Reduced Offsets for Two-Level Multi-Valued Logic Minimization *

Abdul A. Malik †
IBM T. J. Watson Research Center
Yorktown Heights, N.Y 10598

Robert K. Brayton *A. Richard Newton* *Alberto L. Sangiovanni-Vincentelli*
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Abstract

The approaches to two-level logic minimization can be classified into two groups: those that use tautology for expansion of cubes and those that use the offset. Tautology based schemes are generally slower and often give somewhat inferior results, because of a limited global picture of the way in which the cube can be expanded. If the offset is used, usually the expansion can be done quickly and in a more global way because it is easier to see effective directions of expansion. The problem with this approach is that there are many functions that have a reasonable size onset and don't care set but the offset is unreasonably large. It was recently shown that for the minimization of such Boolean functions, a new approach using reduced offsets, provides the same global picture and can be computed much faster. In this paper we extend reduced offsets to logic functions with multi-valued inputs.

1 Introduction

A two-level logic minimization problem is generally posed as the minimization of a logic function given a sum-of-products cover of the onset and a representation of the don't care set. The objective of minimization is primarily to decrease the total number of cubes (product terms) and secondarily the total number of literals in the cover. A logic function can be a function of binary variables or multi-valued variables. The latter is called a mv-function for simplicity. We are primarily concerned with mv-functions in this paper.

In minimizing a two-level function, the number of literals in each cube is reduced (expansion), either to obtain a cube with the minimum possible literals or a cube that contains as many other cubes in the cover as possible. The expanded cube is then added to the cover and those cubes contained in the expanded cube are removed. Expansion can be done in two ways. Let g be the union of the onset and don't care set. The expansion for a cube is valid if and only if the expanded cube is covered by g. The first method uses tautology; the test if an expanded cube is covered by g is converted to a tautology test. This method is used by PRESTO [2]. Testing if a function is a tautology takes exponential time in the worst case. Also, several tautology tests may be necessary for each cube. As a result, this method is usually slow.

The other method uses the fact that if the expanded cube does not intersect the offset $\overline{g}$ then it is covered by g. Although, the computation of the offset is also expensive in the worst case, this needs to be done only once for the entire minimization. Further, using the offset gives a more global picture of the expansion space. Using the offset, it is easy to find the literals that can be removed without affecting any other literals, or the literals that must be retained in any expanded cube. The offset is used by ESPRESSO [1] and MINI [4].

The problem with using the offset is that there are mv-functions which have reasonable size onsets and don't care sets but whose offsets are unreasonably large. One such example can be formed from the binary-valued Achilles' heel function by converting each input variable to a mv-variable. A mv-Achilles' heel function with n cubes is shown below:

$$f = X_1^{\{1,2\}} X_2^{\{0,2\}} X_3^{\{0,1\}} + \cdots + X_{3i-2}^{\{1,2\}} X_{3i-1}^{\{0,2\}} X_{3i}^{\{0,1\}} + \cdots + X_{3n-2}^{\{1,2\}} X_{3n-1}^{\{0,2\}} X_{3n}^{\{0,1\}}$$

where each variable is a 3-valued variable. The don't care set for this function is $d = \emptyset$. It can be shown that the minimum representation for the offset function has 3^n cubes.

Reduced offsets were found to be very effective for minimization of binary valued functions with large offsets [6]. The development of the theorey of reduced offsets was motivated by applications to the multi-level logic minimization problem where two-level binray-valued functions with large offsets are encountered. Minimization of two-level mv-functions is important for several reasons. Binary-valued functions are a special case of mv-functions. Therefore, the study of multi-valued minimization provides a broader picture of binary-valued minimization. The problem of minimizing a multiple output binary-valued function can be treated as that of multi-valued minimization where all the outputs are considered as a single multi-valued variable [9]. This approach is used by ESPRESSO-MV [1, 8]. Two-level mv-minimization has been found useful for the optimal encoding of states of a Finite State Machine [3]. More recently, applications in the mv multi-level domain have appeared [5, 7].

The importance of mv-minimization in logic synthesis and the deficiency of offset-based approach in minimizing mv-functions with large offsets provides the motivation for extension of the theory of reduced offset to the mv domain. The minimum representation of the reduced offset is never larger than that of the offset; yet it can be used in the same way as the offset during cube expansion. Consequently, the quality of minimization is maintained. Efficient algorithms

*This project was supported in part by National Science Foundation under contract number UCB-BSI6421 and Defense Advanced Research Projects Agency under contract number N00039-87-C-0182.

† This work was carried out when the author was a doctoral candidate at University of California at Berkeley.

for computing the reduced offset have been developed. The key ingredient in the algorithms is that the offset of the mv-function is never computed.

In section 2 we present some definitions. In Section 3 we define the reduction operator for mv-functions. In section 4, a unate recursive algorithm is given for computing the reduced offset for mv-functions. Finally in sections 5 and 6, efficient algorithm for computing the overexpanded cube is presented and it is applied to obtain an efficient algorithm for computing the reduced offset. In Section 8 some experimental results are given and in section 9 states some conclusions are stated.

2 Definitions and Terminology

In this section we provide the basic definitions and notations related to mv-functions to be used in this paper. They are similar to those used by Sasao [9] and in [1, 8].

A *mv-function* with n variables is defined as a mapping

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

where $P_i = \{0, 1, \ldots, p_i - 1\}$ and p_i is the number of values that the i^{th} variable may take on. $B = \{0, 1, *\}$. The elements in the domain of the function are called minterms. The domain is partitioned into the onset, offset and the don't care set. The set of all minterms that map to 0, 1, $*$ are called the offset, onset and the don't care set respectively. Any minterm in the don't care set is allowed to have a value 0 or 1.

Let $X_1, X_2, \ldots, X_n$ be the n variables of the function. A *product term* of the function is defined as

$$X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$$

where $S_i \subseteq P_i$ for $1 \leq i \leq n$. The product term is said to contain all minterms in $S_1 \times S_2 \times \cdots \times S_n$.

$X_i^{S_i}$ is called a *literal*. If $S_i = P_i$ then the literal is called a *full literal*. If a variable does not appear in a cube, it is considered to have a full literal and the cube is said not to depend on the variable. If $S_i = \emptyset$ then the literal is called an *empty literal*. If a cube has an empty literal, the cube is called a *null cube* because it does not contain any minterms. A literal $X_i^{S_i}$ is said to be orthogonal to another literal $X_i^{T_i}$ if $S_i \cap T_i = \emptyset$. If two literals of the same variable are not orthogonal, they intersect.

Let $G = X_1^{S_1} X_2^{S_2} \cdots X_n^{S_n}$ and $H = X_1^{T_1} X_2^{T_2} \cdots X_n^{T_n}$ be two cubes. G *contains* H (written as $H \subseteq G$) if $T_i \subseteq S_i$ for $1 \leq i \leq n$. If G contains H then cube G contains all minterms of cube H. If cube G contains all minterms of cube H and cube H is *not* a null cube then $H \subset G$. However, if cube H is a null cube then it is possible that for some i between 1 and n, $T_i \not\subseteq S_i$. In that case, $H \not\subseteq G$. The *complement* of G is obtained by De Morgan Law and is shown below

$$\overline{G} = X_1^{\overline{S}_1} + X_2^{\overline{S}_2} + \cdots + X_n^{\overline{S}_n}$$

where $\overline{S}_i = P_i - S_i$

The *product* or *intersection* of G and H is given by

$$X_1^{S_1 \cap T_1} X_2^{S_2 \cap T_2} \cdots X_n^{S_n \cap T_n}.$$

If the product is a null cube then H and G are *orthogonal*. Otherwise, they *intersect*.

Let $c_0, c_1, \cdots, c_{l-1}$ be product terms in the domain of a completely specified mv-function f such that $\sum_{i=0}^{l-1} c_i = 1$ (i.e tautology) and $c_j c_k = \emptyset$ for $j \neq k$. Then

$$f = \sum_{i=0}^{l-1} c_j f_{c_j}$$

This decomposition is known as the *generalized Shannon expansion*.

A sum-of-products form is *weakly-unate* in variable X_i if there exists a $j \in P_i$ such that for every cube in the cover, the literal corresponding to variable X_i is either a full literal or it does not contain value j. A sum-of-products is weakly unate if it is weakly unate in all variables.

A completely specified mv-function f is *strongly unate* in variable X_i if the elements of P_i can be totally ordered ($\preceq$) such that changing the value of variable X_i from value j to some value k with $j \preceq k$, causes the function value to change from 0 to 1 if it changes at all. If f is strongly unate in all of its variables then f is a strongly unate function.

If f is strongly unate in X_i then it is also weakly unate in X_i but the converse is not true.

3 Reduction operator R_p and Reduced Offset

To facilitate the computation of the reduced offset, the notion of the reduction operator is introduced.

Definition 3.1 *Let g be a sum-of-products and p be a cube. $R_p(g)$ removes every literal from g which is not orthogonal to some literal in p. R_p is called the reduction operator.*

Definition 3.2 *If g is the offset of a function and p is a cube, then $R_p(g)$ is the reduced offset of g for p.*

The removal of the literals mentioned in Definition 3.1 amounts to converting them to full literals which has the effect of enlarging cubes containing them. The larger cubes usually subsume smaller cubes which can then be dropped. This is why the minimum representation of the reduced offset is never larger than that of the offset. In fact, it is usually much smaller.

Definition 3.3 *Let p and c be two cubes. $|R_p(c)|$ is the number of (non-full) literals in $R_p(c)$.*

The usefulness of the offset r in expanding a cube p stems from the fact that it is very easy to check whether a given expansion p' of p is valid. The test is based on the fact that p' is a valid expansion if and only if p' is orthogonal to r. The ease in testing for orthogonality is due to the use of a sum-of-products representation for r. The reduced offset can be used for expansion of cubes in the same way as the offset because it is also provided in the sum-of-products form and the following theorem holds:

Theorem 3.1 *Let $\mathcal{F} = (f, d, r)$ be an incompletely specified mv-function. Let p be a cube and r_p be the reduced offset for it. Let p' be a non-null cube such that $p \subseteq p'$. Then p' is orthogonal to r iff it is orthogonal to r_p.*

4 Unate Recursive Algorithm for the Reduced Offset for a Cube

The reduced offset as defined earlier is produced by the reduction operator acting on the offset. However, this definition is not useful to compute the reduced offset because it requires knowing the offset. The algorithm presented here does not require computation of the offset. The algorithm for computing the reduced offset given here is based on the unate recursive paradigm.

The unate recursive paradigm has been used successfully for several logic operations including complementation of a function [1, 8]. Unate sums-of-products have many nice properties that make operating on them easier than on non-unate sums-of-products. In a unate recursive paradigm, a non-unate cover or sum-of-products is broken down into its unate cofactors using the generalized Shannon expansion recursively. The operation is then applied to the unate cofactors rather than the original sum-of-products and the results are merged together. Obtaining the Shannon expansion takes exponential time in the worst case. Therefore the worst case time complexity of the algorithms based on Unate Recursive Paradigm is exponential in the number of variables. However, the experience with ESPRESSO shows that such algorithms work well in practice.

There are two notions of unateness for mv-functions: strongly unate and weakly unate. In this section, the definition for strongly unate sum-of-products will be presented first. Following that, a unate recursive algorithm for the reduced offset will be developed. The algorithm will use the strongly unate sums-of-products.

Definition 4.1 *Let U be a sum-of-products and X_k be a variable in U. Let $W_k = \{S \subseteq P_i | X_k^S$ is a literal in $U\}$. If the elements of W_k can be completely ordered via $\subseteq$ then U is strongly unate in X_k. If U is strongly unate in every variable then U is a strongly unate sum-of-products.*

The following theorems facilitate getting strongly unate sum-of-products representation for strongly unate functions.

Theorem 4.1 *Let U be a strongly unate sum-of-products. Then U represents a strongly unate function.*

Theorem 4.2 *Let g be a sum-of-products representation for a strongly unate function. Then there exists a strongly unate sum-of-products U that represents the same function. Also, U can be obtained from g.*

4.1 Recursive Shannon Cofactoring

Let p be a cube and $\mathcal{F} = (f, d, r)$ be a function. Let the reduced offset for p be denoted by r_p. By definition $r_p = R_p(\overline{g})$ where $g = f \cup d$. From the generalized Shannon expansion defined in section 2,

$$\overline{g} = L\overline{g}_L + R\overline{g}_R$$

where $L = X_i^{S_l}$, $R = X_i^{S_r}$, X_i is a non-strongly unate variable in g, $S_l \cup S_r = P_i$ so that $L + R = 1$, and $S_l \cap S_r = \emptyset$. The complementation and cofactoring operations commute. Hence,

$$\overline{g} = L\overline{g_L} + R\overline{g_R}$$

Applying the reduction operator on both sides of the equation, we get

$$r_p = R_p(\overline{g}) = R_p(L\overline{g_L}) + R_p(R\overline{g_R})$$

To proceed further, the following theorem is required.

Theorem 4.3 *Let g be a sum-of-products and c be a cube. Then $R_p(c\overline{g_c}) = R_p(c)R_p(\overline{g_c})$.*

Therefore,

$$r_p = R_p(\overline{g}) = R_p(L)R_p(\overline{g_L}) + R_p(R)R_p(\overline{g_R})$$

If either g_R or g_L is not a strongly unate sum-of-products, $R_p(\overline{g}_L)$ or $R_p(\overline{g}_R)$ can be obtained by the recursive application of the above equation.

During recursive application of the above equation, g is not $f \cup d$ after the first time it is cofactored. Instead, it is the result of cofactoring $f \cup d$ with respect to some cube c. If L and R are single literal cubes of variable X_i and c has a literal $M = X_i^{S_m}$ then it is not necessary that $L + R = 1$. All that is required is that $L + R = M$. This is explained in [8].

If g_R or g_L is strongly unate, it is possible to obtain $R_p(\overline{g}_R)$ or $R_p(\overline{g}_L)$ as described below:

4.2 Applying the Reduction Operator to Strongly Unate Sums-of-Products

Theorem 4.4 states how $R_p(\overline{U})$ can be computed for a strongly unate sum-of-products U, without first computing $\overline{U}$.

Theorem 4.4 *Let U be a strongly unate sum-of-products and p be a cube. Let V be obtained from U by removing those cubes that don't contain p. Then*

$$R_p(\overline{U}) = \overline{V}$$

Theorem 4.4 will not hold if U is not a strongly unate sum-of-products even if it represents a strongly unate function. It will also not hold if U is weakly but not strongly unate. This affirms the need to obtain strongly-unate sums-of-products for application of Theorem 4.4.

4.3 Merging

It was mentioned above that $R_p(\overline{g})$ can be obtained by the following equation if g is not a strongly unate sum-of-products:

$$R_p(\overline{g}) = R_p(L)R_p(\overline{g_L}) + R_p(R)R_p(\overline{g_R})$$

where $L = X_i^{S_l}$ and $R = X_i^{S_r}$ for some non-strongly unate variable X_i.

Once $R_p(\overline{g_L})$ and $R_p(\overline{g_R})$ are obtained, $R_p(\overline{g})$ can be obtained by the above equation. If there are n cubes in $R_p(\overline{g_L})$ and m in $R_p(\overline{g_R})$ then this simple approach will give $R_p(\overline{g})$ with $n + m$ cubes. However, it may be possible to combine some cubes in $R_p(L)R_p(\overline{g_L})$ with some in $R_p(R)R_p(\overline{g_R})$ to obtain a smaller representation for $R_p(\overline{g})$. The basic idea is as follows:

1. Suppose $R_p(L) = 1$. Then $R_p(L)R_p(\overline{g}_L) = R_p(\overline{g}_L)$. Let q be a cube in $R_p(R)R_p(\overline{g}_R)$. Then it is possible that q is contained within some cube in $R_p(L)R_p(\overline{g}_L)$. In that case, q can be dropped. In fact, q can be dropped even if it is contained in $R_p(L)R_p(\overline{g}_L)$ but not necessarily in a single cube in $R_p(L)R_p(\overline{g}_L)$. However, testing for containment in $R_p(L)R_p(\overline{g}_L)$ is usually very expensive and is therefore not done. Similarly, cubes can be dropped from $R_p(\overline{g}_L)$ if $R_p(R) = 1$.

Algorithm 1

Input: A cofactor h of $g = f \cup d$ and a mv-cube p.
Output: $R_p(\overline{h})$.

```
compute_rosl(h, p) {
  If (h is a strongly unate sum-of-products) {
    Obtain V by removing those cubes from h
    that don't contain p.
    Return(V)
  } Else {
    Choose a non-strongly unate variable X in h.
    Let M = X^S be a literal in c.
    Choose R = X^{S'} and L = X^{S''} such that
    R + L = M.
    Obtain cofactors h_L and h_R.
    t_L = compute_rosl(h_L, p)
    t_R = compute_rosl(h_R, p)
    t = merge(p, X, L, R, t_L, t_R)
    Return(t)
  }
}
```

2. Let c_1 be a cube in $R_p(L)R_p(\overline{g}_L)$ and c_2 be in $R_p(R)R_p(\overline{g}_R)$ such that they differ only in variable X_i. Let c_1 have literal $X_i^{S_1}$ and c_2 have literal $X_i^{S_2}$. c_1 and c_2 can be merged into c with literal $X_i^{S_1 \cup S_2}$ with all other literals in c the same as in c_1 or c_2.

Subroutine *compute_ros1*() shown in Algorithm 1 contains pseudo-code for computing the reduced offset. It makes use of subroutine *merge*() which is shown in Algorithm 2.

A problem with Algorithm 1 is that every unate cofactor of $g = f \cup d$ is used in *compute_ros1*() which may be very time consuming if g has a very large number of unate cofactors. The algorithm can be improved with the use of the overexpanded cube so that many cofactors need not even be computed.

5 Overexpanded Cube

The overexpanded cube q_p of a mv-cube p is the smallest cube that contains all valid expansions of p. The overexpanded cube is of interest because it will be used to improve Algorithm 1. Two algorithms for the overexpanded cube are well known. One of them uses the offset which is of no use here. The other uses tautology. A new algorithm was discovered during the course of this research which is more effecient than the one based on tautology. The key to this algorithm is in the relationship between the overexpanded cube and the reduced offset.

Theorem 5.1 *Let p be a non-null mv-cube and q_p be its overexpanded cube. Let r_p be the reduced offset for p such that each variable has at most one single literal cube in r_p. Let r'_p be the sum of single literal cubes in r_p. Then*

$$q_p = \overline{r'_p}$$

Theorem 5.1 reduces the problem of finding the overexpanded cube to that of identifying the single literal cubes in

Algorithm 2

Input: Cube p, splitting variable X, partitioning cubes L and R, $t_L = R_p(\overline{g}_L)$ and $t_R = R_p(\overline{g}_R)$.
Output: $r_p = R_p(\overline{g})$.

```
merge(p, X, L, R, t_1, t_2) {
  t_L = R_p(L)t_L
  t_R = R_p(R)t_R
  For each cube q_l in t_L {
    For each cube q_r in t_R {
      if (q_l and q_r differ only in X) {
        Let X^{S'} and X^{S''} be literals of X in q_l
        and q_r respectively.
        Make X^{S' ∪ S''} the literal of X in q_l and
        remove q_r from t_R.
      }
    }
  }
  If (R_p(L) = 1)
    Remove every cube from t_R that is contained
    in a cube in t_L.
  If (R_p(R) = 1)
    Remove every cube from t_L that is contained
    in a cube in t_R.
  Return(t_L + t_R)
}
```

the reduced offset. However, the single literal cubes must be deduced without computing the reduced offset.

To see how the single literal cubes come about in the reduced offset, consider Algorithm 1 described in section 4 for the reduced offset. The recursive application of Shannon expansion in the algorithm amounts to decomposing $g = f \cup d$ into strongly unate cofactors such that

$$g = \sum c_i U_i$$

where U_i is a strongly unate sum-of-products which is a cofactor of g with respect to cube c_i, $c_i c_j = \emptyset$ if $i \neq j$ and $\sum c_i = 1$. It is known from the unate recursive algorithm for complementation [1, 8] that

$$\overline{g} = \sum c_i \overline{U}_i$$

For a given mv-cube p, applying the reduction operator R_p on both sides gives

$$R_p(\overline{g}) = \sum R_p(c_i) R_p(\overline{U}_i)$$

From Theorem 4.4 $R_p(\overline{U}_i) = \overline{V}$. Therefore

$$r_p = R_p(\overline{g}) = \sum R_p(c_i) \overline{V}_i$$

It is clear from the above equation that r_p has a single literal cube c if and only if for some i, $R_p(c_i)\overline{V}_i$ has the single literal cube c. This can happen in one of the following three ways:

1. $R_p(c_i) = 1$ and c is present in $\overline{V}_i$.
2. $R_p(c_i) = c$ and $\overline{V}_i = 1$.
3. $R_p(c_i) = a$ and b is present $\overline{V}_i$ such that a, b and c are all single literal cubes in the same variable and $c = a\,b$.

Algorithm 3

Input: p, overexpanded cube q_p so far, cofactor h of $g = f \cup d$, and cofactoring cube c such that $h = g_c$.
Output: Updated overexpanded cube q_p for p.

```
find_oc1(p, q_p, h, c) {
    If (h is a strongly unate sum-of-products) {
        find_oc_unate(p, c, q_p, h)
    } Else { /* h is not strongly unate */
        Choose a non-strongly unate variable X in h.
        Form single literal partitioning cubes R and L
        of variable X.
        If (|R_p(c R)| ≤ 1) find_oc(p, q, c R, h_R)
        If (|R_p(c L)| ≤ 1) find_oc(p, q, c L, h_L)
    }
}

find_oc_unate(p, c, q_p, h) {
    Form V from h by removing those cubes that
    don't contain p.
    If (|R_p(c)| = 0) {
        For each variable X_j {
            X_j^{S_j} = largest single literal cube of X_j in V̄.
            q_p = q_p * X_j^{S_j}
        }
    } Else { /* |R_p(c)| = 1 */
            X_j^{S_j^{c_i}} = R_p(c_i)
            X_j^{S_j} = largest single literal cube of X_j in V̄.
            q_p = q_p * (X_j^{S_j^{c_i}} + X_j^{S_j})
    }
}
```

Using the above conditions to detect single literal cubes in r_p will require looking at only those cofactors of g for which $R_p(c_i)$ is either 1 or a single literal cube. $R_p(c_i) = 1$ if and only if $|R_p(c_i)| = 0$. $R_p(c_i)$ is a single literal cube if and only if $|R_p(c_i)| = 1$. The single literal cube in that case is $\overline{R_p(c_i)}$. In order to use the above conditions, it is necessary to have an efficient way to test whether $\overline{V}_i$ is a single literal cube. This test can certainly be made by looking at $\overline{V}_i$. However, the following theorem provides a method to make this test that does not require complementing V_i.

Theorem 5.2 *Let* $V = \sum_{i=1}^{m} p_i$ *where* $p_i = X_1^{S_1^{p_i}} X_2^{S_2^{p_i}} \cdots X_n^{S_n^{p_i}}$. *Let* $S_j = \bigcup_{i=0}^{m} S_j^{p_i}$. *Then* $X_j^{\overline{S_j}}$ *is the largest single literal cube of variable* X_j *in* $\overline{V}$.

Subroutine $find\_oc1()$ shown in Algorithm 3 computes the overexpanded cube. It is a recursive algorithm. For the first call to $find\_oc1()$, $q_p = 1$, $c = 1$ and $h = g = f \cup d$.

Algorithm 4

Input: $g = f \cup d$ and a mv-cube $p \subseteq g$.
Output: The reduced offset r_p for p.

```
find_ros1(g, p) {
    q_p = 1
    find_oc1(p, q_p, g, 1)
    If (p = q_p) {
        Return(q̄_p)
    } Else {
        Obtain cofactor g_{q_p} of g with respect to q_p.
        Return(compute_ros1(g_{q_p}, 1, p))
    }
}
```

6 Using the Overexpanded Cube to find the Reduced Offset

The relation between the overexpanded cube and the reduced offset is also the key to speeding up the computation of the reduced offset. It follows from Theorem 5.1 that

$$r_p = \overline{q_p} + t_p$$

where t_p consists of cubes which have more than one literal each.

Any cube c in t_p can be dropped if it is contained in $\overline{q_p}$ or equivalently if it is orthogonal to q_p. Therefore t_p *consists of cubes in* r_p *that are not orthogonal to* q_p. The following Theorem is helpful in eliminating such cubes.

Theorem 6.1 *Let p be a cube, q_p be its overexpanded cube, and r_p be the reduced offset for it. Let $g = f \cup d$. Then*

$$r_p = \overline{q_p} + R_p(\overline{g_{q_p}})$$

Once the overexpanded cube q_p is known, all that is needed to obtain r_p is $R_p(\overline{g}_{q_p})$. As a result of cofactoring g with respect to q_p, some product terms may drop out and some literals may become full literals. Consequently, it is usually much faster to compute $R_p(\overline{g}_{q_p})$ than computing $R_p(\overline{g})$. In the special case where the cube p is a prime cube, the computation of the reduced offset can be greatly simplified:

Theorem 6.2 *Let p be a cube in the onset f of an incompletely specified function $\mathcal{F} = (f, d, r)$ and q_p be its overexpanded cube. If $p = q_p$ then p is a prime cube and its reduced offset $r_p = \overline{q_p}$.*

Subroutine $find\_ros1()$ shown in Algorithm 4 can now be used to obtain the reduced offset. $find\_ros1()$ makes use of $compute\_ros1()$ in Algorithm 1.

7 Storing the Cofactoring Tree

If the first call to subroutine $compute\_ros1()$ in Algorithm 1 is made with $h = g = f \cup d$, rather than with $h = g_{q_p}$ as may be done from subroutine $find\_ros1()$, then the process of recursive Shannon cofactoring will be the same regardless to what p is. This is because no information specific to p is used until unate cofactors are reached. This makes it

Algorithm 5

Input: root of the cofactor tree and a mv-cube $p \subseteq f \cup d$.
Output: The reduced offset r_p *for* p.

```
find_ros2(root, p) {
    q_p = 1
    find_oc2(root, q_p, 1, g)
    If (p = q_p) {
        Return(q_p)
    } Else {
        Return(compute_ros2(root, p, q_p))
    }
}
```

possible to do the recursive Shannon cofactoring only once and reuse it for computing any reduced offsets. Subroutine $find\_oc1()$ shown in Algorithm 3 can be modified in the same way. Since recursive Shannon cofactoring is only done once, this results in some saving of CPU time.

A rooted binary tree called the cofactor tree is used to store the recursive Shannon cofactoring. The root of the tree corresponds to $g = f \cup d$. Each node in the tree corresponds to a cofactor h of g with respect to some cofactoring cube c. Each node has two children. The left child represents cofactor h_L of h with respect to L. Similarly, the right child represents cofactor h_R of h with respect to R. L and R are single literal cubes in the same variable X such that if M is the literal of X in c, then $L + R = M$. The cofactoring cubes corresponding to the left and right children are $c\,L$ and $c\,R$ respectively. A parameter called level is associated with each node. The level of the root node is 1. The level of any other node is one more than that of its parent. A parameter max_level is used to control the size of the tree. If a node whose level is the max-level is encountered then it becomes a leaf node. If a node represents a strongly unate sum-of-products then it becomes a leaf node also. A flag at each leaf node indicates whether it represents a strongly unate cofactor.

Subroutine $compute\_ros1()$ in Algorithm 1 can now be modified to use the cofactor tree. It would be necessary, for the first call, that $h = g = f \cup d$. However, it is still possible to take advantage of the overexpanded cube to reduce the amount of computation.

Theorem 7.1 *Let* h *be a cofactor of* $g = f \cup d$ *and* c *be the cofactoring cube. Let* p *be a cube such that* $p \subseteq g$ *and* q_p *be its overexpanded cube. Let* M *be the literal of some variable* X *in* c. *Let* L *and* R *be single literal cubes in* X *such that* $L + R = M$. *Then*

$$R_p(\overline{h}_{q_p}) = L R_p((\overline{h_L})_{q_p}) + R R_p((\overline{h_R})_{q_p}).$$

Subroutine $find\_ros2()$ shown in Algorithm 5 can now be used to obtain the reduced offset. $find\_ros2()$ makes use of $compute\_ros2()$ which is shown in Algorithm 6. $compute\_ros2()$ makes use of Theorem 7.1 to cofactor V with respect to q_p before complementing. The cofactoring generally results in converting some literals in V to full literals. As a result, it usually takes less CPU time to do the complementation after cofactoring. When $compute\_ros2()$ reaches a leaf node that does not represent a strongly unate sum-of-products, it switches over

Algorithm 6

Input: A node n *in cofactor tree representing* h, *a mv-cube* p *and its overexpanded cube* q_p.
Output: $R_p(\overline{h})$.

```
compute_ros2(n, p, q_p) {
  If (n.leaf = TRUE) {
    h = n.cofactor
    If (n.is_unate = TRUE) {
      Obtain V by removing those cubes from h that
      don't contain p.
      Obtain W by cofactoring V with respect to q_p.
      Return(W')
    } Else {
      Obtain cofactor h_{q_p} of h with respect to q_p.
      compute_ros2(h_{q_p}, n.cofactor_cube, p)
    }
  } Else {
    If (n.L_cube is not orthogonal to q_p)
      t_L = compute_ros2(n.L_child, p, q_p)
    Else t_L = ∅
    If (n.R_cube is not orthogonal to q_p)
      t_R = compute_ros2(n.R_child, p, q_p)
    Else t_R = ∅
    t = merge(p, n.variable, L, R, t_L, t_R)
    Return(t)
  }
}
```

to $compute\_ros1()$. $find\_ros2()$ uses $find\_oc2()$ (not shown) to obtain the overexpanded cube. $find\_oc2()$ is similar to $find\_oc1()$ except that it operates on the cofactor tree and switches to $find\_oc1()$ when a leaf node is reached that represents a cofactor which is not a strongly unate sum-of-products. Subroutine $merge()$ used in Algorithm 6 is shown in Algorithm 2.

8 Experimental Results

Reduced offsets have been implemented in ESPRESSO-MV. They are used instead of the exact offset for cube expansion. The program was run on some industry PLAs and multi-valued examples from the ESPRESSO bench-mark set. The following table contains some representative examples and shows that the quality of the results is unaffected when reduced offsets are used instead of the exact offset. The first column labeled *Initial* shows the number of product terms initially in the function. The column labeled *ESPRESSO* shows the number of product terms obtained when ESPRESSO-MV is used. The last column labeled *RO* shows the number of product terms when simplification is done using the reduced offsets. For these examples, the offsets are not very large and therefore there is no appreciable

speed up when reduced offsets are used.

Initial	ESPRESSO	RO
135	107	107
137	136	136
75	74	74
234	212	212
32	17	17
108	55	55
32	20	20

The comparison was also made for mv-Achilles' heel functions in which each variable X_i is a 3-valued variable. The functions have onsets:

$$f = \sum_{i=1}^{n} X_{3i-2}^{\{1,2\}} X_{3i-1}^{\{0,2\}} X_{3i}^{\{0,1\}}$$

and no don't care sets. The results are shown in the table below. n is the number of product terms in the mv-Achilles' heel function. *C-size* is the number of cubes in the offset. *Ctime* is the CPU time taken by ESPRESSO to compute the offset. *Esp* is the time taken by original Espresso-MV to do the minimization. *RO* is the time taken by the reduced offset based Espresso-MV to do the minimization. All times are in seconds on a VAX 8800. The size of the reduced offset for each cube in each Achilles' heel function was 3 product terms.

n	C-size	C-time	Esp	RO
2	9	0.02	0.07	0.04
4	81	0.48	0.77	0.14
6	729	20.50	23.02	0.43
8	2187	752.3	774.7	1.14
10	-	-	>3600	2.25
15	-	-	>3600	9.57
20	-	-	>3600	27.12
25	-	-	>3600	62.75
30	-	-	>3600	125.9
35	-	-	>3600	229.1
40	-	-	>3600	426.8

In addition, we encountered an *industrial PLA example* which could not be simplified with Espresso-MV even after 50 hours of CPU time on a VAX 8800. The reduced offset based Espresso-MV minimized the function in 781.70 seconds, of which only 112.75 seconds were spent for reduced offset related computations. The function had 198 inputs, 237 outputs and 749 product terms. The minimized PLA had 469 product terms. This is 37% reduction in area.

9 Conclusion

In this paper, we presented an alternative to computing the offset while minimizing two-level mv-functions. This is particularly useful for functions which have such large offsets that their computation requires unreasonably large amounts of CPU time and memory. Such functions occasionally occur as real world PLAs but we feel they may become more common as multi-valued minimization finds more applications in multi-level optimization. Such applications have already begun to appear [5, 7].

The use of mv reduced offsets is recommended for functions that have reasonable size onsets and don't care sets but very large offsets. However, many functions have reasonable size offsets as well. For such functions the reduced offsets are not recommended because computing several reduced offsets instead of a single offset may be more expensive. Although the quality of results will be the same. Espresso-MV with reduced offset ought to be viewed as a special purpose tool, useful for mv-functions with large offsets. A suggested method for doing mv-minimization is to first run the original Espresso-MV with the "-t" option. This option shows each step in minimization as it occurs. If Espresso-MV takes too long to compute the offset or runs out of memory while computing the offset, then the Espresso-MV with the reduced offset should be used. Also, in applications where extreme robustness is desired, the use of reduced offset can be recommended.

References

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

[2] D. W. Brown. A State-Machine Synthesizer - SMS. In *Proceedings of 18th Design Automation Conference*, pages 301–304, Nashvill, June 1981.

[3] G. DeMicheli, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Optimal State Assignment of Finite State Machines. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-4(3):269–284, July 1985.

[4] S. J. Hong, R. G. Cain, and D. L. Ostapko. MINI: A Heuristic Approach for Logic Minimization. *IBM Journal of Research and Development*, 18:443–458, September 1974.

[5] B. Lin and A. Newton. Restructuring State Machines and State Assignment: Relationship to Minimizing Logic Across Latch Boundaries. In *Proceedings of 2nd MCNC International Workshop on Logic Synthesis*, Research Triangle Park, North Carolina, May 1989.

[6] A. A. Malik, R. Brayton, A. R. Newton, and A. Sangiovanni-Vincentelli. A Modified Approach to Two-Level Minimization. In *Proceedings of International Conference on Computer-Aided Design*, pages 106–109, Santa Clara, November 1988.

[7] S. Malik, R. Brayton, and A. Sangiovanni-Vincentelli. Encoding Symbolic Inputs for Multi-Level Logic Implementation. In *Proceedings of 2nd MCNC International Workshop on Logic Synthesis*, Research Triangle Park, North Carolina, May 1989.

[8] R. L. Rudell. Multiple-Valued Logic Minimization for PLA Synthesis. Technical Report M86/65, Electronics Research Laboratory, College of Engineering, University of California at Berkeley, Berkeley, CA 94720, 1986.

[9] T. Sasao. An Application of Multiple-Valued Logic to a Design of Programmable Logic Arrays. In *Proceedings of 8th International Symposium on Multiple Valued Logic*, 1978.