In J. van Leeuwen (ed.), Computer Science Today
Lecture Notes in Computer Science Vol. 1000,
pages 218-233, Springer-Verlag, September 1995.

Differential BDDs*

Anuchit Anuchitanukul, Zohar Manna and Tomás E. Uribe

Computer Science Department
Stanford University
Stanford, California 94305
{anuchit|manna|uribe}@cs.stanford.edu

Abstract. We present a class of Ordered Binary Decision Diagrams, Differential BDDs (ΔBDDs), and transformations *Push-up* ($\uparrow$) and *Delta* (δ) over them. In addition to the ordinary node-sharing in normal BDDs, isomorphic substructures can be collapsed further in ΔBDDs and their derived classes, forming a more compact representation of boolean functions. The elimination of isomorphic substructures coincides with the repetitive occurrences of small components in many applications of BDDs. The reduction is potentially exponential in the number of nodes and proportional to the number of variables, while operations on ΔBDDs remain efficient.

1 Introduction

Binary Decision Diagrams (BDDs) [1] are a representation of boolean functions widely accepted as an important tool in many applications. Ordered Binary Decision Diagrams (OBDDs) [9] are BDDs with canonical forms and efficient manipulation algorithms, which make them the standard representation used in most applications of BDDs.

Many problems in digital system design, verification, optimization of combinatorial circuits, and test pattern generation, can be formulated as a sequence of operations on boolean functions. With the demand to apply OBDDs to increasingly larger problems, close attention has been paid to efficient implementation techniques [8], as well as refinements that improve speed and save memory. These include complement edges [19] and attributed edges [22] that have been shown to reduce the size of the graphs in many practical cases.

Numerous other variations of the basic OBDD idea have been proposed to suit the needs of particular applications. These include zero-suppressed BDDs [21], multi-terminal BDDs [14], multi-valued BDDs [16], edge-valued BDDs [18], BDD trees [20], functional BDDs [3, 5], generalized branching tests [17], domain transformations [6], BMDs [12], and first-order BDDs [23], among others.

* This research was supported in part by the National Science Foundation under grant CCR-92-23226, the Advanced Research Projects Agency under NASA grant NAG2-892, the United States Air Force Office of Scientific Research under grant F49620-93-1-0139, and the Department of the Army under grant DAAH04-95-1-0317.

We define and analyze a new class of BDDs, called Differential BDDs (ΔBDDs), and explore two transformations over them, Push-up ($\uparrow$) and Delta (δ). These transformations have the potential to significantly reduce the size of the graphs, while preserving canonicity and allowing for efficient operations.

We present efficient algorithms for manipulating the general class of $\uparrow^n \Delta$BDDs, derived from ΔBDDs by n push-up transformations. $\uparrow\Delta$BDDs (when $n = 1$) coincide with the class of BDDs with "variable shifter" attributed edges proposed by Minato *et. al.* [22], who show that they can produce more compact circuit representations.

Some BDD variants, such as extended BDDs [15, 2] and IBDDs [7], can give a polynomial-size representation of functions for which OBDDs are always exponential in the number of variables, but lose canonicity and efficiency of operations. The graph-driven BDDs of [24] retain canonicity and efficiency of most operations, but require finding a background oracle graph suitable for all the functions of interest. Similar results apply to the domain transformations of [6]. While our proposed variants cannot achieve such reductions, we believe that they are simple and general-purpose enough to be combined with many of the varieties of BDDs mentioned above to achieve further gains.

For a survey of BDDs and their applications, see [11]; a collection of recent BDD work, including further variants and refinements, is [4].

2 BDDs, OBDDs, and ΔBDDs

A Binary Decision Diagram (BDD) over a set of variables $\mathcal{V}$ is a directed acyclic graph (DAG) with two leaf nodes, *true* and *false* (also identified as **1** and **0**), where each internal node is labeled with a variable in $\mathcal{V}$ and has two outgoing edges, labeled 0 and 1. The successors of a node v are denoted by $S(v, 1)$ and $S(v, 0)$. A BDD defines the boolean function over $\mathcal{V}$ evaluated by following the unique path from the root to a leaf that chooses at each node the edge given by the value of the corresponding variable.

A BDD is *reduced* if (a) no two distinct internal nodes have the same label and the same successors, and (b) the successors of each node are distinct (no redundant tests). Every BDD has an equivalent reduced form.

In an *Ordered* BDD, there can be an edge from a node labeled with x_1 to a node labeled with x_2 only if $x_1 \prec x_2$ for a fixed total order $\prec$ over $\mathcal{V}$. Given such an order, the reduced OBDD representation of a boolean function f is *canonical*: there is exactly one reduced OBDD that represents f. Thus, we can identify a boolean function f with the corresponding root node. OBDDs are *shared* when a set of functions is represented with a single multi-rooted graph. An efficient implementation will maintain exactly one node (or pointer) for each distinct boolean function, so function equivalence can be tested in constant time.

The fixed order on the variables also allows implementing all the usual operations on boolean formulas with a complexity polynomial in the size of the argument OBDDs (see Section 5). The main practical concern, then, is the *size* of the OBDDs created for a particular task: as long as OBDDs remain small, operations remain efficient. In this, the particular variable ordering chosen can have a dramatic effect, in some cases determining if the OBDD size is linear or exponential in the number of variables (see [11]).

2.1 ΔBDDs

In a *Differential* BDD (ΔBDD), each node is labeled by an integer that represents a *displacement* in the fixed variable order. The label of a node v, $L(v)$, represents the difference between the index of the variable associated with v (a function of the path traversed so far) and that of its predecessors. For the leaf nodes, $L(\mathit{false}) = L(\mathit{true}) = 0$.

The value of the boolean function represented by a node v evaluated on a vector $\mathbf{x} = [x_1, \ldots, x_m]$ is defined as in the BDD case, except that the variable x_i branched upon at each node is given by the sum i of the node labels encountered so far. The label of the root node is the index of the first variable on which the BDD depends, and the sum of all labels on any path must be less than or equal to the total number of variables. As with reduced OBDDs, we require that any two internal nodes with the same label and the same successors be identical.

Fig. 1 shows the transition from an OBDD to a ΔBDD.[2] In the intermediate graph, variable labels are replaced by the corresponding displacement. For example, the label $L(vh) = 2$ is the distance between x_1 and x_3. The ΔBDD is the reduced form of this graph. The nodes $\{va, vb, vc, vd\}$ are merged into $v5$, $\{ve, vf\}$ into $v4$, and $\{vg, vh\}$ into $v3$. We may drop the 0 and 1 annotations on the edges and assume that the left and right branches of a node v always lead to its successors $S(v, 0)$ and $S(v, 1)$, respectively.

The ΔBDD representation can sometimes yield more nodes than the corresponding OBDD. Fig. 2 demonstrates such a case. In general, we have:

Proposition 1. *ΔBDDs have the following properties:*

1. *(Canonicity) Given a fixed variable ordering, each boolean function has a unique representation (canonical form) as a ΔBDD.*
2. *For a fixed variable ordering, the reduction in size when moving from an OBDD to a ΔBDD is potentially exponential: if n is the number of nodes in an OBDD, then (in the best case) the number of nodes of a ΔBDD representing the same boolean function can be $O(\log n)$. As a function of the number of variables m, the reduction is at best a factor of m.*
3. *In the worst case, the increase in the number of nodes is a factor of n^2 in the number of nodes n, and a factor of m in the the number of variables m.*

Proof Outline: It is easy to show that for any ΔBDD there is exactly one OBDD with the same ordering that represents the same boolean function. From the canonicity of OBDDs, we conclude that ΔBDDs are canonical as well.

[2] For clarity, the figures use multiple copies of the leaf nodes; reduced OBDDs will share the unique copy of each.

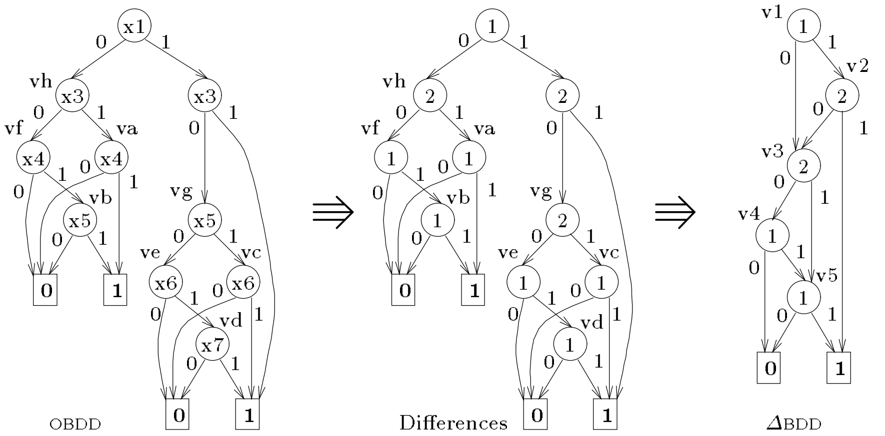

Fig. 1. From OBDD to ΔBDD

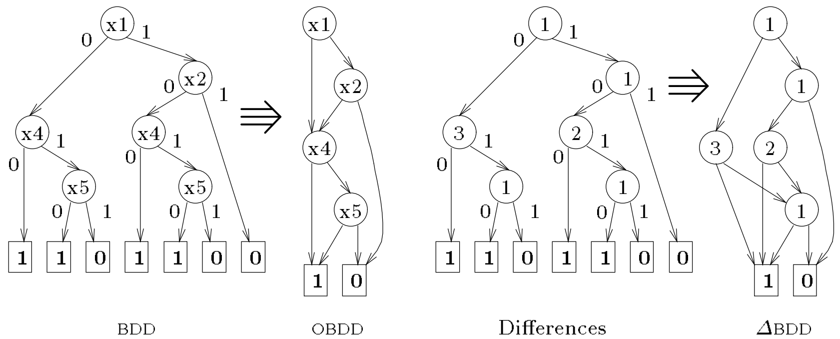

Fig. 2. A bad case for ΔBDDs

We can easily show that two distinct OBDD nodes labeled with the same variable are never reduced to a single ΔBDD node. With m variables, at most m nodes can be collapsed into one, so the reduction is at best a factor of m.

Exponential reduction occurs in the cases where (a) the left and right subgraphs of every OBDD node are isomorphic, and (b) if x_k is the label of a node in the left subgraph then the corresponding node in the right subgraph is labeled by x_{k+c} for some large enough c (*e.g.*, bigger than the index of any label in the left subgraph). In these cases, the left and right subgraphs will be merged together repeatedly when translated into ΔBDDs, and the size of the ΔBDD will be $O(\log n)$.

The only case where an OBDD node is expanded into several $\triangle$BDD nodes is when it has predecessors labeled with different variables: one extra node may be needed for each such label. There are at most n^2 pairs of nodes and predecessors, so the $\triangle$BDD size increase is $O(n^2)$ in the worst case. The factor of m bound follows from the fact that there are at most m different labels. ■

Boolean operations on $\triangle$BDDs can be computed in time polynomial in the size of the BDDs; the operations and their analysis are a special case of the general algorithms described in Section 5.

We now show that for any OBDD we can find a unique $\triangle$BDD of the same size (but that does *not*, in general, represent the same function). Since both representations are canonical, we conclude that the average $\triangle$BDD and OBDD sizes are equal, for a given variable ordering.

Proposition 2. *Given an ordered set of variables $\mathcal{V}$, there is a one-to-one mapping between the sets of* OBDD*s and* $\triangle$BDD*s over $\mathcal{V}$, such that each* $\triangle$BDD *is mapped to an* OBDD *with the same size and structure (*i.e.*, isomorphic up to relabeling of nodes).*

Proof Outline: Assume without loss of generality that the variable indices decrease from the root to the leaves. To construct an OBDD from a $\triangle$BDD, start from the leaves and move upward, adding to each node label the larger of the labels of its two successors. At the end, change each label n to x_n. To check that the result is an OBDD, we show that no duplicate nodes are created, by induction on the structure of the subgraphs, using the fact that there are no duplicate nodes in the original $\triangle$BDD. This construction is reversible, so the one-to-one correspondence follows. ■

In the next section we introduce the $\uparrow$ transformation, which is applied to $\triangle$BDDs to produce $\uparrow\triangle$BDDs. These inherit the potentially exponential reduction property and polynomial-time boolean operations, but are guaranteed to have no more nodes than the corresponding OBDD.

3 The $\uparrow$ Transformation

The *Push-up* ($\uparrow$) transformation takes a DAG with n labels on each node[3] and pushes one of the labels back through the incoming edges, into the predecessors of the node.

A binary DAG with n-ary labels, $\mathcal{G} = (\mathcal{N}, S, L)$, is a set of nodes $\mathcal{N}$, a successor function $S : \mathcal{N} \times \{0,1\} \mapsto \mathcal{N}$, and a labeling function L that maps each node $v \in \mathcal{N}$ to an n-tuple label $L(v) = [d_0, \ldots, d_{n-1}]$. For each node $v \in \mathcal{N}$, we write $d(v,j)$ for the j-th component of the label of v. Given a binary DAG with n-ary labels $\mathcal{G}$, the transformation $\uparrow_i$ (or simply $\uparrow$ if $i = 0$) transforms $\mathcal{G}$ in two steps:

[3] Or, alternatively, an n-ary label in each node; this distinction will always be clear from context.

1. Relabel each node v with $[d_0, \ldots, d_{i-1}, d_{i+1}, \ldots, d_{n-1}, d_{s0}, d_{s1}]$, where $d_{s0} = d(S(v,0), i)$ and $d_{s1} = d(S(v,1), i)$. If v has no successor (i.e., it is a sink node), then $d_{s0} = d_{s1} = 0$.
2. Merge all the nodes with the same successors and the same label.

The result is a binary DAG with labels of length $n+1$ that represents $\mathcal{G}$. From now on, we focus on the case where $i = 0$, $\uparrow$. Fig. 3 shows the result of applying $\uparrow$ once ($\uparrow\Delta$BDD) and twice ($\uparrow^2\Delta$BDD) to a ΔBDD. The nodes in each dashed box are merged into one node on the right. Unlike OBDDs and ΔBDDs, where a function is represented by a single node, in $\uparrow\Delta$BDDs a function is identified by a node *and* an integer, which is the by-product of the $\uparrow$ transformation.

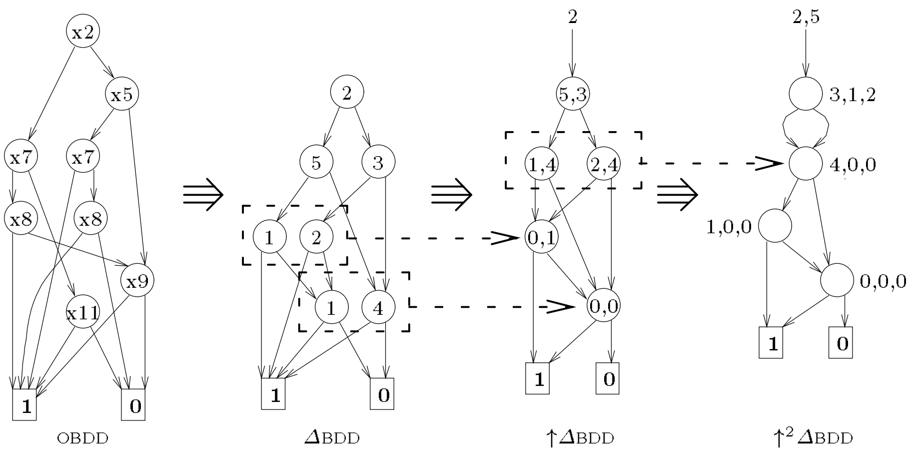

Fig. 3. Push-up transformation

In general, if $\mathcal{C}$ is a class of BDDs, then $\uparrow\mathcal{C}$ is the class of BDDs obtained from a push-up operation on the elements of $\mathcal{C}$.

Proposition 3. *Each application of $\uparrow$ to a class of* BDD*s has the following properties:*

1. *The number of nodes for each* BDD *is reduced up to a factor of m, where m is the number of variables.*
2. *Canonicity is preserved.*

Proof. The factor of m reduction is clear when we observe that no new nodes are created and that the labels of two collapsed nodes can differ only in their first element. For the canonicity property, observe that the $\uparrow$ transformation is reversible. Therefore, two different BDDs cannot be transformed into the same graph. If the original class is canonical, so is the derived class. ■

Furthermore, we will see in Section 5 that boolean operations on the derived class of BDDs remain polynomial in the size of the component BDDs.

Even though the $\uparrow$ transformation offers a best-case reduction of m, it introduces one additional label per node. Therefore, to save memory the following condition must hold:

$$\frac{n_{elim}}{n} > \frac{1}{1 + \frac{|l|}{|dx|}}, \text{ where :}$$

- n is the total number of nodes before the transformation.
- n_{elim} is the number of nodes eliminated by the transformation.
- $|l|$ is the total bit-length of all the labels and pointers associated with each node before the transformation. For example, for OBDDs and ΔBDDs, $|l| = \log(m) + 2b$, where m is the number of variables and b is the bit-length of a pointer.
- $|dx|$ is the number of bits needed to hold the additional label for each node, which should, in fact, be $\log(m)$.

Two main arguments support using the $\uparrow$ transformation:

1. $|dx|$ is usually small since there is usually not a large number of variables. On the other hand, $|l|$ is relatively large since pointers could be 32- or 64-bit long. Thus, $1 + |l|/|dx|$ will usually be of the order of $5, 6, 7, 8\ldots$, and we only need to eliminate a small fraction of nodes to save space.
2. Some unused bits may be used to hold the additional label. For example, if a word in the system is 32 bits long, but we only need 16 bits per label (*i.e.*, there are fewer than 65,536 variables), then we should apply $\uparrow$ and keep the additional label in the unused bits. If this is the case, $\uparrow$ will never increase memory usage, since it will never increase the number of nodes.

We also need to consider the space overhead in the *result cache* (see Section 5): each node in a cache entry will have an extra integer associated with it. However, this cache is usually of a fixed size, and the overhead incurred is proportional to it; alternatively, the absolute cache size can be held constant, at the expense of some efficiency loss. As above, no extra space is needed if the new label can be fitted into the existing words. Note that the *unique table* causes no additional storage overhead if its entries are pointers back to the original nodes.

4 $\uparrow\Delta$BDDs

Starting with $\uparrow\Delta$BDDs, $\uparrow^n\Delta$BDDs are guaranteed to have no more nodes than the corresponding OBDDs:

Proposition 4. *The number of nodes in a $\uparrow\Delta$BDD is always less than or equal to the number of nodes in an OBDD for the same boolean function with the same variable ordering.*

Proof Outline: We can show that each node in an OBDD is translated into at most one node in a $\uparrow\Delta$BDD by induction on the structure of subgraphs. In the inductive case, consider any OBDD node v with successors $S(v,0)$ and $S(v,1)$. By the induction hypothesis, both $S(v,0)$ and $S(v,1)$ are translated into unique $\uparrow\Delta$BDD nodes v'_0 and v'_1. Therefore, the node v is translated into the unique node v' such that v'_0 and v'_1 are the successors of v', and whose label is uniquely determined by the labels of v, $S(v,0)$ and $S(v,1)$. ■

Recalling the analysis in the previous section, this property means that space savings relative to OBDDs can be more frequently obtained, even when accounting for the extra node label. The reduction in the number of nodes is potentially exponential, since the possible savings in the original ΔBDD are retained.

In applications such as combinational circuit representation, eliminating isomorphic structures in the BDD representation is a significant advantage. Many circuits contain identical components that cause their OBDD representation to have substructures that differ only in variable labeling (the input signals to each component). If the relative variable positions in each component are the same, the substructures will be collapsed into a single $\uparrow\Delta$BDD. Note that partitioned and hierarchical representations facilitate such structure sharing.

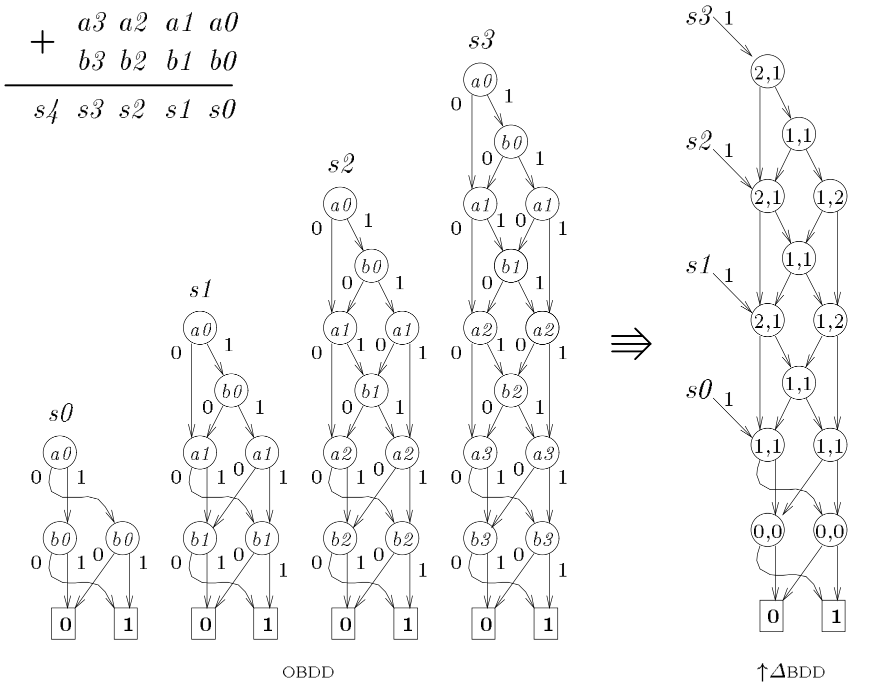

Fig. 4. OBDD and $\uparrow\Delta$BDD for a 4-bit adder (suboptimal variable ordering)

Fig. 4 shows the $\uparrow\Delta$BDD representation of a four-bit adder. The two operands are $[a_3, a_2, a_1, a_0]$ and $[b_3, b_2, b_1, b_0]$, and the variable ordering is $[a_0, b_0, \ldots, a_3, b_3]$. This ordering is not the optimal one for standard OBDDs, which is in fact the reverse one. Under the optimal ordering, the OBDD size is linear in the number of bits, as opposed to the quadratic case of Fig. 4. However, the $\uparrow\Delta$BDD produces a minimal-size (linear) BDD for both orderings.[4] This example suggests that $\uparrow\Delta$BDDs can give some extra flexibility in choosing good orderings, an important problem in the practical use of BDDs.

The BDDs with "variable shifter" edges presented by Minato *et.al.* [22] are an alternate presentation of $\uparrow\Delta$BDDs, where the two node labels are associated with the outgoing edges rather than the node itself.

5 Boolean Operations

We now present the main algorithm used to efficiently compute boolean operations on $\uparrow\Delta$BDDs and derived classes. As in [8], we can implement all binary operations using a single operator, $ite(f, g, h)$, which takes as input the BDD representation of boolean functions f, g, h and returns that of the function

$$ite(f, g, h)(\mathbf{x}) \stackrel{\text{def}}{=} \text{if } f(\mathbf{x}) \text{ then } g(\mathbf{x}) \text{ else } h(\mathbf{x}) \enspace .$$

OBDDs are based on the *Shannon expansion* of boolean functions:[5] for any function f,

$$f \equiv \text{if } x \text{ then } f_x \text{ else } f_{\overline{x}},$$

where f_x and $f_{\overline{x}}$ are the *restrictions* of f (also called *cofactors*) when the variable x is assigned the value 0 or 1, respectively. Given procedures to compute f_x and $f_{\overline{x}}$, the *ite* function can be implemented recursively as follows:

$$ite(f, g, h) = assemble\,(x, ite(f_x, g_x, h_x), ite(f_{\overline{x}}, g_{\overline{x}}, h_{\overline{x}}))\ ,$$

where x is the first variable in the fixed ordering appearing in f, g or h, and $assemble(x, f_1, f_2)$ returns the unique OBDD node labelled with x that has f_1 and f_2 as its right and left successors. To ensure the uniqueness of each OBDD, *assemble* checks a hash table, called the *unique table*, for the existence of such a node before creating a new one. If $f_1 = f_2$, then *assemble* returns f_1, avoiding the redundant test on x.

Choosing x to be the smallest of all variables in f, g, h makes the operations f_x and $f_{\overline{x}}$ particularly easy to compute: if x labels the root node of f, we follow the left or right successor; otherwise, f does not depend on x, and the operation is the identity. In the remainder of this section, we reserve f_x and $f_{\overline{x}}$ for such restriction operations (*i.e.*, under an extra assumption on f and x); the general restriction algorithm is described in Section 5.2.

[4] For the optimal ordering, the $\uparrow\Delta$BDD is still smaller (see Section 7).

[5] Some BDD variants [3, 5, 12] use other expansions, but the push-up transformation should be compatible with most.

In the case of $\uparrow\Delta$BDDs, a boolean function f is identified by a pair $\langle o, N\rangle$, for an integer o, which we call the *offset* of f, and a node N. The constant functions *true* and *false* have an offset of 0. We need to define an appropriate *assemble* function, as well as redefining the operators $f_x, f_{\overline{x}}$ for an integer offset x.

When computing *ite*, we take x to be the smallest offset of the three arguments. If f is the function $\langle o, N\rangle$ for an offset o and a node N, then

$$f_x = \langle o, N\rangle_x = \begin{cases} \langle o - x, N\rangle & \text{if } x < o \\ \langle r, S(N,1)\rangle & \text{if } x = o \end{cases}$$

where r is the second label of N, and $S(N, 1)$ is the right descendant. The case of $f_{\overline{x}}$ is analogous. These operations are illustrated in Fig. 5. Note that the resulting BDD should be interpreted relative to the offset x in all cases.

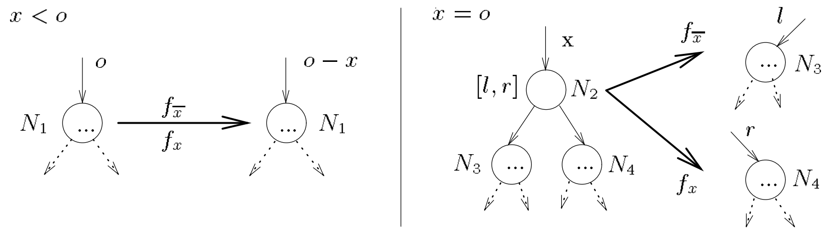

Fig. 5. $f_x, f_{\overline{x}}$ for $\uparrow\Delta$BDDs

The recursive calls to *ite* return two functions, $f_1 = \langle o_r, N_r\rangle$ and $f_2 = \langle o_l, N_l\rangle$, which are combined with x to produce the final result as follows:

$$assemble(x, \langle o_r, N_r\rangle, \langle o_l, N_l\rangle) = \begin{cases} \langle o_r + x, N_r\rangle & \text{if } f_1 = f_2 \\ \langle x, N'\rangle & \text{otherwise} \end{cases}$$

where N' is the node labeled with $[o_l, o_r]$ whose left and right successors are N_l and N_r respectively.

Like [8], we use a second hash table, the *result table*, to avoid multiple evaluations of *ite* on each triple $\langle f_1, f_2, f_3\rangle$. Furthermore, triples can be normalized so that equivalent operations (including those equivalent modulo some initial offset) are mapped to a single table entry. We assume a good implementation of the hash tables where operations, on average, can be done in constant time. Let n_a, n_b, n_c be the size, in nodes, of the arguments, and m be the number of variables over which the functions range. There are $n_a n_b n_c m$ unique tuples to be evaluated, so the complexity of the algorithm is $O(n_a n_b n_c m)$.

Although the algorithm seems to have a higher complexity than that for normal OBDDs, which is $O(n'_a n'_b n'_c)$ in the sizes of the corresponding OBDDs, this is not a disadvantage. First, recall that the graphs are usually larger for

OBDDs. Second, there is a one-to-one (but not always onto) mapping from each evaluation tuple $\langle f_1, f_2, f_3 \rangle$ to a tuple $\langle v_1, v_2, v_3 \rangle$ where v_1, v_2, v_3 are nodes in the OBDD arguments. This means that the number of recursive calls to *ite* for $\uparrow\Delta$BDDs is less than or equal to the number of recursive calls for OBDDs that represent the same boolean functions. In essence, switching to $\uparrow\Delta$BDDs will never introduce any unnecessary computation.

This implementation is compatible with further refinements such as the use of complement edges for constant-time negation [19, 8].

5.1 $\uparrow^n\Delta$BDDs

In the general case of $\uparrow^n\Delta$BDDs, n push-up operations have been done. A function f is now identified by a pair $\langle l, N \rangle$, where l is a tuple of length n, called the *tag* of f, and N is a BDD node with a label of length $n+1$. The first element of the tag l is again called the *offset* of f, and comes from the root of the original ΔBDD. The constant functions *true* and *false* have an all-zero tag.

Assume that the $n+1$ components of each node label are marked indicating if they come from the left or right successor in the push-up operation[6]. We can do the same for the components of each function tag. Let o be the offset of f; we write $[o : l]$ for the tag of f. Assuming $x \leq o$, f_x is defined as follows:

$$f_x = \langle [o : l], N \rangle_x = \begin{cases} \langle [(o-x) : l], N \rangle & \text{if } x < o \\ \langle l', S(N, 1) \rangle & \text{if } x = o \end{cases}$$

where l' is obtained by appending the elements of l that are marked as coming from the right with the corresponding elements in the label of N. The $f_{\overline{x}}$ operation is defined analogously.

We also need to construct the $\uparrow^n\Delta$BDD obtained from branching on an offset x, leading to the functions (relative to this offset) $f_1 = \langle l_1, N_1 \rangle$ and $f_2 = \langle l_2, N_2 \rangle$. We interleave the elements of l_1 and l_2 and then split the resulting list into two parts: the first, l, of length $n-1$, and the second, l', of length $n+1$. Then

$$\mathit{assemble}(x, \langle l_1, N_1 \rangle, \langle l_2, N_2 \rangle) = \langle [x : l], N' \rangle,$$

where N' is the node with l' as its label, and N_1 and N_2 as its right and left successors.

The *ite* operation is now defined exactly as before, again choosing x to be the smallest offset. Note that in all cases $f = \mathit{assemble}(x, f_x, f_{\overline{x}})$, and that in the differential cases f_x and $f_{\overline{x}}$ give a result relative to the offset x.

5.2 Restriction

Another important BDD operation is the general restriction operation. Given a boolean function $f(x_1, \ldots, x_n)$, this operation fixes a variable x_j to be equal

[6] In our notation, if n is even, the left components are in the odd-numbered positions (starting from 0); if n is odd, the reverse applies.

to c, for $c \in \{0,1\}$, yielding a function of $n-1$ variables that we write as $restrict(f,j,c)$.

Restriction is used to "quantify" over boolean variables, and is commonly used, for example, in model checking of finite-state systems [13]. $\exists x_j f$ is identified with $restrict(f,j,0) \vee restrict(f,j,1)$, while $\forall x_j f$ is $restrict(f,j,0) \wedge restrict(f,j,1)$. Restriction can be computed efficiently for $\uparrow\Delta$BDDs in a way similar to standard OBDDs. We define the procedure recursively: let o be the offset (the first element of the tag) of f. If $j < o$, the restriction is the identity:

$$restrict(f,j,1) = restrict(f,j,0) = f \quad .$$

If $j = o$,

$$restrict(f,j,1) = add\_offset(j, f_j) \quad \text{and} \quad restrict(f,j,0) = add\_offset(j, f_{\overline{j}}),$$

where $add\_offset(j,f)$ adds j to the offset of f. If $j > o$, we recurse on the left and right cofactors and then put the result back together:

$$restrict(f,j,X) = assemble(o, restrict(f_o, j-o, X), restrict(f_{\overline{o}}, j-o, X)) \quad .$$

6 The δ Transformation

There are variations of the $\uparrow$ transformation that also exploit regularity in the graph to reduce the number of nodes. One such variation is the *Delta* (δ) transformation. As demonstrated in Fig. 6, the δ transformation takes a DAG with at least two integer labels per node, computes the difference between two of the labels, pushes one of the two labels up and keeps the difference.

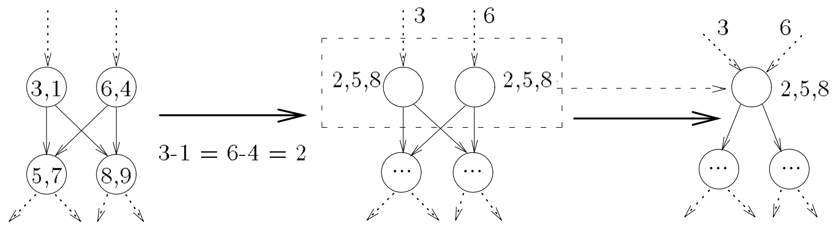

Fig. 6. δ transformation

Fig. 7 shows the application of δ to an entire $\uparrow\Delta$BDD. Like the $\uparrow$ transformation, δ preserves the essential properties of canonicity and efficient manipulation, and gives a best-case reduction proportional to the number of variables. $\delta\uparrow\Delta$BDDs also have the property that all OBDD nodes with the same successors are collapsed into a single $\delta\uparrow\Delta$BDD node.

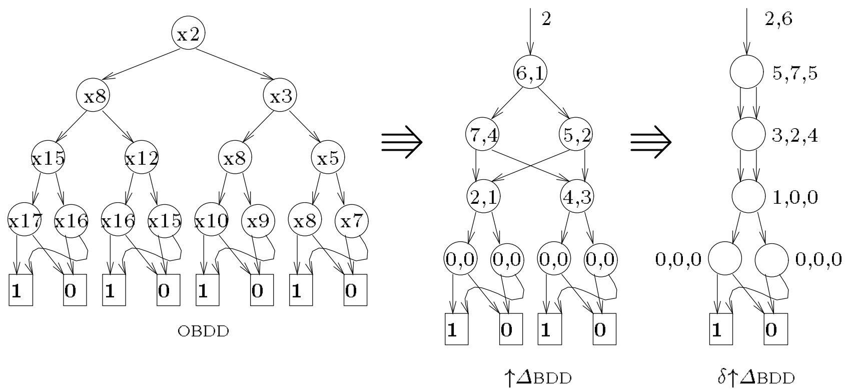

Fig. 7. From OBDD to $\delta{\uparrow}\Delta$BDD

It is easy to adapt the previous algorithms to a class of BDDs derived by a δ transformation. Fig. 8 describes the computation of the immediate restriction functions, f_x and $f_{\overline{x}}$, for $\delta{\uparrow}\Delta$BDDs. Fig. 8 also describes the construction of $assemble(x, f_1, f_2)$ for $f_1 \neq f_2$; if $f_1 = f_2$, then x is added to the first element of the tag of f_1 and the resulting pair is returned. Given f_x, $f_{\overline{x}}$ and *assemble*, the *ite* and *restrict* operations are defined exactly as in Section 5.

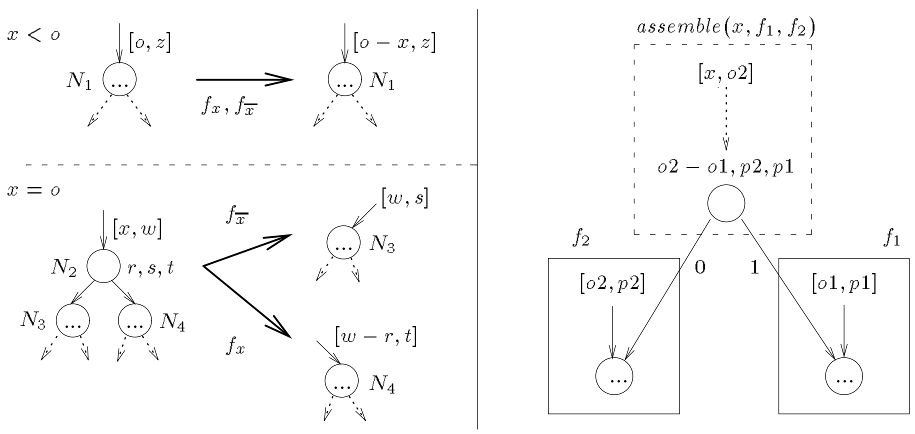

Fig. 8. f_x, $f_{\overline{x}}$ and $assemble(x, f_1, f_2)$ for $\delta{\uparrow}\Delta$BDDs

7 Experimental Results

Table 1 shows the number of OBDD nodes needed to build the representation of a 128-bit adder, together with the percentage of nodes *removed* by ΔBDD variants for 4 different variable orderings. In this example, the δ transformation does not improve the size of the BDD representation; a second push-up transformation increases the node savings for the two best variable orderings. Note that even ΔBDDs display the more dramatic reductions, for the two worst orders. In most cases, the representation for each individual output bit has the same size across the BDD variants; the savings come from *sharing* among the BDD representation of the 128 output functions (see Section 4).

Table 1. Nodes for 128-bit adder (and % OBDD nodes removed)

Order	OBDD	ΔBDD	$\uparrow\Delta$BDD	$\delta\uparrow\Delta$BDD	$\uparrow^2\Delta$BDD
optimal	1,538	2,048 (-33%)	1,154 (25%)	1,154 (25%)	1,025 (33%)
sub-1	3,647	4,839 (-33%)	3,314 (9%)	3,314 (9%)	3,056 (16%)
sub-2	196,737	5,587 (97%)	3,058 (98%)	3,058 (98%)	3,057 (98%)
sub-3	388,888	206,618 (47%)	201,862 (48%)	201,732 (48%)	201,477 (48%)

Of the satisfiability and verification problems discussed in [25], ΔBDDs and their variants were most useful in problems where OBDDs already had some success. Table 2 presents results for problems *not* generally amenable to BDD encoding, which yielded the smallest reductions: random satisfiability problems and multiplication circuits. Note, however, that the savings increase when a set of multipliers is encoded, rather than a single one (3- through 8-bits, in this case; input variables are shared).

Table 2. Hard problems for OBDDs

Problem	OBDD	ΔBDD	$\uparrow\Delta$BDD	$\delta\uparrow\Delta$BDD	$\uparrow^2\Delta$BDD
random1	3,033	4,024 (-33%)	2,909 (4%)	2,908 (4%)	2,789 (8%)
random2	3,480	4,604 (-32%)	3,382 (3%)	3,278 (6%)	3,268 (6%)
mult.8	134,401	134,700 (0%)	130,295 (3%)	128,678 (4%)	129,337 (4%)
mult.3–8	188,820	162,960 (14%)	155,632 (18%)	151,470 (20%)	154,313 (18%)

For other standard combinational circuits, such as ALUs, node savings were between 45% and 80%, of which a second δ and $\uparrow$ transformation contributed up to 10%. However, it was often the case that for one of the transformations the number of nodes remained essentially the same.

Our implementation followed [8] (see Section 5) and included complement edges. The $\uparrow\Delta$BDD, $\delta\uparrow\Delta$BDD, and $\uparrow^2\Delta$BDD variants were generally slower than the original OBDDs, due to the overhead in hash-table and *ite* operations caused by the extra labels. However, in most BDD applications it is space, not time, that presents the main limitation. The time overhead was never more than a factor of 3 (and can be reduced by a more careful implementation), and was offset by the smaller BDD sizes in the cases where the best savings were obtained.

A number of boolean functions, including multipliers, have an exponential-size OBDD (in the number of variables m) for *any* ordering of the variables [10]. Since the $\uparrow$ and δ transformations can only reduce the OBDD size by a factor of m, these problems will still be intractable. However, we hope that the δ and $\uparrow$ transformations can also be used to improve the efficiency of the BDD variants specialized to have small representations for these cases [6, 24].

8 Conclusion

The $\uparrow$ and δ transformations on ΔBDDs are general-purpose techniques to reduce the number of nodes in a class of BDDs, potentially reducing memory usage and speeding up operations. These techniques present a trade-off between the number of nodes and the number of labels per node, but we show that there can be a net gain. The transformations can be performed until the space required to hold the extra labels offsets the space saved by the reduction of nodes. In some environments, such as a machine with long words, unused bits can be utilized to store additional labels.

ΔBDDs give some measure of abstraction over the identity of particular boolean variables. The best space savings occur when representing related functions over different sets of variables. It would be interesting to investigate if these properties can be exploited in particular applications, such as symbolic model checking [13], including new encoding and variable ordering methods.

Acknowledgements

We thank Anca Browne, Nikolaj Bjørner, Henny Sipma, Mark Stickel and Howard Wong-Toi for their helpful comments.

References

1. Akers, S. Binary decision diagrams. *IEEE Transactions on Computers 6*, 27 (June 1978), 509–516.
2. Ashar, P., Ghosh, A., and Devadas, S. Boolean satisfiability and equivalence checking using general binary decision diagrams. In *IEEE International Conf. on Computer Design: VLSI in Computers and Processors* (1991), pp. 259–264.
3. Bani-Eqbal, B. Exclusive normal form of boolean circuits. Tech. Rep. UMCS-92-4-1, Department of Computer Science, University of Manchester, 1992.
4. Becker, B., Bryant, R., Coudert, O., and Meinel, C., Eds. *Report on the Third Workshop on Computer Aided Design and Test* (Feb. 1995). `http://www.informatik.uni-trier.de/Design_and_Test/`.
5. Becker, B., Drechsler, R., and Theobald, M. OKFDDs versus OBDDs and OFDDs. In *International Colloquium on Automata, Languages and Programming* (July 1995), LNCS. To appear.

6. Bern, J., Meinel, C., and Slobodová, A. Efficient OBDD-based boolean manipulation in CAD beyond current limits. In 32^{nd} *Design Automation Conf.* (1995).
7. Bitner, J., Jain, J., Abadir, M., Abraham, J., and Fussell, D. Efficient algorithmic computation using indexed BDDs. In 24^{th} *Intl. Symposium on Fault-Tolerant Computing* (1994), pp. 266–275.
8. Brace, K. S., Rudell, R. L., and Bryant, R. E. Efficient implementation of a BDD package. In 27^{th} *Design Automation Conf.* (1990), pp. 40–45.
9. Bryant, R. E. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers 35*, 8 (Aug. 1986), 677–691.
10. Bryant, R. E. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. on Computers 40*, 2 (Feb. 1991), 205–213.
11. Bryant, R. E. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys 24*, 3 (Sept. 1992), 293–318.
12. Bryant, R. E., and Chen, Y.-A. Verification of arithmetic circuits with Binary Moment Diagrams. In 32^{nd} *Design Automation Conf.* (June 1995). To appear.
13. Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. Symbolic modelchecking: 10^{20} states and beyond. *Information and Computation 98*, 2 (June 1992), 142–170.
14. Clarke, E. M., McMillan, K., Zhao, X., Fujita, M., and Yang, J. Spectral transforms for large boolean functions with applications to technology mapping. In 30^{th} *Design Automation Conf.* (June 1993).
15. Jeong, S., Plessier, B., Hachtel, G., and Somenzi, F. Extended BDD's: Trading off canonicity for structure in verification algorithms. In *IEEE Intl. Conf. on Computer-Aided Design* (1991), pp. 464–467.
16. Kam, T. Y. K., and Brayton, R. K. Multi-valued decision diagrams. Technical Report UCB/ERL M90/125, University of California, Berkeley, Dec. 1990.
17. Karplus, K. Using if-then-else DAGs for multi-level logic minimization. In *Advance Research in VLSI* (1989), C. Seitz, Ed., MIT Press, pp. 101–118.
18. Lai, Y.-T., and Sastry, S. Edge-valued binary decision diagrams for multi-level hierarchical verification. In 29^{th} *Design Automation Conf.* (1992), pp. 608–613.
19. Madre, J., and Billon, J. Proving circuit correctness using formal comparison between expected and extracted behaviour. In 25^{th} *Design Automation Conf.* (June 1988), pp. 205–210.
20. McMillan, K. L. Hierarchical representations of discrete functions, with application to model checking. In *Proc.* 6^{th} *Intl. Conf. on Computer-Aided Verification* (June 1994), vol. 818 of *LNCS*, Springer-Verlag, pp. 41–54.
21. Minato, S. Zero-supressed BDDs for set manipulation in combinatorial problems. In 30^{th} *Design Automation Conf.* (1993), pp. 272–277.
22. Minato, S., Ishiura, N., and Jajima, S. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In 27^{th} *Design Automation Conf.* (1990), pp. 52–57.
23. Possega, J., and Ludäscher, B. Towards first-order deduction based on Shannon graphs. In *Proc.* 16^{th} *German Conf. on AI (GWAI-92)* (1992), vol. 671 of *LNAI*, Springer-Verlag, pp. 67–76.
24. Sieling, D., and Wegener, I. Graph driven BDDs—a new data structure for boolean functions. *Theoretical Comput. Sci. 141*, 2 (Apr. 1995).
25. Uribe, T. E., and Stickel, M. E. Ordered binary decision diagrams and the Davis-Putnam procedure. In *Proc.* 1^{st} *Intl. Conf. on Constraints in Computational Logics* (Sept. 1994), vol. 845 of *LNCS*, pp. 34–49.