Exploring Multi-Valued Minimization Using Binary Methods

Alan Mishchenko
Department of EECS
UC Berkeley
Berkeley, CA 94720
alanmi@eecs.berkeley.edu

Robert K. Brayton
Department of EECS
UC Berkeley
Berkeley, CA 94720
brayton@eecs.berkeley.edu

Tsutomu Sasao
Center for MS and Dept. of CSE
Kyushu Institute of Technology
Iizuka, 820-8502, Japan
sasao@cse.kyutech.ac.jp

Abstract

A transformation of multi-valued input binary-output functions, called co-singleton transform (CST)*, was introduced in* [11] *to reduce algebraic multi-valued (MV) operations to binary. In this paper, we explore its potential for a number of problems related to MV SOP minimization, such as computing ISOPs, the set of all primes, and the set of all essential primes. Experimental results show that in some cases these problems can be solved more efficiently than by the traditional MV SOP minimization approaches represented by ESPRESSO-MV, but that generally there is no clear method-of-choice.*

1 Introduction

Several approaches have been proposed to minimize an SOP representation of binary multi-output logic functions [10][2][8] and to solve related problems, such as the computation of ISOPs [20][17][5], the set of all primes [20][5], and the set of essential primes [25][5]. The problem of SOP minimization has been solved successfully for multi-valued-input binary-output functions (MV functions, for short) [10][23]. This approach stores and manipulates MV SOPs in their original MV form, as sets of MV cubes represented in positional notation using bit-strings.

In contrast, the co-*singleton transform* (CST) [11], can be used to transform a binary solution into a solution of the original MV problem. The approach [11] was applied successfully to algebraic extraction of logic in MV networks [3][22], resulting in comparable quality and noticeable reduction in runtime, compared to a purely MV algebraic approach [9]. Experiments were done only in applications where algebraic operations were performed.

In is natural to try to extend this success to apply the CST to Boolean problems, in particular those related to MV SOP minimization. The binary problems resulting from the CST can be solved using binary BDD-based algorithms, which are often simpler than dedicated MV algorithms. To extend the CST to the Boolean domain, induced don't cares have to be introduced and algorithms are modified to generate only positive unate covers.

For algebraic problems, it was shown that the CST was the unique smallest encoding possible that preserved certain desirable properties. For Boolean problems, no such statement exists; indeed, an equivalent alternate method would be to use the "1-hot" code (the CST is a "1-not-hot code"). Induced don't cares for both are similar but the targeted cover is negative unate in the 1-hot case.

In fact, use of 1-hot codes was the way that MV minimization problems were solved originally using ESPRESSO-binary. ESPRESSO-MV replaced such methods since it was much faster. Besides better data-structures, one reason for this is that induced don't cares are not necessary because MV variables implicitly require a single value at a time. The other reason is that the number of variables is less with the MV representation than with the 1-hot one, although each MV-variable is more complicated.

This paper re-visits these issues, motivated by the success of CST for algebraic problems and with the hope that better data-structures and new ideas may offer interesting alternative methods. Although the 1-hot encoding could be used as well, in this paper we discuss all the ideas in terms of the CST for uniformity with the algebraic methods.

The contributions of this paper are to propose, for MV functions, implicit algorithms to compute a) MV-ISOPs [17], b) the set of all MV-primes, and c) the set of all MV-essential primes. Except for an implicit algorithm to compute the primes of an MV function [13], no comparable algorithms are known.

The paper is organized as follows. Section 2 introduces terminology and notation. Section 3 defines the CST. Sections 4, 5, and 6 discuss algorithms for ISOP, prime, and essential prime computation, respectively. Section 7 compares traditional solutions of these problems with solutions obtained by the algorithms proposed in this paper. Section 8 concludes the paper and outlines some directions for future work.

2 Preliminaries

The following terminology is assumed familiar to the reader:

(1) Binary SOP minimization [2] and, in particular, concepts of literal, cube, prime, essential primes.
(2) Multi-valued SOP minimization [23] and extension of the above concepts for the MV case.
(3) Binary Decision Diagrams (BDDs) [4], in particular, concepts of cofactoring, ITE operator, the generic structure of a BDD traversal procedure.
(4) Zero-suppressed Binary Decision Diagrams (ZDDs) [16] and, in particular, ZDD representation of combinatorial sets and cube covers.
(5) MV functions and relations, as defined in [19].

We refer to completely specified functions as CSFs and incompletely specified functions as ISFs. Unless specifically stated, a CSF or ISF has multi-valued-inputs and a binary output.

An ISF can be represented by two CSFs, L and U, called the *lower bound* and the *upper bound*, respectively. The lower bound of an ISF is its on-set, i.e. the set of minterms where the ISF *must* have value 1. The upper bound is the sum of its on-set and don't-care-set, i.e. the set of minterms where the ISF *can* have value 1. The offset is the complement of the upper bound.

A CSF F is *compatible* with an ISF (L, U) iff $L \Rightarrow F \Rightarrow U$, where symbol "$\Rightarrow$" denotes implication. The *support* of a CSF F, *supp*(F), is the set of variables X, which influence the output value of F. The support size is denoted by |*supp*(F)|.

3 The Co-Singleton Transform

The co-singleton transform (CST), introduced in [11], is defined for MV literals, MV cubes, MV SOPs, and MV functions.

Definition. An MV literal is *non-trivial* if it is not constant 0 (MV literal has the empty value set) nor constant 1 (MV literal has the value set equal to all possible values).

Example. Consider variable X with 5 values. The domain of X is {0,1,2,3,4}. The trivial literals are $X^{\{\}}$ and $X^{\{0,1,2,3,4\}}$. A non-trivial literal is $X^{\{1,3\}}$.

Definition (**CST of MV literal**). Let X be a k-valued MV variable with domain $D_X = \{0, 1, \dots, k\text{-}1\}$. Let l be a non-trivial MV literal of X. Let S_l be the value set of l. Let the set of binary variables $\{x_0, x_1, \dots, x_{k-1}\}$ be put in one-to-one correspondence with the values of D_X. The CST of l is a cube L composed of variables x_i. For each value $v \notin S_l$, the corresponding variable x_v is added to the cube L in the positive polarity: $L = \prod_{v \notin S_l} x_v$

Because l is non-trivial, S_l is a non-empty proper subset of D_X. Thus L contains at least one binary variable x_i.

Example. Consider variable X with 5 values and MV literals $l_1 = X^{\{0,3,4\}}$ and $l_2 = X^{\{2\}}$. The CST of literals l_1 and l_2 are cubes L_1 and L_2, respectively: $L_1 = x_1x_2$, $L_2 = x_0x_1x_3x_4$.

Definition (**CST of MV cube**). Let c be an MV cube $c = l_1l_2\dots l_n$, where l_i are non-trivial literals. The CST of c is a cube C, called *CST cube,* equal to the product of the CSTs of literals l_i.

$$C = L_1L_2\dots L_n.$$

Example. Consider MV cubes $c_1 = A^{\{0,1\}}B^{\{0,3\}}C^{\{3\}}$ and $c_2 = A^{\{0\}}C^{\{1,2,3\}}$; A is 3-valued and B and C are 4-valued. The CST of c_1 and c_2 , are $C_1 = a_2b_1b_2c_0c_1c_2$, $C_2 = a_1a_2c_0$, respectively.

Definition (**CST of MV SOP**). Let t be an MV SOP composed of cubes $c_1, c_2, \dots, c_k$, where c_i are MV cubes. The CST of t is the binary SOP T, called *CST SOP*, composed of cubes C_i that are CSTs of MV cubes c_i: $T = \sum_{i=1}^{k} C_i$.

Example. MV SOP composed of the two MV cubes in the previous example is $t = A^{\{0,1\}}B^{\{0,3\}}C^{\{3\}} \vee A^{\{0\}}C^{\{1,2,3\}}$. Then the CST of t is $T = a_2b_1b_2c_0c_1c_2 \vee a_1a_2c_0$.

Definition (**Inverse CST**). Let T be a positive unate SOP composed of binary variables in one-to-one correspondence with the values of a set of MV variables. Then, the *inverse CST* (ICST) is the transform, which translates the CST SOP into an MV SOP using the rules inverse to those of CST.

Example. Let positive unate SOP be

$$T = a_2b_1b_2c_0c_2 \vee b_0c_0c_1 \vee a_1,$$

and suppose the binary values a_i, b_i, and c_i are in one-to-one correspondence with the values of ternary MV variables A, B, and C. Let t be ICST of T. Then,

$$t = A^{\{0,1\}}B^{\{0\}}C^{\{1\}} \vee B^{\{1,2\}}C^{\{2\}} \vee A^{\{0,2\}}.$$

Property. The cardinality of MV SOP and CST SOP is the same.

Proof. CST transforms each MV cube into a binary cube and the ICST transforms each binary cube into an MV cube. *Q. E. D.*

Property (Singleton literal). Let X be a k-valued MV variable. Let binary variables $\{x_0, x_1, \dots, x_{k-1}\}$ be used to derive the CST of MV literals of X. Then, $\bar{x}_i = X^{\{i\}}$.

Proof. Consider literal $l = X^{\{0,1,\dots,i-1,i+1,\dots,k-1\}}$. Its CST is x_i. The complement of l is $X^{\{i\}}$. Therefore, $\bar{x}_1 = X^{\{i\}}$. *Q.E.D.*

When an MV object (literal, cube or SOP) is subject to CST, a don't-care set is implied, because some of the combinations of the binary variables cannot occur in the transformed object. The following property shows how to compute the CST-generated don't-cares.

Property (**Induced don't-care**). When a k-valued MV variable undergoes CST, the following CST-related don't-care (*induced don't-care*, or *CST DC*) is generated in the binary domain:

$$x_0 \dots x_{k-1} + \sum_{\substack{0 \le i,j < k \\ i \ne j}} \bar{x}_i \bar{x}_j \ .$$

Proof. By the above property, $\bar{x}_i = X^{\{i\}}$ and $\bar{x}_j = X^{\{j\}}$, $i \ne j$. The product of the left-hand sides is $\bar{x}_i\bar{x}_j$. The product of the right-hand sides is $X^{\{\}}$, the literal with the empty set of values. Literal cube $X^{\{\}}$ can be added to any MV SOP without changing the function represented by it. Therefore, $\bar{x}_i\bar{x}_j$ is a don't-care. Similarly, $x_0 \dots x_{k-1} = X^{\{\}}$. Adding all don't-cares of this kind, we get the property. *Q. E. D.*

Definition (**CST of CSF**). Let f be a MV CSF and t be an MV SOP of f. Let T be CST SOP of t and D be the CST DC. Let F be the Boolean function of T. The CST of f is the binary ISF represented by the interval $(F \wedge \overline{D}\ , F \vee D)$.

Definition (**CST of ISF**). Let $f = (f_1, f_2)$ be an MV ISF. Let t_1 and t_2 be MV SOPs of f_1 and f_2. Let T_1 and T_2 be CST SOPs of t_1 and t_2. Let D be the CST DC. Let F_1 and F_2 be Boolean functions of T_1 and T_2. The CST of f is the binary ISF represented by the interval $(F_1 \wedge \overline{D}\ , F_2 \vee D)$.

The following properties are important for SOP minimization.

Property. The CST of a CSF or an ISF is unique and does not depend on MV SOPs used to generate it.

Property. The size of the domain (the total number of minterms in the domain) of the transformed function increases compared to the domain of the original MV function. The number of onset and offset minterms remains the same. It follows that the ratios of on-set size and off-set size to the domain size decrease.

Property. When CST don't-cares are used, minimality of the MV SOP and CST SOPs translates across the domains.

Property. A cube of the original function is prime iff a transformed cube is a prime of the CST function interval.

Property. The primes of the CST function form equivalence classes, distinguished by the polarity of variables. The set of all

primes of the MV function can be computed by computing the CST primes of only one polarity, for example, positive unate.

Property. Primeness and irredundancy of covers translates across the domains.

Property. All minterms composed of the on-set and the off-set of the CST function interval are distance two or more.

Example. Consider $f(A, B) = A^{\{0\}}B^{\{1,2\}} \vee B^{\{1\}} \vee A^{\{1\}}B^{\{0,1\}}$, where A is binary and B is ternary. Let F be the CST of f and D be the CST DC. Then,

$$F = a_1b_0 + b_0b_2 + a_0b_2$$

$$D = \overline{a}_0\overline{a}_1 + \overline{b}_0\overline{b}_1 + \overline{b}_0\overline{b}_2 + \overline{b}_1\overline{b}_2 + a_0a_1 + b_0b_1b_2$$

$a_0a_1 \setminus b_0b_1b_2$	000	001	011	010	110	111	101	100
00	-	-	-	-	-	-	-	-
01	-	-	0	-	1	-	1	-
11	-	-	-	-	-	-	-	-
10	-	-	1	-	0	-	1	-

Figure 1. MV function transformed by CST.

The function interval ($F \wedge \overline{D}$, $F \vee D$) is shown in Figure 1. Only the positive unate primes are shown. Note that both the CST cube b_0b_2, and the corresponding cube $B^{\{1\}}$ of the original function are redundant. (*End of example.*)

Figure 2 shows a recursive algorithm to generate the CST for a binary function represented by a BDD. In the pseudo-code, the binary variable x is represented using two CST variables, x_0 and x_1. Note that although both the original and the CST functions are binary, the former can be binate while the latter is always unate.

```
function CST( function F )
{
   if ( F = 0 )  return 0;
   if ( F = 1 )  return 1;
   x = TopVariable( F );
   ( F0, F1 ) = Cofactors( F, x );
   R0 = CST( F0 );
   R1 = CST( F1 );
   return x1R0 ∨ x0R1;
}
```

Figure 2. CST for a function represented by the BDD.

Although not shown, the recursive pseudo-code discussed in this and the following figures, uses cache for computed results.

4 Computing ISOP

Computation of ISOP for MV function is of practical interest because, in some cases, it can be used as a fast heuristic SOP minimization procedure. In particular, when the SOP of a function to be minimized is composed of essential primes only, the ISOP is the exact minimum.

For binary functions, an efficient recursive ISOP computation algorithm is known [20][17][5]. Given the function interval of an ISF, it computes the ISOP in one traversal of the shared BDD of the lower and the upper bound of the interval.

This ISOP algorithm has not been extended to MV functions represented by MDDs. However, it is possible to apply the binary ISOP computation to MV functions represented using binary encoded MDDs (BEMDDs). To reduce the number of binary variables, a logarithmic encoding of the values of MV variables is used. The resulting ISOP for the binary function is converted back to the MV domain. This method is fast and often generates MV ISOPs much smaller than the set of disjoint SOPs derived by enumerating the paths to terminal 1 in the BDD. Although the resulting ISOP is prime and irredundant in the binary domain, it may not be prime and irredundant after decoding back to the MV domain; the result may not even be single-cube containment free.

4.1 ISOP for Binary Functions

We briefly review the ISOP computation [20][17][5].

```
cover ISOP( function L, function U )
{
   if ( L = 0 )  return {};
   if ( U = 1 )  return {{}};
   x = TopVariable( L, U );
   ( L0, L1 ) = Cofactors( L, x );
   ( U0, U1 ) = Cofactors( U, x );
   R0 = ISOP( L0 ∧ ¬U1, U0 );
   R1 = ISOP( L1 ∧ ¬U0, U1 );
   R2 = ISOP( L0 ∧ ¬Bdd(R0) ∨ L1 ∧ ¬Bdd(R1), U0 ∧ U1 );
   return Cover( x, R0, R1, R2 );
}
```

Figure 3. ISOP computation.

The procedure in Figure 3 takes the function interval (L, U) representing an ISF and returns the ISOP of the ISF. If the lower bound L is 0, the empty cover is returned. If the upper bound U is 1, the cover composed of the cube without literals (the tautology cube) is returned. Otherwise, the topmost variable x in the BDDs is found and used to cofactor of L and U.

Next, are three recursive calls to ISOP, which return cubes containing variable x in the negative polarity (R_0), in the positive polarity (R_1) and without variable x (R_2).

To compute R_0, it is necessary to cover that part of the negative cofactor of the on-set (L_0), which cannot be covered by cubes with the positive polarity literal and without literal x (U_1). Therefore, the lower bound for the recursive call, which computes R_0, is derived by "sharping" L_0 with U_1 ($L_0 \wedge \overline{U}_1$). The computation of R_1 is similar.

The computation of R_2 is based on the partial solutions, R_0 and R_1. The lower bound is the part of the negative cofactor of the on-set (L_0) not covered by R_0 and the part of the positive cofactor of the on-set (L_1) not covered by R_1. Cubes without variable x belong equally to the negative and the positive parts of the on-set. This is why the upper bound in the last recursive call is the intersection of the cofactors of the initial upper bound.

4.2 ISOP for MV Functions

Figure 4 shows a new algorithm for MV ISOP computation for the CST function using the induced don't-cares. The approach is different from the binate ISOP computation shown in Figure 3 in that it generates a unate cover. The assertion, after cofactoring, ensures that the unate cover of the given function can be computed. The cubes without the literal of variable x (R_2) are computed for the negative cofactor of the on-set, followed by covering the remaining part of the domain of the positive cofactor of the on-set with the cubes containing the positive literal of

variable x (R_1). The resulting cover is constructed using the empty set for the set of cubes with the negative literal (R_0).

```
cover ISOP( function L, function U )
{
   if ( L = 0 )  return {};
   if ( U = 1 )  return {{}};
   x = TopVariable( L, U );
   ( L0, L1 ) = Cofactors( L, x );
   ( U0, U1 ) = Cofactors( U, x );
   assert( L0 ⇒ U1 );
   R2 = ISOP( L0, U0 ∧ U1 );
   R1 = ISOP( L1 ∧ Bdd(R0), U1 );
   return Cover( x, {}, R1, R2 );
}
```

Figure 4. Unate ISOP computation.

Experiments show that the algorithm in Figure 4 results in ISOPs with many more cubes, compared to the covers generated by applying the general-case binate ISOP computation in Figure 3 to the CST function.

One possible reason is that the function interval of the CST function with don't-cares is such that the number of minterms in the on-set and the off-set is the same as in the original function. Meanwhile, the size of the domain has increased. Moreover, all the on-set and off-set minterms are distance two or more from each other; thus 0s and 1s in the care set of the Boolean space of the CST function are surrounded by vast domains of don't-cares. The binate ISOP computation is more "flexible" in trying to distinguish among the 0s and 1s using the cubes with both positive and negative literals, while the unate ISOP computation is more "rigid" in always using only positive unate cubes.

5 Computing Primes

Prime computation for MV functions is important for applications, such as exact SOP minimization. A recursive algorithm was proposed in [20] and efficiently implemented in [8] for multi-output binary functions represented using BDDs. The computation of primes for MV functions in Espresso-MV [2][23] is performed recursively using a bit-set cube representation.

A BDD-based prime computation algorithm for MV functions exists [13] but is not efficient in practice, because of the large intermediate BDD representations. In this section, a new method is given to compute the set of all primes for MV functions using the CST function and the induced don't-cares. In some cases, this method is faster than Espresso-MV, because it uses the implicit representation and exploits unateness of the transformed function.

5.1 Primes of Binary Functions

We briefly review the binary prime computation procedure for CSFs [20][8], shown in Figure 5.

If the function is 0, the set of primes is empty; if the function is 1, the set of primes is the tautology cube. Otherwise, F is cofactored using the topmost variable x in the BDD. Let R_0 and R_1 be primes computed for the cofactors of F. The set of primes without variable x are those primes R_2 which appear as primes of both F_0 and F_1. Alternatively, if is possible to compute R_2 as the set of primes in the intersection of cofactors, $F_0 \wedge F_1$. The sets of primes with negative and positive literal x are derived by removing from R_0 and R_1 the primes R_2.

```
cover Primes( function F )
{
   if ( F = 0 )  return {};
   if ( F = 1 )  return {{}};
   x = TopVariable( F );
   ( F0, F1 ) = Cofactors( F, x );
   R0 = Primes( F0 );
   R1 = Primes( F1 );
   R2 = Primes( F0 ∧ F1 );
   R0 = R0 \ R2;
   R1 = R1 \ R2;
   return Cover( x, R0, R1, R2 );
}
```

Figure 5. Binary prime computation.

5.2 Primes of MV Functions

The computation of all primes for a binary ISF (L, U) is performed in two steps. First, all the primes of U are computed, as shown in Figure 5. Next, the procedure in Figure 8 is applied to select only those primes that overlap with L.

If the binary ISF is the CST transformed function for an MV function, the resulting binary primes are projected to the MV domain using the inverse CST. By the property in Section 3, to compute all the primes of an MV function, it is enough to compute only the positive unate primes for the binary ISF. This is done using the pseudo-code in Figure 6.

The computation of positive primes in Figure 6 is similar to that of all primes in Figure 5, except that the primes R_0, with negative literal x, are not computed, and the empty set is used instead of R_0 when creating the result.

```
cover PositivePrimes( function F )
{
   if ( F = 0 )  return {};
   if ( F = 1 )  return {{}};
   x = TopVariable( F );
   ( F0, F1 ) = Cofactors( F, x );
   R1 = PositivePrimes( F1 );
   R2 = PositivePrimes( F0 ∧ F1 );
   R1 = R1 \ R2;
   return Cover( x, {}, R1, R2 );
}
```

Figure 6. Positive prime computation.

6 Computing Essential Primes

One algorithm to compute essential primes of an MV function [25] uses a bit-set representation of the SOP of the function. We propose two new algorithms for the computation of the set of all essential primes. These operate on the BDDs of interval (L, U) representing the transformed function. The first algorithm requires the computation of all primes; the second does not.

Algorithm 1.

The algorithm shown in Figure 7 computes the set R of all primes of the upper bound U of the CST function. Computed next is the set P of those primes in R that overlap with the on-set L. Domain A is the Boolean space covered by only one cube in P.

Finally, the set of essential primes E is primes in P that overlap with A. If an ISOP C contained in the interval (L, U) is available, C can be used instead of P in the last step because an ISOP contains all essential primes.

```
cover Essentials( interval (L,U), subsets S )
{
   cover R  = FilteredPrimes( U, S );
   cover P  = OverlappingCubes( R, L );
   domain A = SingleCoveredDomain( P );
   cover E  = OverlappingCubes( P, A );
   return E;
}
```

Figure 7. Essential prime computation using primes.

The procedures used in Figure 7, are detailed in Figure 8 and Figure 9. These work for an arbitrary cover C. When the cover is positive unate, they can be simplified by setting C_0 to the empty set.

```
cover OverlappingCubes( cover C, function F )
{
   if ( F = 0 or C = {} )  return {};
   if ( F = 1 )            return C;
   x = TopVariable( C, F );
   ( C0, C1, C2 ) = Cofactors( C, x );
   ( F0, F1 )     = Cofactors( F, x );
   R0 = OverlappingCubes( C0, F0 );
   R1 = OverlappingCubes( C1, F1 );
   R2 = OverlappingCubes( C2, F0 ∨ F1 );
   return Cover( x, R0, R1, R2 );
}
```

Figure 8. Computing cubes overlapping with domain.

```
function SingleCoveredDomain( cover C )
{
   if ( C = {}   )  return 0;
   if ( C = {{}} )  return 1;
   x = TopVariable( C );
   ( C0, C1, C2 ) = Cofactors( C, x );
   F0 = SingleCoveredDomain( C0 ∪ C2 );
   F1 = SingleCoveredDomain( C1 ∪ C2 );
   return ITE( x, F1, F0 );
}
```

Figure 9. Computing domain covered by a single cube.

Algorithm 2.

This algorithm in Figure 10 takes the function interval (L, U) and returns the set of positive unate primes and the *essential domain*, i.e. the sub-domain of L covered by only one prime. Returning the essential domain and the set of essential primes allows for not computing the set of all primes. To simplify the pseudo-code, the subset S used for filtering primes is not shown.

After checking for the trivial cases and cofactoring the argument functions in Figure 10, the problem is solved recursively for the domains inside and outside the intersection of cofactors. Because L is positive unate, the domain outside the intersection can be covered by cubes with the positive literal. The subsequent steps compute the essential domain and essential primes inside and outside of the intersection of cofactors, assuming that the partial problems have been solved, and the return values are created. Note that the positive cofactor of the essential domain is the sum of the essential domains inside and outside the intersection, while the negative cofactor is the essential domain inside the intersection.

```
(cover, function) Essentials( interval (L,U) )
{
   if ( L = 0 )  return ( {},   0 );
   if ( U = 1 )  return ( {{}}, L );
   x = TopVariable( L, U );
   ( L0, L1 ) = Cofactors( L, x );
   ( U0, U1 ) = Cofactors( U, x );
   assert( L is positive unate );
   // recursively solve in the intersection
   ( P2, E2 ) = Essentials( L0, H0 ∧ H1 );
   // recursively solve outside the intersection
   ( P1, E1 ) = Essentials( L1 - L0, H1 );
   // essential primes in the intersection
   P2 = P1 ∩ P2;
   // essential domain in the intersection
   A2 = CoveredDomain( E1, P2 );
   // essential area outside the intersection
   A1 = E1 - (H0 ∧ H1);
   // essential primes outside the intersection
   P1 = OverlappingCubes( P1, A1 );
   return ( Cover(x,{},P1,P2), ITE(x, A1∨A2, A2) );
}
```

Figure 10. Computing essential primes without computing primes.

7 Experimental results

The algorithms for ISOP and prime computation were implemented in C using the CUDD package [27] and the EXTRA library [18]. The implementation was tested using a 994GHz 256Mb RAM computer under Windows XP.

Table 1 compares several algorithms for ISOP computation. The MV-input binary-output benchmarks were derived from Espresso benchmarks using pair-decoding [24]. Pair-decoding consists of replacing pairs of binary variables by 4-valued variables. For an odd number of binary variables, one variable is left unpaired. The output MV variable is not changed.

Table 1 shows the benchmark parameters: the number of MV variables after pair-decoding ("vars"), the total number of all values ("Σv"), and the total number of binary variables needed for the logarithmic encoding of each variable in the function ("Σb"). This encoding is used to derive Binary Encoded MDD (BEMDD), to which a binary ISOP computation is applied.

Columns "Disj SOP" gives the number of cubes in the disjoint SOP of the given function derived by enumerating the paths to the terminal 1 in the BEMDD of the function.

Columns "ISOP-log" shows the number of cubes and runtime in seconds to compute the MV SOP by first computing the binary SOP of the BEMDD. The resulting cover is single-cube containment free but is not necessarily prime or irredundant. Note that the BEMDD depends on "Σb" binary variables. Also, note that unlike CST, the BEMDD does not provide any don't cares induced by the encoding.

Columns "ISOP-cst" shows the number of cubes and runtime to compute the MV SOP of the same function using the algorithms proposed. This cover is irredundant but not necessarily prime (conjecture). Note that the CST-based representation of the function depends on "Σv" binary variables. The results in this and

the previous section were verified by deriving the BEMDD for the resulting MV ISOPs and comparing them with the original BEMDD.

The final columns contain the results of running Espresso-MV with three different options. “Espr-fast” performs only one EXPAND and IRREDUNDANT without MAKE_SPARSE. “Espr-heu” runs the heuristic MV SOP minimization loop until no improvement. “Espr-exact” minimizes the functions exactly. The dashes in the table mean the program did not finish within 1 minute. The results were internally verified by Espresso.

Table 1 shows that CST-based minimization is generally a failure; it is always slower than ISOP-log, and only in 8 examples does it derive fewer cubes than ISOP-log. Also, the CST-based computation does not finish in reasonable time for some benchmarks because the CST function depends on more variables (“Σv”) than the BEMDD (“Σb”). However, ISOP-log (BEMDD) seems to provide a useful trade-off; although (as expected) it is always worse (in terms of cubes) than Espresso-fast, it is always (except for one example) faster.

Table 2 shows the results of prime computation for both the original and bit-paired benchmark functions. The parameters section of the table includes the number of inputs (“Ins”), outputs (“Outs”), and the number of values (“Values”). Sections “CST-based” show the sizes of the ZDD representations of the sets of primes, and the number of primes computed using the CST-based approach. The number of primes is the same in all the prime computation methods (of course). The subsequent three columns contain the runtime of the CST-based computation, the output-1-hot-encoded approach [1][2][8], and the explicit MV prime computation in Espresso-MV [23].

The 1-hot-encoded representation was only implemented for binary-input functions. Therefore, column “1-hot” is not listed on the right for the benchmark functions obtained by bit-pairing.

The conclusion from

Table 2 is that CST/ZDD prime computation, for the MV-input examples, is generally superior to the explicit computation in Espresso-MV; in 14 of the 17 cases CST is faster. For some benchmarks (*apex2*) CST is more than two orders of magnitude faster. However, CST is slower, on binary input examples, than the computation of primes using output-1-hot encoding. The reason is similar to why the CST is slower than the ISOP computed from the BEMDD; the output-1-hot encoded representation depends on fewer variables (Ins + Outs) compared to the CST representation (2*Ins + Outs).

Experiments have not been done yet for generating essential primes.

8 Conclusions

In this paper, the approach of reducing multi-valued algebraic operations to binary using the co-singleton transform [11] was extended to a number of problems related to SOP minimization. The approach reduces the minimization of the general-case MV functions to the minimization of binary unate functions with don’t-cares.

Algorithms were given to solve typical problems related to SOP minimization of MV functions, such as computation of ISOPs, computation of all primes, and essential primes. The CST results for minimization were generally inferior to those using logarithmic encoding (ISOP/BEMDD). For generating primes for MV-input examples, the new method was almost always faster than ESPRESSO-MV prime generation. For both high quality and reasonable runtime ESPRESSO-MV-heuristic [23] is still the method of choice for minimizing MV SOP functions.

The ISOP/BEMDD approach can be useful as an alternative to ESPRESSO-MV-fast for trading off fast MV-minimization for less optimal results. Generally, fast ISOP computation can be a useful starter for ESPRESSO-MV, but the latter can be time consuming, especially for functions with large offsets. Future work will focus on achieving a better runtime/quality trade-off in the SOP minimization for MV-functions and/or MV-relations arising in the optimization of MV networks in MVSIS [21].

Acknowledgements

The first and second authors acknowledge the generous long-term support of the SRC under contract 683.004, as well as the GSRC, and the California Micro program with our industrial sponsors, Cadence and Synplicity. The first and third authors acknowledge the support from Kyushu Institute of Technology under the 75th Commemoration Fund Program for Foreign Researchers, which partially sponsored this research during the first author’s visit in Japan.

References

[1] T. C. Bartee, “Computer design of multiple-output logical networks”. *IRE Trans. Electr. Comp.*, March 1961, pp. 21-30.

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

[3] R. K. Brayton and C. McMullen, “The decomposition and factorization of Boolean expressions,” *Proc. ISCAS ‘82*, pp. 29-54.

[4] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comp.*, Vol. C-35, No. 8 (August, 1986), pp. 677-691.

[5] O. Coudert and J. C. Madre, “Implicit and incremental computation of primes and essential primes of Boolean functions”, *Proc. DAC ’92,* pp. 36-39.

[6] O. Coudert, J. C. Madre, H. Fraisse, H. Touati, "Implicit prime cover computation: An overview", *Proc. SASIMI '93,* Nara, Japan.

[7] O. Coudert and J. C. Madre, “Towards a symbolic logic minimization algorithm”, *Proc. VLSI Design*, January 1993.

[8] O. Coudert, “Two-level logic minimization: An overview”, *Integration*, 17-2, pp. 97-140, Oct. 1994.

[9] M. Gao and R. K. Brayton, “Semi-algebraic methods for multi-valued logic”, *Proc. IWLS ‘00*, pp. 73-80.

[10] S. J. Hong, R. G. Cain, and D. L. Ostapko, “MINI: A heuristic approach for logic minimization”, *IBM J. Res. Develop.*, Sept. 1974, pp. 443-458.

[11] J.-H. R. Jiang, A. Mishchenko, and R. K. Brayton, “Reducing multi-valued algebraic operations to binary”, *Proc. DATE ’03*, to appear.

[12] T. Kam, T. Villa, R. Brayton, A. Sangiovanni-Vincentelli. *Synthesis of Finite State Machines: Functional Optimization.* Kluwer Academic Publishers, 1997.

[13] B. Lin, O. Coudert, and J-C. Madre, “Symbolic prime generation for multiple-valued functions”, *Proc. DAC ‘92*, pp. 40-44.

[14] A. A. Malik, R. Brayton, A. R. Newton and A. Sangiovanni-Vincentelli, “Reduced offsets for two-level multi-valued logic minimization”, *Proc. DAC’ 90*, pp. 290-296.

[15] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincentelli, "Espresso-Signature: A new exact minimizer for logic functions," *Proc. DAC ’93*, pp. 618-624.

[16] S. Minato, “Zero-suppressed BDDs for set manipulation in combinatorial problems”, *Proc. of DAC ‘93*, pp. 272-277.

[17] S. Minato, “Fast generation of irredundant sum-of-products forms from binary decision diagrams”, *Proc. SASIMI '92*, pp. 64-73.

[18] A. Mishchenko, *EXTRA Library of DD procedures*. http://www.ee.pdx.edu/~alanmi/research/ extra.htm

[19] A. Mishchenko and R. K. Brayton, “Simplification of non-deterministic multi-valued networks”, *Proc. ICCAD ‘02*, pp. 557-562.

[20] E. Morreale, “Recursive operators for prime implicant and irredundant normal form determination”, *IEEE Trans. Comp.*, C-19(6), 1970, pp. 504-509.

[21] MVSIS Group. *MVSIS.* UC Berkeley. http://www-cad.eecs.berkeley.edu/mvsis/

[22] J. Rajski and J. Vasudevamurthy, “The testability-preserving concurrent decomposition and factorization of Boolean expressions”, *IEEE Trans. CAD*, Vol. 11(6), June 1992, pp. 778-793.

[23] R. L. Rudell and A. Sangiovanni-Vincentelli, “Multiple-valued minimization for PLA optimization”, *IEEE Trans. CAD*, Vol. 6(5), pp. 727-750, Sep. 1987.

[24] T. Sasao, "Multiple-valued decomposition of generalized Boolean functions and the complexityof programmable logic arrays," *IEEE Trans. Comp.*,Vol. C-30, No. 9, pp. 635-643, Sept. l981.

[25] T. Sasao, "An algorithm to derive the complement of a binary function with multiple-valued inputs," *IEEE Trans. Comp*. Vol. C-34, No. 2, pp. 131-140, Feb. 1985.

[26] E. Sentovich, et al, “SIS: A system for sequential circuit synthesis”, *Tech. Rep. UCB/ERI, M92/41, ERL*, Dept. of EECS, Univ. of California, Berkeley, 1992.

[27] F. Somenzi, BDD package “CUDD v. 2.3.0.” http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

Table 1. Comparision of ISOP computation algorithms.

Bench mark	Parameters			Disj SOP	ISOP-log		ISOP-cst		Espr-fast		Espr-heu		Espr-exact	
	vars	Σv	Σb	Cubes	Cubes	Time	Cubes	Time	Cubes	Time	Cubes	Time	Cubes	Time
5xp1	5	24	11	395	69	0.01	110	0.02	47	0.04	46	0.05	46	0.11
9sym	6	19	10	148	148	0.01	112	0.01	42	0.01	31	0.07	26	0.09
alu4	8	36	17	17,790	637	0.24	1273	0.62	304	0.51	288	2.24	280	15.69
apex1	24	135	51	17,482,296	746	0.60	-	-	216	2.62	197	6.20	-	-
apex4	6	37	14	1,570	940	0.04	1133	1.61	671	0.49	466	3.51	-	-
b12	9	39	19	2,796	48	0.01	121	0.02	34	0.01	32	0.04	30	1.15
clip	6	23	12	826	146	0.01	91	0.01	42	0.02	42	0.02	40	0.11
con1	5	16	8	30	9	0.01	18	0.01	8	0.01	8	0.01	8	0.01
cordic	13	48	24	79,936	1180	0.01	918	0.01	78	1.74	78	2.36	78	13.16
cps	13	157	31	17,510	511	0.09	-	-	174	0.54	148	1.39	144	9.55
duke2	12	73	27	93,472	188	0.03	-	-	86	0.08	76	0.22	75	0.93
e64	34	195	72	466,152	65	0.05	-	-	65	0.20	65	12.50	65	12.50
ex1010	6	30	14	3,325	1210	0.08	1202	0.92	543	0.67	241	1.36	-	-
ex5	5	79	14	1,679	211	0.02	-	-	87	0.16	70	0.55	-	-
inc	5	23	11	104	46	0.01	63	0.01	36	0.01	29	0.01	27	0.01
misex1	5	23	11	76	21	0.01	72	0.01	13	0.01	13	0.01	12	0.01
misex2	14	68	30	2,228	29	0.01	70	0.64	28	0.02	27	0.01	27	0.01
misex3	8	42	18	11,640	1104	0.08	2236	1.01	631	1.21	498	6.87	-	-
misex3c	8	42	18	62,031	311	0.24	925	0.61	171	0.20	147	0.98	-	-
rd53	4	13	7	32	31	0.01	26	0.01	13	0.01	13	0.01	12	0.01
rd73	5	17	9	169	147	0.01	77	0.01	38	0.01	37	0.01	37	0.01
rd84	5	20	10	291	258	0.01	171	0.01	56	0.01	54	0.02	54	0.04
sao2	6	24	12	139	60	0.01	112	0.01	39	0.01	39	0.02	38	0.02
seq	22	117	47	1,717,537	1148	0.27	-	-	309	5.31	222	9.46	-	-
spla	9	78	22	88,885	595	0.23	-	-	204	0.29	185	0.78	-	-
squar5	4	18	8	66	28	0.01	33	0.01	24	0.01	21	0.01	21	0.01
table3	8	42	18	1,971	421	0.04	709	0.20	201	0.19	169	0.37	166	0.27
table5	10	49	21	2,145	366	0.03	1062	0.70	143	0.33	119	0.53	119	0.44
vg2	14	58	28	29,690	110	0.06	932	0.14	88	0.16	88	0.33	88	1.97
xor5	4	11	6	16	16	0.01	9	0.01	4	0.01	4	0.01	4	0.01
z5xp1	5	24	11	410	71	0.01	96	0.01	60	0.01	53	0.03	51	0.10
Total				20,085,355	10870	2.26			4455	14.9	3506	49.98		

Table 2. Comparision of prime computation algorithms.

Bench mark	Parameters			Original PLAs					Pair-decoded PLAs			
				CST-based		Runtime, sec			CST-based		Runtime, sec	
	Ins	Outs	Values	ZDD size	Primes	CST	output-1-hot	Espr	ZDD size	Primes	CST	Espr
5xp1	8	10	24	1212	390	0.04	0.01	0.02	1649	482	0.04	0.06
9sym	10	1	19	98	1680	0.01	0.01	0.08	126	264	0.01	0.06
alu4	15	8	36	9551	7145	0.72	0.27	2.47	10598	6150	1.34	2.70
apex2	40	3	81	1135	13403	0.03	0.01	37.01	5205	2217	0.49	3100.0
b12	16	9	39	958	1490	0.03	0.01	0.04	3244	1095	0.08	0.04
clip	10	5	23	533	865	0.01	0.01	0.02	698	376	0.01	0.03
cordic	24	2	48	161	1754	0.01	0.01	0.82	436	225	0.01	6.83
misex2	26	18	68	418	42	0.01	0.01	0.01	715	45	0.01	0.01
misex3	15	14	42	15200	6731	0.45	0.16	1.60	20195	7704	0.52	4.13
rd73	8	3	17	91	211	0.01	0.01	0.01	129	77	0.01	0.01
rd84	9	4	20	143	633	0.01	0.01	0.03	192	149	0.01	0.01
sao2	11	4	24	195	184	0.01	0.01	0.01	285	98	0.01	0.01
t481	17	1	33	58	481	0.01	0.01	0.05	74	32	0.01	0.08
table3	15	14	42	1817	539	0.05	0.02	0.03	2460	625	0.07	0.09
table5	18	15	49	1531	462	0.04	0.01	0.04	2646	431	0.08	0.14
vg2	26	8	58	370	1188	0.02	0.01	0.07	5307	1149	0.35	0.33
z5xp1	8	10	24	1045	390	0.03	0.01	0.02	1527	479	0.03	0.02