Boolean Function Representation Based on Disjoint-Support Decompositions.[*]

Valeria Bertacco and Maurizio Damiani[**]
Dipartimento di Elettronica ed Informatica
Università di Padova, Via Gradenigo 6/A, 35131 Padova, ITALY

Abstract

The Multi-Level Decomposition Diagrams (MLDDs) of this paper are a canonical representation of Boolean functions expliciting disjoint-support decompositions. MLDDs allow the reduction of memory occupation with respect to traditional ROBDDs by decomposing logic functions recursively into simpler - and more sharable - blocks. The representation is less sensitive to variable ordering, and because of this property, analysis of the MLDD graphs allows at times the identification of better variable orderings. The identification of more terminal cases by Boolean algebra techniques makes it possible to compensate the additional - small- CPU time required to identify the disjoint-support decomposition. We expect the properties of MLDDs to be useful in several contexts, most notably logic synthesis, technology mapping, and sequential hardware verification.

1 Introduction

Reduced, Ordered Binary Decision Diagrams (ROBDDs) [1] are probably the most powerful data structure known so far for the manipulation of large logic functions, and for this reason they have become pervasive in logic synthesis and verification environments [2, 3, 4, 5]. Ongoing research is attempting to extend their applicability to other domains, such as word-level verification [6], the solution of graph problems and integer-linear programming [7, 8].

Still, some key inefficiencies (an exponential blowup for some classes of functions, the unpredictability of the ROBDD size and shape with respect to the variable ordering chosen, etc ...) motivate an increasing research activity in this area, including: Efficient implementations [9, 10], development of ordering heuristics [11, 12, 13], and alternative representations altogether [14, 15, 16, 17, 18, 6, 19].

ROBDDs are closely related to deterministic automata: input bits are evaluated sequentially, one at a time, along the graph [19]. In this paper, we add to the basic ROBDD representation the capability of decomposing a function into an **arbitrary**, **multiple-level tree** of disjoint-support sub-functions. Unlike ROBDDs, nodes represent not only two-input MUXes, but also unlimited-fanin OR / AND (or NAND-only, NOR-only) trees of gates.

The novel representation retains most of the properties of ROBDDs, namely, canonicity, a directed-acyclic graph structure, a recursive construction technique, and constant-time complementation. Tree decomposition brings several advantages. First, the representation is significantly less order-sensitive than ROBDDs. For instance, fully decomposable functions are represented by minimal tree circuits, of size linear in the number of variables, regardless of variable ordering. while ROBDD size can range from linear to exponential. Moreover, for this class of functions, some difficult problems (like, Boolean NPN matching [20]) can be solved in linear time by tree matching techniques.

Second, the new representation is usually more compact than ROBDDs, because functions are decomposed in simpler , more sharable blocks. More interestingly, the new representation gives us a more systematic and exact insight on the role of the input variables of a logic function, otherwise deferred to special-purpose heuristics such as dynamic reordering.

Eventually, we show that the additional CPU time for decomposition is provably small. Moreover, the new representation allows us to identify more terminal cases, and therefore to obtain faster and shallow recursions. For instance, the computation of $f + g$ can be carried out in constant time if f and g share no variables, and the size of the result is $|f| + |g| + 1$, regardless of variable order. Other simplifications arise from the recognition of common terms in the decomposition of the operands. In practice, we found the CPU time always to be close and often better than that of reference ROBDD packages.

For reasons of space, theorem proofs are not included. They will be available upon request.

2 Disjoint support decomposition

We consider the decomposition of functions into the NOR (NAND, OR, AND) of disjoint-support subfunctions, whenever possible. This notion will lead to a recursive (e.g. tree) decomposition style and to the definition of MLDDs.

Definition 1. *Let $f : B^n \rightarrow B$ denote a Boolean function of n variables $x_1, \cdots, x_n$. We say that f* **depends** *on x_i if $\partial f / \partial x_i$ is not identically 0. We call* **support** *of f (indicated by $S(f)$) the set of Boolean variables f depends on.*

A set of non-constant functions $\{f_1, \cdots f_k\}$, $k \geq 1$, with respective supports $S(f_i)$ is called a **disjoint-support NOR <u>decomposition</u>** *of f if:*

$$\overline{f_1 + \cdots + f_k} = f; \quad S(f_i) \cap S(f_j) = \emptyset, \quad i \neq j \qquad (1)$$

A disjoint support NOR decomposition is **maximal** *if no function f_i is further decomposable in the OR of other functions with disjoint support. We define disjoint support OR, AND, NAND decompositions in a similar fashion. We indicate by* $\mathbf{D_{NOR}(f)}$ *any such maximal decomposition.* □

[*]Research partially funded by the EC ESPRIT III Basic Research Programme contract No. 9072 (Project GEPPCOM) and by CNR grant # 95.02061.CT07.

[**]Currently with Synopsys, Inc. 700 East Middlefield Road, Mtn.View, CA 94043

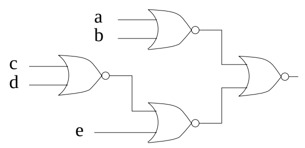

Figure 1. A recursively decomposable function.

Example 1. The function $f = (ab + a'c)(d + e)$ has the following maximal disjoint-support decompositions:

- AND: $\{f_1 = (ab + a'c); f_2 = (d + e)\}$;
- NOR: $\{f_1 = (ab + a'c)'; f_2 = (d + e)'\}$;
- NAND and OR: $\{f\}$.

□

In the rest of the paper, maximal disjoint-support decompositions are referred to as decompositions, for short. Moreover, we focus only on NOR decomposition, as the results for the other decompositions can be obtained readily by standard Boolean algebra.

By decomposing recursively a logic function, we obtain a NOR-tree representation of F:

Example 2. The function $F = (a + b)(c'd' + e)$ is recursively NOR decomposable. From the first decomposition we obtain $f_1 = (a + b)'$ and $f_2 = [e + (c + d)']'$. These functions are then again decomposable until reaching the input variables, as in Fig. (1). □

Definition 2. *A* **tree decomposition** *of a logic function f is a decomposition of f into a NOR-only tree of subfunctions, where the functions at the inputs of each NOR are maximally decomposed. We indicate by* $\mathbf{TD_{NOR}}$ *the decomposition tree. Similarly we can define TD_{NAND} and $TD_{\text{AND/OR}}$.* □

Theorem (1) below states a relevant intuitive result:

Theorem 1. *For a given function f, 1) there is a unique D_{NOR} ; and 2) there is a unique TD_{NOR} .* □

3 Multi-Level Decomposition Diagrams

We exploit tree decompositions to derive a hybrid representation style. The model is based on applying tree decomposition whenever possible, and then Shannon expansion until reaching primary input variables or their complements.

Example 3. The function $f = (a'b + ac'd' + e'f')'$ has TD_{NOR} as in Fig. (2.a). Note that we could not decompose $ac'd' + a'b'$ because of the disjoint support constraint. Applying Shannon expansion, in Fig. (2.b) we obtain a TD_{NOR} for each input of MUX. □

Definition 3. *A MLDD is a directed acyclic graph, with leaf vertices labeled by a Boolean constant or variable and two kinds of internal vertices: NOR vertices have a non-empty set of outgoing edges, each pointing to a MLDD; MUX vertices have two outgoing edges, labeled 0 and 1 and each pointing to a MLDD. MUX vertices are labeled by a Boolean variable.*

A MLDD defines recursively a logic function with the following rules:

- *A terminal vertex labeled by variable or constant x denotes the function x.*
- *A MUX m labeled by x defines:*
$$F_m = \overline{x}F_{\mathbf{0}}(m) + xF_{\mathbf{1}}(m) \qquad (2)$$
- *A NOR vertex n with k outgoing edges defines the function:*
$$F_n = \overline{f_1 + \cdots + f_k} \qquad (3)$$
where f_i, $i = 1, \ldots, k$ is the function defined by the MLDD pointed by edge i.

□

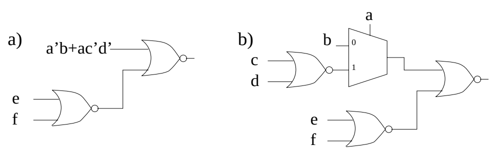

Figure 2. a) Tree decomposition of the function in Example (3). b) The same function with the addition of BDD nodes.

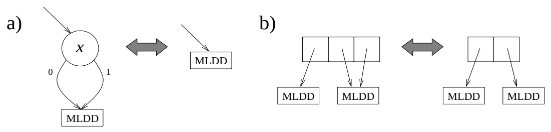

Figure 3. Second reduction rule. a) Mux vertices. b) NOR vertices

In a MLDD, while MUX vertices correspond to ROBDD nodes, NOR vertices indicate a function decomposition. Just like ROBDDs, we impose **reduction** and **ordering rules** to obtain a more compact and canonical representation:

- There are no two identical subgraphs;
- For each vertex, no two pointers point to the same MLDD;
- Each path from root to a terminal must traverse subsequent MUX nodes in respect of the variable ordering and each variable is evaluated at most once on each path.

The MLDD of a function matches a multi-level logic circuit in the obvious way. In the subsequent drawings, circles represent MUXes, while arrays of squares represent NORs.

It is worth noting that, unlike ROBBDs, second reduction rule bears different consequences on the two kinds of internal vertices. As sketched in Fig. (3), a reduction of a NOR vertex does not cause its deletion. In addition to ROBDD-like rules, in order to grant canonicity we must impose **decomposition rules**:

- the functions pointed by a NOR vertex must represent a maximal decomposition.
- a function is represented by a MUX iff it is not decomposable, nor its complement.

The following result is a direct consequence of the canonicity of tree decompositions and reduction rules:

Theorem 2. *Reduced Ordered Decomposed MLDDs are canonical.* □

The following results on D_{NOR}s are useful in the construction of the core procedures:

Theorem 3. *Suppose $\{f_1, \cdots, f_k\}$ is a D of some function. Then, by erasing elements from the set, the new set is also a D.* □

```
    NOR (mldd op1, mldd op2, int i) {
1   if (terminal case) return (terminal value);
2   D(opc) = D(op1) ∩ D(op2);
3   D(op1) = D(op1) \ D(opc);
4   D(op2) = D(op2) \ D(opc);
5   if (terminal case) return ($\overline{\text{D(opc)} \cup \text{terminal value}}$);
6   res = comp_lookup(op1, op2);
7   if (res != NULL) return ($\overline{\text{D(opc)} \cup \text{res}}$);
8   x = top_var(op1, op2);
9   le=NOR (op1.x', op2.x', i-1);ri=NOR (op1.x, op2.x, i-1);
10  res = mldd_find (le, ri, x);
11  comp_insert (op1, op2, res);
12  return ($\overline{D(opc) \cup \text{res}}$); }
```

Figure 4. Pseudocode of `NOR()`

Theorem 4. *If* $D_{\text{NOR}}\ (f) = \{f_1, \cdots, f_k\} \cup \{p_1, \cdots, p_h\}$ *and* $D_{\text{NOR}}\ (g) = \{g_1, \cdots, g_l\} \cup \{p_1, \cdots, p_h\}$, *where* $g_i \neq f_j, i = 1, \cdots l, j = 1, \cdots k$, *then:*

1. $D_{\text{NOR}}\ (\overline{f+g}) = (\{p_1, \cdots, p_h\} \cup$
   $\cup D_{\text{NOR}}\ ([(f_1 + \ldots + f_k)' + (g_1 + \ldots + g_l)']'))'$
2. *Let* x *denote a variable, and suppose* $f = (x+g)'$. *Then,* $D_{\text{NOR}}\ (f) = \{x\} \cup D_{\text{NOR}}\ (g'_{x'})$
3. *Let* x *denote a variable not in the support of* f *or* g. *Then:* $D_{\text{NOR}}\ (x'f + xg) = \{p_1, \cdots, p_h\} \cup D_{\text{NOR}}\ ([x'(f_1 + \ldots + f_k)' + x(g_1 + \cdots + g_l)']')$

□

4 MLDD manipulation routines

We tested two distinct implementations of MLDDs. In the first implementation, vertices are realized uniformly with n-tuples, the first element being an integer, all the others being pointers to other MLDDs. In the first element we encode the type of node (*i.e.*, MUX or NOR vertex), the number of elements in the n-tuple (2 for MUX vertices) and the top variable of the function represented. In the second implementation, NOR vertices are implemented by linked lists. Although the memory occupation of a single list is twice than that of the corresponding array, this implementation allows the sharing of list elements. In practice, we found little difference in terms of CPU time or memory occupation between the two lists. In either case, we maintain the structure in strong canonical form, (*i.e.*, no two copies of the same graph exist), by the usual hashing.

We implemented Boolean operation routines. Fig. (4) reports the pseudo-code for the NOR of two functions.

`NOR` is invoked by the netlist parser. For each call, the parser knows the support of the two functions and it determines an upper bound `i` on the recursion depth, namely, the depth of the last variable in common between `op1, op2`.

The recursion is structured as follows. First, terminal cases are identified. Some terminal cases are induced by the decomposition. They are reported in lines 4-6 of Table (1). In line 4, we recognize whether `op1, op2` are of type

$$op1 = f + g..; \quad op2 = f' + h + \ldots.$$

In this case, `op1´, op2´` **contain** f, f' and we return 0. This is more general than just checking `op1 = op2´`. We also check whether one operand appears as a component of the other. Since scanning the two lists at each recursion step is expensive, only the first list elements are actually compared, after the two top variables have been determined.

	terminal case	return value
1	op1 = 1 or op2 = 1	**0**
2	op1 = 0 and op2 = 0	op2', op1'
3	op1 = op2	op1'
4	$\exists x \in$ `DSD(op1´)`; $x' \in$ `DSD(op2´)`	**0**
5	S(op1) ∩ S(op2) = $\emptyset$	{op1, op2}
6	op1 = $f(x, \ldots)$, op2 = x	$\{f_{x'}, x\}$

Table 1. Terminal cases for `NOR()`

In line 5, we detect if `op1, op2` have disjoint support. In this case, we create and return a 2-input NOR, with inputs pointing to `op1, op2`, respectively. Since this reduces to a test to the input parameter `i`, it takes constant time, regardless of the variable ordering. In line 6, we construct $f_{x'}$ (in time linear to the size of f, in the worst case) and return a NOR gate representing $(f_{x'} + x)'$.

Rows 2, 3 and 4 of `NOR` are the application of Theorem 4, case 1. `D(op)` indicates the set of elements of the decomposition of `op`. In a NOR vertex `op`, it is the set of all outgoing pointers (\ indicates set difference operation). We seek common elements in the operands and remove them from the recursion. This removal can result in faster execution, as the new operands have fewer variables. In practice, only the first element of the two lists is checked, for reasons of speed. After this removal, we check the operation to see if it reduces again to a terminal case.

A computed table maintains partial results. It is looked up in Line 6. The removal of common subfunctions reduces table overwrites, as we exploit the entry $\overline{F+G} = H$ for retrieving results of every operation of type $\overline{Ff + Gf} = Hf$. If the search fails, recursion starts.

```
    evaltop (mldd op, boolean value) {
1   if (op is a MUX node) return (op.value);
2   i = element of op such that op.topvar = op.i.topvar;
3   opr = evaltop(op.i, value);
4   D(op) = D(op) \ op.i;
5   D(res) = D(opr) ∪ D(op);
6   return (res); }
```

Figure 5. Pseudocode of `evaltop()`

Unlike ROBDDs, cofactoring may be nontrivial. Procedure `evaltop(f, value)` returns the cofactor $f_{x=value}$ assuming x is the top variable of f. After recursion, `mldd_find()` creates a MLDD from a `top_var` and its cofactors. We now analyze in more detail cofactoring and MLDD creation.

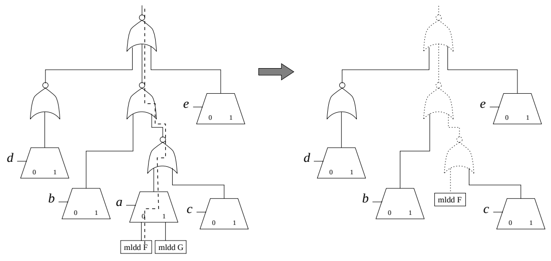

Figure 6. An example of `evaltop()` application

The pseudo-code of `evaltop` is in Fig. (5), and Fig. (6) illustrates its operation. `evaltop()` recursively goes down

```
   mldd_find(mldd left, mldd right, top_var x) {
1    if (left == right) return (left);
2    if (right == 0) {
3      new_vertex = find_or_create(0, 1, x);
4      D(res) = new_vertex ∪ D(right);
5      return(res);
     }
6    if (right == 1) { similarly }
7    if (left == 1 or left == 0) { symmetric case }
8    D(opc) = D(left) ∩ D(right);
9    if ( D(opc) = ∅ ) return( find_or_create(left, right, x) );
10   D(left) = D(left) \ D(opc);
11   D(right) = D(right) \ D(opc);
12   new_vertex = mldd_find(left, right, x);
13   D(res) = D(opc) ∪ new_vertex ;
14   return(res); }
```

Figure 7. Pseudocode of `mldd_find()`

the decomposition tree (Line 3) until it reaches the MUX node labeled with the top variable of the MLDD (Line 1). Since each gate contains the indication of the critical input with the top variable, only one path of the tree is traversed. Returning from recursion, it takes the cofactor of the reached MUX and substitutes NOR vertices with newly generated ones, (dotted in Fig. (6.b)) while maintaining canonicity. In practice, since often the graph of $f_{x=value}$ was created by a previous computation, new gates are rarely generated. Moreover, the recursion is typically very shallow (one or two levels in most benchmark examples).

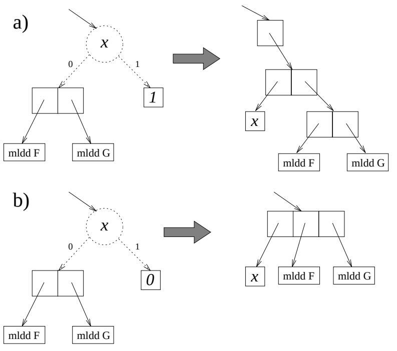

Figure 8. Identification of D during traversal - terminal cases

Procedure `mldd_find()` is responsible for discovering decompositions. Its pseudo-code is in Fig. (7). It builds a MLDD discovering every possible 'common term' from the two cofactors. It performs two distinct operations. First, it considers cases (rows 2 to 7) in which one of the two cofactors is a constant. For instance, rows 2 to 5 examine the case *right = 0, i.e.*, the function to generate is $f = x' \cdot \textit{left} = (x + l_1 + \cdots + l_n)'$, where l_i are the components of *left*. Fig. (8) shows these terminal cases.

Lines 8-13 deal with the general case. Common elements between left and right MLDD are factored out (Fig. (9)). This applies case 2 of Theorem 4.

As mentioned, `evaltop()` and `mldd_find()` replace cofactoring and `find_or_create()` in ROBDDs. While these operations are constant-time in ROBDDs, they may take $O(d)$ time in MLDDs, where d denotes the tree depth.

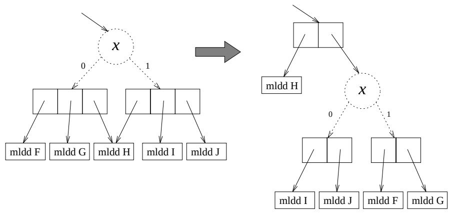

Figure 9. Identification of D during traversal - general case

To this regard, we observe that d is bound by the number of variables and it is rather small in practice.

Constant-time complementation
Complement edges (*i.e.*NOT gates) in ROBDDs allow us to represent f and f' with the same structure, and constant-time checking of equality $f = g'$ speeds up execution. NOT gates give rise to canonicity problems, as one may have a representation of g as NOT(f) and another representation rooted at a ROBDD node. this problem is solved in ROBDDs by applying appropriate rules when returning from recursion [9]. That approach is extended to MLDDs with the help of the following result:

Theorem 5. *If a logic function F is NOR-decomposable, then its complement F' is not.* □

From Theorem (5), we group functions in three classes: decomposable, with decomposable complement, and not decomposable. Our goal is to always represent functions of the second class as the NOT of a function of the first class. From Theorem (4), however, if F is in the second class, then its cofactors must be in that class as well. Hence, the situation prior to the call to `mldd_find` must be as in Fig. (10), and the standard complement edge reduction rules apply.

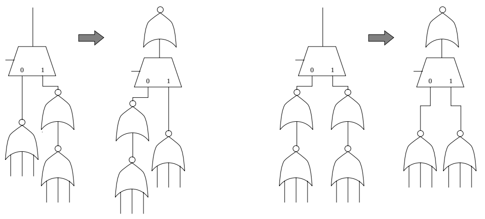

Figure 10. Equivalent MLDDs

5 MLDDs vs. ROBDDs

In this section we present some comparisons in representing functions with MLDDs and ROBDDs.

Exponential growth
MLDDs are less sensitive to variable ordering. Consider the functions:

$$F_n = (x_1 + x_2)(x_3 + x_4) \cdots (x_{2n-1} + x_{2n}) \qquad (4)$$

An improper ordering of the variables (for example, placing the odd-labeled variables up top) results in a ROBDD for F_n with over 2^n nodes [1]. Moreover, in spite of the simplicity of the function, most variable orderings for F_n can be proved bad. The MLDD of the function, instead, corresponds to the natural, 2-level NOR realization with n+1 NOR gates, and is of size linear in the number of variables, regardless of the variable order.

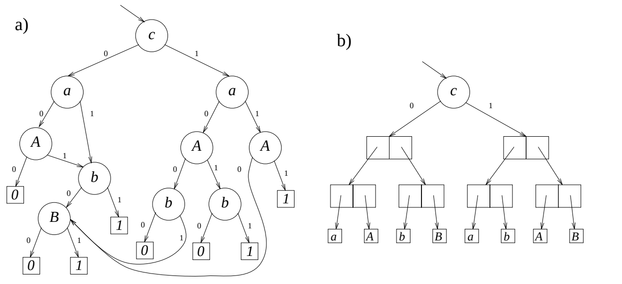

Figure 11. a) ROBDD structure for the function of Example (4). b) MLDD for the same function.

Example 4. Consider the function $f = (a + A)(b + B)c' + (a + b)(A + B)c$, with an ordering of variables placing c on top. Since $f_{c=0} \neq f_{c=1}$, any ROBDD has the aspect shown in Fig. (11.a). In general, we may think of a case where the two cofactors look like a function f_n of Eq. (4), but with a different combination of products. Any ordering of a, A, b, B which optimizes one branch is bound to be sub-optimal for the other branch of the ROBDD. Fig. (11.b) illustrates the MLDD for the same function. Both branches are automatically decomposed optimally. □

Sharing of logic.

Decomposition makes it possible to share blocks of logic that could not be shared with ROBDDs:

Example 5. Fig. (12.a) shows ROBDDs of functions: $F = (x'+y')(p+q)$, $G = (x'+y')(a+b)$, $H = (p+q)(a+b)$. If we have to represent those three simultaneously, whichever order we choose, we can share at most two subgraphs, either $(a + b)$ or $(p + q)$ or $(x' + y')$. This is because ROBDDs represent the AND of two disjoint-support functions putting one above the other. The MLDD representation, instead, can exploit maximal sharing. □ .

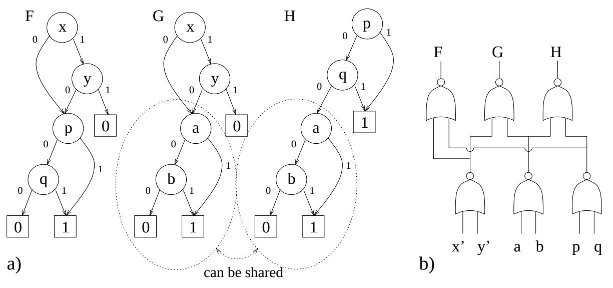

Figure 12. a) ROBDDs for Example (5). b) equivalent MLDDs

5.1 Experimental results

We compared MLDDs against ROBDDs on a number of benchmark circuits. Benchmarks are divided in three sections: multi-level circuits, two-level and the combinational part of synchronous circuits [21]. For the first set of tests, we used the Berkeley variable ordering [3], and no dynamic reordering took place in either package. We assumed bare-bone implementations, in which in particular each ROBDD node takes three machine words. Moreover ROBDDs have complement edges. For MLDD vertices we assumed an implementation where each node consists of an array. The first element stores node information, while others are pointers. Complement edges are used for NOT gates.

CPU-time was taken on a HP Vectra 5/133 with 48Mbytes of RAM. From Table (2), MLDDs turn out to be more compact on average of 18%. Some benchmarks give particularly good results. For example, *comp* and *pair* are decomposed very effectively. For decomposable functions, MLDDs often result also in a better CPU time, because term sharing can be used effectively. The largest benchmarks, however, resistant to decomposition, and in these cases MLDDs result in larger CPU time expenditure without a significant memory saving.

We implemented dynamic reordering in our model with a sifting-based algorithm [13]. Over ROBDDs, we have the advantage to know more about a ‘good variable order’ directly from the data structure.

In Table (2) we make comparisons using for each benchmark the order given by our sifting (interestingly, the final ordering differs from that given for ROBDDs.) Variable ordering took place only at the end of execution.

Results show that, after sifting, MLDDs improve slightly further over ROBDDs. This is because during sifting we exploit our better knowledge of the function’s structure and can avoid to go through orderings that give a small advantage but block further improvements.

6 Conclusions and future work

MLDDs have proved themselves efficient in making explicit the Ds of logic functions. This property allows us to reach a more compact, flexible and robust graph-based representation. Moreover, this representation is more informative on the role of the support variables of a function. We expect these properties to be useful in diverse applications, most notably technology mapping for combinational circuits and especially Boolean matching /reachability analysis for verification / ATPG in sequential circuits, where the ability of decomposing functions in simpler blocks is useful for drawing implications among next-state functions.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, 35(8):677–691, August 1986.

[2] O. Coudert and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proc. ICCAD*, pages 126–129, November 1990.

[3] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. ICCAD*, pages 6–9, November 1988.

[4] Y. Matsunaga and M. Fujita. Multi-level logic optimization using binary decision diagrams. In *Proc. ICCAD*, pages 556–559, November 1989.

[5] H. Touati, H. Savoj, B. Lin, R.K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD’s. In *Proc. ICCAD*, pages 130–133, November 1990.

[6] R. E. Bryant. Binary decision diagrams and beyond: enabling technologies for formal verification. In *Proc. ICCAD*, pages 236–243, 1995.

Benchmark	Berkeley ordering						After dynamic reordering					
	ROBDDs			MLDDs			ROBDDs			MLDDs		
	nod	mem	CPU	nod	mem	CPU	nod	mem	CPU	nod	mem	CPU
MULTILEVEL												
apex6	1171	3513	0.2	903	3377	0.1	669	2007	0.1	537	2147	0.1
C1355.iscas	45922	137766	4.8	44156	150231	6.8	30460	91380	2.8	30043	95622	3.8
C432.iscas	31178	93534	4.3	16147	82676	5.9	1300	3900	0.1	2478	11054	0.8
C499.iscas	45922	137766	3.8	44156	150231	5.6	30460	91380	2.2	30043	95622	3.0
C880.iscas	12841	38523	0.8	9173	31476	1.1	6969	20907	0.4	4294	14392	0.6
comp	5476	16428	0.2	434	1459	0.3	152	456	0.0	66	381	1.4
DES	31508	94524	6.7	28185	90660	8.8	9515	28545	2.2	8058	28898	2.8
example2.blif	869	2607	0.1	223	1362	0.0	646	1938	0.1	183	1061	0.0
k2	28336	85008	3.3	27437	86341	3.8	1426	4278	0.2	617	3796	0.3
pair	41128	123384	1.4	8053	26641	1.3	6032	18096	0.6	4283	15485	1.0
rot	12537	37611	1.1	7796	27463	2.8	7069	21207	0.6	3708	14185	1.5
too_large	7096	21288	2.1	4876	18153	5.6	1113	3339	0.9	578	2529	1.9
vda	4345	13035	0.2	4203	13235	0.3	534	1602	0.1	289	1648	0.1
TWOLEVEL												
apex5.pla	2679	8037	1.0	1088	5259	1.2	1440	4320	1.0	881	3935	1.2
e64.pla	1441	4323	0.4	66	2404	0.2	732	2196	0.1	66	2404	0.2
misex3.pla	1301	3903	1.8	814	3929	2.3	760	2280	1.8	185	1170	1.9
misex3c.pla	828	2484	0.2	552	2571	0.3	617	1851	0.1	201	1190	0.2
sao2.pla	155	465	0.0	48	319	0.1	133	399	0.0	45	289	0.0
vg2.pla	1044	3132	0.1	520	2429	0.1	414	1242	0.1	59	403	0.1
FSM												
ex1	769	2307	0.0	118	1785	0.1	639	1917	0.0	104	1568	0.1
s1196	3387	10161	0.2	2216	9523	0.3	806	2418	0.2	318	1862	0.1
s1238	3087	9261	0.2	2018	8998	0.3	813	2439	0.2	311	1812	0.1
s1494	654	1962	0.2	500	2072	0.2	468	1404	0.2	178	1156	0.2
s386	160	480	0.0	90	490	0.0	137	411	0.0	45	346	0.0
s400	180	540	0.0	96	511	0.0	148	444	0.0	91	467	0.0
s713	903	2709	0.1	228	1480	0.1	747	2241	0.1	173	1110	0.1
s820	409	1227	0.1	217	1178	0.1	280	840	0.1	90	664	0.1
s838	300	900	0.1	148	668	0.1	293	879	0.1	147	665	0.1
s953	474	1422	0.1	201	1302	0.1	498	1494	0.1	158	1081	0.1

Table 2. ROBDD vs. MLDD in size and performance

[7] F. Corno, P. Prinetto, and M. Sonza Reorda. Using symbolic techniques to find the maximum clique in very large sparse graphs. In *Proc. EDAC*, pages 320–324, March 1995.

[8] Y-T. Lai and S. Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *Proc. DAC*, pages 240–243, June 1992.

[9] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. DAC*, pages 40–45, June 1990.

[10] H. Ochi, K. Yasuoka, and S. Yajima. Bredth-first manipulation of very large binary decision diagrams. In *Proc. DAC*, pages 48–55, November 1993.

[11] K. M. Butler, D. E. Ross, R. Kapur, and M. R. Mercer. Heuristics to compute variable orderings for the efficient manipulation of binary decision diagrams. In *Proc. DAC*, pages 417–420, June 1991.

[12] S. J. Friedman and K. J. Supowit. Finding the optimal variable ordering for binary decision diagrams. *IEEE Trans. on Computers*, 39:710–713, 1990.

[13] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. ICCAD*, pages 42–47, November 1993.

[14] U. Kebschull, E. Schubert, and W. Rosentiel. Multilevel logic based on functional decision diagrams. In *EuroDAC, Proceedings of the European Conference on Design Automation*, pages 43–47, September 1992.

[15] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of of switching functions based on ordered kronecker functional decision diagrams. In *Proc. DAC*, pages 415–419, June 1994.

[16] Kevin Karplus. Representing boolean functions with if-then-else dags. Technical Report UCSC-CRL-88-28, Baskin Center for ComputerEngineering & Information Sciences, 1988.

[17] S.-I. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proc. DAC*, pages 272–277, June 1993.

[18] Y.-T.Lai, M. Pedram, and S. B. K. Vrudhula. Evbdd-based algorithms for ilp, spectral trnasform and function decomposition. *IEEE Trans. on CAD/ICAS*, 13(8):959–975, August 1994.

[19] V. Bertacco and M. Damiani. Boolean function representation using parallel-access diagrams. In *Sixth Great Lakes Symposium on VLSI*, March 1996.

[20] F. Mailhot and G. DeMicheli. Algorithms for technology mapping based on binary decision diagrams and on boolean operations. *IEEE Trans. on CAD/ICAS*, pages 599–620, May 1993.

[21] S. Yang. Logic synthesis and optimization benchmark user guide, version 3.0. *MCNC*, January 1991.