Augmented Sifting of Multiple-Valued Decision Diagrams

D. Michael Miller
Department of Computer Science
University of Victoria
Victoria, BC
CANADA V8W 3P6
mmiller@csr.uvic.ca

Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen
GERMANY
drechsle@informatik.uni-bremen.de

Abstract

Discrete functions are now commonly represented by binary (BDD) and multiple-valued (MDD) decision diagrams. Sifting is an effective heuristic technique which applies adjacent variable interchanges to find a good variable ordering to reduce the size of a BDD or MDD.

Linear sifting is an extension of BDD sifting where XOR operations involving adjacent variable pairs augment adjacent variable interchange leading to further reduction in the node count. In this paper, we consider the extension of this approach to MDDs. In particular, we show that the XOR operation of linear sifting can be extended to a variety of operations. We term the resulting approach augmented sifting.

Experimental results are presented showing sifting and augmented sifting can be quite effective in reducing the size of MDDs for certain types of functions.

1. Introduction

BDDs [1][2][3][8][14][17][18] and MDDs [10][11] are commonly used in a wide variety of applications. The variable ordering can significantly affect the size of a decision diagram and there has thus been considerable work on determining good orderings. Sifting [15] is a very effective technique applicable to BDDs and MDDs.

The size of a BDD can be further reduced by a technique called linear sifting [9]. In this approach, certain variables are replaced by the XOR of variables so that the realization of a system of functions *F* consists of a linear prefilter made up of XORs that permutes the input space to a BDD representing a system of functions *G* which together realize the given system *F* at lower overall cost. This is in fact the linearization scheme discussed by Karpovsky [7] who gave an analytical solution applicable to a system with a small number of inputs. Linear sifting is a heuristic application of linearization applicable to large problems.

The purpose of this paper is to examine the extension of linear sifting to the MDD case. We consider mod-*p* sum sifting which is based on replacing XOR by summation mod-*p* and also augmented sifting where a variety of extensions to XOR are considered.

The work reported in this paper was supported in part by a Research Grant from the Natural Sciences and Engineering Research Council of Canada.

2. Preliminaries

We consider $f_i(x_1, x_2, ..., x_n), 1 \le i \le m$, a system of totally-specified *p*-valued functions where the x_i are also *p*-valued. The functions are totally-specified so each x_i takes on all values 0…*p-1*. A particular function may take on a subset of the values 0…*p*-1. In particular, we shall consider the case of multiple-valued input, binary output functions. We denote the mod-*p* sum as $x \oplus y$ and use $x^k = (x+k) \bmod p, 1 \le k \le p-1,$ to denote the *k* possible cyclic negations.

The type of function considered can be represented by a *multiple-valued decision diagram* (MDD) which is a directed acyclic graph (DAG) with up to *p* terminal nodes each labelled by a distinct value 0,1,…,*p*-1. Every non-terminal node is labelled by an input variable and has *p* outgoing edges; one corresponding to each logic value. These are termed the 0-edge, 1-edge, *etc.*

An MDD is *ordered* if the variables adhere to a single ordering on every path in the graph, and no variable appears more than once on any path from the root to a terminal node. Finding a variable ordering to minimize the number of nodes in an MDD is a critical issue.

A *reduced* MDD has no node where all *p* outgoing edges point to the same node and no isomorphic subgraphs. Clearly, no isomorphic subgraphs exist if, and only if, no two non-terminal nodes labelled by the same variable, have the same direct descendants. Throughout this paper we assume all MDDs are reduced and ordered.

For a system of function (multiple-output problem), we represent the functions by a single DAG with multiple top nodes, a structure called a *shared* MDD. When $p = 2$, the MDD structure becomes the well-known BDD.

We use cyclic negation as an edge attribute in our MDDs as developed in [10][11] as a generalization of edge negations in BDDs [1][13]. Every edge in an MDD points to a function. When an edge has an associated cyclic negation, it means that edge points to the cyclic negation of the function rather than the function itself. The representation is normalized so that there is no cyclic negation on any 0-edge. Our MDDs always have a single terminal node with value 0. Note that a cyclic negation may be required for realizing a desired output function.

We note that the normalization process used in our MDDs differs from that often used in BDDs [16]. We have chosen the normalization rules for our package to best accommodate different values of p and to allow for easier extension to mixed-radix MDDs..

Figure 1 shows an MDD representing the sum (F1) and carry (F2) for the addition of two 3-valued inputs. The three edges from each non-terminal node are drawn solid for the 0-edge, dashed for the 1-edge, and dotted for the 2-edge. A number immediately to the right of an edge indicates a cyclic negation associated with that edge.

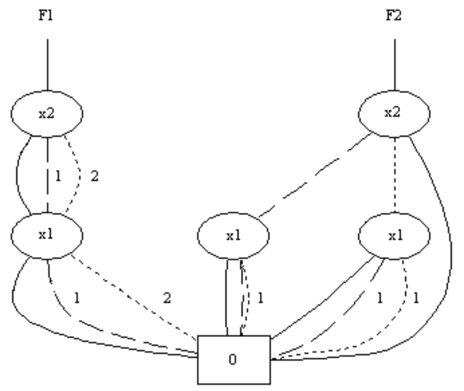


Figure 1 MDD representing sum (F1) and carry (F2).

3. Sifting of BDDs and MDDs

Sifting is a very effective heuristic variable ordering technique developed by Rudell [15] which is now available in commonly used packages such as CUDD [16].

3.1 Sifting of BDDs

The principal step in sifting is the interchange of a pair of adjacent variables in the current variable ordering. The key to the efficiency of sifting is that such a variable interchange can be done as a local operation affecting only nodes labelled by the two variables in question and no others. Use of a unique table [3][16] makes these nodes directly accessible.

In general terms, sifting proceeds as follows:

Sifting Procedure

i) select a variable y – a simple heuristic is to choose the variable that labels the most nodes in the BDD,

ii) sift y to the bottom of the BDD by a sequence of adjacent variable interchanges,

iii) sift y to the top of the BDD by a sequence of adjacent variable interchanges,

iv) during steps (ii) and (iii) a record is kept of the position of y that yields the smallest node count in the BDD, so now sift y back down to that position,

v) repeat steps (i) to (iv) until each variable has been sifted into the *best* position noting that once a variable is selected for sifting, it is not selected a second time.

There are $n!$ orderings of n variables. Sifting examines on the order of n^2 orderings, yet does extremely well at identifying good variable orderings.

In the above procedure, each variable is shifted to its 'best' position. The whole process can be iterated until there is no further improvement which is termed sifting to convergence [16]. All sifting in this work is sifting to convergence.

3.2 Sifting of MDDs

Sifting an MDD requires an efficient means of performing adjacent variable interchange. Such a method was given by the present authors in [11]. We here briefly outline this method as it will be used in modified form to implement mod-p sum and augmented sifting. Full detail can be found in [11].

We consider the interchange of x_i and x_j where the former immediately precedes the latter in the variable ordering and assume for simplicity that all non-terminal nodes have p descendants. For each node η labelled x_i, matrix $\mathbf{T}$ is constructed with $\mathbf{T}_{qr}$ set to

(a) the r-th descendant of the q-th descendant of η if the q-th descendant points to a node labelled x_j,

(b) the q-th descendant of η, otherwise.

Given $\mathbf{T}$ formed as described above, the new nodes labelled x_i are constructed using the columns of $\mathbf{T}$ to determine the descendants and then using the nodes so constructed as the descendants of the new node labelled x_j. In simplest terms, the required rearrangement is accomplished by filling $\mathbf{T}$ by rows and then applying it by columns. There are a number of implementation issues to consider which are given in detail in [11].

Given this method for adjacent variable interchange, sifting of MDDs is readily implemented using the same overall approach as for the BDD case.

4. Linear Sifting of BDDs

Linear sifting was introduced by Meinel, Somenzi and Theobold [9] and further discussed by Günther and Drechsler [4][5][6]. In this extension to sifting, the simple interchange of two adjacent variables x_i and x_j in steps (ii) and (iii) of the procedure outlined above is replaced by the following:

(a) Variables x_i and x_j are interchanged. Let k_1 be the number of nodes in the BDD after this interchange.

(b) Apply the linear transformation $x_j \leftarrow x_i \oplus x_j$. Let k_2 be the resulting number of nodes in the BDD.

(c) If $k_1 \leq k_2$ then the transformation is undone.

Undoing the transformation is accomplished by simply reapplying it since it is its own inverse. Note that the above is described in terms of XOR due to the normalization rules we use for decision diagrams. The original description in [9] is in terms of equivalence due to different normalization rules. The concept is the same.

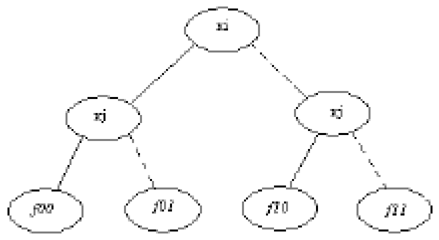
Figure 2 illustrates the two basic operations used in linear sifting. (a) shows a BDD structure before transformation. 0-edges are solid and 1-edges are dashed. The nodes labelled f_{00} through f_{11} are the top nodes of sub-DAGs representing subfunctions (not shown). (b) shows the effect of interchanging x_i and x_j which is to interchange the subfunctions f_{01} and f_{10}. (c) shows the effect of subsequently applying $x_j \leftarrow x_i \oplus x_j$ which is to interchange the subfunctions f_{10} and f_{11} in (b).

The function represented by each of the diagrams in Figure 2 is the same and the representations of the subfunctions f_{00} through f_{11} are not affected by the transformations. Hence both the interchange of variables and the linear transformation are local operations affecting only two adjacent levels in the BDD. Using cyclic negations does not change this locality property.

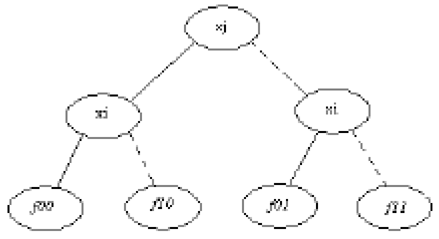
Symmetry would suggest application of the transformation $x_i \leftarrow x_i \oplus x_j$ should be considered. In fact, it is since sifting will encounter the variables in the two possible orderings. Trying both $x_j \leftarrow x_i \oplus x_j$ and $x_i \leftarrow x_i \oplus x_j$ for each orientation duplicates effort.

5. Mod-*p* Sum and Augmented Sifting of MDDs

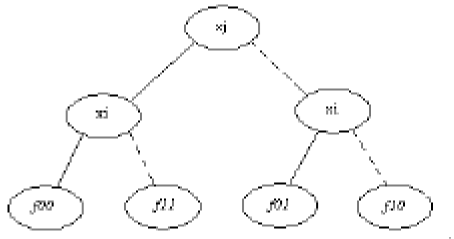
We next consider the extension of linear sifting to MDDs. We first address the case of replacing XOR by the mod-*p* sum which results in an approach we term *mod-p sum sifting*. Based on that, we then consider other operations as extensions to XOR. The full method, which we term *augmented sifting*, allows for the consideration of multiple operations during a single sifting process.



(a) initial structure



(b) after interchange of x_i and x_j



(c) after subsequent transformation $x_j \leftarrow x_i \oplus x_j$

Figure 2 Linear sifting transformations.

5.1 Mod-*p* Sum Sifting of MDDs

As the first step in extending the idea of linear sifting to MDDs, we consider the replacement of XOR in $x_j \leftarrow x_i \oplus x_j$ with the mod-*p* sum. A critical difference to note is that while XOR is its own inverse, mod-*p* summation is not its own inverse and to undo a mod-*p* sum transformation we must apply mod-*p* subtraction. Fortunately, both transformations can be implemented using essentially identical matrix procedures.

In general, the interchange of the two variables x_i and x_j results in the subfunction interchanges

$$f_{st} \leftrightarrow f_{ts}, s \neq t, 0 \leq s,t \leq p-1$$

Similarly, application of the transform $x_j \leftarrow x_i \oplus x_j$ results in the subfunction substitutions

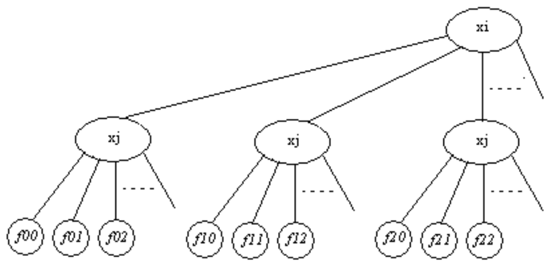
$$f_{st} \leftarrow f_{s,s\oplus t}, 0 \leq s,t \leq p-1$$

The interchange of two variables and the transformation $x_j \leftarrow x_i \oplus x_j$ is implemented using the matrix based procedure described above in Section 3.2.

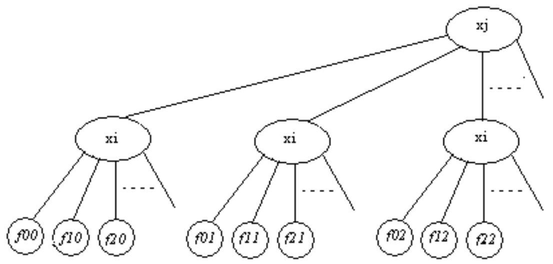
The method proceeds as illustrated in Figure 3. In general, consider a node γ labelled x_j. We construct a matrix $\mathbf{T}$ with *p* rows and *p* columns. For *i*=0,1,…, *p*-1,

(a) If the *s*-edge from γ leads to a node δ labelled x_i, then for *t*=0,1,…,*p*-1, $\mathbf{T}_{s\oplus t,t}$ is set to point to the node pointed to by the *t*-edge of δ with the edge cycles being the composition of the edge cycles on the *s* edge from γ and the *t* edge from δ.

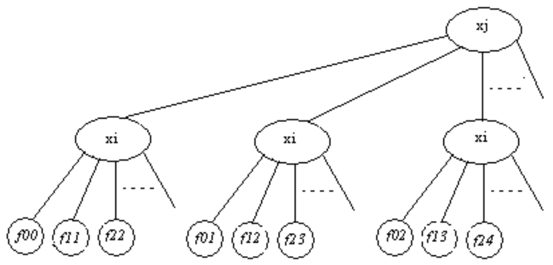
(b) If the *s*-edge from γ leads to a node δ not labelled x_i, then $\mathbf{T}_{s\oplus t,t}$ is set to the *s*-edge from γ for *t*=0,1,…,*p*-1.



(a) initial structure



(b) after interchange of x_i and x_j



(c) after transformation $x_j \leftarrow x_i \oplus x_j$

Figure 3 Mod-p sifting transformations.

Once **T** is constructed as above, the transformation is made by setting each *s*-edge from γ, *s*=0,1,…,*q*-1 to point to a node labelled x_i whose *t*-edge, *t*=0,1,…,*p*-1, points to the node pointed to by $\mathbf{T}_{s\oplus t,t}$. During this construction, the edge cycle operations are normalised to ensure there is no cycle operation on any 0-edge.

The complete transformation is accomplished by performing the above for all nodes originally labelled x_j.

In the same fashion as discussed above for variable interchange, it is clear that this is a local transformation of the MDD affecting only the x_i and x_j levels.

The same procedure is used for the reverse transformation, *i.e.* to undo a transformation when it does not improve the MDD node count. The difference is that reference is made to $\mathbf{T}_{s\odot t,t}$ where the mod-*p* difference is $s \odot t = (s - t + p) \bmod p$.

Once the variable currently being considered has been sifted to the bottom and then to the top it must be positioned to yield the smallest decision diagram. As noted in step (iv) of the sifting procedure presented in Section 3.1, for sifting, only a sequence of downward variable interchanges is required.

For linear or mod-*p* sum sifting, an ordered record must be kept of the $x_j \leftarrow x_i \oplus x_j$ transformations. Putting the variable under consideration into the correct position, requires the $x_j \leftarrow x_i \oplus x_j$ and variable interchanges be undone in reverse order back to but not undoing the interchanges and transformations that put the variable into the best position during the sifting process. The bookkeeping required is straightforward but the computation in undoing the operations back to the best position can be substantial and in general can be equal to the computation required in the sifting down and up of the variable. The latter is certainly the case when the original position is optimal for the variable.

5.2 Other Operations

We confine our attention to *p* = 2, 3, 4. The extension to higher values of *p* should be clear. Table 1 shows the sum and difference operations modulo-*p*. The critical properties for the work here are

(a) The operations are reversible so that a transformation that does not reduce the size of a decision diagram can be undone.

(b) The (0,0) entry is 0 which means the subfunction on the 0-0 path does not move so the transformation of the decision diagram is a local operation. Replacing this subfunction with another could require a normalization requiring edge operation changes higher in the diagram thereby destroying the locality of the transformation.

Given (a) and (b), XOR is the only choice when *p* = 2. If we require just (a) and (b), there are a number of alternatives to mod-*p* sum when *p* > 2. For example, for *p* = 3, one alternative pair is shown in Table 2.

To limit the number of choices to a reasonable number both in terms of the computation and the bookkeeping required, we add a third constraint that the generalizations of $\oplus$ must satisfy

(c) The first row and the first column of the table defining $\oplus$ should contain 0, 1, …, *p*-1 in order.

Mod-*p* sum is then the only generalization of $\oplus$ for *p* = 3. For *p* = 4, mod-4 sum is a proper generalization as are the operations $\oplus_1, \oplus_2, \oplus_3$ listed in Table 3.

p=2

$x \oplus y$	y	
	0	1
0	0	1
x 1	1	0

$x \odot y$	y	
	0	1
0	0	1
x 1	1	0

p=3

$x \oplus y$	y		
	0	1	2
0	0	1	2
x 1	1	2	0
2	2	0	1

$x \odot y$	y		
	0	1	2
0	0	2	1
x 1	1	0	2
2	2	1	0

p=4

$x \oplus y$	y			
	0	1	2	3
0	0	1	2	3
x 1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

$x \odot y$	y			
	0	1	2	3
0	0	3	2	1
x 1	1	0	3	2
2	2	1	0	3
3	3	2	1	0

Table 1 Sun and difference mo-p for p=2, 3, 4.

$x \hat{\oplus} y$	y		
	0	1	2
0	0	2	1
x 1	2	1	0
2	1	0	2

and

$x \hat{\odot} y$	y		
	0	1	2
0	0	2	1
x 1	2	1	0
2	1	0	2

Table 2 Alternative generalization for p = 3.

5.3 Augmented Sifting

Our augmented sifting method follows the same computational procedure as linear sifting. The difference is that while linear sifting for p = 2 need only consider XOR operations between a pair of variables, for p = 3 or 4 our method tries each of the appropriate generalizations of $\oplus$, and when they are different the corresponding generalizations of $\odot$. Hence, for every adjacent variable interchange while a variable is sifted to the bottom of the MDD and then to the top, the augmented sifting method tries transformations based on the operations:

- p = 2: XOR;
- p = 3: mod-3 sum, mod-3 difference;
- p = 4: mod-4 sum, mod-4 difference, the 5 distinct functions in Table 3.

At each step, the augmented sifting method chooses the transformation (if any) from amongst those tried that yields the greatest reduction in the MDD node count. The implementation of all these transformations is as described for mod-p sum in Section 5.1.

$x \oplus_1 y$	y			
	0	1	2	3
0	0	1	2	3
x 1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

$x \odot_1 y$	y			
	0	1	2	3
0	0	1	2	3
x 1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

$x \oplus_2 y$	y			
	0	1	2	3
0	0	1	2	3
x 1	1	0	3	3
2	2	3	1	0
3	3	2	0	1

$x \odot_2 y$	y			
	0	1	2	3
0	0	1	3	2
x 1	1	0	2	3
2	2	3	0	1
3	3	2	1	0

$x \oplus_3 y$	y			
	0	1	2	3
0	0	1	2	3
x 1	1	3	0	2
2	2	0	3	1
3	3	2	1	0

$x \odot_3 y$	y			
	0	1	2	3
0	0	2	1	3
x 1	1	0	3	2
2	2	3	0	1
3	3	1	2	0

Table 3 Alternative operations for p=4.

6. Experimental Results

Augmented sifting has been implemented in the MDD package discussed by the present authors in [11] and [12]. As noted above, cyclic negations are used. The MDDs are build using recursive implementations of MIN and MAX and unique and compute tables as discussed in [12]

We here present the results of applying the procedure to a variety of functions using our MDD package on a Sun Blade 1000 with one 750 MHz. UltraSPARC III CPU with 512 Mb RAM. The binary examples presented for comparison were also done with our MDD package with p = 2 in which case augmented sifting is linear sifting.

p	in	out	initial size	sifted size	mod-p sifted size	# transfor-mations
2	4	3	13	11	10	1
	6	4	32	21	17	1
	8	5	71	34	24	2
	10	6	150	50	31	3
	12	7	309	69	38	4
	14	8	628	91	45	5
	16	9	1267	116	52	6
3	4	3	22	15	9	2
	6	4	71	28	13	3
	8	5	219	45	17	4
	10	6	664	66	21	5
4	4	3	32	18	11	2
	6	4	133	34	16	3
	8	5	538	55	21	4

Table 4 Sifting and mod-p sifting of adder function.

Table 4 shows the results for *p*-valued addition of two *n*-bit numbers where each example has 2*n* *p*-valued inputs and *n*+1 *p*-valued outputs. The column labelled 'initial size' is the number of MDD nodes for the input ordering $a_n, a_{n-1}, ..., a_1, b_n, b_{n-1}, ..., b_1$ that is the inputs of the two numbers being added one following the other. The column 'sifted size' is the number of MDD nodes after sifting is applied. The variable ordering found by sifting is $a_n, b_{n,}, a_{n-1}, b_{n-1}, ..., a_1, b_1$.

The column 'mod-*p* sifted size' is the node count after mod-*p* sifting is applied. The node count is substantially reduced by mod-*p* sifting with just a few transformations.

Sifting and mod-*p* sifting are clearly very effective for adders since they are both symmetric in corresponding positions for the numbers being added and also highly dependent on the $\oplus$ operation. Augmented sifting gives no further improvement for adders.

Multiplication is a difficult case for decision diagram representation, and mod-*p* and augmented sifting do not help. For example, multiplication of two 6-bit binary numbers, a problem with 12 inputs and 12 outputs, has 1,158 nodes in the simple one number after the other variable order, and 1,098 nodes after applying sifting. Applying mod-*p* or augmented sifting yields the same result as sifting with no transformations selected.

p	*n*	initial size	mod-*p* sifted size	# transfor-mations
2	5	17	12	3
	6	24	16	6
	7	31	22	6
	8	38	29	8
	9	53	40	10
	10	64	48	11
	11	75	57	12
	12	90	66	16
	13	105	78	16
	14	120	94	19
	15	135	110	16
	16	150	121	22
3	2	6	5	1
	3	10	7	3
	4	14	11	3
	5	24	19	4
	6	36	29	5
	7	49	41	6
	8	62	53	7
	9	75	65	8
	10	88	77	9
4	2	7	6	1
	3	12	9	2
	4	17	13	3
	5	22	17	4
	6	39	33	5
	7	59	51	6
	8	80	71	7

Table 5 Mod-p sifting of summation functions.

p	in	out	initial size	sifted size	augmented sifted size	# transfor-mations
2	4	3	13	11	10	1
4	2+	3	12	12	6	2
4	2*	3	8	6	6	0
2	6	4	32	21	17	1
4	3+	4	25	23	23	0
4	3*	4	15	9	9	0
2	8	5	71	34	24	2
4	4+	5	65	35	15	2
4	4*	5	24	12	12	0
2	10	6	150	50	31	3
4	5+	6	115	53	53	0
4	5*	6	35	15	15	0
2	12	7	309	69	38	4
4	6+	7	264	68	23	3
4	6*	7	48	18	18	0
2	14	8	628	91	45	5
4	7+	8	477	123	123	0
4	7*	8	63	21	21	0

(+ unsifted order conversion; * sifted order conversion)

Table 6 Binary and quaternary coded adders.

Table 5 shows the results for the summation of *n* *p*-valued inputs. The number of outputs in each case is $\lceil \log_p(n \times p) \rceil$ and is the *p*-valued representation of the arithmetic sum of the inputs.

Table 6 is a comparison of the BDD size of binary adders and the size of two distinct MDDs derived from each. Each case has three rows. The first gives the results for the binary adders which are those from Table 4. The second row is for the MDD where each quaternary input is derived from a pair of binary inputs from left to right where the binary inputs are in the order $a_n, a_{n-1}, ..., a_1, b_n, b_{n-1}, ..., b_1$, *i.e.* the digits of the first number followed by the second which we call unsifted order. The natural binary to quaternary conversion is used, *i.e.* $(00 \rightarrow 0; 01 \rightarrow 1; 10 \rightarrow 2; 11 \rightarrow 3)$

The outputs are left as binary so the derived functions are quaternary-input binary-output and do not represent the quaternary-input, quaternary-output adder in Table 4.

The third row of the table is for the MDD constructed in the same fashion but using the sifted variable order found in the binary case which as noted above is $a_n, b_{n,}, a_{n-1}, b_{n-1}, ..., a_1, b_1$.

A number of observations can be made. First it is clear that basing the conversion of binary inputs to quaternary inputs on the binary sifted order is better than using the unsifted order. In particular, we conjecture the MDD constructed from the sifted binary order has $3n$ nodes whereas the corresponding linear sifted BDD has $7n-4$ nodes where *n* is the number of bits in each of the binary numbers being added..

We also note that augmented sifting is beneficial for the BDDs (in fact linear sifting) and also for the MDDs derived from the unsifted binary inputs when n is even. Augmented sifting does not help the MDDs when n is odd because the quaternary encoding combines the least significant bit of a with the most significant bit of b which precludes the transformations found in the even case where this is not the situation.

It is also interesting to note that for the MDDs constructed from the sifted binary order, augmented sifting of the MDD itself is of no benefit. This is the situation because the binary variable pairing used to construct the quaternary inputs captures the linearity.

Table 7 (at the end of the paper) shows the results for a number of commonly used benchmark problems. Three representations are presented for each problem: the BDD for the original binary problem, the 4-valued input, binary-output MDD for the given variable order, and the 4-valued input, binary-output MDD for the variable order found by sifting for the BDD.

Three scenarios are presented: (A) sifting followed by augmented sifting, (B) sifting followed by mod-*p* sum sifting (we show only the case where the result can differ from scenario A, and (C) mod-*p* sum sifting not preceded by regular sifting.

We note that in this paper we are not concerned with the best way to transform a binary problem to a quaternary one. They are here only used as a source of examples. However, we do note that in general smaller MDDs arise when the BDD sifted variable ordering is used. The exceptions to this arise when this approach pairs variables that are not in the support set of the majority of output functions. For example, this results in the MDDs derived for example e64 being larger than the BDD. The quaternary pairing has in fact introduced variable dependency not present in the original problem. That situation must be avoided in the case where the objective is to find a good conversion of a binary problem to quaternary. Nevertheless, the BDD sifting order is a good starting point.

7. Concluding Remarks

This paper has considered the augmented and mod-*p* sum sifting of MDDs. The experimental results presented indicate these approaches work well for certain classes of functions such as adders and weight functions. We expect they will work well for many 'arithmetic' types of functions with the notable exception of multipliers.

In general, our results indicate mod-*p* sum sifting is as effective as the more general augmented sifting while requiring considerably less computation. The results also indicate that as found in [5] for linear sifting of BDDs, it is best to apply regular sifting followed by mod-*p* sum or augmented sifting. The regular sifting determines a good threshold by variable swapping after which transformations are applied only when they further reduce the node count.

Optimization of our implementations of sifting, mod-*p* sum and augmented sifting is ongoing. At present, our implementations are rather slow, especially in comparison to a highly optimised package such as CUDD [17]. For example, the problem apex1 (45 inputs and 45 outputs) treated as binary requires 1.8 CPU sec. for sifting and 4.8 sec. for augmented (linear) sifting using our package. When apex1 is converted to a quaternary problem, sifting takes on the order of 1.7 sec. and mod-*p* sum sifting takes about 2.5 sec. Augmented sifting for this examples takes about 23 sec. We have found that mod-*p* sum sifting takes on the order of 2 to 5 times longer than sifting whereas augmented sifting takes on the order of 7 to 10 times longer than mod-*p* sum sifting.

In contrast, the time required to read and sift apex1 using CUDD is negligible on the same machine. A major part of the difference is that our package treats a BDD as a special case of an MDD which leads to a slower implementation due to the flexibility required, *e.g.* a variable number of edges from each node, cyclic negation as opposed to simple binary negation *etc*.

The MDD package used in this work is available at www.csr.uvic.ca/~mmiller/MDD.

References

[1] K. S. Brace, R L. Rudell and R. E. Bryant, “Efficient implementation of a BDD package,” *Proc. Design Automation Conference*, pp. 40-45, 1990.

[2] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Trans. on Computers*, V. C-35, no. 8, pp. 677-691, 1986.

[3] R. Drechsler and D. Sieling, “Binary decision diagrams in theory and practice,” *Int. Journal on Software Tools for Technology Transfer*, 3, pp. 112-136, 2001.

[4] W. Günther and R. Drechsler, “Linear transformations and exact minimization of BDDs,” *IEEE Great Lakes Symposium on VLSI*, pp. 325-330, Lafayette, LA, Feb. 1998.

[5] W. Günther and R. Drechsler, “BDD minimization by linear transformations,” *Conf. on Advanced Computer Systems*, pp. 525-532, Szczecin, Poland, Nov. 1998.

[6] W. Günther and R. Drechsler, “Minimization of BDDs using linear transformations based on evolutionary techniques,” *IEEE International Symposium on Circuits and Systems*, pp. I:387-390, Orlando, FL, May 1999.

[7] M. G. Karpovsky, *Finite Orthogonal Series in the Design of Digital Devices*, John Wiley and Sons., 1976.

[8] H. T. Lau and C.-S. Lim, “On the OBDD representation of general Boolean functions,” *IEEE Trans. on Comp.,* C-41, No. 6, pp. 661-664, 1992.

[9] C. Meinel, F. Somenzi, and T. Theobold, “Linear sifting of decision diagrams,” *Proc. Design Automation Conference*, pp. 202-207, 1997.

[10] D. M. Miller, “Multiple-valued logic design tools,” (Invited Address) *Proc. 23rd Int. Symp. on Multiple-Valued Logic*, pp. 2-11, May 1993.

[11] D. M. Miller and R. Drechsler, “Implementing a multiple-valued decision diagram package,” *Proc. 28th Int. Symp. on Multiple-Valued Logic*, pp. 52-57, May 1998.

[12] D. M. Miller and R. Drechsler, “On the construction of multiple-valued decision diagrams,” *Proc. 32nd Int. Symp. on Multiple-Valued Logic*, pp. 245-253, May 2002.
[13] S. Minato, N. Ishiura and S. Yajima, “Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation,” *Proc. ACM/IEEE Design Automation Conference*, pp. 52-57, 1990.
[14] S. Minato, “Graph-based representations of discrete functions,” *Proc. IFIP WG 10.5 Workshop on the Application of Reed-Muller Expansion in Circuit Design*, pp. 1-10, 1995.
[15] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” *Proc. IEEE/ACM ICCAD*, pp. 43-47, 1993.
[16] F. Somenzi, “CUDD: CU Decision Diagram Package,” http://bessie.colorado.edu/~fabio/ CUDD
[17] F. Somenzi, “Efficient manipulation of decision diagrams,” *Int. Journal on Software Tools for Technology Transfer*, 3, pp. 171-181, 2001.
[18] A. Srinivasan, T. Kam, S. Malik, and R.E. Brayton, “Algorithms for discrete function manipulation,” *Proc. ICCAD*, pp. 92-95, 1990.

						Scenario A		Scenario B		Scenario C	
example	*p*	in	out	initial size	sifted size	augmented sifted size	# trans.	mod-*p* sifted size	# trans.	direct mod-*p* sifted size	# trans.
alu2	2	10	8	114	60	60	0			60	0
	4	5+	8	90	54	53	1	54	0	54	0
	4	5*	8	48	48	48	0			48	0
alu4	2	14	8	1093	570	540	3			430	7
	4	7+	8	786	573	573	0			573	0
	4	7*	8	408	381	381	0			381	0
apex1	2	45	45	4876	1307	1278	2			1344	21
	4	23+	45	3081	874	873	1	873	1	873	3
	4	23*	45	933	874	874	0			874	2
apex2	2	39	3	5613	400	400	0			400	5
	4	20+	3	3470	646	646	0			646	1
	4	20*	3	299	274	274	0			274	0
apex3	2	54	50	1044	904	896	2			895	4
	4	27+	50	597	563	563	0			563	0
	4	27*	50	766	602	594	14	595	4	595	5
bw	2	5	28	112	99	99	0			99	2
	4	3+	28	88	69	69	0			68	1
	4	3*	28	81	63	63	0			63	2
seq	2	41	35	2153	1201	1122	12			1156	23
	4	21+	35	1300	857	853	3	857	0	880	2
	4	21*	35	933	857	843	1	843	1	843	1
e64	2	65	65	1444	129	129	0			129	0
	4	33+	65	1019	592	582	3	582	3	593	7
	4	33*	65	162	162	162	0			162	0
duke2	2	22	29	769	369	338	4			347	25
	4	11+	29	561	339	339	0			356	7
	4	11*	29	279	278	277	1	277	1	277	1
misex1	2	8	7	71	36	36	0			36	1
	4	4+	7	47	25	25	0			27	1
	4	4*	7	33	33	33	0			33	0
misex2	2	25	18	114	82	79	1			80	1
	4	13+	18	114	81	79	1	79	1	79	1
	4	13*	18	78	64	63	1	64	0	64	0
misex3	2	14	14	652	480	478	1			476	6
	4	14+	14	433	321	321	0			321	0
	4	14*	14	318	317	317	0			317	0
sao2	2	10	4	126	86	79	3			70	11
	4	5+	4	81	64	59	2	61	3	61	2
	4	5*	4	56	56	50	3	54	2	54	2
sn74181	2	14	8	626	564	560	1			560	1
	4	7+	8	402	390	390	0			390	0
	4	7*	8	467	458	458	0			458	0

(+ unsifted order conversion; * sifted order conversion)

Table 7 Standard benchmark functions.