COMPUTING BOOLEAN EXPRESSIONS WITH OBDDs

by

Thomas R. Shiple, Robert K. Brayton, and
Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M93/84

10 December 1993

COMPUTING BOOLEAN EXPRESSIONS WITH OBDDs

by

Thomas R. Shiple, Robert K. Brayton, and
Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M93/84

10 December 1993

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Computing Boolean Expressions with OBDDs

Thomas R. Shiple Robert K. Brayton
Alberto L. Sangiovanni-Vincentelli
Department of EECS, University of California, Berkeley, CA 94720

December 10, 1993

Abstract

We present a method to compute the BDD for an arbitrary Boolean expression, where the operands are themselves BDDs. Such expressions are usually computed by the successive application of binary operators. However, cases exist where this method performs wasteful intermediate computations and creates BDD nodes not used in the final result. In contrast, our method never creates a BDD node unless it is present in the final result. We tested the new method on the application of building BDDs for the nodes in a multi-level logic network. Although the new method uses fewer BDD nodes, its runtime is much worse. By analyzing the reasons for this, we further our understanding of BDD operations.

1 Introduction

A reduced ordered binary decision diagram (BDD) is a data structure used to store and manipulate Boolean functions [2]. Researchers are applying BDDs to a growing number of problems, such as formal design verification, logic synthesis, and graph-theoretic problems. Forming new BDDs by applying Boolean operators to existing BDDs, is at the heart of most BDD-based applications, so any technique to reduce the memory and time required for these operations will have an impact on such applications.

Boolean operations on BDDs are performed by applying Shannon's expansion recursively. For example, to compute $f(x_1,\ldots,x_n)\cdot g(x_1,\ldots,x_n)$, where x_1 occurs first in the BDD variable ordering, $f_{x_1}\cdot g_{x_1}$ and $f_{\overline{x_1}}\cdot g_{\overline{x_1}}$ are recursively computed, and then combined by forming $x_1\cdot f_{x_1}\cdot g_{x_1}+\overline{x_1}\cdot f_{\overline{x_1}}\cdot g_{\overline{x_1}}$.

In the BDD packages with which we are familiar, complicated Boolean expressions are formed by successively applying binary operators (e.g. AND, OR, XOR). The problem with this approach, called the *binary method*, is that unnecessary runtime and memory may be consumed in building intermediate expressions that are not explicitly used in the final result.[1] For example, to construct $(f\cdot g)+h$, the term $f\cdot g$ is built first, and then combined with h using OR. In the case where h is

[1]The problem of intermediate memory consumption is somewhat alleviated by the garbage collection of dead BDD nodes (see Section 4).

ONE (i.e. the constant one),[2] the work in building $f \cdot g$ is obviously wasted, since the final result is just ONE.

A second example illustrating the weakness of the binary method is the expression $f_1 \cdot f_2 \cdot \ldots \cdot f_k$. Since there are k operands, $k-1$ binary AND operations are needed. However, the order in which the operands are processed can drastically affect the runtime and memory consumption, since the recursion on a binary AND terminates as soon as one of the operands becomes ZERO. Thus, the sooner a ZERO branch is found at some point in the BDD for *one* of the operands, the sooner it is known that the corresponding branch need not be explored in *any* of the other operands. An extreme case occurs when f_i is ZERO for some i; obviously, it is best to discover this as early as possible. Hence, when using just binary operators, we may be faced with the difficult problem of ordering the operands to improve efficiency in time and space.

By exploiting implicit don't cares in an expression, the binary method can be extended to overcome some of its limitations. For example, when computing $(f \cdot g) + h$, each of f and g can be simplified using h as a don't care set.[3] That is, for any minterm where h is ONE, the final result will be ONE on that minterm regardless of the value of f or g. Thus, we can use a heuristic, such as *restrict* [3], to minimize each of f and g (with respect to the don't care set h) before forming their product. In the special case where h is ONE, f and g will each be simplified to constants, thus making their product trivial. However, cases exist where *restrict* actually increases the size of f and g, thus exacerbating the problem.

We propose a method, the *multi-way method*, to avoid unnecessary intermediate computations and the problem of ordering operands. The method is motivated by the observation that binary Boolean operators in BDD packages, by themselves, do not perform wasteful intermediate computations nor create unnecessary nodes. We extend this feature to arbitrary Boolean expressions, including cofactoring by a cube, by recursively applying Shannon's expansion to all the operands in an expression. Therefore, the multi-way method does not create any intermediate nodes—a node is created only if it is present in the final result.

To illustrate the multi-way method, again consider the example $(f \cdot g) + h$. If x_1 is the top variable, then $(f_{x_1} \cdot g_{x_1}) + h_{x_1}$ and $(f_{\overline{x_1}} \cdot g_{\overline{x_1}}) + h_{\overline{x_1}}$ are recursively computed and then combined to form

$$x_1((f_{x_1} \cdot g_{x_1}) + h_{x_1}) + \overline{x_1}((f_{\overline{x_1}} \cdot g_{\overline{x_1}}) + h_{\overline{x_1}}).$$

At any step in the recursion, if the third operand (the "h" operand) becomes ONE, then the recursion terminates and returns ONE from that step.[4] In this case, f and g are not explored beyond this point.

In Section 2, an overview of the algorithm is given, and then each part is discussed in detail. We assume that the reader is already familiar with BDDs. Section 3 offers analysis and discussion of the algorithm, and Section 4 provides some preliminary experimental results. Section 5 presents

[2] It may be unlikely that h is ONE at the top-level of recursion, but this can easily occur at some intermediate step in the recursion.

[3] The full don't care set for f is $\overline{g} + h$, and for g is $\overline{f} + h$. However, there is an ordering problem, since these don't care sets cannot be used independently.

[4] Of course, there are other terminating conditions, as will be explained later.

conclusions and future work.

2 Algorithm

2.1 Overview

The input to the algorithm is a Boolean expression.

Definition 1 *Boolean expression* is defined inductively:

1. Any BDD f is a Boolean expression.
2. If f_1 and f_2 are Boolean expressions, then NOT(f_1), AND(f_1, f_2), OR(f_1, f_2), and XOR(f_1, f_2) are Boolean expressions.
3. If f is a Boolean expression, and BDD c is a cube, then COF(f, c) is a Boolean expression (f cofactored by a cube).

A Boolean expression can be represented by a tree with internal nodes labeled by NOT, AND, OR, XOR, and COF. The leaves of the tree are *operands.*

The outline of our approach for computing the BDD for a Boolean expression follows:

1. Input processing
   (a) Convert the user's input into a tree representing the Boolean expression to be computed.
   (b) Create a new Boolean expression by substituting each operand in the original expression by a new, auxiliary variable, and build the BDD for this new expression using standard binary operators (cofactors are ignored at this point). This BDD is used to test for terminating conditions in the recursion, as explained below.
   (c) For each operand, record by which variables it is cofactored. This information is stored with each operand, and is used when traversing the operand BDDs to decide which branches to follow.
2. Core routine
   Recursively apply Shannon's expansion to all the operands of the expression. When an operand becomes a constant during the recursion, evaluate its corresponding auxiliary variable in the terminal condition BDD. When the variable support of the terminal condition BDD falls to one or zero variables, this signifies a terminating condition of the recursion. Store the result of each recursive step in a cache to avoid recomputing the same intermediate expression more than once.

2.2 Input Processing

The user creates two arrays to represent a Boolean expression. The first, the "expression" array, is an array of elements from the set {OPRND, AND, OR, NOT, XOR, COF} representing the expression in postfix notation. For example, the expression

$$(\overline{f_1} \cdot f_2) \oplus \overline{(f_3 + f_4)}_{f_5}$$

is represented by the array

[OPRND, NOT, OPRND, AND, OPRND, OPRND, OR, NOT, OPRND, COF, XOR].

The second, the "operand" array, is an array of BDDs corresponding to the OPRND elements of the expression array (e.g. $[f_1, f_2, f_3, f_4, f_5]$ for the above example). In general, we use k to denote the length of the operands array. The operands belong to the *principal* BDD manager.

The **first** input processing step is to combine the information in the expression and operand arrays into a single tree, where each node of the tree is an operator and each leaf is an operand. Each operand is associated with an auxiliary variable in a new variable space.[5] Thus, the leaf for operand f_i contains two pieces of information: a pointer to the BDD for f_i, and a pointer to the BDD for the corresponding auxiliary variable, denoted by y_i. Figure 2 shows the expression tree for the expression $((f \cdot g)_{x_2} + h)_{\overline{x_4}}$.

The **second** step is to build the terminal condition BDD, working from the expression tree. This BDD, denoted by TC, is defined over the auxiliary variables $y_1, \ldots, y_k$. TC is built by the recursive procedure *build_terminal_condition*, shown if Figure 1. Note that COF operations are ignored when building TC—cofactor information is accounted for separately, as explained shortly. As an example, TC for $((f \cdot g)_{x_2} + h)_{\overline{x_4}}$ is $(y_1 \cdot y_2) + y_4$ (y_3 corresponds to the operand x_2, and y_5 corresponds to $\overline{x_4}$).

As a preview, TC is used in the core recursive procedure to detect terminal conditions of the recursion. A copy of TC is made at each step of the recursion,[6] and the copy is evaluated on those auxiliary variables corresponding to operands which just became constants in the current step. That is, if operand f_i becomes, say ONE, then TC is evaluated at $y_i = 1$. For example, if h becomes ONE in the expression $(f \cdot g) + h$, then TC $= (y_1 \cdot y_2) + y_3$ evaluated at $y_3 = 1$ yields TC $= 1$. This is a terminal condition, and indicates that the value of the user's expression on this branch is just ONE. The use of TC is further explained in Section 2.3.

The **last** step in processing the input is to gather the cofactor information. Figure 2 shows the cofactor information for the expression $((f \cdot g)_{x_2} + h)_{\overline{x_4}}$. For each operand, we record by which literals to cofactor the operand. For example, in the above expression, both f and g are cofactored by the cube $x_2 \cdot \overline{x_4}$. This is done by again traversing the expression tree: whenever a COF node is reached, the literals present in the cofactor cube are noted. This information is recorded for every operand within the scope of the cofactor. It is illegal to cofactor an operand by the same variable in

[5] The auxiliary variables are stored and manipulated in a separate BDD manager so that BDD operations on the auxiliary variable space can be counted separately from those on the principal variable space.

[6] Copying is a constant time operation in a BDD package that uses a strong canonical form [1].

```
function build_terminal_condition(expression tree) {
  switch (type of node) {
  case OPRND:
    return (auxiliary variable corresponding to operand);
  case AND:
    left = build_terminal_condition(left child of expression);
    right = build_terminal_condition(right child of expression);
    return (bdd_and(left, right));
  case OR:
    left = build_terminal_condition(left child of expression);
    right = build_terminal_condition(right child of expression);
    return (bdd_or(left, right));
  case XOR:
    left = build_terminal_condition(left child of expression);
    right = build_terminal_condition(right child of expression);
    return (bdd_xor(left, right));
  case NOT:
    return (bdd_not(build_terminal_case(child of expression));
  case COF:
    return (build_terminal_condition(child of expression being cofactored));
  }
}
```

Figure 1: Procedure for building the terminal condition BDD.

opposite phases (e.g. x and $\overline{x}$). Since the cofactor information may be identical for many operands (such as for f and g above), only one copy of each distinct pattern of cofactor variables is stored. The number of distinct patterns is bounded by the number of COF operators in the expression. An array is used to store a pattern; the length is the number of variables in the principal BDD manager, and the ith entry contains two bits encoding whether variable x_i is a cofactor variable, and if so, in which phase.[7]

2.3 Core Routine

The BDD for a Boolean expression is computed by recursively applying Shannon's expansion to the entire expression, taking into account the cofactor operations. The major steps of this recursive routine are:

1. If a terminating condition is satisfied, then return the corresponding terminal value. The terminal value will be either the constant ONE or ZERO, or a sub-BDD of one of the original operands.
2. If the same expression is found in the cache, then return the previously computed result.
3. Get the *then* and *else* branches of the operands, subject to the cofactor information of each operand.
4. Recursively build the BDD for the *then* operands and for the *else* operands.
5. Create a new BDD node to combine the results from step 4 using Shannon's expansion.
6. Insert the expression and result of step 5 into the cache. Return the result of step 5.

Each of these steps will be discussed in detail. Figure 3 shows the pseudo-code for the core routine.

The core routine works by performing a simultaneous depth-first search (DFS) on all the operands. At each step of the recursion, the user's expression must be computed using the sub-BDDs at the current step as the operands of the expression (i.e. this process is just Shannon's expansion). Two pieces of information are carried along and updated on the recursive descent. The first is a set of flags indicating which operands have "fallen" to constants. The second is the TC BDD, evaluated on those auxiliary variables corresponding to operands that have fallen to constants. For example, if the ith operand is currently a constant, say ZERO, then TC is evaluated at $y_i = 0$.

Upon taking a recursive step, the **first** task of the core routine determines if a terminating condition has been reached. This is done by evaluating the local copy of TC on the auxiliary variables corresponding to those operands that just became constants on this recursive step (a *bdd_cofactor* is used to perform this evaluation).

[7] Actually, it is not necessary to gather cofactor information and to assign an auxiliary variable for those operands corresponding to the cofactoring cubes. As a minor improvement, the code should be changed to reflect this.

original expression: $((f \cdot g)_{x_2} + h)_{\overline{x_4}}$

expression array: [OPRND, OPRND, AND, OPRND, COF, OPRND, OR, OPRND, COF]

operand array: $[f, g, x_2, h, \overline{x_4}]$

terminal condition TC: $(y_1 \cdot y_2) + y_4$

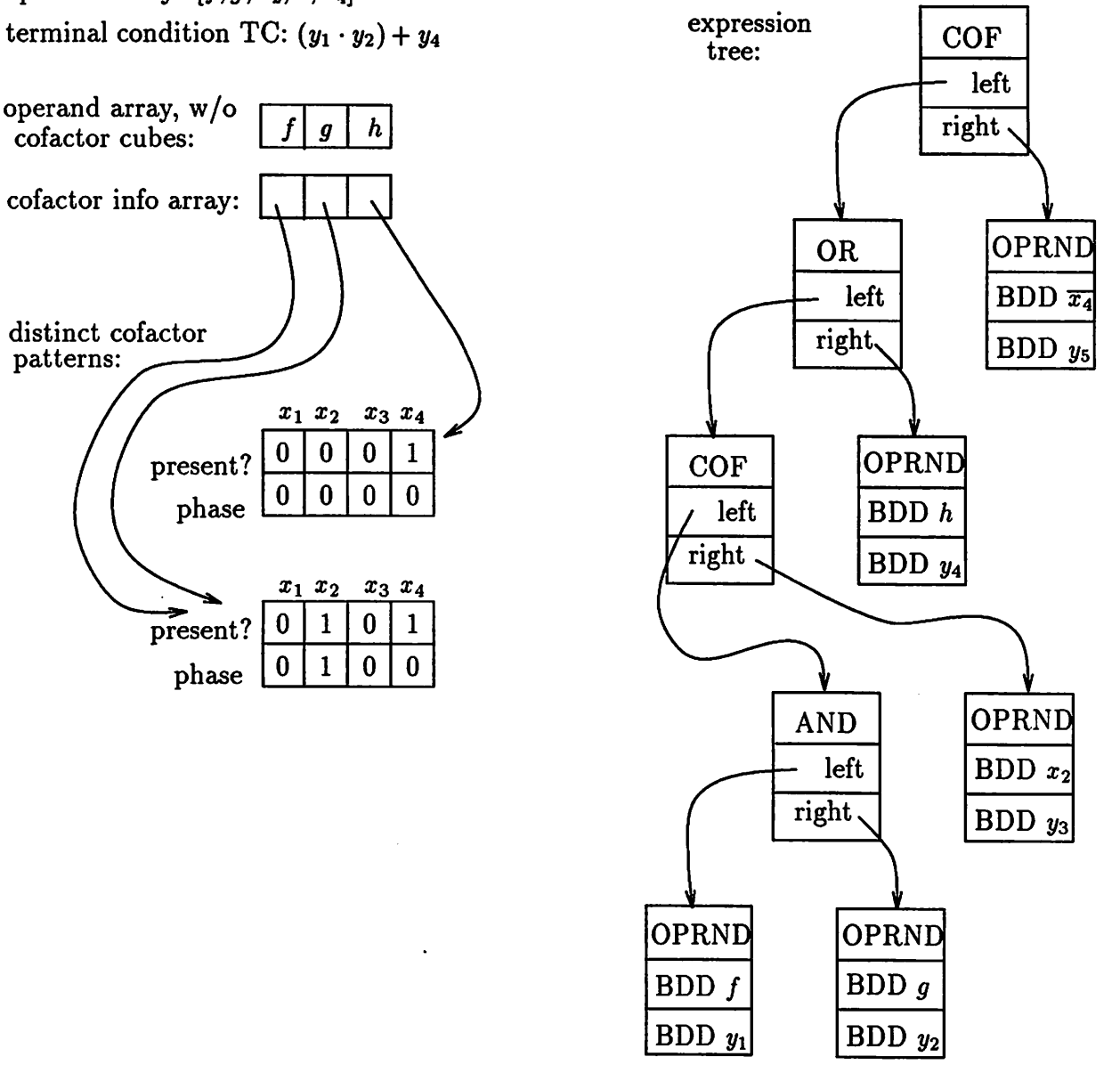

Figure 2: Expression tree and cofactor information for $((f \cdot g)_{x_2} + h)_{\overline{x_4}}$. Note that the functions f, g, and h are defined over x_1, x_2, x_3, and x_4.

```
bdd_node *
function compute_expression(operands, TC, cofactor_info, constant_flags)
{
      /* update constant_flags and TC, based on the input operands arrary */
     new_constant_flags = update_constant_flags(operands, constant_flags);
     new_TC = update_TC(operands, new_constant_flags, TC);

      /* Step 1:  test for terminal conditions */
     if (test_terminal_case(operands, cofactor_info, new_TC, &result)) {
        return result;
     }

      /* Step 2:  check if this operands array has already been seen */
     if (cache_lookup(operands, &result)) {
        return result;
     }

      /* Step 3:  get the then and else branches of each operand, subject */
      /* to the value of top_id and the cofactor info */

      /* First, determine the minimum of the top variable IDs of the operands */
     top_id = min_i{top_var(operand_i → id)};

      /* Next, based on the value of top_id and the cofactor_info, get the */
      /* appropriate then and else branches for each node in the operands array */
     get_branches(top_id, operands, &then_nodes, &else_nodes, cofactor_info);

      /* Step 4:  recursively compute the then and else results */
     then_result = compute_expression(then_nodes, new_TC, cofactor_info, new_constant_flags);
     else_result = compute_expression(else_nodes, new_TC, cofactor_info, new_constant_flags);

      /* Step 5:  combine the then and else results to create a node representing */
      /* the function indicated by the input operands array */
     result = ITE(top_id, then_result, else_result);

      /* Step 6:  insert the result into the cache and return the result */
     cache_insert(operands, result);
     return result;
}
```

Figure 3: Psuedo-code for the core routine.

If TC is a now a constant, then the value of the expression on this recursion path is just the value of TC. If TC is not a constant, then the variable support of TC is determined.[8] If only one auxiliary variable remains in the support, then we simply return the corresponding operand in the proper phase, as the value of the expression. (There is an exception to this: if that operand still must be cofactored by variables not yet reached in the recursion, then we cannot terminate the recursion.) If more than one variable remains in the support of TC, then a terminal condition has not been reached yet. However, to improve efficiency, we further analyze the variable support of TC. If an auxiliary variable corresponding to a non-constant operand has dropped out of the support, then that implies that that operand no longer has an effect on the value of the expression. We replace such operands by a constant to avoid needlessly applying Shannon's expansion to them. For example, if the expression is $(f_1 \cdot f_2) + (f_3 \cdot f_4)$ and f_1 becomes ZERO, then TC evaluated at $y_1 = 0$ is $y_3 \cdot y_4$. Thus, TC no longer depends on y_2, indicating that it is no longer necessary to recurse on f_2.

This method of testing for terminal conditions misses some obvious terminal conditions. For example, recursion does not terminate on the expression $(f \cdot \overline{f}) + h$, because the TC function is $(y_1 \cdot y_2) + y_3$, which does not satisfy any of the above conditions. As another example, the expression $f \cdot f$ is missed, since TC is $y_1 \cdot y_2$. These sorts of checks could be added, although the tradeoff is unknown between the time needed to perform the checks and the time saved in further recursion. Note that it is not sufficient to detect these sorts of pathological cases in the original user's expression, because they may occur at any step of the recursion.

If a terminal condition has not been reached, then in the **second** step of the core routine, the cache is checked to see if this expression has been previously computed. The cache maps an array of operands to a BDD representing the value of the user's expression on these operands. Note that a key of the cache is just an array of operands—there is no reference to the operators of the expression. Thus, it is assumed that all entries in the cache correspond to the same user's expression. For this reason, the cache must be flushed after the core routine finally returns and the user's expression has been fully computed. This is in contrast to the ITE cache of a standard BDD package, which persists between user's calls to ITE.

The cache is represented by an open hash table (i.e. each bucket contains at most one entry). Given an array of operands (BDD pointers), the hash function sums the elements of the array, after first shifting each element by a varying number of bits.[9] The operand array $[f_1, f_2, \ldots, f_k]$ is considered to match the array $[g_1, g_2, \ldots, g_k]$, thus achieving a cache hit, only if $f_i = g_i$, for all i. Note that this implies $(f \cdot g) + h$ and $(g \cdot f) + h$ do not match.

Assuming that the result was not found in the cache, the **third** step determines the appropriate BDD branches of each operand for the succeeding recursive calls. First, we compute the topmost BDD variable ID, *top_id*, of all the operands. For example, if the variables are ordered $x_1, \ldots, x_n$, and there are three operands, with topmost variables x_2, x_5, and x_4, respectively, then *top_id* is 2. Next, the *then* and *else* branches for each operand f are determined based on *top_id* and the

[8] To improve efficiency, we could implement a specialized cofactor routine that also returns the variable support of the resulting BDD.

[9] It may be worth experimenting with different hash functions, as the performance of the overall algorithm strongly depends on the cache hit rate.

cofactor information. Figure 4 shows the pseudo-code fragment for doing this.

```
f_id = get top variable ID of f;
if (f_id > top_id) {
    /* f is below top_id, so don't split f yet */
  then = f;
  else = f;
} else {  /*  f_id == top_id */
  if (f cofactored by top_id) {
     if (cofactor is in positive phase) {
          /* follow the positive branch for both succeeding recursive calls */
        then = f_{top_id};
        else = f_{top_id};
     } else {  /* cofactor is in negative phase */
          /* follow the negative branch for both succeeding recursive calls */
        then = f_{~top_id};
        else = f_{~top_id};
     }
  } else {  /* f not cofactored by top_id, so just split on top_id */
     then = f_{top_id};
     else = f_{~top_id};
  }
}  /* it's not possible that f_id < top_id */
```

Figure 4: Procedure for determining the *then* and *else* branches for recursion for operand f.

Thus, from the input operands array, we have created two new operand arrays representing the *then* and *else* branches. The **fourth** step recursively builds the BDD for the expression represented by each of these two arrays.

The **fifth** step creates a new node whose variable ID is *top_id*, and whose *then* and *else* pointers are the results from the recursive calls in step 4. This node may already exist in the BDD manager, and if so, it is used.

The **sixth** and last step simply creates a new entry for the cache, where the entry's key is the array of operands in this recursive step, and the entry's value is the BDD node created in step 5. Finally, the core routine returns the BDD node created in step 5, to be used at the previous level of recursion to build up the BDD of the final result.

We illustrate the core routine on the expression $(f \cdot g)_{x_2} + h$ (this is the same expression used earlier, except that the cofactor by $\overline{x_4}$ has been dropped to make the computation more interesting). Figure 5 gives the BDDs for the operands f, g, h, and the result of the expression. Figure 6 shows the recursive calls to the core routine. The first column shows which branch (of the Shannon cofactor

tree) was followed to reach a particular point in the recursion. The annotation "(*)" is a reminder that the first and second operands are being cofactored by x_2, so even though the recursion is following the $x_2 = 0$ branch, the $x_2 = 1$ branches are used for the first and second operands. The second column shows the array of operands input to each recursive call. These arrays are laterally shifted in the figure to indicate the depth of recursion. The third column shows the node returned by the recursive call. The annotation "term. cond." indicates that a terminating condition was hit on that call.

Theorem 1 The multi-way method never creates unnecessary intermediate BDD nodes. That is, every node that is created in the principal BDD manager during the procedure belongs to the BDD of the final result.

Proof The important observation is that the **fifth** step of the core routine is the *only* step that creates nodes in the principal BDD manager. Thus, we must analyze the nodes created by the fifth step; we do this by induction on the recursive calling structure of the core routine. In the base case, the fifth step just returns the constant ZERO or ONE or a sub-BDD of one of the original operands, all of which already exist in the principal BDD manager. By induction, the BDD nodes returned by the *then* and *else* branches of the recursion must appear in the final result BDD. Now, the node created in the fifth step combines the *then* and *else* results by calling ITE(*top_id*, *then_result*, *else_result*). If the *then* and *else* results are identical, then the fifth step just returns *then_result*, which by induction is needed in the final result. If they are not the same, then the final result needs a node at the *top_id* level to distinguish the *then* and *else* results. ∎

The fact that a procedure can be created that has this property should not be surprising. This follows since the value of an expression on a given minterm can be computed knowing just the value of each operand on that minterm. Thus, a simplistic algorithm would perform a simultaneous DFS on the operands, *always* recursing down to the point where all the operands are constants. Since a BDD package with a strong canonical form is used, the final BDD is automatically reduced when the recursion unwinds. What we have done beyond this simplistic algorithm is to avoid (in some cases) having to recurse all the way down to constants, by using a cache of previous computations, and by terminating the recursion as soon as only one non-constant operand remains.

2.4 Specialized Multi-way AND

We have specialized the general routine for building Boolean expressions to the case of multi-way AND (i.e. $f_1 \cdot f_2 \cdot \ldots \cdot f_k$). Doing this greatly simplifies the procedure. First, there is no expression array to parse—the input is simply an array of operands. Second, there is no cofactor information to gather and process. Third, we take advantage of the commutativity of AND by sorting the array of operands by their addresses; this increases the cache hit ratio (e.g. $f \cdot g \cdot h$ and $g \cdot f \cdot h$ are not distinguished).

Last, and most important, the test for terminal conditions is hard-coded (there is no need for the auxiliary variables or the BDD TC), and it is more sophisticated. Besides the cases which are detected by the general routine operating on a multi-way AND, namely:

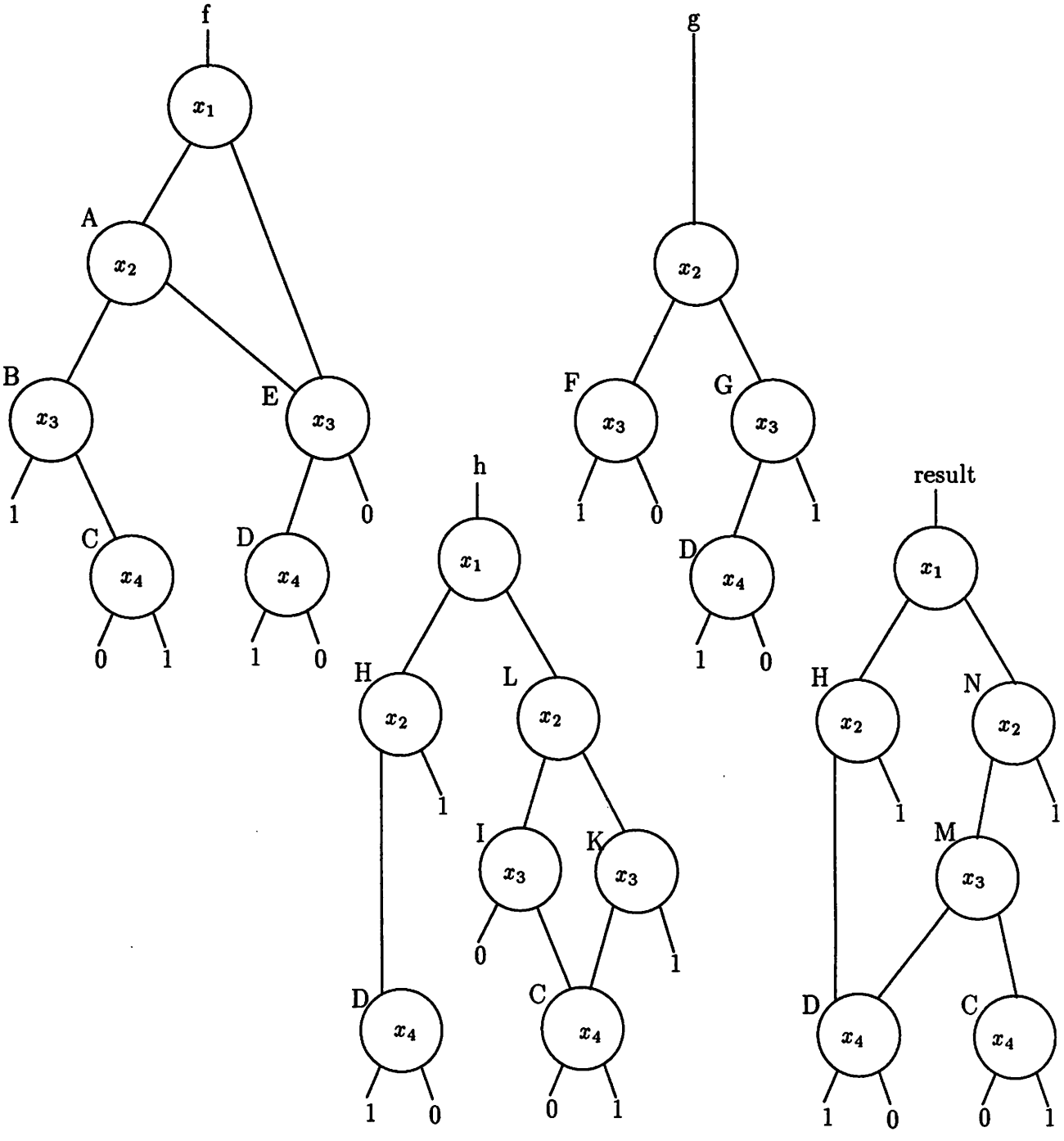

Figure 5: BDDs for the operands used to illustrate the core routine. The letter labeling each node is for future reference. Complement edges are not used to simplify the example. The BDDs are not drawn showing the sharing of nodes, but they are labeled to reflect sharing. Finally, the left edge is the "0-edge" and the right edge is the "1-edge".

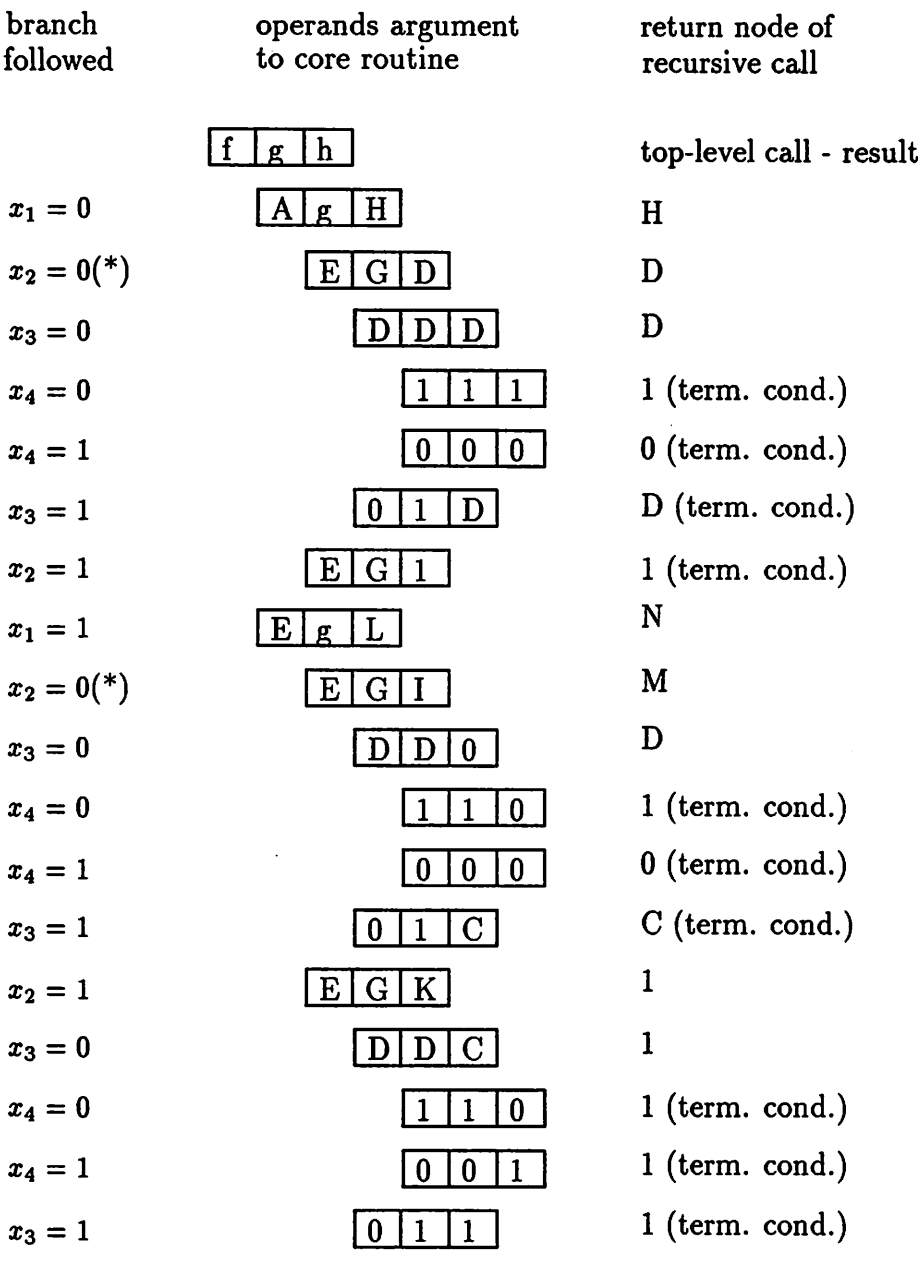

Figure 6: Illustration of the core routine operating on $(f \cdot g)_{x_2} + h$.

- one of the operands is ZERO,
- all of the operands are ONE, or
- there is only one non-constant,

the specialized code also detects the following conditions:

- two operands are complements of each other,[10] or
- there is only one *unique* non-constant.

Also, at each step in the recursion, duplicate entries and constants are removed from the operands array, so that the length of the array gradually decreases along each recursion path.[11]

A specialized multi-way OR is implemented by applying DeMorgan's Law (BDD complement is constant time) and invoking the specialized multi-way AND.

3 Analysis and Discussion

In this section, we briefly analyze the algorithm and make a few observations.

First, if there are k operands, then the terminal case BDD TC has the usual bound of $O(2^k)$ on its size. After building TC at the top-level, TC is only cofactored by cubes, so the size of TC monotonically decreases with recursion depth.

Second, the number of recursive steps in the algorithm, and hence the size of the final BDD, is bounded by the product of the sizes of the operands.[12] This is a generalization of the result that the complexity of a binary operation is the product of the sizes of the two operands. Hence, the asymptotic behavior of the multi-way method is the same as the binary method.

There are examples where the multi-way method requires more recursive steps than the binary method. Consider the expression $(x_n \cdot \overline{x_n}) + f(x_1, \ldots, x_n)$, where x_n is the last variable in the ordering. The binary method requires two steps: the first to compute $(x_n \cdot \overline{x_n})$ = ZERO, and the second to compute ZERO + f = f. On the other hand, the multi-way method requires $O(|f|)$ steps,[13] since the first two operands remain unchanged until the recursion reaches the last variable. As mentioned earlier, the terminal condition could be made more sophisticated to detect this case.

It also seems that the multi-way method is not able to fully exploit previous computations, due to the coarse granularity of the cache entries. For example, suppose that we are computing the AND of three functions, and during the recursion we first reach $f \cdot g \cdot h$ and somewhere later we

[10] Complemented operands are adjacent in the sorted array since their addresses differ by one.

[11] This is not yet implemented, but may have a significant impact since much time is spent sorting the array.

[12] The bound on the number of steps assumes a closed hash table, where *all* previous computations are stored.

[13] $|f|$ denotes the number of nodes in the BDD for f.

reach $f \cdot g \cdot e$. Further suppose that $f \cdot g$ = ZERO, but $f \neq \overline{g}$, so it is not trivial to realize that $f \cdot g \cdot h = f \cdot g \cdot e$ = ZERO. Using the binary method, the necessary recursion will be performed to realize that $f \cdot g$ = ZERO. Then, when $(f \cdot g) \cdot h$ and $(f \cdot g) \cdot e$ are reached, it is trivially realized that the result is ZERO.

On the other hand, for the multi-way method, the necessary recursion is performed on $f \cdot g \cdot h$ to realize the result is ZERO, and this fact is stored in the cache. However, since $h \neq e$, $f \cdot g \cdot e$ will not be found in the cache, and hence recursion must be performed on $f \cdot g \cdot e$ to compute the result. So, in summary, there are examples where the binary method has an inherent advantage over the multi-way method, because the operations are done at a finer level of granularity, and hence previous computations are more likely to be repeated and reused. Note that the specialized multi-way AND also suffers from this problem.

Although existential or universal quantification is not one of the basic operations of the multi-way method, the multi-way method can be used to perform quantification. For example, the expression $\exists_{w,z} f$ can be rewritten as:

$$f_{\overline{w}\overline{z}} + f_{\overline{w}z} + f_{w\overline{z}} + f_{wz}$$

and hence can be computed using the multi-way method with *no* intermediate memory consumption in the principal BDD manager. Of course, this translation is exponential in the number of quantifying variables, so it is probably limited to 5-10 variables.

4 Experimental Results

The new methods are implemented in the Berkeley BDD package. We tested these methods on the application of building the BDD for every node of a multi-level logic network, in terms of the primary inputs. In particular, we experimented with logic networks in SIS [4], where the function at every node is represented as a sum of products in terms of the node's immediate fanins. BDDs are built for the network by starting at the primary outputs and recursively building the BDD at each node, always in terms of the primary inputs. Within this framework, we used four different methods to build the BDD for the sum of products at each node:

Method 1. Binary: use binary AND to build up each product term, and then binary OR to conjunct each product term as it is formed.

Method 2. Specialized multi-way AND: use specialized multi-way AND to construct each product term, and then one application of specialized multi-way OR (dual form of AND) to form the sum.

Method 3. General multi-way AND: same as 2, but use the general algorithm, rather than the specialized code. (This method serves as a control to see how much is gained by specializing the multi-way method to multi-way AND.)

Method 4. Multi-way: build the BDD for the entire sum of products in one application of the multi-way method.

We ran each of these methods on 41 benchmark circuits, after first applying the SIS commands `sweep` and `eliminate -1`, to remove single fanout and trivial nodes. The experiments were run on a DEC 7000 Model 610 AXP with one gigabyte of memory. A time limit of 20000 CPU seconds was used for each experiment. For each method, Table 1 lists the runtime (in CPU seconds) for creating the BDDs, the peak number of BDD nodes (in the principal BDD manager), and the number of BDD garbage collections (GC) performed. [14] Table 1 shows that methods 2, 3 and 4 are roughly 2, 8 and 10 times slower, respectively, than method 1. However, there are examples where methods 2, 3 and 4 are more than 100 times slower, or do not complete at all due to an explosion in the number of recursive steps. The runtimes will be explained later when analyzing Table 2.

The peak number of BDD nodes for method 2 is consistently lower than method 1.[15] The biggest percentage difference comes in example *frg1*: 601 for method 2 vs. 3206 for method 1. However, there is one example, C499, where method 2 has a larger peak than method 1. This can happen because in method 2, it is necessary to keep around all the product terms until the full sum is formed, whereas in method 1, the product terms can be freed as they are conjuncted to form the partial sum.

As expected, when method 4 is able to complete, it always has a lower peak usage than method 1 (by construction, the peak usage for method 4 is exactly the number of nodes needed for the final BDDs of the network). The largest absolute difference comes in example C3540: 196K vs. 297K nodes. This verifies the basic motivation behind the multi-way operations: unnecessary intermediate computations can be avoided. However, there is a factor which undermines this motivation: BDD nodes are garbage collected, so that nodes in intermediate BDDs that are no longer needed can be reused. This means that the peak node usage for the binary method need not be much larger than for the other methods. Indeed, in example C432, the binary method requires a *working space* of only 11K nodes above the 73K nodes that are needed for the final BDDs. The extra garbage collections needed by the binary method to keep the working space clean do not greatly impact the overall runtime.

Table 2 lists the number of recursive steps required for each method. For the binary method, this is the number of ITE operations. For the other methods, this is the number of Boolean expression (BE) operations (core routine calls, described in Section 2). In addition, for the last two methods, we give the total number of recursive calls to ITE and cofactor associated with maintaining the terminal condition BDD.

The examples broadly fall into two classes:

1. those where the number of recursive steps for methods 2, 3 and 4 is smaller than method 1, and

[14] *Peak node usage* is measured as the greatest number of nodes in use at any point during the lifetime of the BDD manager. Immediately after a garbage collection, all nodes in use are *live*. However, in general, "nodes in use" may include both live and dead nodes. When a user "frees" a BDD, some of the nodes in that BDD may become dead (depending on sharing with other BDDs), but they remain as "nodes in use" until the next garbage collection. Note that for the Berkeley BDD package running on a 64-bit machine (e.g. DEC 7000 Model 610 AXP), the first garbage collection is triggered when the number of nodes in use reaches 5110.

[15] Method 3 has the same peak as method 2 since the same intermediate BDDs are computed.

	Binary			Specialized AND			General AND			Multi-way		
Name	Time sec.	Peak Nodes	GC	Time sec.	Peak Nodes	GC	Time sec.	Peak Nodes	GC	Time sec.	Peak Nodes	GC
sbc	0.2	4452	0	0.4	2957	0	1.5	2957	0	1.8	2115	0
s641	0.1	2008	0	0.2	1585	0	0.6	1585	0	0.7	1108	0
scf	0.2	4671	0	0.5	1229	0	2.1	1229	0	2.0	1229	0
styr	0.2	4569	0	0.3	901	0	1.2	901	0	1.1	901	0
cbp.32.4	0.3	5110	3	0.8	5110	2	2.5	5110	2	1.3	3806	0
minmax5	0.7	8176	5	2.0	7714	3	7.4	7714	3	7.0	5863	1
key	1.1	6643	8	1.7	6643	5	5.8	6643	5	4.0	4942	0
alu2	0.1	2406	0	0.2	1348	0	0.8	1348	0	1.1	560	0
apex6	0.2	4086	0	0.3	2877	0	1.2	2877	0	1.8	1552	0
apex7	0.1	2191	0	0.2	1630	0	0.6	1630	0	0.5	872	0
des	9.8	32193	45	19.6	32193	10	66.6	32193	10	56.4	22703	4
frg1	0.2	3206	0	22.3	601	0	1041.8	601	0	-	-	-
frg2	0.8	5110	5	2.0	5110	3	10.9	5110	3	23.0	3739	0
i2	0.3	5110	1	42.2	2755	0	-	-	-	-	-	-
i3	0.1	2835	0	0.1	779	0	0.4	779	0	0.4	319	0
i4	0.1	2055	0	0.7	1099	0	22.0	1099	0	-	-	-
i6	0.1	988	0	0.1	830	0	0.3	830	0	0.2	418	0
i7	0.1	1332	0	0.1	1110	0	0.4	1110	0	0.3	508	0
i8	1.3	8569	10	2.7	7665	5	9.2	7665	5	10.5	6373	1
i9	0.4	5110	2	0.8	5110	1	3.3	5110	1	1.9	2298	0
i10	200.1	1101716	40	345.3	896294	19	1014.4	896294	19	1105.1	786444	13
k2	0.6	5110	3	1.5	2914	0	5.5	2914	0	5.6	2779	0
myadder	0.1	4253	0	0.3	3122	0	0.9	3122	0	0.4	834	0
pair	3.2	56006	7	13.0	54166	7	37.0	54166	7	65.3	44251	6
rot	2.5	17885	17	4.7	15432	7	17.6	15432	7	52.1	12382	3
t481	1.4	5523	6	10.1	5110	1	35.8	5110	1	104.0	3327	0
term1	0.2	4804	0	0.7	1988	0	6.0	1988	0	45.5	922	0
toolarge	4.3	7154	24	135.3	7106	3	2390.7	7106	3	-	-	-
ttt2	0.1	984	0	0.2	555	0	1.1	555	0	2.4	254	0
vda	0.2	3404	0	0.3	857	0	1.5	857	0	1.3	845	0
x1	0.2	4464	0	1.2	1713	0	60.1	1713	0	412.4	799	0
x3	0.2	4679	0	0.6	3063	0	3.9	3063	0	10.0	1526	0
C432	12.2	83758	18	25.5	79716	13	86.4	79716	13	125.1	72888	7
C499	6.4	52122	13	21.7	55364	12	70.8	55364	12	47.4	41686	6
C880	1.2	15948	8	2.5	14207	5	8.1	14207	5	8.0	12016	3
C1355	6.4	55188	13	21.1	53655	12	69.0	53655	12	48.7	41686	6
C1908	14.0	82820	19	34.7	77090	11	106.0	77090	11	98.9	60751	6
C2670	28.3	195713	16	101.9	194143	11	396.5	194143	11	400.9	115251	8
C3540	31.1	297419	19	78.3	287182	16	257.3	287182	16	159.4	195610	9
C5315	4.5	65521	12	17.5	61672	10	1247.6	61672	10	1219.2	52913	6
alu4	0.5	5110	2	0.9	4727	0	3.5	4727	0	5.8	2049	0

Table 1: Time, peak number of BDD nodes, and number of garbage collections, for each method.

	Binary	Spec. AND	General AND		Multi-way	
Name	ITE ops	BE ops	BE ops	Aux ops	BE ops	Aux ops
sbc	11060	8636	8908	35774	7192	58116
s641	4461	3800	3928	11148	3417	18775
scf	12489	4524	4962	97544	4721	97659
styr	11935	3256	3582	46535	3457	46402
cbp.32.4	21714	19982	20122	26509	8102	16281
minmax5	46808	42403	42759	209242	19134	277408
key	48648	34431	34335	121183	11860	138163
alu2	7644	4711	4895	25155	1945	48804
apex6	8947	7117	7447	31398	6117	62169
apex7	4893	3902	3960	14039	2068	16359
des	597693	341632	351544	1673988	209455	1656013
frg1	13277	48726	846952	54702463	-	-
frg2	50618	39984	56856	316039	42449	880290
i2	22143	193419	-	-	-	-
i3	5336	1502	1510	13322	458	18368
i4	5468	9544	91246	618738	-	-
i6	2361	1774	1780	6623	713	5775
i7	3100	2273	2273	8707	764	7976
i8	80718	54637	54757	220742	32216	327922
i9	32635	16885	19773	89799	5045	76304
i10	9022956	4200340	4568100	14191269	2454852	16641171
k2	31638	9832	10828	252325	9285	264652
myadder	8616	6460	6448	11149	1680	6401
pair	170633	171181	176589	919840	257774	2237645
rot	153851	79927	94413	277846	100958	1071991
t481	99609	76147	116795	1083715	61056	4949938
term1	15702	10267	27381	189071	75776	1620958
toolarge	331092	136716	775670	94982853	-	-
ttt2	3333	3027	5879	36116	6069	97647
vda	9666	4092	4774	55455	3871	55414
x1	14340	9379	67673	3163571	126306	9489548
x3	12066	10557	20231	117893	22759	369309
C432	680090	504666	519124	1356541	353561	4036215
C499	395858	535724	542424	777587	335706	543782
C880	70077	51160	52466	128790	34195	189919
C1355	398382	507388	521480	748299	341294	551500
C1908	870447	669700	662174	1515758	503750	1862897
C2670	1714189	1920193	2470637	4460198	1366375	6852376
C3540	1601538	1734221	1857443	3244524	899391	2480295
C5315	234946	349515	5405375	43556073	5255133	43341119
alu4	33330	19076	20390	102175	9816	244863

Table 2: Number of recursive steps for each method. The last two methods have additional operations in the auxiliary variable space.

2. those where the number is *much* greater, or did not complete.

Those in the first class satisfy the expectation that grouping multiple binary operations into a single expression reduces the total number of operations, since "global" terminating conditions are discovered earlier. However, even though the number of operations is smaller, the runtimes are larger due to the extra overhead of processing general expressions. For example, method 2 spends significant time sorting the array of operands, and methods 3 and 4 spend much time evaluating the terminal condition BDD.

The more fundamental problem lies in the second class. The explosion in the number of BE operations is thought to be due to:

- poor cache performance, due to the coarse granularity of computations, and to the fact that the operations are not cached across top-level expressions, and
- not identifying expressions which differ by a permutation of symmetric operands, and not detecting some terminal cases (excluding method 2).

The difference in granularity between methods 1 and 2 is demonstrated in example *frg1*, which has two small network nodes, and one big node with 25 fanins, 112 cubes, and 780 literals. Method 2 requires a reasonable 1400 operations to form the 112 product terms of the big node. However, forming the OR of these terms requires 47K operations, as compared to a total of 13K operations with method 1.

It is interesting to compare the specialized multi-way AND method to the general multi-way AND method. For those examples where the number of BE operations is similar, the general method is roughly four times slower. This is due to the enormous overhead of building and maintaining the terminal condition BDD. Indeed, for the general method the number of operations performed in the auxiliary variable space is significantly greater than the number of BE operations: on average, each BE step induces many operations to evaluate the terminal condition BDD. The specialized code avoids these operations by efficiently hardcoding the terminal condition for a multi-way AND. This overhead is present also in the multi-way method, and hence partly explains why the multi-way method is slower than the binary method.

Continuing the comparison between methods 2 and 3, in the other examples the number of BE operations is much greater in method 3 than in method 2. For example, method 3 requires 847K operations for *frg1*, whereas method 2 requires only 49K. This is because the specialized method sorts the operands before caching them, thus increasing the cache hit ratio, and because the terminal cases are detected where two operands are complements of each other or where there is exactly one unique non-constant operand. The difference in BE operations in examples like these emphasizes the importance of developing techniques to terminate the recursion as early as possible.

We did not perform any experiments comparing the multi-way method with the binary method extended to use implicit don't cares (hereafter, the *binary DC method*). Nor do we have a conjecture as to the outcome of such experiments. However, it seems that any advantage that the binary DC method may have over the multi-way method would stem from the binary DC method's finer

granularity of computation, rather than its use of don't cares. To see this, again consider the example $(f \cdot g) + h$. Using h as a don't care to simplify f (using the *restrict* operator, f and h must be traversed simultaneously to the point where h is ONE, in order to realize how f can be modified. In contrast, the multi-way method may also need to traverse to the point, however, once it reaches there, it knows that the final answer on this branch is ONE (since h is ONE), and hence can simply return. In summary, it seems that the work performed by the binary DC method to discover how the don't cares can be exploited, can just as well be used to compute the final answer.

5 Conclusions and Future Work

We have developed a new method for computing the BDD for a general Boolean expression, where the operands are themselves BDDs. The method is able to avoid unnecessary intermediate computations, and it never creates a new BDD node unless that node appears in the final BDD for the expression. We also specialized the general algorithm to the case of a multi-way AND.

We tested the new methods on the application of building BDDs for nodes in a multi-level logic network. The new methods take much longer than the binary method due to the coarse granularity of computation, the less sophisticated caching and terminal case checking, and the overhead incurred by allowing more general Boolean expressions. Also, although the new methods use fewer BDD nodes, this gain is mitigated by the presence of a garbage collected working space in the BDD package.

Nonetheless, there is some cause for optimism. On many examples, the new methods require fewer recursive steps. Thus, the challenge is to reduce the time spent on each step. We saw that by specializing the general multi-way code to the case of multi-way AND, the speed decreased by a factor of four. There may be further optimizations possible in the specialized multi-way AND, making it more competitive in runtime with the binary method.

The problem of explosion in the number of BE operations needs to be addressed. An explosion seems to occur when there are many operands present in the expression (say, more than 100). Possibly by splitting such expressions into smaller pieces, we can avoid explosion due to coarse granularity, while still enjoying a reduced number of recursive steps, relative to the binary method.

We have experimented with only one application so far, and the new methods did not perform well. However, it is easy to devise examples which *cannot* be completed using the binary method. Suppose $|f \cdot g| = O(|f|\,|g|)$, where $|f|, |g| = 10000$. Then the binary method would not be able to compute $(f \cdot g) +$ ONE. Hence, maybe special applications can be identified where it is clear that the multi-way method wins.

Besides trying to make the new algorithms more efficient, there are two avenues for further research. The first is to create a multi-way AND-smooth, that is, to compute $\exists x_1, \ldots, x_p(f_1, \ldots, f_k)$. Whatever improvements are made to multi-way AND can be directly used in a multi-way AND-smooth.

The second avenue is to incorporate BDD minimization using don't cares into building BDDs for

Boolean expressions. The BDD minimization problem for a single incompletely specified function is to find a cover of the function with a small BDD [5]. We could extend the input to the multi-way method to allow a don't care function for each operand in the expression. Then the goal would be to use the don't cares present in each operand to make the final BDD for the expression as small as possible. We have already outlined an algorithm to do this.

Acknowledgements

We wish to thank Rick McGeer and Alex Saldanha for their helpful comments. This work was supported by SRC grant 93-DC-008, and by an equipment grant from Digital Equipment Corporation. In addition, the first author was supported by an SRC Fellowship.

References

[1] Karl S. Brace, Richard L. Rudell, and Randal E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th Design Automat. Conf.*, pages 40–45, June 1990.

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

[3] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer-Verlag, June 1989.

[4] Ellen M. Sentovich, Kanwar Jit Singh, Cho Moon, Hamid Savoj, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Sequential circuit design using synthesis and optimization. In *Proc. Int'l Conf. on Computer Design*, October 1992.

[5] Thomas R. Shiple, Ramin Hojati, Alberto L. Sangiovanni-Vincentelli, and Robert K. Brayton. Heuristic minimization of BDDs using don't cares. Technical Report UCB/ERL M93/58, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, July 1993.