Manipulation of *BMDs

Rolf Drechsler
Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
drechsle@informatik.uni-freiburg.de

Stefan Höreth
Dept. of Electrical Engineering
Darmstadt University of Technology
64283 Darmstadt, Germany
sth@rs.tu-darmstadt.de

Abstract—Multiplicative Binary Moment Diagrams (*BMDs) have recently been introduced as a data structure for verification. Using *BMDs it was for the first time possible to verify multiplier circuits with up to 256 bits.

In this paper we use a modification of *BMDs, called positive *BMDs (p*BMDs), that allows to apply dynamic variable ordering, that is the most promising minimization technique for decision diagrams, to *BMDs. Furthermore, we study *BMDs representing Boolean functions. We show that in this case for some operations polynomial algorithms can be given, while the general case of integer-valued functions requires exponential effort.

Experimental results demonstrate that p*BMDs clearly outperform *BMDs with respect to runtime during dynamic minimization, while keeping (nearly) all advantages.

I. INTRODUCTION

Most formal approaches in verification nowadays make use of function representation by Decision Diagrams (DDs). In this context Ordered Binary Decision Diagrams (OBDDs) [4] have intensively been studied and frequently applied. Unfortunately OBDDs fail for some functions with high practical relevance, like multipliers [5]. For this, several extensions of the basic OBDD concept have been proposed over the last few years (see e.g. [19, 11, 9, 10, 15]).

Recently, a new DD type for representation of integer-valued functions, called Multiplicative Binary Moment Diagram (*BMD) [7], has been proposed. Using *BMDs it is possible to verify multipliers of large bit length. In the meantime several extensions have been proposed and *BMDs have been integrated in verification tools [1, 8]. In [18] a polynomial upper bound has been proven for verification of multipliers using *BMDs.

Since *BMDs are at least in principle similar to OBDDs the question turns out whether successful concepts known can be transferred. OBDDs as well as *BMDs are very sensitive to the variable ordering. The most promising approach for determining good orderings for OBDDs is based on dynamic variable ordering [13, 21].

In this paper we show that it is not possible to directly transfer dynamic reordering to *BMDs without big penalties. But we use a slight modification that restricts the root nodes of the *BMD to positive values. (For this we call the resulting data structure positive *BMDs (p*BMDs) [15].) By this restriction exchanging neighboring variables in a *BMD becomes a local operation. Experimental results underline the efficiency of our approach.

Furthermore, we have a closer look at (p)*BMDs representing Boolean functions. It turns out that in this case several operations known to be “difficult” can be solved efficiently, i.e. in polynomial time. This observation leads to the conclusions that a straightforward application of the standard (word-level) algorithms as proposed in [6, 7] can result in inefficient methods, while specialized algorithms can solve the problem simpler.

The paper is structured as follows: In Section II we review basic notations and definitions of DDs. The difficulties with using dynamic variable ordering for *BMDs and p*BMDs are described in Section III. (p)*BMDs representing Boolean functions are considered in Section IV. In Section V we discuss our experiments. Finally, the results are summarized.

II. PRELIMINARIES

All data structures considered in the following are graph-based representations, called in general Decision Diagrams (DDs). (For illustration see Figure 1.) For simplicity we focus less on a mathematical exact description. Instead we describe the main ideas. (For more details see [4, 11, 7].)

At each (non leaf) node v a decomposition of the function (represented by this node) into two (simpler) sub-functions (the low-function f_low(v) and the high-function f_high(v)) is performed. The decomposition is done with respect to a variable x, the resulting sub-functions are independent of x and finally, by recursive application of decompositions constant sub-functions are obtained. The function represented at a node v can be reconstructed by combining the sub-functions and edge values (if existent) according to the decomposition used at node v.
If in each node a (Boolean) Shannon decomposition
\[ f = \tau f_{\text{low}(v)} + x f_{\text{high}(v)} \]
is carried out the resulting DD is called Ordered Binary Decision Diagram (OBDD) [4]. (All data structures in the following are assumed to be ordered and reduced.) If in each node instead a positive Davio (pD) decomposition
\[ f = f_{\text{low}(v)} \oplus x f_{\text{high}(v)} \]
is performed this results in Ordered Functional Decision Diagrams (OFDDs) [1, 11]. OBDDs and OFDDs can only be used for Boolean function representation.

For representation of integer-valued functions Multiplicative Binary Moment Diagrams (*BMDs) [7] have been proposed. In *BMDs the values at the edges are multiplied with the functions represented by the node. Furthermore, a different decomposition is used. An edge with (integer) weight \( m \) to a node \( v \) in a *BMD represents the function
\[ m, f = m(f_{\text{low}(v)} + x f_{\text{high}(v)}). \]

Using the notation above we call the *BMD decomposition also the pD decomposition analogously to OFDDs.

Since the reduction of the graph sizes is an important issue in the following we describe it in more detail: In OBDDs (OFDDs) we simply remove all redundant nodes in the graph by identifying isomorphic subgraphs and by removing nodes where both outgoing edges point to the same node (where high pointers point to constant zero). By this reductions OBDDs and OFDDs become a canonical data structure for Boolean functions.

For *BMDs an additional restriction is needed to normalize edges. For this, the greatest common divisor is extracted. By this also *BMDs become a canonical representation for Boolean as well as integer valued functions.

It is known from [2]:

**Theorem 1** The OFDD for a Boolean function represented by a *BMD can be obtained by performing a modulo 2 operation in each node and reducing the resulting graph.

A simple consequence of this theorem is that the OFDD size is a lower bound with respect to number of nodes for the *BMD for Boolean functions.

**III. Dynamic Variable Ordering for *BMDs**

Dynamic Variable Ordering (DVO) [13, 21, 20, 11] is the state-of-the-art method for finding good orderings for OBDDs and OFDDs. The basic operation is an exchange of neighboring variables, that is a local operation in these cases. In this section we show that a direct extension of this concept to *BMDs is not possible, since the exchange of adjacent variables may change values in upper levels, i.e. it is not a local operation. (Notice that nodes in lower levels are not influenced.) We give a simple further restriction to the *BMD reduction rules described above, that solves this problem. Even though the modification seems to be small the improvements in running time are tremendous (see Section V).

**A. Difficulties**

Exchanging neighboring variables can easily be performed by redirecting pointers in the case of OBDDs (see Figure 2) and for this the basic operation of DVO is a local operation that only affects the two neighboring levels. A straightforward computation shows that the same holds for BMDs, but not for *BMDs, since values at the edges in upper levels might change. We give a small example to show the difficulties:

**Example 1** In Figure 3 two *BMDs for function \( f = x - 2y + 2xy \) under different variable orderings are given. As can easily be seen the root node of the two *BMDs differ, but the ordering of the left *BMD can be transformed to the ordering of the right *BMD by only swapping one pair of neighboring variables. Thus, DVO is not a local operation for *BMDs.
For this, in a straightforward approach the *BMD has to be traversed after each exchange and a reduction has to be carried out to retain a canonical representation. The runtime is given by the size of the graph, while the execution time for a local exchange is given by the number of nodes in this level that is normally much smaller (see also Section V).

### B. Restriction to positive Values

A more detailed analysis shows that these problems can be avoided, if we restrict the root nodes to positive values only [15]. (In the following we review the main concepts.)

A sufficient condition to make dynamic reordering an automated (background) process is that incoming edges to a sifted node must not change during variable swapping. If this condition holds, the user application can safely copy and manipulate graph pointers (edges) without interfering with the reordering mechanism.

For *BMDs, a function obtains its root label by computing the greatest common divisor \( \text{gcd} \) of the root labels of successor functions. Conversely, the function represented at an edge is obtained by multiplying the edge value \( m \) with the node function:

\[
f = m \cdot \left( \frac{f_{\text{low}}}{m} + x \cdot \frac{f_{\text{high}}}{m} \right).
\]

Starting from terminal values, the root label is obtained in a bottom-up manner. Since for *BMDs hold \( f_{\text{low}} = f_{x=0} \), \( f_{\text{high}} = f_{x=1} - f_{x=0} \), and

\[
m = \text{gcd}(f_{\text{low}}, f_{\text{high}}) = \text{gcd}(f_{x=0}, f_{x=1} - f_{x=0}) = \text{gcd}(f_{x=0}, f_{x=1})
\]

the root label of a *BMD can also be obtained from the Shannon expanded tree representation of the graph. As a consequence, the root label of a *BMD is simply the greatest common divisor of all function values. Its absolute value is independent of variable ordering and will not change during sifting.

However, the sign of the root label is obtained from the sign of the successor in the low direction. If \( f_{\text{low}} \) is the constant function zero, the sign is obtained from \( f_{\text{high}} \). Since \( f_{\text{low}} \) might change during sifting, the sign of the root label will probably change as well.

To overcome this limitation, it is sufficient to restrict root labels to positive values. By this modification, the size of the resulting graph is doubled at most. Multiplying a *BMD with a positive number is still a constant-time operation, but multiplication with negative numbers additionally requires to copy the graph and to negate terminal values.

**Example 2** The *BMDs for the function from Example 1 are given in Figure 4. As can easily be seen the value at the root node does not change by an exchange of variables.

A straightforward computation shows that this also holds in the general case. Thus, for *BMDs the exchange of neighboring variables is a local operation. In Section V it will be demonstrated by experiments, how tremendous runtime can be improved by the slight modification of the data structure described above.

### IV. Representation of Boolean Functions by *BMDs

While *BMDs have mainly been suggested for representation of integer-valued functions they can also be used as a data structure for Boolean function representation. In [7] it has been shown that for “several common Boolean functions” the *BMD has about the same size as the corresponding OBDD. In contrast in [2] several functions have been constructed for which an exponential trade-off between OBDDs and *BMDs exist.

Furthermore, the algorithms suggested for manipulation of *BMDs representing Boolean functions were derived from the manipulation algorithms for integer-valued functions (that have exponential worst case behavior).

In this section we show that in the case of *BMDs representing Boolean functions other algorithms can guarantee polynomial bounds.
Theorem 2 Let f be a function over Boolean variables.

1. Finding an x with f(x) = 0 is NP-hard.

2. If f only assumes Boolean values finding an x with f(x) = 0 can be done in polynomial time.

Proof:

1. Can be reduced from subset sum [6].

2. Let f be given as a *BMD. Then the corresponding OFDD representing the same function can be obtained by Theorem 1. For this, the graph has to be traversed and a modulo operation has to be computed locally followed by a reduction run. As can easily be seen all this can be done in time and space $O(|G|)$, where $G$ denotes the underlying graph of the *BMD. Then a satisfying assignment of the OFDD can be constructed in $O(n)$ by following the low edge until constant one is reached. If by following the low edge the path ends at a constant zero a backtrack of one step has to be performed and the high edge has to be followed. Due to the reduction rules of the OFDD it is easy to see that by this simple traversal method we end up at a node labeled with constant one. The variables along the path then describe a satisfying assignment.

Thus, we have an example where a “difficult” problem becomes easy for Boolean functions.

Theorem 3 Let $f, g \in B_n$ be two Boolean functions represented by *BMDs. $f \leq g$ and $f \cdot g = 0$ can be computed in polynomial time, respectively.

Proof: Again the proof follows from the reduction of the corresponding *BMDs to OFDDs and by application of the result in [12].

Notice that for $f \cdot g = 0$ the AND is not computed, since also the OFDD has exponential worst case behavior for this operation.

Our result shows that a “trivial” mapping of the Boolean operations to the word-level manipulation algorithms (as suggested in [7]) can lead to inefficient algorithms, while polynomial operations can be constructed.

V. Experimental Results

In this section we describe experiments carried out on a Sun UltraSparc-170 workstation with 256 MByte of main memory. All runtimes are given in CPU seconds. The DD sizes are always given by the number of nodes. We compare the size reduction that could be obtained by sifting p*BMDs and give a comparison to the runtime behavior of “classical” *BMDs. For all our experiments on dynamic minimization we use a reordering heuristic similar to [21].

In a first series of experiments we demonstrate the efficiency of our reordering method for p*BMDs. The results are given in Table I. The name of the benchmark is given in the first column. #in (#out) denotes the number of inputs (outputs) of the function. We first constructed an OBDD. The resulting size and the time needed for the construction is given in column Bit. Then the OBDD is transformed to a p*BMD by constructing a weighted sum over all outputs. (For more details see [7, 10, 15].) By this a p*BMD with a single output is obtained. The size of the resulting p*BMD and the time needed for the transformation are given in column Word. Then we applied our dynamic reordering algorithm to the p*BMDs. The resulting sizes and needed runtimes are reported in column Sift. As can easily be seen (analogously to OBDDs [21, 20]) the reduction in size can be up to 98%, while the runtime remains moderate, since we use p*BMDs, not *BMDs (see below). In the next two columns we give some more detailed informations on the sifting process:

1. #swaps denotes the number of exchanges of neighboring variables during the whole process. For the largest circuit more than 68000 exchanges are performed in less than 270 CPU seconds.

2. peak size gives the maximal number of nodes that were needed during the sifting process. This number is in most cases around 10% larger than the starting point.

The total running time is given in the last column. As can be seen the p*BMD sifting is very efficient. Also functions with more than 100 variables can be minimized very fast. (This becomes even more obvious, if we compare the performance of p*BMDs with “classical” *BMDs below.)

Next we repeated the experiment above, but instead of p*BMDs we also used *BMDs. For *BMDs each exchange of two neighboring variables is followed by an additional “repair run”, i.e. the graph is traversed and a reduction is carried out to remain canonicity (as described in Subsection A). The results are given in Table II. We only report the runtimes and sizes for the columns Word and Sift. As can be seen the initial size of the *BMDs is smaller than for p*BMDs. This can range up to a factor of two. But the runtimes needed for sifting of p*BMDs are much smaller. For functions with more than 100 variables the speed-up is larger than 60. E.g. for circuit 14 *BMD sifting took longer than 4 CPU hours, while reordering based on p*BMDs terminated in about 4 CPU minutes.

All in all, it has clearly been demonstrated that dynamic reordering based on p*BMDs obtains runtimes comparable to OBDDs, while “classical” *BMDs fail.
VI. Conclusions

Positive *BMDs result as a modification of the “classical” *BMDs. p*BMDs allow the application of exchange of neighboring variables as a local operation, while *BMDs have runtime linear in the size of the graph. By this, dynamic reordering methods can now be used as a background procedure.

Furthermore, we had a closer look at (p)*BMD manipulation algorithms. It turned out that in the case of Boolean functions more efficient algorithms can be constructed.

Experiments clearly showed that dynamic minimization based on exchange of neighboring variables can be done very efficiently for p*BMDs, while a straightforward method for *BMDs fails in most cases.

The method presented in this paper can also be extended to more general types of word-level DDs, like K*BMDs [10]. (For more details see [16].)

Recently, a new method for synthesis based on reordering has been proposed for OBDDs [14]. Based on the results presented in our paper it might now also be possible to extend these ideas to *BMDs.

References


<table>
<thead>
<tr>
<th>Circuit</th>
<th>Word size</th>
<th>p*BMD time</th>
<th>Sift size</th>
<th>*BMD size</th>
<th>Sift size</th>
<th>*BMD time</th>
</tr>
</thead>
<tbody>
<tr>
<td>alu4</td>
<td>4110</td>
<td>13s</td>
<td>2146</td>
<td>3.4s</td>
<td>3538</td>
<td>1.8s</td>
</tr>
<tr>
<td>apex1</td>
<td>3004</td>
<td>2.9s</td>
<td>1757</td>
<td>8.5s</td>
<td>1987</td>
<td>4.8s</td>
</tr>
<tr>
<td>apex2</td>
<td>18901</td>
<td>17.1s</td>
<td>1555</td>
<td>23.6s</td>
<td>9547</td>
<td>76.3s</td>
</tr>
<tr>
<td>apex3</td>
<td>1424</td>
<td>47.0s</td>
<td>887</td>
<td>12.5s</td>
<td>1142</td>
<td>55.7s</td>
</tr>
<tr>
<td>apex4</td>
<td>446</td>
<td>0.7s</td>
<td>446</td>
<td>0.4s</td>
<td>445</td>
<td>0.8s</td>
</tr>
<tr>
<td>apex6</td>
<td>1768</td>
<td>1.1s</td>
<td>551</td>
<td>43.4s</td>
<td>1157</td>
<td>1.5s</td>
</tr>
<tr>
<td>cps</td>
<td>907</td>
<td>2.3s</td>
<td>311</td>
<td>3.3s</td>
<td>488</td>
<td>3.5s</td>
</tr>
<tr>
<td>dalu</td>
<td>4739</td>
<td>5.4s</td>
<td>2072</td>
<td>20.2s</td>
<td>3186</td>
<td>8.5s</td>
</tr>
<tr>
<td>frg1</td>
<td>928</td>
<td>3.2s</td>
<td>354</td>
<td>4.7s</td>
<td>550</td>
<td>5.5s</td>
</tr>
<tr>
<td>frg2</td>
<td>10497</td>
<td>9.8s</td>
<td>747</td>
<td>60.6s</td>
<td>6310</td>
<td>14.1s</td>
</tr>
<tr>
<td>i3</td>
<td>88788</td>
<td>18.5s</td>
<td>378</td>
<td>93.8s</td>
<td>45453</td>
<td>47.9s</td>
</tr>
<tr>
<td>i4</td>
<td>168050</td>
<td>108.3s</td>
<td>1761</td>
<td>262.9s</td>
<td>87247</td>
<td>161.5s</td>
</tr>
<tr>
<td>i5</td>
<td>1955</td>
<td>0.5s</td>
<td>671</td>
<td>41.1s</td>
<td>1282</td>
<td>0.6s</td>
</tr>
<tr>
<td>i8</td>
<td>6466</td>
<td>8.6s</td>
<td>2598</td>
<td>56.4s</td>
<td>3725</td>
<td>10.2s</td>
</tr>
<tr>
<td>i9</td>
<td>5175</td>
<td>6.5s</td>
<td>1562</td>
<td>27.9s</td>
<td>2992</td>
<td>7.5s</td>
</tr>
<tr>
<td>k2</td>
<td>4048</td>
<td>4.0s</td>
<td>1824</td>
<td>8.1s</td>
<td>2876</td>
<td>6.5s</td>
</tr>
<tr>
<td>misex3</td>
<td>2008</td>
<td>1.2s</td>
<td>2287</td>
<td>3.5s</td>
<td>2175</td>
<td>1.4s</td>
</tr>
<tr>
<td>seq</td>
<td>2991</td>
<td>17.4s</td>
<td>1150</td>
<td>12.1s</td>
<td>1532</td>
<td>27.7s</td>
</tr>
<tr>
<td>table3</td>
<td>2308</td>
<td>2.1s</td>
<td>1525</td>
<td>2.4s</td>
<td>1379</td>
<td>2.6s</td>
</tr>
<tr>
<td>tooLarge</td>
<td>22241</td>
<td>11.0s</td>
<td>1809</td>
<td>24.4s</td>
<td>11126</td>
<td>28.9s</td>
</tr>
<tr>
<td>xl</td>
<td>4637</td>
<td>0.9s</td>
<td>1807</td>
<td>9.5s</td>
<td>2990</td>
<td>1.4s</td>
</tr>
<tr>
<td>x3</td>
<td>1579</td>
<td>1.3s</td>
<td>556</td>
<td>43.8s</td>
<td>1208</td>
<td>1.4s</td>
</tr>
</tbody>
</table>