Int J STTT (2001) 3: 171–181 / Digital Object Identifier (DOI) 10.1007/s100090100042

Efficient manipulation of decision diagrams

Fabio Somenzi

Department of Electrical and Computer Engineering, University of Colorado at Boulder, Boulder, CO 80309-0425, USA;
E-mail: Fabio@Colorado.edu

Published online: 15 May 2001 –

Abstract. Over the last decade significant progress has been made in the efficient implementation of algorithms for the manipulation of decision diagrams. We review the main issues involved in designing a decision diagram package, with special emphasis on the basic data structures and their management, on fast variable reordering, and on the collection of statistics to guide the tuning of the algorithms. Our analysis focuses on depth-first manipulation of reduced, ordered binary decision diagrams.

Key words: Binary decision diagrams – Strong canonical form – Variable reordering

1 Introduction

The interest in decision diagrams as a versatile data structure for the manipulation of Boolean functions has led to the development of several efficient packages that support one or more forms of diagrams. In this article we refer to these packages as BDD packages, and discuss the main issues that must be taken into account by their implementors. We concentrate on ordered reduced binary decision diagrams (BDDs henceforth), but the majority of our observations apply to other varieties as well.

Though BDDs have been around for several decades (see Sect. 1 in [13] for a brief history) it is the work of Brace, Rudell, and Bryant [6] that drew the attention of researchers to the advantages of efficient implementations of BDD packages. That work combined ideas that had been proposed previously [22, 26] with new insights to produce what may be considered the first modern BDD package.

This work was supported in part by SRC contract 98-DJ-620.

Since then, several ideas have been proposed to increase the efficiency of BDD packages. At the obvious risk of excluding some important contributions, we mention attributed edges [30], dynamic variable ordering [15, 34], levelized processing [3, 31, 36, 42], the use of operation nodes to combine reordering and synthesis [17], and, finally, the use of generational garbage collection for a "cache-friendly" implementation [24].

Originally designed to support the standard operations on Boolean functions, BDD packages have grown to incorporate a very wide range of operations, some of which were specifically meant for the graph representation of functions [11, 29, 33]. Many variants of decision diagrams have been devised (see Sect. 3 of [13] for an overview), and BDD packages often support two or more of these variants. It is not uncommon for a BDD package to export more than a hundred functions. Last but not least, the sharp decline in the cost of semiconductor memory over the last few years has made it possible for BDD-based applications to explore the trade-off between space and time in new ways.

The combined result of all these factors is that today's packages tend to be complex and sophisticated pieces of software striving to offer rich functionality and efficient execution with a variety of approaches. In the rest of this article we will concentrate on one such approach, which can be described as depth-first manipulation of BDDs. This is the standard approach, already used in [8], discussed in Sect. 2.3 [13], and adopted in most BDD packages. It is not always the most efficient, but as discussed in Sect. 6, it is quite competitive for symbolic model checking [28] and other demanding applications.

After the preliminaries of Sect. 2, we examine the "guts" of a BDD package: Sect. 3 is devoted to the main data structures required for BDD manipulation

(the unique and computed table), while Sect. 4 discusses garbage collection. We then review in Sect. 5 several aspects related to variable ordering. In Sect. 6 we cover the important issue of gathering statistics and tuning a package for performance. We try to do justice to a number of neglected topics in Sect. 7. The discussion of several aspects of a BDD package is obviously influenced by the solutions adopted in the CUDD package [38]. This influence is acknowledged here and not further mentioned in the sequel.

2 Preliminaries

For an introduction to BDDs and for the basic definitions, we refer the reader to [13]. Unless otherwise stated, we consider shared, reduced, ordered binary decision diagrams. Throughout this article we assume the use of complement attributes as described in Sect. 2.2 [13]. In the drawings, the complement attribute is indicated by a black dot. A circle instead of a dot is used to designate the 0-edge out of a node when it is not complemented. To reduce clutter, the constant 1 node is not shown, and dangling edges are understood to connect to it. The boxes labeled $f, g, \ldots$ designate the roots of the BDD.

The importance of the complement attributes is not so much in their ability to reduce the memory requirements – the theoretical limit of 50% is seldom approached – as it is in the ability of a package using them to perform complementation in constant time. Complementation in constant time allows one to freely use De Morgan laws to reduce coding: given, for instance, an algorithm for conjunctive decomposition, disjunctive decomposition of a BDD is easily obtained by complementing the results of conjunctively decomposing the complement of the BDD. Furthermore, recursive algorithms can test for additional termination condition. For instance, $f \wedge \overline{f} = 0$ can be detected in constant time.

Additional attributes (input inverters and variable shifters) were proposed in [30]. Unfortunately, they are not compatible with the algorithms for variable reordering of Sect. 5.1. Therefore, in spite of their effectiveness, they are not widely used nowadays, and we do not discuss them further.

3 Unique and computed tables

It would not be an exaggeration to claim that the main reason for the efficiency of BDDs is their extensive reliance on table lookups. The maintenance of a canonical representation is accomplished by storing all BDD nodes in a dictionary called the *unique table*. The efficient implementation of almost all recursive manipulation algorithms is made possible by the *computed table*. In this section we describe these two data structures that are fundamental to most BDD packages.

3.1 Unique table

The unique table is usually implemented as a hash table [1]. For greater flexibility, open hashing with collision chains are normally used. The collision lists can be kept sorted [24] to reduce the number of memory accesses required for a lookup on average.

Recently, closed hashing has been proposed: its advantage is that the BDD node can be made smaller by not storing the collision chain pointer. In implementations that do not store reference counts in the nodes, and that impose limitations on the number of nodes and variables, the closed hashing scheme allows one to store a BDD node in a mere 8 bytes [4]. In the following, however, we shall assume the usual scheme that requires 16 bytes per node.

The unique table is usually divided in subtables – one for each variable. This organization allows fast access to all nodes labeled by a given variable, which facilitates variable reordering. An additional table may be used for the constant nodes, especially in implementations that support multiple terminals. For an internal node, the variable index is used to select the subtable, and the pointers to the two children of the node are used as hash key. The size of each subtable is either a prime number or a power of two. With the latter, multiplicative hash functions work well [10]; they have the form

$$((t \cdot p_1 + e) \cdot p_2)/2^{b-k} ,$$

where t and e are the pointers to the two children of the node, p_1 and p_2 are suitably chosen primes, b is the word length used for the computation of the hash function, and 2^k is the size of the hash table. The division, implemented as a shift, has the effect of extracting the k most significant bits from the result. Using the most significant bits is important because low-order bits of the pointers are mostly zeroes due to alignment requirements.

The subtables start small and grow as more nodes are created. The collision lists should be short to keep the lookup time within acceptable bounds. Therefore, the load factor of the table is usually less than 4; if that value is exceeded, the subtable is either subjected to garbage collection to remove *dead* nodes – nodes not currently in use – or is resized. If the number of dead nodes is low, garbage collection may be ineffective. Furthermore, implementations that do not maintain a count of the references to each node may prefer to separate the decision to reclaim dead nodes from the one of resizing the unique subtables, because without reference counts it is necessary to scan the unique table to count the dead nodes.

In any case, the relative frequency of garbage collection and table resizing determines the average fraction of nodes stored in the unique table that are dead. From the point of view of memory efficiency, this fraction should be low. However, from the point of view of performance, it is sometimes very important to allow a substantial accumulation of dead nodes in between garbage collections.

The obvious effect is to amortize the cost of one garbage collection over a larger number of dead nodes. A more important consequence, however, is to improve – sometimes dramatically – the performance of the computed table. To understand the effect of retaining dead nodes on the computed table, one has to consider that:

- A lookup in the computed table may return a pointer to a dead node. In such a case, the node is "resuscitated."
- Some operations on BDDs – most notably matrix multiplication algorithms for various algebraic structures like the *AndExist* algorithm used in model checking – produce many intermediate results that are disposed of. While it is impractical to completely prevent the collection of these many dead nodes, the algorithms critically rely for performance on not recomputing these intermediate results multiple times.
- Garbage collection causes the computed table to be flushed of all entries that point to dead nodes. (In some implementations, the computed table is completely flushed.) Frequent garbage collections reduce the hits in the computed table, especially for the intermediate results mentioned above.

Not all applications equally benefit from keeping many dead nodes around. Fortunately, the rate at which dead nodes are resuscitated can be used to determine a suitable ratio of dead nodes to live nodes in a healthy unique table.

The trade-off between speed and memory should obviously take into account the available resources. In particular, the growth of the unique table should slow down when it approaches a threshold determined by the available physical memory.

3.2 Computed table

The computed table acts as a software cache for the results of the recursive operations on BDDs. A *lossless* computed table – one that stores all results – guarantees polynomial cost for the basic synthesis operations. In small to medium-scale experiments lossless computed tables perform very well. However, they are not feasible for applications that manipulate many large BDDs. It is therefore customary to implement *lossy* computed tables, at least for those operations for which losslessness is not a requirement.

We distinguish three forms of computed tables: the first form of computed table consists of a *field in the BDD node data structure*. In its simplest possible form, it is one or two bits, usually taken from the least significant bits of pointers. A field in the BDD node may be adequate to store the results of unary operators (operators that work on one BDD). The cached results are usually flushed at the end of the operation, which leads to no interoperation caching. A typical example is given by the counting of the nodes reachable from a set of roots. In this case, only one bit is required to mark a node that has been previously reached. Lossiness is not acceptable in this computation. At the end of the computation, the mark of each node is cleared.

A second form of computed table is a *temporary* table – usually a hash table – created at the beginning of a BDD operation, and released at the end of the recursion. There are at least two reasons why a temporary cache may be preferred to a permanent one. Some operations may have operands or results that do not fit in a *shared*, permanent computed table, yet they may be used too infrequently to justify a *dedicated* permanent computed table.

Other operations may require a temporary table to guarantee some properties of the result. We briefly discuss two cases here. In the *LI-compaction* algorithm [20], and in the *Remapping Underapproximation* algorithm [33] – to cite two examples – the action to be taken at a given node depends on the roots on which the operation was invoked. Saving results across different invocations of the operations would therefore produce incorrect results. The second reason why a lossless computed table may be preferable has to do with the reproducibility of results on different *computing platforms*. (By this term we denote the combination of hardware, operating systems, compilers, compiler switches, and other factors that uniquely determine one execution of a program.) One of the main problems in insuring reproducibility of results in BDD packages is the dependence of hash values on the memory addresses of nodes. Those addresses normally change with the platform. Here, we only consider the following aspect of the problem.

Suppose an operation on BDDs returns a result for a node that depends on the length of the path first traversed to reach that node. The result is stored in the computed table and reutilized when the node is reached successive times, provided the corresponding computed table entry is not evicted. If the result is recomputed because of an eviction, it may not be the same as the original one. Though this behavior may be perfectly acceptable from the point of view of correctness, it may cause two different traces of execution on two different platforms. For instance, a bug may no longer be observable when the program is run under a debugger.

Fortunately, not many operations require – for one reason or the other – a temporary computed table. Therefore, we can concentrate in our discussion on the third form of computed table: the *permanent* table, which we assume to be lossy. A permanent table does not incur the overhead of creation and annihilation at each top-level call of an operation, and it allows interoperation caching. Therefore, it tends to be more efficient than a temporary computed table.

Some implementations use separate tables for each operation; others use a *shared* computed table. Hybrid solutions are also possible. With the separate computed tables the entries need not store the operation type; operations with arbitrary numbers of operands can be accommodated, and some cases of conflict misses are easier

to handle. On the other hand, a shared computed table is easier to manage, especially for applications that interleave many different operations. In the following we assume a shared computed table.

An entry of the computed table consists of pointers to the operands and the result; for a shared table, the operation type is also part of the entry. Since most BDD operators have two or three operands, it is common to reserve the shared computed table to such operators. Exploiting the fact that there are relatively few three-operand operations, it is then possible to store an entry in 16 bytes (for 32-bit pointers). This is desirable, because it allows alignment of the computed table entries to the cache line boundaries. (The size of the cache line is usually a multiple of 16 bytes.)

Most computed tables are one-way associative: each combination of operands and operator maps to exactly one position in the table. Though the hit rate (i.e., the fraction of successful lookups) for two-way associative tables is better, the improvement may not be enough to offset the overhead involved in the higher associativity. The key to an entry is given by all the operands and the operation type. The quality of the hash function is especially critical for the computed table, because it is not uncommon to have tables with 16 million entries. In that case the hash values should be 24 "random" bits, while a pointer typically contains 26 non-constant bits or less. Fortunately, hash functions similar to the one discussed for the unique subtables perform well in practice.

Like the unique table, the computed table starts small and possibly grows during the execution of the application, within the limits imposed by the available resources. Some applications are weakly affected by rather substantial changes in the size of the computed table. For them, a small table is desirable, because it leaves more room for the BDD nodes, and can be flushed more quickly during garbage collection. Other applications (e.g., large model checking runs) rely critically on a large computed table for performance. The sizing policy should try to accommodate these different requirements.

One important observation in the sizing of the computed table is that the hit rate is largely independent of the size of the table itself over a wide range of sizes, while it depends strongly on the operation and the operands. This observation can be justified as follows. On the one hand, the operation and the operands determine how many repeated subproblems will be generated. On the other hand, when a large problem is solved more than once, so are its recursively generated subproblems. If a miss occurs for a given result that had been evicted from the computed table, there is still a good chance that there will be hits for the subproblems.

These observations suggest a *reward-based* policy for the growth of the computed table. If a given run generates a high hit rate, it is likely to benefit from a larger computed table. Therefore the table grows if the hit rate is high. However, there is little point in keeping a computed table that is much larger than the unique table. Every time garbage collection is invoked, the computed table is flushed of all dead entries. A computed table that is much larger than the unique table is therefore less than fully utilized.

Since memory is a precious resource for BDD applications, care should be exercised in not abusing the computed table. First of all, it is normally counterproductive to save trivial results, like the ones generated by terminal cases of the recursive procedures. One can also standardize the entries to the computed table to increase the hit rate. One common approach is to impose an arbitrary but fixed order on the operands of commutative operations. For instance, in computing the conjunction of two BDDs, the operands can be sorted according to the addresses of their top nodes. Standardization is applicable also to non-commutative operations. When checking whether $f \leq g$, an implementation that uses complement pointers can always force g to be a regular pointer. Indeed, if f is regular and g is complemented, the recursion has reached a terminal case because $f(1,1,\ldots,1) = 1 \not\leq 0 = g(1,1\ldots,1)$; if, on the other hand, f and g are both complemented, one can check for $\bar{g} \leq \bar{f}$. The transformation from $f \leq g$ to $\bar{g} \leq \bar{f}$ can also be applied when f is complemented and g is regular, if the pointers to the top nodes of the two operands are not in the proper order. The standardization scheme for the *if-then-else* operation is even more elaborate; the interested reader is referred to [6].

The last technique we consider to increase utilization of the computed table is to only store in the table results for which there is a reasonable expectation that they will be needed again. Consider two operands with a reference count of 1. They can only be visited once for each top-level call to most operations. Therefore, the result produced during the first visit is unlikely to be needed again and the recursive procedure may refrain from storing it in the computed table. This decrease the probability of evicting a valuable result, and also saves the cost of an insertion and a lookup. (If the reference counts are 1, no lookup is performed.)

4 Garbage collection

An application using BDDs often creates many intermediate BDDs. An implementation should therefore provide a mechanism to recover the memory used by BDDs that are no longer needed. The most common approach is based on garbage collection. If shared BDDs are used – an assumption that we shall tacitly make – then the nodes of a BDD that is no longer needed cannot be simply returned to a free list, because they may be reachable from other roots that are still in use. If a reference count is kept for all nodes, then in principle it is possible to return a node to the free list as soon as its reference count goes to

0. (We then say that the node is *dead*.) However, there are good reasons not to do so.

First of all, a node may be referenced by an entry in the computed table (see Sect. 3.2) because it was either an operand or the result of a recently computed operation. The entries of the computed table do not usually contribute to the reference count, because there would be too many nodes kept alive by computed table entries. Therefore, when the reference count of a node goes to 0, it would be necessary to locate all entries in the computed table pointing to it. This would require scanning the table, or using extra memory.

Another important reason to use periodical garbage collection is that a certain amount of dead nodes may be beneficial, as mentioned in Sect. 3.1: it is often the case that dead nodes are resuscitated, and the cost of bringing a BDD back from the dead is usually much smaller than the cost of creating it anew.

Moreover, the cost of garbage collection is amortized over a large number of nodes. This makes it possible to perform maintenance operations on the free list that would be too expensive to perform on individual nodes. For instance, it is possible to increase the locality of access of the algorithms by (partially) sorting the free list.

Finally, garbage collection is inevitable in implementations that do not use reference counts. Such implementations usually implement some variant of the *mark and sweep* approach. It is also possible to use reference counts, but not keep them constantly updated. In this case garbage collection is again the only practical approach to reclaiming the nodes of BDDs no longer in use.

In describing different schemes for garbage collection we shall assume that BDD packages keep a free list of nodes organized in *pages*. Each page may store about one thousand nodes, and is allocated with a single call to the operating system. Garbage collection algorithms endeavor to concentrate the nodes in use in as few pages as possible, and to keep the distance between successive nodes of the free list as low as possible. This is done to increase the locality of access to memory.

A simple garbage collection scheme for implementations that use reference counts proceeds in two phases. In the first phase the computed table is scanned and the entries pointing to dead nodes are invalidated. The unique table is then scanned, and the dead nodes are returned to the free list. If the reference counts are not kept up-to-date at all times, they are updated before garbage collection. To improve the locality of access, the nodes recovered from the unique table may be sorted before being returned to the free list.

An alternative scheme scans the free list before scanning the unique table. The dead nodes are linked in address order, and pages that are no longer in use can be identified and freed. Finally the dead nodes are unlinked from the unique table [25].

The techniques discussed so far do not move nodes during garbage collection. The approach proposed in [24], on the other hand, is based on the assumption that the relative age of two nodes can be inferred from their addresses in memory. Assuming that the heap grows toward higher addresses, then a more recent node has a higher address. This property, if maintained, speeds up the search of the collision lists of the unique table. It requires, however, that a compacting garbage collection strategy be used. One advantage that tends to outweigh the cost of the compaction step is the ability to apply a generational garbage collection scheme: the BDD package keeps track of the lowest address in a BDD that has been freed. The portion of memory below this address does not need to be scanned during garbage collection, because it is guaranteed not to contain any dead nodes.

Garbage collection should be performed when there are enough dead nodes to justify it. In packages that keep reference counts up-to-date, the number of dead nodes and the total number of nodes are known, thus allowing garbage collection to be triggered when precise conditions are met. Otherwise, the package has to assume that that a given growth in the total number of nodes has resulted in the creation of sufficiently many dead nodes to warrant garbage collection.

Keeping accurate reference counts has therefore the advantage of providing more accurate control over when to collect garbage. However, the cost of maintaining accurate reference counts may be high. The overhead may go from unnoticeable to quite significant, depending on the application and the case at hand. In particular, when the same intermediate results are computed several and disposed of in between the computations, decreasing the reference counts when BDDs are freed, and then increasing them when many of the same BDDs are found in the computed table may account for a large fraction of the entire CPU time. Therefore, hybrid solutions can be used that provide an estimate of the number of dead nodes, rather than an exact count.

The reference count takes up space in a node. Therefore, when 32-bit pointers are used, more than 16 bits are very seldom reserved to the reference count. In experiments involving large BDDs, the reference counts of nodes in the lowest levels of the BDDs can easily exceed $2^{16}-1$. Fortunately, even with millions of nodes, few nodes reach high reference counts. Hence, a solution based on *saturating increments* works well in practice when 16 bits are used for the reference counts. A saturating increment leaves unchanged a reference count if it has reached its maximum value. A saturating decrement does the same. Therefore, a node whose reference count saturates will never be reclaimed by garbage collection, even if it becomes unused. This corresponds to a memory overhead that is quite acceptable in practice. With fewer than 16 bits for the reference counts the situation changes. More sophisticated approaches are needed (for instance, keeping an overflow table for large reference counts [9]) to prevent memory from filling up with nodes whose reference counts have saturated.

5 Variable reordering

Variable reordering is often crucial when dealing with "difficult" BDDs. It is, however, a time-consuming process, which requires a careful implementation. It was already mentioned in Sect. 3.1 that the unique table is organized in subtables to allow fast access to all the nodes labeled by the same variable. In this section we complete our review of the data structures and algorithms used in variable reordering.

5.1 Swapping adjacent variables

Our discussion will start from the basic operation of most reordering algorithms: the swap of two variables adjacent in the order. It is possible to perform such a swap in time proportional to the number of nodes labeled by the two variables, and to the sizes of the two subtables used to store those nodes. However, this requires that the swap be performed *in place*, that is, by modifying the existing BDDs, rather than creating a modified copy. This is possible thanks to the following important theorem [14]:

Theorem 1. *Permutation of a prefix of the variable order has no effect on the nodes labeled by variables not in the prefix (the bottom part of the BDD is not affected by rearrangement of the top variable). Likewise, permutation of a suffix of the variable order has no effect on the nodes labeled by variables not in the suffix.*

This result is illustrated in Fig. 1, which suggests that reordering of the variables between levels i and j does not require knowledge of the remaining parts of the BDD (shown as a shaded area).

We now examine the swap of the variables in positions i and $i+1$ in the order. Let these variables be x and y, respectively. (We assume that position i is closer to the roots of the BDD than position $i+1$.) The use of a separate subtable for each variable allows the reordering algorithms to access all the nodes labeled x in time independent of the number of nodes labeled by other variables. Each node labeled x is the root of a sub-function f of the BDD to which we can then apply the following identity:

$$\begin{aligned} f &= \bar{x}(\bar{y} f_{|x=0,y=0} \vee y f_{|x=0,y=1}) \vee \\ &\quad x(\bar{y} f_{|x=1,y=0} \vee y f_{|x=1,y=1}) \\ &= \bar{y}(\bar{x} f_{|x=0,y=0} \vee x f_{|x=1,y=0}) \vee \\ &\quad y(\bar{x} f_{|x=0,y=1} \vee x f_{|x=1,y=1}) \,. \end{aligned}$$

Suppose that $f_{00} = f_{|x=0,y=0}$, $f_{01} = f_{|x=0,y=1}$, $f_{10} = f_{|x=1,y=0}$, and $f_{11} = f_{|x=1,y=1}$ correspond to four distinct BDD nodes. Then reordering sub-function f amounts to rearranging these four nodes as shown in Fig. 2. One has to be careful, though, in dealing with the nodes labeled x and y. After swapping, there is one node labeled y in f. This node should be a child of every node that had the old root of f as child. However, most BDD packages do not keep lists of the nodes pointing to each node; searching the levels of the BDD above i for nodes pointing to the old root of f would make the swap time depend on the overall size of the BDD. The solution is to utilize the old root node as new root, so that pointers do not need to be updated. The node is moved from the x subtable to the y subtable, and its label is changed from x to y.

The nodes corresponding to $f_0 = f_{|x=0}$, and $f_1 = f_{|x=1}$, on the other hand, cannot be reutilized for the nodes corresponding to $f_{|y=0}$, and $f_{|y=1}$, because the new nodes correspond to different functions, and the old nodes may be still required after reordering. Therefore, new nodes are created for $f_{|y=0}$, and $f_{|y=1}$. For each node, a reference count is kept. At the end of the variable swap, the y subtable is scanned to collect all nodes whose reference counts have become 0. These nodes were exclusively pointed by nodes in the x subtable before the swap.

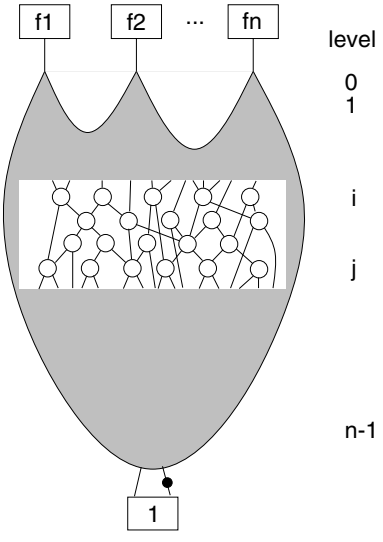

Fig. 1. The fundamental lemma for reordering

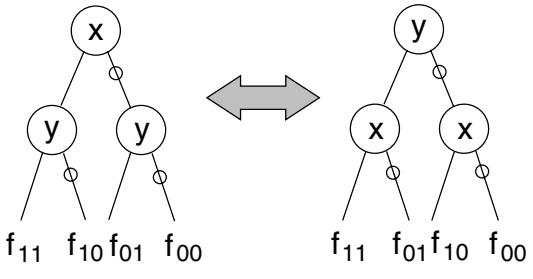

Fig. 2. Variable swap for one node

5.2 Interacting variables

If $f_{00} = f_{01}$ and $f_{10} = f_{11}$, then f does not depend on y. Therefore, it is not moved or changed by the swap. If this is the case for all the nodes in the x subtable, we say that

x and y do not interact. Swapping two variables is very inexpensive if it is known in advance that they do not interact, because one can avoid scanning the subtables altogether [35]. It is often very advantageous to collect information on what pairs of variables interact before reordering starts. In many practical cases over 90% of the variable swaps can be performed inexpensively thanks to this preprocessing. Suppose that the BDD to be reordered contains variables x and y with the following property: for every sub-function f of the BDD,

$$f_{|x=0} = f_{|x=1} \vee f_{|y=0} = f_{|y=1} \,. \tag{1}$$

In words, no f depends on both x and y. Then it is easy to see that x and y will never interact, regardless of the variable order. In addition, it is sufficient to check the roots of the BDD, because any sub-function f is reachable from at least one root. Let the *interaction matrix* I be the square matrix over $\{0,1\}$ such that $I_{ij} = 0$ if and only if the i-th and j-th variables satisfy (1). Then I can be computed by performing a depth-first search from each root of the BDD. Though this has a worst-case cost that is quadratic in the number of nodes, in practice the number of roots of large BDDs is much smaller than the number of nodes. Hence, the cost of computing the interaction matrix is negligible compared to the total cost of reordering a large BDD. The space requirements are quadratic in the number of variables. However, the matrix is symmetric, and it needs only one bit per element. Hence, even for a BDD with 2000 variables the interaction matrix requires only one quarter of a megabyte.

It should be pointed out that the interaction matrix based on (1) does not identify all inexpensive swaps. This is illustrated in Fig. 3, in which the same function is shown with two variable orders. With the order on the left, y and z do not interact; with the order on the right they do. The interaction matrix will conservatively say that the two variables always interact.

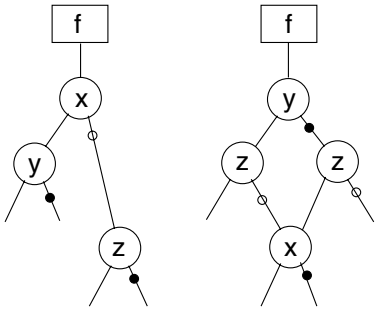

Fig. 3. Limitations of the interaction matrix

5.3 Lower bounds on the BDD size

In the sifting algorithm [34] for reordering, a variable is moved by a sequence of swaps until it visits all positions in the order. It is then moved to a position for which the size of the BDD was minimal. (See Fig. 4.) It is often the case that letting a variable take all positions in the order is not necessary to find the best one: it has been observed empirically that the size of the BDDs are only minimally affected if movement of one variable in one direction is abandoned as soon as the BDD grows by a small percentage (less than 20%) over its initial size. Besides limiting the excursion of a variable if the BDD grows too large, implementations can also compute bounds on the minimum size achievable by the BDD when a variable is moved either up or down.

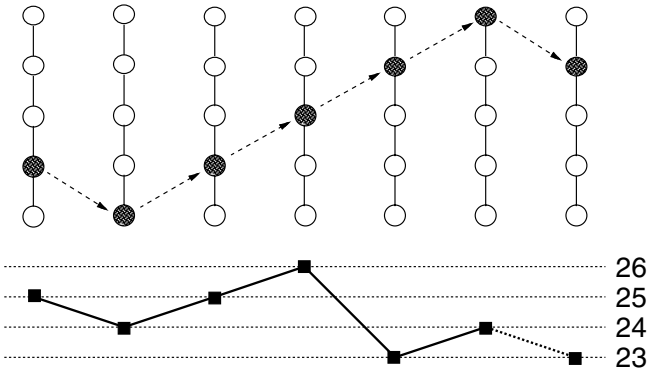

Fig. 4. The sifting algorithm. The upper part of the figure shows the movement of a variable, while the lower part shows the changes of the BDD size that result from it

Let N_i, $0 \le i < n$ be the number of nodes at level i in the BDD. Suppose variable x is currently at position j in the order and is being moved up. Then the size of the BDD cannot be reduced below:

$$LB^{\uparrow} = N_0 + \sum_{j<i<n} N_i \,. \tag{2}$$

Equation (2) states that the part of the BDD below the variable being sifted is unaffected, and the nodes at the top of the order cannot disappear. If x is being moved down, then the size of the BDD cannot be reduced below:

$$LB^{\downarrow} = N_j + \sum_{0\le i<j} N_i \,. \tag{3}$$

In words, the part of the BDD above the variable being sifted is unaffected. The node count for the variable x_j being moved down may change, but the pointers pointing to the x_j-nodes before the swap still point to different nodes after the swap, so that the number of nodes in the bottom part is still bounded by N_j. These bounds can be updated in constant time while sifting a variable. The bounds can be improved by using the interaction matrix. If a variable does not interact with x, then its node count will not change as a result of sifting x. This improvement is particularly significant when the BDD has many roots, each depending on a small set of variables.

Figure 5 shows an example in which the bounds of (2) and (3) are met. However, assuming that the entire

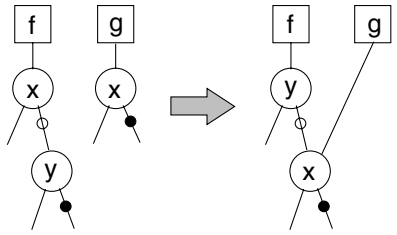

Fig. 5. Lower bounds on the size of a BDD. Equations (2) and (3) both predict a minimum size of 2

segment of the BDD ahead of x in the direction of its movement will vanish (with the possible exception of the first level) when x moves up in the order, often leads to underestimation of the lower bounds. A more accurate technique has been proposed in [12]. It is based on the ideas of [5], from which one can derive the following bounds:

$$LB^{\uparrow} = E^{\uparrow} + \sum_{j<i<n} N_i \,, \tag{4}$$

$$LB^{\downarrow} = E^{\downarrow} + \sum_{0\le i<j} N_i \,, \tag{5}$$

with $E^{\uparrow}$ and $E^{\downarrow}$ given by:

$$E^{\uparrow} = \min_{0\le k<j} \left[\max\left(N_k, (j-k) + \frac{N_j}{2^{j-k}} \right) + \sum_{0<i<k} N_i \right] ,$$

$$E^{\downarrow} = \min_{j<k<n} \left[\max\left(N_j, 1 + \frac{1}{2} \sum_{j<i\le k} N_i \right) + \sum_{k<i<n} N_i \right] .$$

The expression for $E^{\downarrow}$ can be simplified to:

$$E^{\downarrow} = \max\left(N_j, 1 + \frac{1}{2} \sum_{j<i<n} N_i \right) ,$$

and therefore $LB^{\downarrow}$ can be updated in constant time while moving a variable. The cost of updating $LB^{\uparrow}$, on the other hand, is proportional to the number of variables.

5.4 Reordering algorithms

Various algorithms have been proposed to find good variable orders of BDDs, e.g., Sect. 2.4 in [13]. The algorithms that compute an optimal order are based on dynamic programming [16], and are only applicable to small problems, typically less than 30 variables. Heuristic algorithms can be loosely classified into expensive (those based on simulated annealing or evolutionary algorithms) and not-so-expensive (window permutation, sifting, and its variants). The former are mostly limited by the size of the BDDs and tend to become very slow beyond 100,000 nodes. On the other hand, they tend to produce very good orders. The latter can be used even for millions of nodes, and usually produce acceptable results.

All these algorithms – both exact and heuristic – can be implemented in terms of swaps of adjacent variables. The natural consequence of this approach is that only one BDD is kept in memory during the computation. For algorithms that work on a family of BDDs (e.g., dynamic programming and evolutionary algorithms) only one BDD and the orders for each member of the family are stored. When a given BDD is needed it is constructed by applying the desired variable permutation to the BDD in memory. This strategy obviously saves memory, and works well thanks to the efficiency of the in-place variable swap.

Among the heuristic algorithms, sifting [34] is the most used because of the good balance between cost and quality of results. One of the shortcomings of the original sifting algorithm is that it only tries to move one variable at the time. Sometimes a variable is in the wrong place in the order, but moving it actually increases the size of the BDDs unless some other variables are moved at the same time. This is typically the case of symmetric variables. In these cases, the solution consists of sifting groups of variables. Criteria to decide aggregation can be derived by relaxing the variable symmetry conditions [32]. These conditions specify, for instance, that if x and y are adjacent variables in the order with x preceding y, for all nodes labeled x, $f_{10} = f_{01}$ must hold. Relaxing this rule leads to allowing a small fraction of x-nodes for which the condition is violated.

Another reason for moving groups of variables instead of individual variables is to enforce constraints on the positions of related variables. One common heuristic used in sequential verification is to force pairs of corresponding current state-next state variables to remain adjacent. The advantage of this scheme is to minimize the work required to translate the BDDs representing sets of states between the two sets of state variables. In some cases, however, this advantage is paid at a fairly high cost in terms of BDD sizes [19].

Reordering algorithms may be invoked explicitly by the application, or they may be invoked *dynamically.* In the latter case, the BDD package must decide when to trigger reordering. The following simple policy is used in several implementations. The first reordering takes place as soon as the BDD reaches a given size. The thresholds for the successive reorderings are based on some combination of the previous threshold and the size of the BDD after reordering. Using the size of the BDD after reordering to determine the next threshold tends to cause more frequent reorderings. This works better when building BDDs for combinational circuits than when performing, for instance, a model checking experiment. In general, application of reordering to sequential verification is more problematic, because of the very large BDDs that may be involved, and the more complex evolution of the experiments, in which BDDs are repeatedly created and destroyed. Several heuristics have been recently proposed to address these issues [18, 21, 37, 40], for which a satisfactory solution is not yet available.

6 Statistics

Many implementation decisions that we have described in the previous sections are based on experimental evaluation more than on theoretical analysis. Hence, in the development of a BDD package, one must pay due attention to the collection of statistics. This collection takes two forms.

First, the package must gather useful information on each run. This information concerning the amount of resources used and the work done is useful at three levels:

- The developer examines the profiles of the runs to make implementation choices and detect inefficiencies.
- The package relies on some statistics for various policies, including the sizing of the tables.
- The sophisticated user examines the profiles to choose the optimal values of parameters for his/her application.

A typical aspect that a package should monitor is the performance of the hash functions. Consider in particular the hash function for the computed table. By counting the number of insertions, α, and the number of empty slots, ϵ, one can compare the actual performance of the hash function to an ideal one, which uniformly distributes the accesses. For a sufficiently large number of slots σ, the expected number of unused slots can be estimated by assuming a Poisson distribution, which yields:

$$\frac{\epsilon}{\sigma} \approx e^{-\frac{\alpha}{\sigma}} .$$

This sanity check on the performance of the hash function can be implemented at negligible cost in terms of memory and time. In collecting access counts it is important to avoid overflow. Since long runs can easily produce billions of recursive calls, and since 64-bit integers are not yet universally available, it is good practice to use double-precision floating point numbers for the statistical counters.

The second important form of analysis of a BDD package is through the use of profiling tools. These include both tools for the counting of basic blocks, and tools that evaluate effects like cache and translation look-aside buffer (TLB) hit rates [2, 39]. Basic-block counting is the main tool to fine tune the code, but it does not take into account the impact of the memory hierarchy and other architectural effects. Figure 6 shows a summary produced by running DCPI [2] to monitor the execution of a set of symbolic model checking [28] experiments. DCPI classifies CPU stalls according to their causes. Since it is based on sampling, the tool reports an interval for each cause of dynamic stalls. The report of Fig. 6 says that almost two-thirds of the CPU cycles are stalls. It also points out that data cache misses and data TLB misses are responsible for about half of those stalls. DCPI can also provide a break-down of the cache or TLB misses according to the functions causing them. In the case of the runs summarized in Fig. 6, accesses to the computed table are responsible for about half of the misses.

I-cache (not ITB)	0.8%	to	4.1%	
ITB/I-cache miss	4.6%	to	4.9%	
D-cache miss	24.5%	to	26.5%	
DTB miss	9.2%	to	11.4%	
Write buffer	0.1%	to	0.8%	
Synchronization	0.0%	to	0.0%	
Branch mispredict	1.1%	to	4.6%	
IMUL busy	0.0%	to	0.0%	
FDIV busy	0.0%	to	0.0%	
Other	1.4%	to	1.4%	
Unexplained stall	2.5%	to	2.5%	
Unexplained gain	-0.5%	to	-0.5%	
Subtotal dynamic				49.2%
Slotting	5.2%			
Ra dependency	6.1%			
Rb dependency	2.5%			
Rc dependency	0.0%			
FU dependency	0.9%			
Subtotal static				14.8%

Fig. 6. Performance analysis report produced by DCPI [2] for a set of model checking runs

A combination of tools and appropriate instrumentation of the code can shed light on the nature of BDD computations and can lead to remarkable improvements in performance. As an example, consider the comparison between ordinary, depth-first implementation of BDDs operations, and levelized implementation (improperly dubbed breadth-first). The latter is clearly superior in reducing paging. However, even for the more efficient levelized algorithms, performance degrades quickly when paging occurs. Hence, the main interest in these algorithms is due to the fact that in certain cases, they outperform the depth-first approaches even when the number of page faults is negligible. A plausible explanation that has been advanced is that the increased memory locality due to the levelized processing leads to better utilization of the memory hierarchy. However, the measurements of [27] point out that the main reason for differences in performance is often the lossless computed table, which is intrinsic to the levelized processing. The improved memory locality, on the other hand, is often offset by an increase in memory references. Further insight on the comparison between the two approaches is provided by the study of B. Yang et al. [41], who noticed that the different mixes of operations encountered in building BDDs for combinational circuits and in model checking make levelized processing much more effective for the former application. Another important contribution of [41] is the establishment of a framework for the comparative evaluation of different BDD packages.

7 Conclusions

Many applications are built today around a BDD package. Though, in a sense, a BDD package is not unlike a linked list package, or a balanced tree package, in an important respect it is different: in many cases significant algorithmic advances in an application come from the development of new operations on BDDs. (It suffices to mention the constrain operator [11].) The boundary between a symbolic model checker and its BDD package is often blurred. In spite of this it is advisable to keep as clean an interface as possible between an application and its BDD package. This can be done by interposing a translation layer between the application and the BDD package. As an example, the model checker VIS [7] currently provides interfaces to three different BDD packages [23, 36, 38], and will be able to accommodate new packages that may become available. However, some algorithms in VIS are only available when a specific BDD package is used.

The cost of the additional layer in terms of efficiency is typically negligible. (Though there are obviously exceptions.) It is quite common for a large model checking run to spend 90% of the time in the *AndExist* function (which computes the product of two operands and quantifies some variable out of the result). If dynamic reordering is enabled, the function that swaps adjacent variables will get the lion's share of the CPU time. The time spent in the interface, if it is designed properly, may be less than 1%. This opens the possibility for mixed-language implementations. The cores of BDD packages are usually written in C or C++. However, the applications can be written in Perl, Lisp, or Java, to name a few. In particular, it is possible to rely on the automatic garbage collection features of these languages, without fear of inefficiency.

As recent advances [24, 41] indicate, the evolution of BDD packages is still very rapid, both from the point of view of the fundamental data structures, and from the point of view of the coordination between package and application. In taking leave from this article, we hope it will help in the development of new, better implementations.

Acknowledgements.

The large number of "personal communications" in the references is an indication of how much this author is indebted to friends and colleagues. Special thanks to Rick Rudell, Dave Long, and Jean Christophe Madre for sharing their BDD lore over the years, and to all the participants in the FMCAD BDD study that Bwolen Yang enthusiastically coordinated.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: Data Structures and Algorithms. Addison-Wesley, Reading, Mass., USA, 1983
2. Anderson, J., Berc, L.M., Dean, J., Ghemawat, S., Henzinger, M.R., Leung, S.-T., Sites, R.L., Vandevoorde, M.T., Waldspurger, C.A.: Continuous profiling: where have all the cycles gone? ACM Trans. on Computer Systems, pp. 357–390, November 1997
3. Ashar, P., Cheong, M.: Efficient breadth-first manipulation of binary decision diagrams. In: Proc. Int. Conf. on Computer-Aided Design, pp. 622–627, San Jose, Calif., USA, November 1994
4. Biere, A.: Personal communication, 1998
5. Bollig, B., Löbbing, M., Wegener, I.: On the effect of local changes in the variable ordering of ordered decision diagrams. Inf. Process. Lett. 59(5): 233–239, 1996
6. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient implementation of a BDD package. In: Proc. 27th Design Automation Conference, pp. 40–45, Orlando, Fla., USA, June 1990
7. Brayton, R.K. et al.: VIS: A system for verification and synthesis. In: Henzinger, T., Alur, R. (eds.): 8th Conference on Computer Aided Verification (CAV'96). Rutgers University. LNCS 111. Berlin, Heidelberg, New York: Springer-Verlag, 1996, pp. 428–432
8. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. on Computers C-35(8): 677–691, August 1986
9. Bryant, R.E.: Personal communication, 1995
10. Coudert, O.: Personal communication, 1998
11. Coudert, O., Berthet, C., Madre, J.C.: Verification of sequential machines using boolean functional vectors. In: Claesen, L. (ed.): Proc. IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, pp. 111–128, Leuven, Belgium, November 1989
12. Drechsler, R., Günther, W.: Using lower bounds during dynamic BDD minimization. In: Proc. Design Automation Conference, pp. 29–32, New Orleans, La., USA, June 1999
13. Drechsler, R., Sieling, D.: Binary decision diagrams in theory and practice. Software Tools Technol. Transfer. This issue
14. Friedman, S.J., Supowit, K.J.: Finding the optimal variable ordering for binary decision diagrams. IEEE Trans. on Computers 39(5): 710–713, May 1990
15. Fujita, M., Matsunaga, Y., Kakuda, T.: On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In: Proc. European Conference on Design Automation, pp. 50–54, Amsterdam, February 1991
16. Held, M., Karp, R.M.: A dynamic programming approach to sequencing problems. J. SIAM 10(1): 196–210, 1962
17. Hett, A., Drechsler, R., Becker, B.: MORE: alternative implementation of BDD-packages by multi-operand synthesis. In: Proc. European Design Automation Conference, pp. 164–169, 1996
18. Higuchi, H.: Personal communication, 1998
19. Higuchi, H., Somenzi, F.: Lazy group sifting for efficient symbolic state traversal of FSMs. In: Proc. Int. Conf. on Computer-Aided Design, pp. 45–49, San Jose, Calif., USA, November 1999
20. Hong, Y., Beerel, P.A., Burch, J.R., McMillan, K.L.: Safe BDD minimization using don't cares. In: Proc. Design Automation Conference, pp. 208–213, Anaheim, Calif., USA, June 1997
21. Kamhi, G., Fix, L.: Adaptive variable reordering for symbolic model checking. In: Proc. Int. Conf. on Computer-Aided Design, pp. 359–365, San Jose, Calif., USA, November 1998
22. Karplus, K.: Using if-then-else DAG's for multi-level minimization. In: Decennial Caltech Conference on VLSI, May 1989
23. Long, D.E.: Robdd package, 1993
24. Long, D.E.: The design of a cache-friendly BDD library. In: Proc. Int. Conf. on Computer-Aided Design, pp. 639–645, San Jose, Calif., USA, November 1998
25. Madre, J.C.: Personal communication, 1996
26. Madre, J.C., Billon, J.P.: Proving circuit correctness using formal comparison between expected and extracted behavior. In: Proc. 25th Design Automation Conference, pp. 205–210, June 1988
27. Manne, S., Grunwald, D.C., Somenzi, F.: Remembrance of things past: locality and memory in BDDs. In: Proc. Design Automation Conference, pp. 196–201, Anaheim, Calif., USA, June 1997
28. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic, Boston, Mass., USA, 1994

29. McMillan, K.L.: A conjunctively decomposed Boolean representation for symbolic model checking. In: Alur, R., Henzinger, T.A. (eds.): 8th Conference on Computer Aided Verification (CAV'96). LNCS 1102. Berlin, Heidelberg, New York: Springer-Verlag, 1996, pp. 13–25
30. Minato, S.-I., Ishiura, N., Yajima, S.: Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In: Proc. Design Automation Conference, pp. 52–57, Orlando, Fla, USA, June 1990
31. Ochi, H., Yasuoka, K., Yajima, S.: Breadth-first manipulation of very large binary-decision diagrams. In: Proc. Int. Conf. on Computer-Aided Design, pp. 48–55, Santa Clara, Calif., USA, November 1993
32. Panda, S., Somenzi, F.: Who are the variables in your neighborhood. In: Proc. Int. Conf. on Computer-Aided Design, pp. 74–77, San Jose, Calif., USA, November 1995
33. Ravi, K., McMillan, K.L., Shiple, T.R., Somenzi, F.: Approximation and decomposition of decision diagrams. In: Proc. Design Automation Conference, pp. 445–450, San Francisco, Calif., USA, June 1998
34. Rudell, R.: Dynamic variable ordering for ordered binary decision diagrams. In: Proc. Int. Conf. on Computer-Aided Design, pp. 42–47, Santa Clara, Calif., USA, November 1993
35. Rudell, R.: Personal communication, 1994
36. Sanghavi, J.V., Ranjan, R.K., Brayton, R.K., Sangiovanni-Vincentelli, A.: High performance BDD package based on exploiting memory hierarchy. In: Proc. Design Automation Conference, pp. 635–640, Las Vegas, Nev., USA, June 1996
37. Slobodova, A., Meinel, C.: Sample method for minimization of OBDDs. In: SOFSEM'98: Theory and Practice of Informatics. LNCS 1521. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 419–428
38. Somenzi, F.: CUDD: CU Decision Diagram Package. University of Colorado at Boulder, `ftp://vlsi.colorado.edu/pub/`
39. Srivastava, A., Eustace, A.: ATOM: A system for building customized program analysis tools. In: Proc. ACM SIGPLAN'94 Conference on Programming Language Design and Implementation, pp. 196–205, June 1994
40. Stangier, C.: Accelerating variable reordering. Presented at the Dagstuhl Seminar on Decision Diagrams, Concepts and Applications, January 1999
41. Yang, B., Bryant, R.E., O'Hallaron, D.R., Biere, A., Coudert, O., Janssen, G., Ranjan, R.K., Somenzi, F.: A performance study of BDD-based model checking. In: Gopalakrishnan, G., Windley, P. (eds.): Formal Methods in Computer Aided Design. LNCS 1522. Berlin, Heidelberg, New York: Springer-Verlag, 1998, pp. 255–289
42. Yang, B., Chen, Y.-A., Bryant, R.E., O'Hallaron, D.R.: Space- and time-efficient BDD construction via working set control. In: Proc. of Asia and South Pacific Design Automation Conference (ASPDAC'98), pp. 423–432, Yokohama, Japan, February 1998