Boolean Factorization Using Multiple-Valued Minimization

Stan Liao, Srinivas Devadas
Department of EECS
MIT, Cambridge, MA

Abhijit Ghosh
Mitsubishi Electric Research Laboratories
Sunnyvale, CA

Abstract

We show that the problem of factoring a sum-of-products representation of a logic function can be transformed into one of multiple-valued prime generation followed by branch-and-bound covering.

We give a factorization method that generates potential Boolean factors by generating the primes of a multiple-valued function with an associated don't-care set. A covering problem is solved wherein a set of primes with minimal cost is selected to obtain a Boolean factorization. This method can exploit Boolean identities in factorization such as $a \cdot \overline{a} = 0$ and $a \cdot a = a$. Common factors across a set of Boolean functions can be identified by using multiple-output prime generation and covering.

We show how all the kernels of an expression can be generated by generating the primes of a multiple-valued function. A covering step can be used to arrive at an algebraic factorization.

1 Introduction

Multilevel logic synthesis has been an active research area in the past two decades [4]. Several programs exist today (*e.g.* [2], [6]) that are capable of optimizing large multilevel circuits for area, delay or testability.

The optimization methods used most often in existing optimization systems are algebraic in nature. Algebraic methods are used to factor a single logic expression to reduce the literal count of the expression and to identify common factors among a set of expressions. Algebraic methods are *weak* in that they cannot identify Boolean factors. However, they are CPU time efficient and can be repeatedly used on large circuits.

To augment the capabilities of algebraic methods, don't-care-based optimization methods have been proposed. Observability and satisfiability don't-care sets can be defined for each node in a Boolean network [1]. Two-level logic minimization of the function at any node with an appropriate don't-care set can result in additional simplifications of the network that are not possible using purely algebraic techniques [6].

Boolean resubstitution methods can also be used to augment algebraic optimization. In Boolean resubstitution, also called strong division, a node g is resubstituted into a node f by (1) adding an extra input to f corresponding to g, call it g' (2) defining a don't-care set corresponding to $g \cdot \overline{g'} + \overline{g} \cdot g'$ and (3) minimizing f' which is f with the extra input added. If f' has a smaller cost than f, the resubstitution is performed, else it is discarded. While Boolean resubstitution is a well-defined operation, finding appropriate g functions that will result in a minimized f' with a significantly smaller cost than f is a much harder problem.

An effort to identify good Boolean factors of a logic expression using multiple-valued minimization was presented in [5]. As we will show in Section 3, this method is incapable of finding certain useful Boolean factors of a logic expression because it does not exploit Boolean identities such as $a \cdot \overline{a} = 0$.

Our method of identifying algebraic and Boolean factors in a logic function involves multiple-valued prime generation and covering. The method is capable of exploiting Boolean identities such as $a \cdot \overline{a} = 0$ and $a \cdot a = a$. Given a sum-of-products representation of a logic function, a subset of inputs to the logic function are duplicated to create a new Boolean function. A don't-care set corresponding to each duplicated pair of inputs being unequal can be defined for this function. The duplicated inputs are then represented as a multiple-valued variable, and the Boolean function is converted into a multiple-valued input, binary-valued output function.

We show that the prime implicants of the multiple-valued function correspond to Boolean factors of the original function. We can either select a prime implicant of the multiple-valued function and perform strong division to obtain a Boolean factorization, or solve a covering problem to directly arrive at a Boolean factorization. We define the cost function in the covering step to correspond to the literal count in the factorization.

We construct a multiple-valued function corresponding to a given function such that the primes of the multiple-valued function correspond to algebraic kernels of the given function. A covering step can be used to algebraically factor the function given its kernels.

2 Preliminaries

For basic terminology pertaining to Boolean functions, the reader is referred to [3]. We will give the terminology for multiple-valued functions here.

A **multiple-valued variable** m^p can take on values $0, 1, \ldots, p-1$. In particular, a binary-valued variable m^2 can take on values of 0 or 1. If the superscript, p, is omitted, it is assumed to be 2.

Let p_i for $i = 1, \ldots, n$ be positive integers representing the number of values for each of n variables. Define the sets $P_i \equiv \{0, \ldots, p_i - 1\}$ for $i = 1, \ldots, n$ which represent the p_i values that variables m_i may assume, and define $B_j \equiv \{0, 1, *\}$ which represents the value of the function. A **multiple-valued input, binary-valued output function** with k outputs, $P_{n \times k}$, (hereafter known as a **multiple-valued function**) is a map-

ping

$$P_{n\times k} : P_1 \times P_2 \ \cdots \ \times P_n \ \longrightarrow B_1 \times \ \cdots \ \times B_k$$

The function is said to have n multiple-valued inputs, $m_i^{p_i}$, k binary-valued outputs, and variable i takes on p_i possible values.

The **cover** of a multiple-valued function, $P_{n\times k}$ is a collection of **cubes** or **implicants** $\{c_i, 1 \leq i \leq |P|\}$ where

$$c_i = (c_{i1}, c_{i2}, \ \ldots, \ c_{in}, b_{i1}, \ \ldots, \ b_{ik})$$

$|P|$ is the number of cubes in the cover. The first n entries represent the **input part** of the cube, and the next k entries represent the **output part** of the cube. (In the sequel, when no confusion can arise, we will only list the input part of a cube). $c_{ij} \subseteq P_j$ is a collection of values of $m_j^{p_j}$ that appear in cube c_i, and is called a literal of c_i.

We will use a bit-vector representation of length p_j for $c_{ij} = [\ c_{ij}(0), \ \ldots, \ c_{ij}(p_j - 1)\ ]$, where

$$c_{ij}(k) = \ 1 \ if \ k \in c_{ij} \ else \ 0$$

Notions such as **prime implicant**, **essential prime implicant** *etc.* can be defined for multiple-valued functions in a manner similar to Boolean functions.

The inputs to a binary-valued logic function will be denoted x_i. The **support** of a logic function g, denoted $sup(g)$ is the set of input variables of the logic function. A logic function g is an **algebraic divisor** or **weak divisor** of f if $f = g \cdot h + r$, where h and r are logic functions, $sup(g) \cap sup(h) = \phi$ and $g \cdot h \neq 0$. A logic function g is a **Boolean divisor** or **strong divisor** of f if $f = g \cdot h + r$, where h and r are logic functions and $g \cdot h \neq 0$.

The set of algebraic divisors is defined as $D(f) = \{g | f/g \neq \phi\}$. The **primary divisors** of f are defined as $P(f) = \{f/c | c \ is \ a \ cube\}$.

A function g is termed **cube-free** if the only cube that divides g evenly is 1. The **kernels** of f are defined as $K(f) = \{k | k \in P(f), \ k \ is \ cube \ free\}$. The cube c that was used to generate the kernel k is called the **co-kernel** of k.

3 Illustrative Examples

In this section we illustrate our method for obtaining Boolean factors with a simple example. Consider the sum-of-products expression of a function f shown to the left of Figure 1. Denote the inputs of f as a, b and c. The function has three implicants. [1]

We duplicate each input to obtain a six-input function as shown to the right of Figure 1. The new inputs are A, B and C.

We can add a don't-care set to this function corresponding to combinations that cannot occur, *e.g.* $a \cdot \overline{A}$, $\overline{a} \cdot A$ etc. This is shown to the left of Figure 2.

[1] Recall that an implicant of a function should be contained in the union of the ON-set and the DC-set of the function. A prime implicant is an implicant that is not contained by another implicant.

```
10-  1          10-10-  1
0-1  1          0-10-1  1
-10  1          -10-10  1
```

Figure 1: Function and Duplicated-Input Function

```
10-10-  1        10- 00001100  1
0-10-1  1        -10 00100010  1
-10-10  1        0-1 01010000  1
1--0--  -        1-- 11110000  -
0--1--  -        0-- 00001111  -
-1--0-  -        -1- 11001100  -
-0--1-  -        -0- 00110011  -
--1--0  -        --1 10101010  -
--0--1  -        --0 01010101  -
```

Figure 2: Function with Don't-Care Set and Multiple-Valued Function

Next, we represent the three new inputs A, B and C by a single 8-valued variable, where the first value of the variable represents the combination 000 over A, B and C, the second value the combination 001 and so on. This is illustrated to the right of Figure 2. For instance the implicant `10- 00001100` corresponds to $a \cdot \overline{b} \cdot \{A \cdot \overline{B} \cdot \overline{C}, \ A \cdot \overline{B} \cdot C\}$ and can be represented as the binary-valued function $a \cdot \overline{b} \cdot (A \cdot \overline{B})$.

We generate all the prime implicants of the multiple-valued function of Figure 2. The total set of primes is shown in Figure 3.

The first six primes are uninteresting because they simply correspond to the primes of the original function (with all possible values of the multiple-valued variable). For instance, the first prime is $\overline{a} \cdot b$. The prime `--- 01111110` is also uninteresting because it represents the entire ON-set of f over A, B and C.

The first prime in the 3-tuple of primes with the multiple-valued part `01111111` corresponds to $\overline{a} \cdot (\overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C} + A \cdot B \cdot C)$. This can be written as $\overline{a} \cdot (A + B + C)$, where $(A + B + C)$ is the minimized binary-valued representation of the `01111111` multiple-

```
01- 11111111  1
10- 11111111  1
0-1 11111111  1
-01 11111111  1
1-0 11111111  1
-10 11111111  1
--- 01111110  1
0-- 01111111  1
-0- 01111111  1
--0 01111111  1
1-- 11111110  1
-1- 11111110  1
--1 11111110  1
```

Figure 3: Primes of the Multiple-Valued Function

```
0-- 01111111  1     0-- (1-- + -1- + --1)  1
-0- 01111111  1     -0- (1-- + -1- + --1)  1
--0 01111111  1     --0 (1-- + -1- + --1)  1
```

Figure 4: A Boolean Factorization

valued part. Similarly for the other primes. The set of primes shown at the left in Figure 4 corresponds to a cover for the multiple-valued function of Figure 2, and is an implementation of the binary-valued function of Figure 2 as $(\overline{a}+\overline{b}+\overline{c})\cdot(A+B+C)$ (shown to the right in Figure 4). The chosen cover for the multiple-valued function corresponds to a Boolean factorization of the original function of Figure 1. Since $A = a$ and $B = b$ and $C = c$, we can write $f = (\overline{a}+\overline{b}+\overline{c})\cdot(a+b+c)$.

The Boolean factorization of Figure 4 cannot be discovered by the method of [5] because inputs are not duplicated and the don't-care set corresponding to conditions like $a \cdot \overline{A}$ are not used in that method.

We will now give another example, this time of an algebraic factorization of a function. Consider

$$f \;=\; a\cdot c + a\cdot d + b\cdot c + b\cdot d$$

Clearly the optimum factorization corresponds to:

$$f \;=\; (a+b)\cdot(c+d)$$

Assume that we duplicate all the inputs and represent the duplicated inputs as a 16-valued variable. The first position of the 16-valued variable corresponds to $\overline{A}\cdot\overline{B}\cdot\overline{C}\cdot\overline{D}$, the fifth position to $\overline{A}\cdot B\cdot\overline{C}\cdot\overline{D}$ and the last position to $A\cdot B\cdot C\cdot D$. A multiple-valued cover with 2 product terms for the function with duplicated inputs is given below.

```
1--- 1111111101110111  1
-1-- 0111111111110111  1
```

The factorization corresponds to:

$$f \;=\; a\cdot(C+D+\overline{A}) + b\cdot(C+D+\overline{B})$$

This factorization has a literal count of 8. Due to the don't-care set that has been added, the primes in the above factorization have unnecessary $a\cdot\overline{A}$ and $b\cdot\overline{B}$ terms. However, if we reduce the multiple-valued parts of the chosen implicants, removing redundant 1's from the multiple-valued parts, we can obtain the cover shown below which corresponds to the optimum factorization with cost 4.

```
1--- 0111011101110111  1
-1-- 0111011101110111  1
```

The above example motivates the selection of non-prime implicants in a cover during the search for an optimal factorization. Exploiting the $a\cdot a = a$ identity also requires the selection of non-prime implicants.

4 A Boolean Factorization Method

4.1 Overall Strategy

Given a Boolean function f, we will generate a multiple-valued function f_m as described below.

1. Given a single-output logic function $f(x_1, \cdots, x_n)$ and a sum-of-products representation (or cover) for f, namely F, we create a cover F' for $f'(x_1, \cdots, x_n, x_{n+1}, \cdots, x_{2n})$ by taking each cube[2] $c_i \in F$ where

$$c_i \;=\; (c_{i1}, \cdots, c_{in})$$

and replacing it with a cube over $2n$ inputs namely

$$c'_i \;=\; (c_{i1}, \cdots, c_{in}, c'_{i(n+1)}, \cdots, c'_{i(2n)})$$

where $c'_{i(n+k)} = c_{ik},\ 1 \le k \le n$.

2. A don't-care set D corresponding to

$$\sum_{i=1}^{n} x_i \cdot \overline{x_{n+i}} + \overline{x_i}\cdot x_{n+i}$$

is added to F'. This don't-care set allows the exploitation of Boolean identities such as $a\cdot\overline{a} = 0$ in the factorization.

3. f' (F') is converted to a multiple-valued function f_m (F_m). Each cube

$$c'_i \;=\; (c_{i1}, \cdots, c_{in}, c'_{i(n+1)}, \cdots, c'_{i(2n)})$$

$\in F'$ is converted to

$$c_i \;=\; (c_{i1}, \cdots, c_{in}, c_{i(n+1)})$$

where $c_{i(n+1)}$ is a literal of a 2^n-valued variable m^{2^n}. Each minterm over $x_{n+1}, \cdots, x_{2n}$ has a unique position in m^{2^n}. $c_{i(n+1)}$ will contain a 1 in all the positions where the corresponding minterm is in $(c'_{i(n+1)}, \cdots, c'_{i(2n)})$.

4. Minimize the multiple-valued logic function f_m under a particular cost function.

5. Identify all the distinct multiple-valued literals in the implicants of the cover. Let the set of distinct literals be $\{c_{d_1(n+1)}, c_{d_2(n+1)}, \cdots, c_{d_K(n+1)}\}$.

6. We represent f as a Boolean factorization given by:

$$f \;=\; \sum_{i=1}^{K} (c_{d_i(n+1)}) \cdot \Big(\sum_{\substack{over\ all\ l \\ such\ that \\ c_{l(n+1)} = c_{d_i(n+1)}}} \prod_{j=1}^{n} c_{lj} \Big) \quad (1)$$

Note that in the factorization each $c_{d_i(n+1)}$ is implemented as a sum-of-products expression.

[2] We will not specify the output part of the cube.

4.2 The Cost Function

The cost function for the covering step clearly cannot be the number of product terms in the multiple-valued cover, otherwise we would always obtain a single implicant in the cover. This implicant would correspond to the binary-valued part being the universe, and the multiple-valued part containing exactly those values that correspond to minterms in the *ON*-set of the original function. In our example of Figure 3, we would select the implicant `--- 01111110`.

Operating from first principles, it is clear that we wish to obtain a multiple-valued cover such that if we convert this cover into a Boolean factorization for f, as given by Eqn. 1, this Boolean factorization should have a minimum number of literals. In general, if we select the single implicant with all don't-cares in the binary part, we will obtain the original f in sum-of-products form and the factorization will not have resulted in any reduction in literal count.

Any given implicant has a multiple-valued part. We can compute the number of literals that this multiple-valued part corresponds to in the Boolean factorization by creating a single-output function corresponding to the disjunction of all the binary-valued combinations that are in the multiple-valued part. For example, the implicant `1-- 11111110` has the multiple-valued literal `11111110`. The binary-valued function corresponding to this multiple-valued literal is $\overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot C + \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot \overline{B} \cdot C + A \cdot B \cdot \overline{C}$. We have to minimize this function to obtain the "cost" of the multiple-valued part. The minimized function is $\overline{A} + \overline{B} + \overline{C}$ containing 3 literals. The cost of the multiple-valued literal `11111110` is therefore 3.

Given a multiple-valued cover, $F_m = \{c_i,\ 1 \leq i \leq |F_m|\}$, we calculate the cost of a cover as follows:
1. Identify all the distinct multiple-valued literals in the implicants of the cover. Let the set of distinct literals be $\{c_{d_1(n+1)},\ c_{d_2(n+1)},\ \cdots,\ c_{d_K(n+1)}\}$. Compute

$$M = \sum_{j=1}^{K} cost(c_{d_j(n+1)}) \tag{2}$$

where $cost(c_{d_j(n+1)})$ is the number of literals in the minimized binary-valued function corresponding to $c_{d_j(n+1)}$.
2. Count the number of literals in the binary-valued parts of each implicant in the multiple-valued cover. That is, compute

$$B = \sum_{i=1}^{|F_m|} \sum_{j=1}^{n} cost(c_{ij}) \tag{3}$$

where $\sum_{j=1}^{n} cost(c_{ij})$ is the number of literals in the binary-valued part of the cube c_i.
3. The cost of the cover is:

$$cost(F_m) = M + B \tag{4}$$

The number of literals in the factorization of Eqn. 1 is given by Eqn. 4.

4.3 Prime Generation and Covering

The prime generation and covering steps we use are similar to the corresponding steps in ESPRESSO-MV [8]. The modifications required for our problem are explained below.

1. Only a subset of primes of the multiple-valued function are generated. In particular, only those primes whose binary-valued parts strictly contain the prime implicants of the original binary-valued function are generated. In our example we will only generate the last seven primes of Figure 3. Primes whose binary-valued parts are the universe are discarded.

This selective generation can be done without generating the entire set of primes [8].

2. The cost of a partial selection of primes is computed using Eqn. 4. The problem with only using prime implicants, and checking for strict equality of multiple-valued literals in evaluating Eqn. 4 was illustrated with an example in Section 3.

For an implicant $c_i = (c_{i1}, \cdots, c_{in}, c_{i(n+1)})$, compute $q_i = (c_{i1}, \cdots, c_{in}, q_{i(n+1)})$ as the largest implicant (containing the maximum number of minterms) such that $q_i \subseteq D$, where D is the don't-care set. There is only one such q_i. We define a **maximally reduced implicant** ${c_i}^r = ({c_{i1}}, \cdots, {c_{in}}, {c_{i(n+1)}}^r)$ of the implicant $c_i = (c_{i1}, \cdots, c_{in}, c_{i(n+1)})$ as ${c_{i(n+1)}}^r = c_{i(n+1)} - q_{i(n+1)}$. The binary-valued literals of c_i and ${c_i}^r$ are the same, and the multiple-valued part of ${c_i}^r$ has the minimal number of 1's. However, the implicants c_i and ${c_i}^r$ cover the same *ON*-set minterms in f_m. Consider a function of variables a, b, c, *etc.* which is being factored. Consider an implicant in the multiple-valued function whose binary part is $a \cdot b$. To obtain the maximally reduced implicant, we have to multiply the multiple-valued part of the implicant with the duplicated variables corresponding to the cube in the binary part, *i.e.* $A \cdot B$. For example, the prime implicants `1--- 1111111101110111` and `-1-- 0111111111110111` in the second example of Section 3 can be maximally reduced to `1--- 0000000001110111` and `-1-- 0000011100000111` respectively.

The **range** $range(c_i)$ of a prime implicant c_i defines an interval of functions $[{c_i}^l,\ {c_i}^u]$, and is initially set to $[{c_i}^r,\ c_i]$.

During the covering step, assume that we have a selected set of prime implicants, call it S. We now select a new prime implicant c. We will recur twice, once adding c to S and the other time deleting c without adding it to S.

If any prime implicant $p \in S$ is such that $range(p) \cap range(c) \neq \phi$, during the first recursion where we add c to S, we do not count the cost of the multiple-valued part of c. We further update the ranges of both p and c to $range(p) \cap range(c)$ during this recursion. For two ranges $[c_1^l, c_1^u]$ and $[c_2^l, c_2^u]$ that intersect, their range intersection is given as $[c_1^l + c_2^l, c_1^u \cdot c_2^u]$. Note that S is unchanged for the second recursion, where c is not selected.

To find the cost of a multiple-valued literal in a

prime c_i, we place the minterms corresponding to all the positions which have a 1 in ${c_i}^l$ in the *ON*-set of a constructed function, and the minterms corresponding to the remaining positions which have a 1 in ${c_i}^u$ in the *DC*-set of the the function. The literal count of the minimum-literal implementation of the constructed function is used as the cost in Eqn. 2. Costs may have to be recomputed when the ranges are altered.

3. Given a partial selection of primes, a lower bound on the required number of primes to cover the uncovered minterms can be found using the maximal independent set heuristic [8]. We can also use the maximal independent set to obtain a lower bound on the number of literals we will add to the current cost in the process of obtaining a complete cover. This is done by finding the prime with least weight among each set of primes that covers each uncovered minterm. We compute the weight as the number of literals in the binary-valued part of the prime. We cannot count the number of literals in the multiple-valued part since we do not know if the primes that are already in the cover or the other primes with the least weight have the same multiple-valued part or not (see Eqn. 4). This bound is thus not as tight as in the case of minimizing cover cardinality.

Dominated primes can only be deleted if the cost of the binary-valued and multiple-valued parts of the dominating prime is equal to or less than the cost of the binary-valued part of the dominated prime.

The cover with the minimum cost is selected. Each prime implicant in the cover has an associated range for its multiple-valued literal. We find the best implementation of the multiple-valued literal in the same manner as we calculated the cost of the literal (described in Step 2 above).

4.4 Common Factors Among Expressions

The methods described in this section can be *directly* used to identify common Boolean factors amongst multiple functions simply by using multiple-output, multiple-valued minimization.

4.5 Decomposition Using the Complement

The factorization method can be trivially extended to a decomposition method. In the case of decomposition, we can select factors such that their complements can be used as well.

The bitwise NOT of a multiple-valued literal corresponds to the complement of the factor. For example if `0101` represents $\overline{x_1} \cdot x_2 + x_1 \cdot x_2$, then `1010` represents $\overline{\overline{x_1} \cdot x_2 + x_1 \cdot x_2}$. We alter the cost function given by Eqn. 4 slightly to look for the distinct literals $\{c_{d_1(n+1)}, c_{d_2(n+1)}, \cdots, c_{d_K(n+1)}\}$ such that no $c_{d_i(n+1)}$ is the bitwise NOT of another $c_{d_j(n+1)}$. (If necessary, we can add 1 to the cost if a literal is used in both true and complemented form, modeling the cost of the inverter that is required in the decomposition.)

5 Algebraic Factorization

We describe a procedure to determine an optimal factorization of a function. This procedure generates co-kernel/kernel pairs and then performs covering to determine the factorization.

```
                    0100 0011
                    0001 0101
                    0010 1010
                    1000 1100
1010 1000           1010 1000
1001 0100           1001 0100
0110 0010           0110 0010
0101 0001           0101 0001
```

Figure 5: Kernels and Multiple-Valued Primes

All possible co-kernel/kernel pairs for a given expression can be generated using the methods proposed in [2]. Here, we construct a multiple-valued function such that the primes of the function correspond to kernels of the given expression. Given any matrix, a multiple-valued function can be constructed so that its prime implicants correspond to the prime rectangles [4] of the matrix. This formulation allows for the use symbolic prime generation methods for multiple-valued functions [7]. We construct a multiple-valued function similar to the cube-literal matrix in [4]. This function has two multiple-valued variables, where the first has a number of values equaling the number of distinct literals in the original expression, and the second has a number of values given by the number of cubes in the expression.

The multi-valued function that we construct for the function $f = a \cdot c + a \cdot d + b \cdot c + b \cdot d$ is shown to the left of Figure 5. The first variable's four positions correspond to the four literals a, b, c and d. The second variable's four positions correspond to the four cubes in f. The four implicants correspond to the four cubes in f. For a given implicant in the multiple-valued function, the first variable has `1`'s in the positions corresponding to the literals that are in the corresponding cube of f.

The primes of the multiple-valued function are shown to the right of Figure 5. The first four primes correspond to the co-kernel/kernel pairs of f. The last four primes correspond to the cubes of f. Consider the first prime. Its first multiple-valued variable has a single `1` in the position corresponding to literal a. The co-kernel corresponding to this implicant is therefore a. The set of cubes corresponding to the second multiple-valued variable in the first prime is $a \cdot b + a \cdot c$. Dividing this set of cubes by the co-kernel will give us the kernel $b + c$. Similarly for the other primes.

Consider a covering matrix where each row corresponds to a cube in a given sum-of-products expression for the given function f. Each column corresponds to either a co-kernel/kernel pair of f or a cube of f. A position in the matrix has a x if the co-kernel/kernel pair covers the corresponding cube. For example, for our function f the matrix is:

	$a \cdot (c+d)$	$b \cdot (c+d)$	$c \cdot (a+b)$	$d \cdot (a+b)$
$a \cdot c$	x		x	
$a \cdot d$	x			x
$b \cdot c$		x	x	
$b \cdot d$		x		x

The first two columns are co-kernel/kernel pairs, where a is the co-kernel of the kernel $c+d$ and b is the co-kernel of the kernel $c+d$. Similarly for the last two columns. We additionally add all the cubes of f as columns – we have not shown this in the above matrix. Optionally, prime kernel intersections can be added as columns to the matrix.

The cost of each column is given by the sum of the literals in the co-kernel and kernel corresponding to that column. The columns in the matrix all have cost 3. The covering problem to be solved corresponds to selecting a subset of columns such that there is a x in every row minimizing the total cost of the selected columns. However, there is one additional qualification. If two or more columns are selected which have the same kernel, we only count the cost of the kernel *once*. For example, if we select the first two columns, the two columns will have a cost of $3+1=4$ rather than $3+3=6$.

Once a subset of columns have been selected, we can obtain a factorization by grouping all the co-kernels together for each distinct kernel. We will obtain the factorization shown below for the above example with the minimum cost 4 by selecting the first two columns or the last two columns.

$$f = (a+b)\cdot(c+d)$$

In order to partially exploit Boolean identities, we can define a range for each kernel. The kernel $c+d$ in the co-kernel/kernel pair $a\cdot(c+d)$ can be expanded into an interval $[a\cdot c+a\cdot d,\ c+d+\overline{a}]$. Similarly, the kernel $c+d$ in the co-kernel/kernel pair $b\cdot(c+d)$ can be expanded into an interval $[b\cdot c+b\cdot d,\ c+d+\overline{b}]$. If the range of the kernel corresponding to a column intersects the range of the kernel corresponding to another column, the cost of only one kernel is counted.

The ranges above allow for the exploitation of the $a\cdot a=a$ and $a\cdot\overline{a}=0$ Boolean identities. Since a cube (row) in the matrix may be multiply covered by a selection of columns, the $a+a=a$ Boolean identity may also be exploited.

6 Experimental Results

We briefly present some results obtained by applying our Boolean factorization method to arithmetic functions. The functions were recursively factored using the methods described. We compare this method to the algebraic factorization methods implemented in the MISII program [2]. We emphasize that we are comparing the two different factorization methods in isolation from other multilevel minimization techniques.

In Table 1, we have the number of inputs (inp), outputs (out) and the literal counts of some arithmetic circuits when each output is represented in single-output minimized sum-of-products form (SOP). Each output in each function was factored separately. Running the algebraic factorization procedure available in MISII (`factor -g`) results in the literal counts under the column ALG. The CPU times required for algebraic factorization are given. The literal counts obtained using Boolean factorization are given under the column BOOL along with the CPU times required. The times are in seconds on a IBM R/S 6000 Model 320.

CKT	i/o	SOP	ALG		BOOL	
			Time	Lit	Time	Lit
add2	5/3	50	1.2s	29	17s	26
add4	9/5	206	5.2s	81	36s	75
addsub2	6/3	85	1.1s	51	21s	46
addsub4	10/5	397	3.2s	144	98s	102

Table 1: Comparisons Between Factorization Methods

7 Acknowledgements

This research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-91-J-1698 and in part by a NSF Young Investigator Award with matching funds from Mitsubishi and IBM Corporation.

References

[1] K. Bartlett, R. K. Brayton, G. D. Hachtel, R. M. Jacoby, C. R. Morrison, R. L. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang. Multi-level Logic Minimization Using Implicit Don't Cares. In *IEEE Transactions on Computer-Aided Design*, pages 723–740, June 1988.

[2] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wan g. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, CAD-6(6):1062–1081, November 1987.

[3] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[4] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. Multilevel Logic Synthesis. In *Proceedings of the IEEE*, pages 264–300, February 1990.

[5] S. Devadas, A. R. Wang, A. R. Newton, and A. Sangiovanni-Vincentelli. Boolean Decomposition in Multi-Level Logic Optimization. In *Journal of Solid State Circuits*, volume 24, pages 399–408, April 1989.

[6] D. Bostick et. al. The Boulder Optimal Logic Design System. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 62–65, November 1987.

[7] B. Lin, O. Coudert, and J-C. Madre. Symbolic Prime Generation for Multiple-Valued Functions. In *Proceedings of the* 29^{th} *Design Automation Conference*, pages 40–44, June 1992.

[8] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. In *IEEE Transactions on Computer-Aided Design*, pages 727–751, September 1987.