Reducing Multi-Valued Algebraic Operations to Binary

ABSTRACT

Algebraic operations have been developed for binary logic synthesis and later were extended to apply to multi-valued (MV) logic. The operations in the MV domain were previously considered more complex and slower. This paper shows that MV algebraic operations are essentially as easy as binary ones, with only slight overhead (linear in expressions) in transformation and inverse transformation.

By introducing *cosingleton sets* as a new basis, any MV sum-of-product expression can be rewritten and parsed to a binary logic synthesizer for fast execution; optimized results can be directly interpreted in the MV domain. This process, called EBD, reduces MV algebraic operations to binary.

A pure MV operation differs its corresponding EBD one mainly in that the former possesses "semi-algebraic" generality, which has not been implemented for binary logic. Experiments show that the proposed methods are significantly faster with little or no loss in quality when run in comparable scripts of sequences of logic synthesis operations.

1. INTRODUCTION

In high-level hardware design, system descriptions are inherently multi-valued. This nature inspires researchers in logic synthesis to pursue optimality from the binary domain to the MV domain. The MVSIS project [2] is an example. Traditionally, logic synthesizers are designed for binary optimization. That is, to minimize MV expressions, they should be first encoded into binary before logic synthesizers can come into play. The disadvantages are two-folded: first, the optimization is restricted to a particular encoding; second, compact structures might be destroyed by the encoding and become more obscure to extract. Moreover, MV optimization problems were considered to be more sophisticated than binary ones; it was not so obvious whether it is affordable to work on the MV domain. In this paper we conquer all of these serious obstacles at once. To achieve this, without resorting to binary encoding, we rewrite MV expressions in terms of *cosingleton sets* while the structures of these expressions are preserved. Thereby, the new representations, which are *binary in disguise*, can be parsed to a binary logic synthesizer. Also, the results directly reflect optimized structures in the MV domain. This approach, called *EBD* (Execution in Binary-in-Disguise), can be significantly faster than previous MV algebraic operations [1, 5, 6].

In [1, 5, 6], semi-algebraic operations for multi-valued input, binary output functions have been implemented in MVSIS for synthesizing multi-valued multi-level networks. The methods are fairly complex and tend to be slow on large examples, but were a significant improvement over previously known methods [8]. On the other hand, their algebraic binary counterparts are very fast. Another MV algebraic method [10] applies to the multi-valued output case. It does the factoring with the MIN and MAX operators instead of OR and AND, and thus is not directly comparable with our formulation.

Semi-algebraic in the binary domain refers to using the Boolean (non-algebraic) identity $xx = x$.[1] Thus it allows the following factorization:

$$abd + aef + bcdf + cef = (cf + a)(bd + ef).$$

Although semi-algebraic methods have been proposed for the binary domain, they have not been implemented. Thus all algebraic operations in the binary domain in this paper are restricted to algebraic and not semi-algebraic. We will use the term algebraic in the MV domain to mean the more general, semi-algebraic[2], and in the binary domain, to mean purely algebraic and not semi-algebraic.

We show how all MV semi-algebraic operations can be mapped into operations on purely binary-input functions using cosingletons to rewrite each multi-valued literal. Then the binary algebraic operations of *extraction*, *factoring*, *decomposition*, and *algebraic division* can be applied. After the result is obtained, the answer is translated back into the original MV domain. This set of binary operations are referred to as *the EBD operations*.

[1] In logic synthesis it is common to refer to algebraic as those algorithms which manipulate expressions like polynomials. Certainly $xx = x$ is algebraic in the Boolean algebraic sense, but we refer to such rules as Boolean instead.

[2] In the MV-domain, the purely algebraic operations have not been defined because it seems that the absorption rule has to be used in this domain. Thus, the generalization to semi-algebraic was made in the MV-domain. While such could be done in the binary domain, use of absorption would significantly slow down the algebraic operations. As it is, the algebraic operations in the binary domain are extremely fast.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DATE '03 Messe Munich, Germany

Copyright 2002 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

We also show, modulo that the EBD operations do not have semi-algebraic equivalents, that the EBD methods have no obvious loss of quality, in that there are results that can be obtained using the multi-valued method, which cannot be obtained by the EBD method and vice versa. Thus, the dual methods are incomparable, but we show that when restricted to purely algebraic methods, they only differ in their use of the flexibility available during the algebraic division process in the MV domain, each representing opposite extremes.

The rest of this paper is organized as follows. In Section 2, we define our representations for MV logic. Section 3 introduces the cosingleton transform with an EBD factorization example. For each algebraic operation, the pure MV method and its corresponding EBD method are compared in Section 4. We then extend the discussion further to non-algebraic operations in Section 5. Experimental results and conclusions are given in Sections 6 and 7 respectively.

2. PRELIMINARIES

As a generalization of Boolean functions, we define

DEFINITION 1. *A* multi-valued function $\mathcal{F}(a, b, \ldots)$ *is a function* $\mathcal{F}: A \times B \times \cdots \mapsto T$, *where* a, b, ... *are* multi-valued variables *taking on values from sets* A, B, ... *respectively, and* T *is the codomain of* $\mathcal{F}$.

In particular, we consider finite sets with size ≥ 2. (This paper uses nature numbers to represent elements of a set. An n-element set has elements $0, 1, \ldots, n-1$.) Like Boolean functions, MV functions can also be expressed in sum-of-product (SOP) forms. For example,

$$\mathcal{F}^{\{\tau\}} = (a^{S_a} b^{S_b} \cdots) + (a^{S'_a} b^{S'_b} \cdots) + \cdots,$$

where $\emptyset \subset S_a, S'_a, \ldots \subseteq A$, $\emptyset \subset S_b, S'_b, \ldots \subseteq B$, ..., and $\tau \in T$.

Also, from set theory, we define

DEFINITION 2. *Given a set* U *with a singleton* $S \subset U$, *the* cosingleton *of* S *is the complement set of* S *with respect to* U.

It is immediate that

PROPOSITION 1. *Given a set* U, *for any subset of* U *there is a unique representation in the form of a conjunction of cosingletons* $\subset U$.

3. THE COSINGLETON TRANSFORM

Consider the following problem. Given an arbitrary multi-valued SOP expression $\mathcal{E}$ and an oracle Ω which optimally factors a binary SOP input, can we take advantage of Ω to factor $\mathcal{E}$? If yes, how?

To achieve this attempt, we have two criteria:

1. $\mathcal{E}'$ must "look like" binary, and
2. $\mathcal{E}''$ must "directly" reflect an optimal factored form of $\mathcal{E}$,

where $\mathcal{E}'$, modified from $\mathcal{E}$, is the input of Ω, and $\mathcal{E}''$ is the resultant output of Ω.

Obviously any binary encoding, which associates a value of an MV variable with a code of a vector of binary variables, satisfies the first criterion. However, none satisfies the second for all possible $\mathcal{E}$. For example, 1-hot codes [3], negative 1-hot codes [4] and all logarithmic codes fails. Historically, there were many instances encoding multi-valued applications into binary. For example, the initial version of Espresso [3] was binary, but it used the idea of treating the multiple outputs of a PLA as a single multi-valued input. These were coded with a 1-hot representation and don't cares were added which stated that two values, on at the same time, was a don't care (plus the all 0s code). Also, in most MDD implementations [7], multi-valued variables are encoded with some logarithmic code and a regular BDD package is used. These are just to name a few.

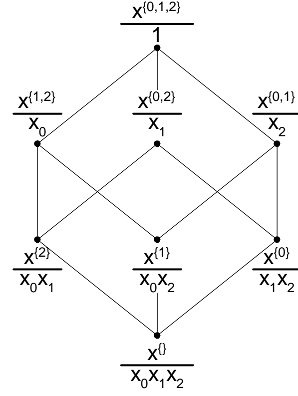

Figure 1: The Hasse diagram of the power set of $X = \{0, 1, 2\}$: For an MV variable x taking on values from X, all of its literals can be expressed by products of cosingletons. The product-of-cosingleton expressions are listed under the corresponding literals.

To proceed, consider the following example. Given an MV expression with two 4-valued variables a and b,

$$a^{\{2,3\}}b^{\{0,1\}} + a^{\{0,3\}}b^{\{1,2\}} + a^{\{1,2\}}b^{\{0,3\}} + a^{\{0,1\}}b^{\{2,3\}},$$

it can be factored as

$$\begin{aligned} = \quad & (a^{\{0,2,3\}}b^{\{0,1,2\}} + a^{\{0,1,2\}}b^{\{0,2,3\}}) \cdot \\ & (a^{\{1,2,3\}}b^{\{0,1,3\}} + a^{\{0,1,3\}}b^{\{1,2,3\}}). \end{aligned}$$

We can observe that, unlike binary SOP representations, multi-valued ones still have disjunctive operations (due to MV literals) in each product term. To eliminate such disjunctive operations, we need to re-express each MV literal in a pure product form. Also we require a bijection between the new expressions and the original MV literals. More importantly, after factorization, we should be able to recover MV literals.

So here is a solution, called *the cosingleton transform.* Let x be a variable taking on values from $X = \{0, \ldots, n-1\}$. In the rest of this paper, we use x_i to denote $x^{\{0,\ldots,i-1,i+1,\ldots,n\}}$ (i.e. the literal with cosingleton set $\{0, \ldots, i-1, i+1, \ldots, n\}$), and use $\overline{x}_i$ to denote $x^{\{i\}}$ (i.e. the literal with singleton set $\{i\}$). From Proposition 1, we know that any literal of x can be uniquely expressed as a product of some cosingleton literals. Figure 1 illustrates the case for $n = 3$. With this convention, the previous example can be rewritten as

$$a_0a_1b_2b_3 + a_1a_2b_0b_3 + a_0a_3b_1b_2 + a_2a_3b_0b_1.$$

With no excuse for rejecting wrong formats, Ω factors this

expression in the binary domain and yields

$$= (a_1b_3 + a_3b_1)(a_0b_2 + a_2b_0).$$

One can check that in this case the EBD factorization gives the same result as the MV factorization. In general, the two factorizations may lead to different but related results as discussed in Section 4. Notice that, even for non-algebraic factorization, we can still transform back to the MV domain since all set operations are legal. More discussions can be found in Section 5.

As one might expect from the previous example, the procedure of the cosingleton transform is as follows. Given an MV SOP expression, for each literal x^S we replace it with $\prod_{i \notin S} x_i$. To perform inverse transform, $\prod_{i \in T} x_i$ are replaced with $x^{\{j|j \notin T\}}$.

So far we have answered the questions raised in the beginning of this section. One might be still curious whether the cosingleton transform is the most compact way.

PROPOSITION 2. *To rewrite all possible n-valued literals as a product form, at least n binary variables should be introduced.*

As the cosingleton transform introduces n "binary variables" in this case, it is optimal. However, for a particular expression, it is possible that one can rewrite an n-valued literal with fewer binary variables than n.

Although the cosingleton transform looks like binary encoding, one should notice their fundamental difference. That is, x_i (or $\overline{x}_i$) is not a binary variable, rather a symbol denoting some cosingleton (or singleton). There is no sense to valuate x_i and $\overline{x}_i$. Accordingly, we cannot treat a transformed representation as a binary implementation for an MV expression.

4. ALGEBRAIC OPERATIONS

4.1 Factorization and Decomposition

One difference between the current MV algebraic operations and the binary ones is the set of divisors used. In the MV algebraic case, only divisors which have no "common cube" are considered. An expression has a common cube if there is a literal of some cube that contains all other literals of that variable appearing in the other cubes of the expression. In that case, the dominating literal can be factored out to obtain a factored expression with no increase in literals. An example which has a common cube is

$$a^{\{0,1,3\}}b^{\{2\}} + a^{\{1,3\}}b^{\{1\}} = a^{\{0,1,3\}}(b^{\{2\}} + a^{\{1,3\}}b^{\{1\}}).$$

In contrast, the notion of being "cube-free" is that for each variable, the supercube (literal) of all literals of that variable appearing in that expression is computed. If the supercube is 1 (the universe literal containing all values for that variable) for all variables appearing in the expression, then the expression is cube-free. Making an expression cube-free by factoring out the supercube literals of an expression is likely to increase the number of literals in the factored form of the expression. For example,

$$a^{\{0,1\}}b^{\{2\}} + a^{\{1,3\}}b^{\{1\}} =$$
$$a^{\{0,1,3\}}b^{\{1,2\}}(a^{\{0,1,2\}}b^{\{0,2,3\}} + a^{\{1,2,3\}}b^{\{0,1,3\}})$$

where the expression in the parenthesis has been made cube-free. Thus in the MV domain the notion of common-cube free replaced the notion of cube-free (used in the binary domain) for algebraic operations[3]. However, making an expression cube-free moves it "nearer" to being prime (which may be good in some sense) since the maximum number of values for each variable is inserted into the expression.

The expression

$$a^{\{0,1\}}b^{\{2\}} + a^{\{1,3\}}b^{\{1\}}$$

has no common cube in the MV domain and so would be a candidate divisor. However, mapping these two cubes into binary using the cosingleton transform, we get

$$a_2a_3b_0b_1b_3 + a_0a_2b_0b_2b_3 = a_2b_0b_3(a_3b_1 + a_0b_2)$$
$$= a^{\{0,1,3\}}b^{\{1,2\}}(a^{\{0,1,2\}}b^{\{0,2,3\}} + a^{\{1,2,3\}}b^{\{0,1,3\}})$$

Thus the corresponding cube-free divisor (in the binary domain, the cube-free notion is used) is

$$a_3b_1 + a_0b_2 = a^{\{0,1,2\}}b^{\{0,2,3\}} + a^{\{1,2,3\}}b^{\{0,1,3\}}$$

On the other hand, in a slightly different example, in the MV domain,

$$a^{\{0,1\}}b^{\{2\}} + a^{\{0,1,3\}}b^{\{1\}} = a^{\{0,1,3\}}(a^{\{0,1\}}b^{\{2\}} + b^{\{1\}})$$

so the associated *common-cube free divisor* is

$$a^{\{0,1\}}b^{\{2\}} + b^{\{1\}}.$$

In the binary case,

$$a_2a_3b_0b_1b_3 + a_2b_0b_2b_3 = a_2b_0b_3(a_3b_1 + b_2)$$
$$= a^{\{0,1,3\}}b^{\{1,2\}}(a^{\{0,1,2\}}b^{\{0,2,3\}} + b^{\{0,1,3\}})$$

so the associated MV *cube-free divisor* is

$$a_3b_1 + b_2 = a^{\{0,1,2\}}b^{\{0,2,3\}} + b^{\{0,1,3\}}.$$

The difference between these two results is that for the cube-free notion associated with the binary case, the divisor has been "lifted" to have the most values possible. Thus since $a^{\{0,1,3\}}$ has been factored out, the value 2 for a can be inserted everywhere in the cube-free factor. Similarly, the values 0 and 3 can be inserted everywhere for b. By doing this lifting, we can transform the common-cube free divisor into the cube-free divisor:

$$a^{\{0,1\}}b^{\{2\}} + b^{\{1\}} \longrightarrow a^{\{0,1,2\}}b^{\{0,2,3\}} + b^{\{0,1,3\}}.$$ [4]

In the MV algebraic case, the divisor values are "lowered" as much as possible. Thus the two methods are incomparable. In general it is not clear which divisor is preferable. In fact, there are other divisors between these two extremes (obtained by inserting **some** of the values allowed); one of these may be preferred, since it may coincide with a divisor of another expression. Thus the most general method would be to find all divisors when looking for common divisors. Unfortunately, doing this appears to be too expensive.

Consider the following example, where the first factorization is done in the MV domain:

$$a^{\{2\}}b^{\{1\}} + a^{\{1\}}b^{\{3\}} + a^{\{0\}}b^{\{1,2\}} + a^{\{0,1\}}b^{\{2\}} =$$
$$(a^{\{0,2\}}b^{\{1,2\}} + a^{\{0,1\}}b^{\{2,3\}})(a^{\{1,2\}}b^{\{1,3\}} + a^{\{0,1\}}b^{\{1,2\}})$$

[3]In the binary domain, *cube-free* and *common-cube free* are equivalent definitions.

[4]To understand that this is cube free, note that literals with all values are suppressed by convention, e.g. $a^{\{0,1,2,3\}} = 1$ is suppressed in this expression.

which translated to the binary domain (and factored further) is

$$= (a_1b_3 + a_2b_1)(a_0b_2 + a_2b_3)a_3b_0$$

Note that this is a semi-algebraic factorization. EBD factorization yields:

$$\begin{aligned} a_0a_1a_3b_0b_2b_3 + a_0a_2a_3b_0b_1b_2 + a_1a_2a_3b_0b_3 + a_2a_3b_0b_1b_3 \\ = (a_2b_3(a_1+b_1) + a_0b_2(a_1b_3+a_2b_1))a_3b_0 \\ = (a^{\{0,1,3\}}b^{\{0,1,2\}}(a^{\{0,2,3\}} + b^{\{0,2,3\}}) + a^{\{1,2,3\}}b^{\{0,1,3\}} \\ (a^{\{0,2,3\}}b^{\{0,1,2\}} + a^{\{0,1,3\}}b^{\{0,2,3\}}))a^{\{0,1,2\}}b^{\{1,2,3\}} \end{aligned}$$

Thus the EBD factorization is different for two reasons. First, no semi-algebraic factorization is done in the EBD method. Second, the EBD result has been lifted to include as many values as possible. The values in the EBD result can be lowered to make it more comparable, by multiplying out the cubes on the outside of the parenthesized expressions (which has the effect of removing values).

It is possible using simple rules to modify one result to be closer to the other. For example, the following rule can be used to convert a binary factorization into one that is more similar to the MV factorization result.

1. In the factorization tree, at a node that is factored as a product of expressions, if a binary literal, x_i, has been factored out and one of the expressions has associated literals x_j everywhere, then multiply x_i into all expressions everywhere but leave terms in any expression with no x_j untouched. Remove x_i as a factor.

2. Otherwise leave x_i as a factor, but in any expression where an x_j occurs, replace it with x_jx_i.

Thus the following expression is transformed,

$$\begin{aligned} (a_1b_1 + b_2)(a_0b_0 + a_2)a_3b_3 \longrightarrow \\ (a_1a_3b_1b_3 + b_2b_3)(a_0a_3b_0b_3 + a_2a_3) \\ = (a^{\{0,2\}}b^{\{0,2\}} + b^{\{0,1\}})(a^{\{1,2\}}b^{\{1,2\}} + a^{\{0,1\}}). \end{aligned}$$

Note that a_3 is not put with b_2 in the first parenthesis since no other a_i occurs there with b_2. Also, note that when a_3 and b_3 are put back, this results in a semi-algebraic factorization (we use $a_3a_3 = a_3$). This is the same result obtained by the MV factorization:

$$= (a^{\{0,2\}}b^{\{0,2\}} + b^{\{0,1\}})(a^{\{1,2\}}b^{\{1,2\}} + a^{\{0,1\}}).$$

As another example,

$$(a_1b_1 + b_2)a_3b_3 \longrightarrow (a_1a_3b_1b_3 + b_2b_3)a_3.$$

using Rule 2 for a_3.

4.2 Algebraic Division

In algebraic division, a divisor, d, and an expression, f are given and we want to obtain a quotient, q, and remainder, r, so that $f = dq + r$. Further, r should have as few cubes as possible. In the MV case, this is called exact algebraic division[5]. Two algorithms were also defined, *matching* and *satisfiability matrix*; the first was defined for the case where the divisor had only two cubes and was much faster.

[5] Another type of division, inexact, was also defined for the MV case, where given d we seek a better result of the form $f = \tilde{d}\tilde{q} + \tilde{r}$ where $d \subseteq \tilde{d}$.

The results obtained by EBD algebraic division and the MV exact division method are comparable but not necessarily identical for the same reasons as for factorization. In both cases the divisor is given. So in the expression $f = dq + r$, f and d are identical in both cases. Thus,

$$f = dq_1 + r_1 = dq_2 + r_2$$

where 1 refers to the EBD result and 2 refers to the MV result. For the reasons discussed for factorization, the two quotients, q_1 and q_2, and two remainders r_1 and r_2, may differ. r_1 and r_2 may be different because the semi-algebraic division may be able to absorb more cubes in the product, and the quotients can differ also because one is maximally raised and the other maximally lowered.

4.3 Common Divisor Extraction

The following is an example of two functions, where a common factor is found in the EBD domain and not in the MV domain. The factorization in the MV domain[6], implies there is no common factor.

$$\begin{aligned} f = a^{\{3\}}b^{\{2,3\}} + a^{\{2,3\}}b^{\{1,3\}} + a^{\{1\}}b^{\{2\}} + a^{\{1,2\}}b^{\{1\}} \\ = (a^{\{1,3\}}b^{\{2,3\}} + b^{\{1,3\}})(a^{\{2,3\}} + a^{\{1,2\}}b^{\{1,2\}}) \\ g = a^{\{0,3\}}b^{\{1,3\}} + a^{\{0,2,3\}}b^{\{2,3\}} + a^{\{0,1\}}b^{\{1\}} + a^{\{0,1,2\}}b^{\{2\}} \\ = (a^{\{0,1,3\}}b^{\{1,3\}} + b^{\{2,3\}})(a^{\{0,2,3\}} + a^{\{0,1,2\}}b^{\{1,2\}}) \end{aligned}$$

However transformed into the binary domain we get

$$\begin{aligned} f = a_0a_1a_2b_0b_1 + a_0a_1b_0b_2 + a_0a_2a_3b_0b_1b_3 + a_0a_3b_0b_2b_3 \\ = (a_3b_3 + a_1)(a_2b_1 + b_2)a_0b_0 \\ g = a_1a_2b_0b_2 + a_1b_0b_1 + a_2a_3b_0b_2b_3 + a_3b_0b_1b_3 \\ = (a_3b_3 + a_1)(a_2b_2 + b_1)b_0 \end{aligned}$$

which has the common factor

$$a_3b_3 + a_1 = a^{\{0,1,2\}}b^{\{0,1,2\}} + a^{\{0,2,3\}}.$$

The expressions,

$$\begin{aligned} f = a^{\{2\}}b^{\{1\}} + a^{\{1\}}b^{\{3\}} + a^{\{0\}}b^{\{1,2\}} + a^{\{0,1\}}b^{\{2\}} \\ = (a^{\{0,2\}}b^{\{1,2\}} + a^{\{0,1\}}b^{\{2,3\}})(a^{\{1,2\}}b^{\{1,3\}} + a^{\{0,1\}}b^{\{1,2\}}) \\ g = (a^{\{0,2\}}b^{\{1,2\}} + a^{\{0,1\}}b^{\{2,3\}})(c^{\{1\}} + d^{\{1\}}) \end{aligned}$$

have a common factor in the MV domain but none using the EBD method. As mentioned, there are examples where both expressions have no common algebraic factors when factored in either the MV domain or the EBD domain, but by selectively adding and deleting values a common factor can be obtained.

Another difference happens when a divisor is extracted and algebraically divided into other expressions. The results can be different between the EBD extraction and the MV extraction e.g. the extraction may result in new nodes $y = k$ and $\tilde{f} = yq + r$ where the EBD q and the MV q may differ.

[6] Actually, our first implementation of factorization in MV-SIS did not factor f and g fully, since it gave:

$$\begin{aligned} f = (a^{\{1,3\}}b^{\{2,3\}} + a^{\{1,2,3\}}b^{\{1,3\}})(a^{\{2,3\}}b^{\{1,2,3\}} + a^{\{1,2\}}b^{\{1,2\}}) \\ g = (a^{\{0,1,3\}}b^{\{1,3\}} + b^{\{2,3\}})(a^{\{0,2,3\}}b^{\{1,2,3\}} + a^{\{0,1,2\}}b^{\{1,2\}}), \end{aligned}$$

and hence did not make the factors common-cube free. Further, the common-cubes, $a^{\{1,2,3\}}$ and $b^{\{1,2,3\}}$ can be eliminated as factors of f and $b^{\{1,2,3\}}$ can be eliminated as a factor of g, saving a total of 3 literals in the factored forms.

5. COMPLEMENTATION, MINIMIZATION, AND NON-ALGEBRAIC OPERATIONS

The interpretation of $\overline{a}_i$ is $a^{\{i\}}$. If we take a binary expression which has come from reduction from a MV expression, the new "binary variables" (cosingletons) occur in only positive form, i.e. the binary expression is positive unate in these variables. Now consider the following expression

$$a_1b_1 + a_2b_2 = a^{\{0,2,3\}}b^{\{0,2,3\}} + a^{\{0,1,3\}}b^{\{0,1,3\}}$$

Using that $\overline{x}_i\overline{x}_j = \emptyset$, the complement of the left-hand side is

$$(\overline{a}_1 + \overline{b}_1)(\overline{a}_2 + \overline{b}_2) = \overline{a}_1\overline{b}_2 + \overline{b}_1\overline{a}_2$$

which translates into

$$a^{\{1\}}b^{\{2\}} + a^{\{2\}}b^{\{1\}}.$$

The complement of the right-hand side is

$$(a^{\{1\}} + b^{\{1\}})(a^{\{2\}} + b^{\{2\}}) = a^{\{1\}}b^{\{2\}} + a^{\{2\}b^{\{1\}}}.$$

Thus we see that we get equal results, in this case, but in general this is not always the case.

EBD manipulations, like simplification, yield equivalent results to the MV versions, but EBD results may not be prime and irredundant. For example,

$$x^{\{1,2\}}y^{\{0\}} + x^{\{1\}} + x^{\{0,1\}}y^{\{1\}} = x_0y_1 + x_0x_2 + x_2y_0$$

is irredundant in the binary domain, but is redundant when translated to the MV domain. Also, being prime does not translate between domains, because in the binary domain, the relations $\overline{x}_i\overline{x}_j = \emptyset$ are unknown, since x_i and x_j are treated as independent variables in the binary domain. Thus in expanding a cube until it just meets the offset in order to generate a prime, we might not expand far enough in the binary domain because there are additional (unknown) cases where intersections are empty.

If the relation, $\overline{x}_i = x_0x_1 \ldots x_{i-1}x_{i+1} \ldots x_{n-1}$ is used, then every expression is positive unate in the new binary variables. But again, the relation $x_0x_1 \ldots x_{n-1} = \emptyset$ is unknown. Of course, this information could be provided by giving

$$\sum(\overline{x}_i\overline{x}_j)$$

as don't cares, but then the minimization process becomes more cumbersome[7]

In general, single cube containment is equivalent between both domains, i.e. $C^i \subseteq C^j$ if and only if $B(C^i) \subseteq B(C^j)$, where $B(.)$ converts an MV cube to a cosingleton-transformed one.

Although not minimal, the EBD Boolean operations may be useful since the EBD result could be improved by a single step of making cubes prime. The result is already minimal with respect to single cube containment. Optionally further redundancy can be removed as well. The EBD operation may be faster since in the binary domain the `nocomp` option, which uses the concept of the reduced offset [9], is available. Thus EBD Boolean operations give a type of sub-optimal MV operations.

[7]This might be useful and practical, but some experimentation needs to be done here.

6. EXPERIMENTAL RESULTS

These ideas have been implemented in MVSIS and the MV algebraic methods compared with the EBD algebraic methods in the setting of optimizing multi-valued multi-level networks. For the benchmarks tested, an identical sequence of operations[8] was applied, except in the EBD version, called EBD_script, all MV algebraic operations of MV_script were replaced with their EBD counterparts. Each EBD algebraic operation consists of

1. converting the MV expression(s) to their transformed "binary" counterpart,
2. applying the binary algebraic operation, and
3. converting the result back to an MV expression.

The results are shown in Table 1 where EBD refers to running EBD_script and MV refers to running MV_script. Time is in seconds and `lits-ff` refers to the total number of literals in the factored forms of the MV expressions in a circuit. The last set of examples are multi-valued.

As expected, the results show that quality is essentially maintained by the EBD operations, but the speed is greatly increased for the larger examples. In some benchmarks, only a slight loss of quality occurs in using the EBD operations. This is possibly due to the lack of semi-algebraic operations in the binary implementation. Also, `EBD_fx` extracts common cubes and some of these are of the type $\prod_i x_i$ which in the MV-domain is just a literal and of no value as a separate node in the network.

7. CONCLUSIONS

The idea of the cosingleton transform puts most MV operations (which include EBD ones) now on an equal footing with the binary ones in terms of speed and quality of results, and so is a large step in achieving our goal, to make MVSIS the system of choice for optimizing multi-level networks, whether the network is MV or binary. In addition, MVSIS includes a number of newer ideas that are not in SIS, so that the quality of the results already exceeds those obtained in SIS. We have also observed that the ability to enter and leave the MV domain during an optimization run, allows more freedom in finding better optimizations and encourages new ideas.

We have not experimented with the MV Boolean operations, mentioned in Section 5, since the binary ones lead to sub-optimal results. In addition, the MV Boolean operations are almost as fast. The exception is that there is no code for the MV counterpart of the reduced offset in `Espresso-MV` even though the theory exists [9]. The reduced offset helps in those cases where the complement of a large function is needed in order to expand to primes during a two-level SOP optimization. During node minimization, it is common to derive a large don't care set, and a subsequent call to `Espresso-MV` requires complementation of the onset plus don't care set. However, we intend to experiment with using EBD node-minimization as a technique for trading lower quality for increased speed.

[8]The basic script used was an improved multi-valued version of `script.rugged` used in SIS.

circuit	EBD time	EBD lits-ff	MV time	MV lits-ff
vg2	2.9	87	2.6	85
sse	2.1	128	2.2	120
b12	2.4	70	2.3	70
cht	1.8	163	1.9	164
sqrt8	1.1	67	1.2	56
clip	5.3	134	7.6	129
duke2	10.7	497	24.6	488
sand	23.6	545	47.5	525
f51m	1.8	108	2.4	97
sao2	2.4	109	4	110
term1	5.2	147	6.2	142
9sym	3	72	4.6	120
alu2	12.5	266	19.4	278
sct	1.9	83	2	90
t481	14.2	36	63.9	40
ttt2	3.1	233	4.4	221
bw	3.2	194	4.4	194
rd84	5.3	87	9.5	106
squar5	1.4	58	1.4	58
z4ml	1.2	38	1.4	38
C432	46.3	185	49.3	195
planet	24.7	605	63.5	611
vda	32.3	763	96	777
cps	93.3	1479	364.1	1524
dk16	7.0	248	9.3	238
S953	18.6	510	29.6	516
k2	269.2	1426	3351	1428
balance	8.1	182	84.1	217
conv35cc	1.2	83	1	72
employ1	1.7	42	1.5	36
mm3	0.9	23	0.8	23
mm5	5.3	137	8.5	130
pal3x	4	114	4.5	100
aluack	1.4	91	1.4	76
iris	1.3	12	1.3	12
mm4	2	75	2.3	60
monks2tr	1.2	51	1.2	43
monksltr	0.9	7	0.8	7
sleep	36.6	33	63.7	37
car	1.2	43	1.3	44

Table 1: Comparison of EBD and MV scripts

8. REFERENCES

[1] R. K. Brayton. Algebraic methods for multi-valued logic. Technical Report UCB/ERL M99/62, Electronics Research Laboratory, University of California, Berkeley, Dec. 1999.

[2] R. K. Brayton and et al. MVSIS. `http://www-cad.eecs.berkeley.edu/Respep/Research/mvsis/`.

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

[4] S. Devadas and A. R. Newton. Exact Algorithms for Output Encoding, State Assignment, and Four-Level Boolean Minimization. *IEEE Trans. Comput.-Aided Design Integrated Circuits*, 10(1):13–27, Jan. 1991.

[5] M. Gao and R. K. Brayton. Semi-algebraic methods for multi-valued logic. In *Proc. of the Intl. Workshop on Logic Synthesis*, May. 2000.

[6] M. Gao and R. K. Brayton. Multi-valued multi-level network decomposition. In *Proc. of the Intl. Workshop on Logic Synthesis*, June 2001.

[7] T. Kam and R. K. Brayton. Multi-valued decision diagrams. Technical Report UCB/ERL M90/125, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Dec. 1990.

[8] L. Lavagno, S. Malik, R. Brayton, and A. Sangiovanni-Vincentelli. MIS-MV: Optimization of multi-level logic with multiple-valued inputs. In *Proceedings of the International Conference on Computer-Aided Design*, 1990.

[9] A. Malik, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. A Modified Approach to Two-level Logic Minimization. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 106–109, Nov. 1988.

[10] H. M. Wang, C. L. Lee, and J. E. Chen. Algebraic division for multi-level logic synthesis of multi-valued circuits. In *International Symposium on Multiple-Valued Logic*, 1994.