 1's in these variables for each cube. This procedure can be viewed as cube reduction applied to each cube, with the reduction retained only for the multiple-valued variables. However, this technique suffers from the same problem as REDUCE, namely that the order in which the cubes are processed can greatly affect the total amount of reduction possible.

Instead, the IRREDUNDANT routine is used to select, for a particular part, which cubes are redundant; this part is set to 0 for the redundant cubes. This way the cube ordering problem is avoided, and the more powerful heuristics of IRREDUNDANT are used to find a good reduction of the sparse variables.

For each value j of a sparse variable X_i, define e_j^i to be the cube of $X_i^{\{j\}}$. By finding an irredundant cover for $(F \cup D)_{e_j^i}$ we can determine which cubes of F can have part j removed. If a cube does not belong to the irredundant subcover of $(F \cup D)_{e_j^i}$, then the part in the cube is redundant and can be removed. These parts are removed, and, after all parts for a variable have been processed, the next variable is processed.

Note that by using the IRREDUNDANT algorithm rather than REDUCE, the order in which the cubes are examined in part j of variable X_i is immaterial. (Further, the order in which the parts of any variable is processed is also immaterial.) But, the order in which the sparse variables are processed does influence the reduction of variables which are not processed first. Depending on the nature of the problem being solved, an appropriate order for the variables is chosen.

2) RAISE_DENSE*—Expand the Dense Variables:* As mentioned earlier, we desire the binary-valued variables and the variables resulting from bit-pairing be dense. After reducing the multiple-valued variables with LOWER_SPARSE, the resulting set of cubes is no longer prime. Hence, we can try to expand this set of cubes by expanding only the dense parts of each cube. This is done with a modified version of EXPAND which removes all of the sparse parts from the *free* set (cf. subsection IV-D) before finding the expansion of a cube. Hence, none of the sparse parts will be expanded.

Interestingly, EXPAND will still check for cubes which, when limited to only the dense variables, can expand to cover another cube. As mentioned earlier, on subsequent iterations of MAKE_SPARSE it is possible for the cardinality of the cover to decrease. If it is possible for a cube to be covered, EXPAND will expand the dense variables so as to cover the cube.

IV. Exact Boolean Minimization

Two methods for generating all of the prime implicants of a Boolean function were presented in subsection III-C (EXPAND), and an algorithm for generating the prime implicant table needed by an exact minimization algorithm was presented in subsection III-D (IRREDUNDANT). Generating the set of all prime implicants, using IRREDUNDANT to generate the prime implicant table, and then solving the covering problem for this table provide an algorithm for determining the minimum solution for a given minimization problem.

In this section, a new technique for guiding a branch and bound solution to the covering problem is presented. This technique is *exact* in that it determines the minimum cover for a matrix. These techniques have been used to solve many large covering problems resulting from Boolean minimization problems. A new approximate algorithm of polynomial complexity (based on this technique with no backtracking), which is more practical for heuristic minimization programs, is also presented. This approximate algorithm has the advantage of providing a lower bound on the cardinality of the exact solution, and hence can sometimes determine that the solution provided is in fact optimum.

A. Minimum Cover Problem

Minimum Covering Problem: Given a binary matrix A, and a cost *cost* $(\cdot)$ for each column of the matrix, find a vector x such that $A \cdot x^t \geq (1, 1, \cdots, 1)^T$ and $\Sigma_{i=1}^{m} x_i cost(i)$ is minimum.

The constraint $A \cdot x^T \geq (1, 1, \cdots, 1)^T$ can be understood as saying that each row of the matrix must have a least one 1 in some column where x has a 1. (In this case, the row is said to be "covered" by the particular "column" of x, and the goal is to cover all rows with a vector of minimum weight.) This problem is NP-hard [19] so that any algorithm which solves the problem can be expected to have a poor worst-case complexity.

In this section, a cost function of 1 for each column of the matrix is used to simplify the explanation. The extensions of the algorithm presented here to a more general cost function is straightforward.

B. Reducing the Size of the Covering Problem

First, we review some results which are of interest in reducing the size of a covering problem.

1) **Partitioning:** If the rows and columns of matrix A can be permuted to yield a block structure of the form

$$\begin{array}{|c|c|}\hline A & 0 \\ \hline 0 & B \\ \hline\end{array}$$

where 0 represents an appropriately sized block of all zeros, then a minimum cover for A can be written as the union of a minimum cover for A, and a minimum cover for B.

2) **Essential elements:** Any row of the matrix A which has only a single 1 identifies an essential column. The solution vector x must have a 1 in the essential column in order to cover the row singleton. After placing a 1 in the essential column, any other rows which become covered can be removed from consideration.

3) **Row dominance:** If row i of A contains another row j of A (i.e., row i contains a 1 for all columns in which row j has a 1), then row i can be removed from the matrix A without changing the minimum solution. Clearly, once row j has been covered, then row i will automtically also be covered; hence row i is providing redundant information in the covering problem.

4) **Column dominance:** If column i of A contains another column j of A (i.e., column i contains a 1 for all rows in which column j contains a 1), then column j can be removed from the matrix A without changing the minimum solution. Clearly, there could be no advantage to choosing column j because choosing column i instead would cover the same set of rows, and perhaps more. Hence, column j is not needed for a minimum solution.

Therefore, the strategy to reduce the size of the matrix is as follows.

1) Look for a block partitioning.
2) Use row dominance and column dominance to reduce the number of rows and columns in the matrix. Note that it is only necessary to apply either transformation once, and the order in which they are applied is irrelevant.
3) Identify essential elements and add them to the covering set. The rows which are now covered and the essential columns are removed from the matrix.
4) Repeat steps 2–4 until no essential elements are detected in step 3.

After using steps 1–4 to reduce the size of the matrix, if a solution has not been reached, an element is selected for branching. The problem is then solved recursively assuming the branching element is in the solution, and then assuming the branching element is not in the solution.

The branch and bound algorithm for solving this problem is shown in Fig. 5. The routine is entered at the top level with the matrix (A) to be covered, a current solution (x) which is initially the empty set, a record (*best*) of the best solution known to be a cover (which is initially a full set), a lower bound (*best_possible*) on the size of the best solution (which is initially ∞), and an indication *level* of the current level in the recursion (which is initially 0). The routine returns a set of the columns of A which is a minimum cover for A.

The routines **remove_row_dominance** and **remove _column_dominance** apply row and column dominance to A to reduce its size. The routine **detect_essential** detects rows with only a single 1, and these are added to the selected set. The function **select_column** applies heuristics to select a column of A for branching. The function **reduce** removes those rows of A which are covered by q and removes the column q, and the function **remove**(A, q) deletes the column q and A.

First a check is made for a simple partition of the covering problem. If this fails, row dominance and column dominance are applied iteratively to reduce the size of the covering problem, and then the essential elements are detected and added to the selected set. Then, using a technique described in the next section, a lower bound is placed on the size of the cover for A, and the search is terminated (or bounded) if the size of the selected set exceeds the best solution possible for A. If there are no more rows in A, then we have reached a new best solution, and the solution is returned. Otherwise, a column is selected heuristically to branch on and recursively compute the solution, assuming that the element is in the covering set and then assuming that the element is not in the covering set.

```
bit_vector minimum_cover(A, x, best, level)
bit_matrix A;                       /* the matrix to be covered */
bit_vector x;                       /* the current solution */
bit_vector best;                    /* the best solution seen so far */
int best_possible;              /* the best solution possible */
int level;                          /* recursion level */
{
        if (partition(A, H_1, H_2)) {            /* check for block partition */
                x_1 ← minimum_cover(H_1, ∅, ∅, best_possible, 0);
                x_1 ← minimum_cover(H_2, ∅, ∅, best_possible, 0);
                return x_1 ∪ x_2;
        }

        do {
                /* reduce the number of rows and columns */
                A ← remove_row_dominance(A);
                A ← remove_column_dominance(A);

                /* Select essentials, and remove rows covered by an essential */
                p ← detect_essential(A);
                x ← x ∪ p;
                A ← reduce(A, p);
        } while (p ≠ ∅);

        independent_set ← maximal_independent_set(A);
        if (level == 0)
                best_possible ← |independent_set|;

        /* if current solution exceeds the best possible from here on, bound the search */
        if (|x ∪ independent_set| ≥ |best|)
                return best;

        /* if no rows left in A, then new best solution */
        else if (numrows(A) == 0)
                return x;

        /* Else branch on some column */
        else {
                q ← select_column(A, independent_set);
                /* recur assuming q belongs to the minimum cover */
                left ← minimum_cover(reduce(A, q), x ∪ q, best, best_possible, level + 1);
                if (|left| < |best|)
                        best = left;
                if (|best_possible| = |best|)
                        return best;

                /* recur assuming q does not belong to the minimum cover */
                right ← minimum_cover(remove(A, q), x, best, best_possible, level + 1);
                if (|right| < |best|)
                        best = right;

                return best;
```

Fig. 5. Covering algorithm pseudocode.

C. Use of the Maximum Independent Set

The most important feature of the above algorithm is in the routine **maximal_independent_set**. This routine finds a maximal set of rows of A all of which are pairwise disjoint (i.e., they do not have 1's in the same column). It should be clear that the number of rows in this independent set is a lower bound on the solution to the covering problem, because a different element must be selected from each of the independent rows in order to cover these rows. Hence, this lower bound can be used to terminate the search if the size of the current solution plus the size of the independent set is greater or equal to the best solution seen so far. Also, the size of the independent set at the first level of the recursion is a lower bound for the final minimum cover. Hence, by recording this value, the search can be terminated if a solution is found which meets this lower bound.

The major drawback of this technique, of course, is that the problem of finding a maximum independent set of rows is itself an NP-hard problem. But this is of no concern. The problem of finding a maximal independent set of rows can be solved heuristically while still providing a correct lower bound on the size of the final solution. (In general, finding the maximum independent set provides the best bound; other minimal solutions provide less precise but, nonetheless, accurate lower bounds.) Hence, even though this problem is itself difficult, a good heuristic algorithm is sufficient for finding a maximal independent set of rows.

To find a large independent set of rows, a graph is constructed where the nodes correspond to rows in the matrix, and an edge is placed between two nodes if the two rows are disjoint. The problem is now equivalent to finding a maximal clique (a maximal, completely connected subgraph) of this graph. To solve this problem, a greedy algorithm is used.

1) Initialize the clique to be empty (contains no nodes).
2) Pick the node of largest degree (and not already in the current clique) and add this node to the clique. Break ties by choosing the node which is connected to the most other nodes of maximum degree.
3) Remove all nodes and their edges from the graph which are not connected to the current clique.
4) Repeat steps 1 and 2 while there are still nodes in the graph not in the current clique.

The node of largest degree in step 2 corresponds to the row which is disjoint with the maximum number of other rows of the matrix. The tiebreaker attempts to preserve as many of the remaining nodes of maximum degree as possible.

Thus, the bounding in the branch and bound algorithm is modified by bounding the search if $|$**maximal_independent_set**$(A) \cup x|$ equals or exceeds the best known solution (rather than waiting until $|x|$ equals or exceeds the best known solution). The goal is to terminate unprofitable searches as early as possible.

There is the further difficulty that the bound provided by the maximum independent set may not be sharp. For example, consider the matrix

$$\begin{array}{ccc} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{array}$$

A maximum independent set of rows for this matrix contains only a single row, but a minimum cover requires at least two columns. The size of the maximum independent set remains lower bound on the size of a minimum cover; the search may just not be terminated as early as possible.

D. Choice of Branching Column

A unique element from each set of the independent set of rows must be in the minimum solution. Once a maximal independent set of rows has been computed, the selection of a branching element is limited to some element which belongs to one of these rows. Each element of each row is given a weight as the reciprocal of the row sum. Then the weights are summed for each column, and the column of maximum weight which is also in the independent set of rows is chosen for the branching variable. This weighting strategy gives the elements of the smaller sets a higher weight. For example, in a set with 2 elements, each element receives a weight of 0.5, whereas in a set with 10 elements, each element receives a weight of 0.1. The larger sets are thought of as "easier" to cover, and the smaller sets are "harder" to cover. The heuristic is to try to force a selection from one of the smaller sets. Another reason for favoring choosing an element from a smaller set (for example, a set with two elements) is to create more essential elements at the next step of the recursion.

E. Heuristic Covering Algorithm

The heuristic covering algorithm used in Espresso-MV is based on the above algorithm for the minimum covering problem. In order to make the running time more predictable, the algorithm is converted into a greedy algorithm in which the first leaf visited is taken as the solution and no backtracking is performed. Note that this greedy algorithm has the property that it can compute a lower bound on the size of a minimum cover (even though it is not guaranteed to generate a minimum cover). (Recall that the size of the maximal independent set of rows at the first level of the recursion is a lower bound for the minimum solution to the covering problem.) Hence, sometimes this greedy algorithm is able to demonstrate that it has achieved a minimum solution.

V. Experimental Results

In this section we report results from an implementation of the Espresso-MV algorithms. At Berkeley we have collected a large set of PLA's, and these have been used to determine the effectiveness of various minimization techniques. In particular, we present a comparison with the McBoole [24] exact minimization algorithm for the standard multiple-output minimization problem. We then present the results of the heuristic algorithms Espresso-MV and compare with the exact solutions when they are known. Some examples are presented to show that Espresso-MV is more efficient that Espresso-IIC, even at solving binary-valued multiple-output minimization problems. Also, as expected, Espresso-MV is much more efficient at solving multiple-valued minimization problems than a binary-valued minimizer with a don't-care set [3, ch. 5]. Finally, we will describe some design examples where multiple-valued minimization was used to optimize PLA from actual chip designs.

A. Espresso-MV

The program Espresso-MV implements the heuristic and exact logic minimization algorithms described earlier, as well as heuristic and exhaustive algorithms for the *output phase assignment* and the *input variable assignment* problems. The program can also be used for manipulating multiple-valued logic functions. Espresso-MV is written in the C language. The exact minimization algorithm contained in Espresso-MV will be referred to as Espresso-EXACT.

B. The PLA Test Set

When research leading to the Espresso-II algorithms began, PLA examples were collected as a vehicle for comparing different minimization algorithms. By the time book *Logic Minimization Algorithms for VLSI Synthesis* was written, 56 PLA examples had been collected. Further contributions to the test set from industry and universities have expanded it to 134 functions. Of these, 111 are designated as industrial examples (implying that their origin is either an industrial or university chip design), and 23 are mathematical functions such as multiply and square root.

1) Grading the Test Set by Problem Difficulty: With a test set so large, it is a challenge to present the results from competing algorithms in a meaningful manner. It can be misleading to merely report the total number of cubes and the total number of literals for each algorithm and then attempt to draw conclusions from these totals. Hence, the first goal is to determine the difficulty of the minimization problem for each PLA in the test set.

For each problem in the test set, the problem is classified as shown in Table II.

TABLE II
PLA CLASSIFICATION BY DEGREE OF DIFFICULTY

Classification	Description
trivial	minimum solution consists of essential prime implicants
noncyclic	the covering problem contains no cyclic constraints
cyclic and solved	the covering problem contains cyclic constraints and the minimum solution is known
cyclic and unsolved	the covering problem contains cyclic constraints but the minimum solution is unknown
too many primes	there were too many primes to be enumerated

TABLE III
COMPARISON OF ESPRESSO-EXACT AND MCBOOLE

		Espresso-EXACT			McBoole		
Type	Total	Number Primes	Number Solved	Time (s)	Number Primes	Number Solved	Time (s)
trivial	9	9	9	120	9	9	271
noncyclic	56	55	54	26524	56	56	35956
cyclic and solved	42	42	41	41330	42	21	11241
cyclic and unsolved	10	7	0		10	0	
too many primes	17	0	0		0	0	
Totals	134	113	104	67974	117	86	47468

The classifications were determined by allowing Espresso-EXACT and the exact minimization algorithm of McBoole to run for 5 hours for each example on an Apollo DN660.[1] (If a program had not terminated after 5 hours, it was aborted.) By examining the results for each program, a classification is determined for each example. If the problem was solved by either of the two exact minimization algorithms, it is easy to decide whether it belongs to the class **trivial, noncyclic,** or **cyclic and solved.** An example is classified as **too many primes** only if neither program was able to enumerate the complete set of prime implicants, and an example is classified as **cyclic and unsolved** only if neither program was able to complete the covering program after having generated the set of all prime implicants.

2) Comparison of Espresso-EXACT and McBoole: Both programs first generate the set of all prime implicants, and then attempt to find a minimum subset of the set of all prime implicants. Further, both programs attempt to solve only the simpler covering problem, namely, to return the cover with the fewest number of cubes without consideration for the number of literals. (Both programs use a "cleanup" step where the number of literals is reduced once the minimum number of rows has been achieved, but both programs solve this problem heuristically.)

[1]Tests show that the Apollo DN660 with version 3.12 of the C compiler executes Espresso-MV at the same speed as a DEC VAX 11/785 with the 4.3BSD portable C compiler. All results in this section were timed on an Apollo DN660 with 4 megabytes of memory.

Table III summarizes the comparison between Espresso-EXACT and McBoole for the 134 PLA's in the test set. *Number primes* is the number of examples for which each program was able to generate all of the primes for, *number solved* is the number of the examples for which each program was able to solve, and *time* gives the total time on an Apollo DN660 (in seconds) taken for those examples which could be solved within the 5-hour time limit. Thus, for example, Espresso-MV took more than 30 000 seconds longer than McBoole for the category **cyclic and solved,** but this involved solving 20 more problems that McBoole.

For examples with no cyclic constraints, both Espresso-EXACT and McBoole are usually able to find the minimum solution. Espresso-EXACT failed to generate the minimum solution for two examples (*al2* and *prom1*). For *prom1*, it was unable to enumerate all of the primes (which has 9179 primes). For *al2*, it was able to generate all of the primes (there were 9326 primes), but was unable to generate the prime implicant table.

However, when there are cyclic constraints, the covering algorithm of Espresso-EXACT is able to find the minimum solution for many more of the PLA's than McBoole. Only for example *intb* did Espresso-EXACT fail to solve an example with cyclic constraints that McBoole was able to solve. (Espresso-EXACT was un-

TABLE IV
ESPRESSO-MV RESULTS

		Espresso-MV			Espresso-MV (SUPER_GASP)		
		Solution		Time	Solution		Time
Type	#	Cubes	Lits	(s)	Cubes	Lits	(s)
trivial	9	234	1683	23	243	1683	23
noncyclic	56	3909	45712	1674	3899	45956	2372
cyclic-s	42	4092	42030	3202	4056	42577	5403
cyclic-us	10	2023	25347	3444	2010	25438	4637
too-many-primes	16	2759	35718	6751	2755	35881	7924
Totals	133	13026	150490	15094	12963	151535	20359

TABLE V
ESPRESSO-EXACT VERSUS HEURISTIC MODE

		Espresso-MV			Espresso-MV (SUPER_GASP)			Espresso-EXACT		
		Solution		Time	Solution		Time	Solution		Time
Type	#	Cubes	Lits	(s)	Cubes	Lits	(s)	Cubes	Lits	(s)
trivial	9	243	1683	23	243	1683	23	243	1683	120
noncyclic	54	3371	34060	1366	3361	34223	2030	3360	34204	26523
cyclic-s	41	3463	36163	2532	3427	36658	4279	3395	36564	41329
Totals	104	7077	71960	3920	7031	72564	6332	6998	72451	67973

ble to generate the prime implicant table for *intb* which has 6522 prime implicants.) Sometimes the results are quite dramatic. The example *sqr6* was allowed to run for 58 hours with McBoole without terminating with the minimum solution; however, Espresso-EXACT is able to complete this same example in only 100 seconds. Also, Espresso-EXACT was able to determine the minimum cover for the example *mlp4* (a 4-bit multiplier) in about 1 hour. Results have been published for both of these examples without presenting the minimum solution [24], [25]. As far as we know, no previous program has successfully completed the minimization of these two examples.

Comparing the efficiency of the prime generation algorithms, we find that in 113 cases both programs could generate all of the prime implicants, in 4 cases (*b4* with 6455 primes, *bc0* with 6595 primes, *prom1* with 9326 primes, and *t1* with 15 135 primes) McBoole was able to generate all of the prime implicants when Espresso-EXACT could not, and in 17 cases neither program was able to generate all of the prime implicants.

Overall, there were 83 examples which both programs could minimize, 3 examples which McBoole could minimize which Espresso-EXACT could not, 21 examples which Espresso-EXACT could minimize which McBoole could not, and 27 examples for which neither program was able to complete the exact minimization (20 percent). For the 83 examples which both programs could minimize, Espresso-EXACT used 38 198 seconds, and McBoole used 28 628 seconds. The Espresso-EXACT result had 51 821 literals, and McBoole had 53 686 literals, indicating that MAKE_SPARSE was more efficient at reducing the number of literals (once the minimum number of terms was determined). Of course, for these 83 examples, both returned the same number of prime implicants, essential prime implicants, and solution cubes.

Including the time each program used on those examples for which a solution was not found, Espresso-EXACT used 6.1 days of computer time and McBoole used 10.3 days of computer time.

3) Espresso-MV Results: We are thus in an excellent position to grade the quality of the results for the heuristic minimization algorithm Espresso-MV as we know the minimum solution for 107 of the 134 examples in the test set.

Table IV shows the totals for 133 examples, broken down by category, for Espresso-MV and Espresso-MV (SUPER_GASP mode). The examples were run on an Apollo DN660. It is evident that the SUPER_GASP option can be expensive; sometimes, however, the extra reduction in the number of terms might be considered worthwhile. Curiously, SUPER_GASP produces more literals in all categories while providing solutions with fewer cubes.

Next we compare the results from Espresso-MV (again, with and without SUPER_GASP), but consider only those examples for which Espresso-EXACT was able to generate the minimum solution. This will allow the comparison of Espresso-MV in its exact and heuristic modes. The results are shown in Table V. It is evident that Espresso-MV provides a high-quality result for all of the examples for which the minimum solution can be generated—the difference between Espresso-MV and Espresso-EXACT

TABLE VI
USING A BINARY-VALUED MINIMIZER FOR MULTIPLE-VALUED FUNCTIONS

Example	States	Binary-Valued Terms	Binary-Valued Time*	Multiple-Valued Terms	Multiple-Valued Time*
DK14	7	26	4.3	26	0.5
DK16	8	55	108.6	55	1.6
PCC	12	-	(3600)	48	4.4
BLUE	93	-	(3600)	775	1053.0

*Time in seconds measured on an IBM 3081 using the Waterloo C Compiler, Version 1.1 under the VM/CMS Operating System.

TABLE VII
ESPRESSO-MV VERSUS ESPRESSO-IIC

Program	Cubes	Literals	Time
Espresso-MV	5993	60322	560
Espresso-IIC	6001	60578	992

*Time in seconds measured on an IBM 3081 using the Waterloo C Compiler, Version 1.1 under the VM/CMS Operating System.

is about 1 percent. Also, Espresso-MV is more than 15 times faster than the exact minimizer on problems that both algorithms can solve.

C. *Multiple-Valued Minimization Results*

1) Multiple-Valued Minimizer versus Binary-Valued with a DC-set: As mentioned in Section I, it is possible to use a binary-valued minimizer to minimize a multiple-valued function. This technique is described in detail in [3, ch. 5].

In this section, we compare the efficiency of this technique for a small set of examples. Table VI compares Espresso-MV running as a multiple-valued minimizer versus translating the problem into an equivalent binary-valued minimization problem (using Espresso-MV as the binary-valued minimizer). The time reported for these examples was measured on an IBM 3081. The examples DK14, DK16, PCC, and BLUE represent problems that are being solved by the state-assignment programs KISS [12]. They have 7, 8, 12, and 93 states, respectively.

Solving a multiple-valued minimization problem using a binary-valued minimization tool can be inefficient. In the two largest cases, the binary-valued minimizer was unable to complete the solution after 1 hour on an IBM 3081. The computation did not terminate for either PCC or BLUE within the 1-hour time limit.

2) Multiple-Output Espresso-IIC versus Espresso-MV: Table VII compares the performance of Espresso-MV against the binary-valued minimizer Espresso-IIC for the 56 examples published in [3].

Comparing Espresso-IIC and Espresso-MV, the quality of the results is almost identical, but the run time has been reduced by almost 50 percent. This is a surprising result, as one might expect the generalization of the algorithms of multiple-valued variables to penalize the performance for binary-valued minimization problems. However, the algorithms are improved by the more uniform treatment of the output part during the multiple-valued minimization. For example, as described in subsection III-C, the OFF-set does not need to be represented with only a single output active in each cube. This leads to a more compact representation of the OFF-set, and to a more efficient EXPAND procedure. Likewise, Espresso-IIC effectively would not split against the output part until reaching a leaf of one of the recursive procedures (e.g., TAUTOLOGY). By allowing the program to split against the output at any step of the procedure, the heuristics of choosing the splitting against the output at any step of the procedure, the heuristics of choosing the splitting variable leads to a more efficient choice of splitting variables.

D. *Multiple-Valued Minimization for Chip Design*

The final results we report come from a chip design currently being performed at the University of California, Berkeley. The project, known as SPUR [26], involves the design of a RISC-style microprocessor and a snooping, multiprocessor cache controller. We present results from performing an optimal assignment of opcodes to the instructions for the instruction decode PLA, and for two of the largest PLA's from the cache controller chip.

The procedure for solving these *input encoding problems* is described in detail in [10]. The following is an overview of the procedure.

1) Create a representation of the function where the possible decodes for a set of binary-valued variables are viewed as a single multiple-valued variable.
2) Minimize the multiple-valued function.
3) Determine an encoding which satisfies all of the *constraints*.

Each multiple-valued product term prescribes a *constraint* to be solved. This constraint specifies that the codes for a set of the values are to lie on a face of the Boolean hypercube. The number of product terms which result in step 2 are a lower bound on the number of product terms needed for any possible encoding of the values. If an encoding can be found which satisfies all of the constraints, then the final PLA will require as many product terms as are in the multiple-valued cover. However, in practice, not all of the constraints can be easily satisfied, so that the final PLA is sometimes larger than this lower bound.

For these examples, the encoding problem was solved manually. However, work is continuing at Berkeley on algorithms for solving this problem efficiently.

1) Optimal Assignment of Instruction Codes: The instruction decoder for the SPUR microprocessor chip is implemented as a PLA with 8 inputs (7 bits of opcode) and 39 outputs. The PLA, after minimization, has 71 product terms. It should be noted that an attempt was made by the designers to choose a good assignment of opcodes to instructions. We mapped the problem into a multiple-valued minimization problem where the 7-bit opcode was replaced with a single multiple-valued variable assuming

128 different values. This was then minimized using the mutliple-valued minimizer and resulted in 37 multiple-valued product terms. An assignment was then made to the opcodes based on the constraints produced by the multiple-valued minimization and the resulting PLA was implemented using only 45 product terms.

2) State Assignment: The controller chip contains a large finite-state machine (known as *Sequencer*) consisting of 74 states, 41 inputs, and 35 outputs. It is intended that a PLA will be used to implement this finite-state machine. Another large combinational block of logic (known as *Decoder*) decodes the current state into a set of control signals for the data paths on the chip and on the microprocessor system board. The finite-state machines are described in the high-level language Endot, and are translated into a symbolic representation of the finite-state machine using the program BDSYN. The input encoding problem is then solved simultaneously for the Sequencer and the Decoder to choose an assignment of binary codes to the states which optimizes the amount of logic needed to map the current state into the outputs of the decoder and the sequencer. (This approximation of state assignment treats each possible next state as a separate problem.)

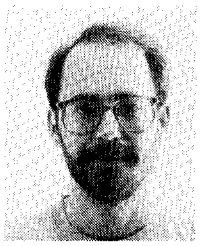
The original description of the Sequencer consists of 2000 transitions in a state-diagram representation of the machine (including *don't-care* conditions). Minimization with an arbitrary encoding of the states results in a PLA with 326 product terms (using the don't-care set provided by the designer). A multiple-valued minimization results in a PLA with 152 multiple-valued product terms. A similar result for the Decoder shows a reduction from 121 product terms (after minimization with the arbitrary assignment) to 45 multiple-valued product terms as a multiple-valued cover. The constraints resulting from both of these minimizations need to be traded off against each other to choose the final encoding for the states. The final size for the sequencer is 180 product terms, and 68 product terms for the decoder.

Hence, in this design example, significant reduction in the area required to implement these functions was achieved using multiple-valued minimization.

References

[1] H. Fleisher and L. I. Maissel, "An Introduction to array logic," *IBM J. Res. Develop.*, vol. 19, pp. 98–109, Mar. 1975.
[2] J. C. Logue, N. F. Brickman, F. Howley, J. W. Jones, and W. W. Wu, "Hardware implementation of a small system in programmable logic arrays," *IBM J. Res. Develop.*, vol. 19, pp. 110–119, Mar. 1975.
[3] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis.* Norwell, MA: Kluwer Academic Publishers, 1984.
[4] T. Sasao, "Input variable assignment and output phase optimization of PLA's," *IEEE Trans. Comput. C-33*, pp. 879–894, Oct. 1984.
[5] G. D. Hachtel, A. R. Newton, and A. Sangiovanni-Vincentelli, "An algorithm for optimal PLA folding," *IEEE Trans. Computer-Aided Design*, vol. CAD-1, pp. 63–76, Jan. 1982.
[6] G. De Micheli and A. Sangiovanni-Vincentelli, "Multiple constrained folding of programmable logic arrays: Theory and applications," *IEEE Trans. Computer-Aided Design*, vol. CAD-2, pp. 151–167, July 1983.
[7] G. H. Mah, "PANDA: A PLA generator for multiple folded PLAs," Tech. Rep. UCB M84/95, University of California Electronics Research Laboratory, May 1984.
[8] W. V. Quine, "A way to simplify truth functions," *Amer. Math. Mon.*, vol. 62, p. 627, Nov. 1955.
[9] E. J. McCluskey, "Minimization of Boolean functions," *Bell Syst. Tech. J.*, vol. 35, pp. 1417–1444, Nov. 1956.
[10] R. L. Rudell, "Multiple-valued logic minimization for PLA synthesis," Master's report, University of California, Berkeley, 1983.
[11] G. De Micheli, "Computer aided synthesis of PLA-based systems," Ph.D. thesis, University of California, Berkeley, 1983.
[12] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Optimal state assignment for finite-state machines," *IEEE Trans. Computer-Aided Design*, vol. CAD-4, pp. 269–285, July 1985.
[13] G. De Micheli, "Symbolic minimization of logic functions," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 1985, pp. 293–295.
[14] T. Sasao, "An application of multiple-valued logic to a design of programmable logic arrays," in *Proc. 18th Int. Symp Mult. Val. Logic*, 1978.
[15] T. Sasao, "An algorithm to derive the complement of a binary function with multiple-valued inputs," *IEEE Trans. Comput.*, vol. C-34, Feb. 1985.
[16] T. Sasao, "Multiple-valued decomposition of generalized Boolean functions and the complexity of programmable logic arrays," *IEEE Trans. Comput.*, vol. C-30, pp. 635–643, Sept. 1981.
[17] Y. H. Su and P. T. Cheung, "Computer minimization of multi-valued switching functions," *IEEE Trans. Comput.*, vol. C-21, pp. 995–1003, 1972.
[18] T. Sasao, "Tautology checking algorithms for multiple-valued input binary functions and their application," in *Proc. 14th Int. Symp. Mult. Val. Logic*, 1984.
[19] M. R. Garey and D. S. Johnson, *Computers and Intractability.* San Francisco: W. H. Freeman, 1979.
[20] S. J. Hong, R. G. Cain, and D. L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBMJ J. Res. Develop.*, 443–458, Sept. 1974.
[21] J. P. Roth, *Computer Logic, Testing, and Validation.* Rockville, MD: Computer Science Press, 1980.
[22] T. Sasao, "Corrections and addition to input variable assignment and output phase optimization of PLA's," private communication.
[23] P. Simanyi, "POP reference manual," in *Berkeley CAD Tools Manual.* Univ. of California, Berkeley, Sept. 1983.
[24] M. R. Dagenais, V. K. Agarwal, and N. C. Rumin, "McBoole: A new procedure for exact logic minimization," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 229–238, Jan. 1986.
[25] T. Sasao, "Comparison of minimization algorithms for multiple-valued expressions," *Draft*, 1982.
[26] R. Katz, *et al.*, "Design decisions in SPUR," *COMPUTER*, pp. 8–22, Nov. 1986.

*



Richard L. Rudell received the B.S. degree in electrical engineering from the University of Minnesota in 1983 and the M.S. degree in electrical engineering from the University of California in 1986. He is currently a Ph.D. candidate in electrical engineering at the University of California, Berkeley.

From 1980 to 1983 he worked part-time at the Honeywell Corporate Computer Science Center in Minneapolis in the area of computer-aided design. This work was in the areas of the test pattern generation, high-level synthesis tools, and floor planning algorithms for VLSI. More recently, he has spent the summers of 1984 and 1985 working at the IBM T. J. Watson Research Center in the area of multiple-level logic synthesis. His current interests are in the area of multiple-level logic optimization, including design specification, factoring of Boolean equations, multiple-level minimization, and optimal technology mapping.

*

Alberto Sangiovanni-Vincentelli (M'74–SM'81–F'83), for a photograph and biography, please see page 693 of this issue.