Efficient Implementation of a BDD Package

Karl S. Brace*
Dept of ECE
Carnegie Mellon
Pittsburgh, PA 15213

Richard L. Rudell
Synopsys, Inc.
1098 Alta Avenue
Mountain View, CA 94043

Randal E. Bryant*
School of Computer Science
Carnegie Mellon
Pittsburgh, PA 15213

Abstract

Efficient manipulation of Boolean functions is an important component of many computer-aided design tasks. This paper describes a package for manipulating Boolean functions based on the reduced, ordered, binary decision diagram (ROBDD) representation. The package is based on an efficient implementation of the if-then-else (ITE) operator. A hash table is used to maintain a *strong canonical form* in the ROBDD, and memory use is improved by merging the hash table and the ROBDD into a hybrid data structure. A *memory function* for the recursive ITE algorithm is implemented using a *hash-based cache* to decrease memory use. Memory function efficiency is improved by using rules that detect when equivalent functions are computed. The usefulness of the package is enhanced by an automatic and low-cost scheme for recycling memory. Experimental results are given to demonstrate why various implementation trade-offs were made. These results indicate that the package described here is significantly faster and more memory-efficient than other ROBDD implementations described in the literature.

1 Introduction

The efficient representation and manipulation of Boolean functions is important for many algorithms in a wide variety of applications. In particular, many problems in computer-aided design for digital circuits (CAD) can be expressed as a sequence of operations performed over a set of Boolean functions. Some examples from CAD are combinational logic verification [1, 2], sequential-machine equivalence [3], logic optimization of combinational circuits [4], test pattern generation [5], timing verification in the presence of false paths [6], and symbolic simulation [7].

Hence it is desirable to develop a general-purpose software package for manipulating Boolean functions which allows variables to be created, and allows standard Boolean operations such as AND, OR, and NOT to be performed on functions. The package should also allow a function to be tested for *tautology* – i.e., to determine whether the function evaluates to *1* for all inputs.

The problem with developing such a package is that the tautology problem is co-NP complete [8]. This implies that all known solutions require time which grows exponentially with the number of variables in the worst case. However, by developing clever representations and efficient manipulation algorithms, it is often possible to avoid an exponential computation.

Many different representations have been proposed for manipulating Boolean functions and each has a corresponding algorithm to test for tautology. However, many of the functions of interest in CAD have an exponential size in the sum-of-products representation [9], and checking tautology in a general Boolean network appears to be intractable [10], making these representations unacceptable for a general package.

The representation we have found most useful for manipulating Boolean functions is the reduced, ordered, binary-decision diagram (ROBDD) [11, 12, 13]. The ROBDD is a canonical form, so the tautology test is a constant-time comparison against the unique representation of the function **1**. While the size of the ROBDD can be exponential in the worst case, ROBDD's remain small for many of the functions we are interested in.

We are aware of several computer implementations of ROBDD's, but few have been put forward as a reusable package, and fewer still have had their performance measured and compared. Our primary goal was to develop a generic package interface that would hide the details of the package implementation, yet still be efficient in computer run-time and memory use. We also wanted to understand the various trade-offs possible in an ROBDD package to tailor such a package for our applications in CAD.

2 Programming Techniques

A *hash table* associates a *value* with a *key*. A *hash function* applied to the *key* selects which of N linked lists the *key,value* pair is stored. The *load factor* of a hash table is defined as $\alpha = n/N$, where n is the number of keys stored in the table.

A *memory function* for the function F is a table of values $(x, F(x))$ that the function has already computed. If F is called with argument x again, $F(x)$ is returned without any computation.

A *hash-based cache* is a hash table where a collision chain is not used to resolve collisions. Instead, at insert time, any existing element at the particular array position is discarded and replaced with the new entry. At lookup time, if the element does not match the stored key, a cache miss occurs and no element is returned.

A *strong canonical form* is a form of pre-conditioning which reduces the complexity of an equivalence test between elements in a set. A *unique id* is assigned to each unique element in the set, so an equivalence test is a simple scalar test between the unique id's of each element.

Garbage collection is a class of techniques to periodically free unused memory. It is useful when references to the structures being freed prevent incremental freeing. The cost of searching for these references is amortized over many free operations.

3 BDD Overview

3.1 Basic Definitions

Basic definitions for binary decision diagrams (also known as function graphs) are given in [13]. We review some of these definitions here for reference.

*This research partially funded by Semiconductor Research Corporation contract number 90-DC-068.

A *binary decision diagram* (BDD) is a directed acyclic graph (DAG). The graph has two sink nodes labeled **0** and **1** representing the Boolean functions 0 and 1. Each non-sink node is labeled with a Boolean variable v and has two out-edges labeled *1* (or *then*) and *0* (or *else*). Each non-sink node represents the Boolean function corresponding to its *1* edge if $v = 1$, or the Boolean function corresponding to its *0* edge if $v = 0$.

An *ordered binary decision diagram* (OBDD) is a BDD with the constraint that the input variables are ordered and every source to sink path in the OBDD visits the input variables in ascending order.

A *reduced ordered binary decision diagram* (ROBDD) is an OBDD where each node represents a distinct logic function.

Bryant [13] was the first to prove that the ROBDD is well-defined. Bryant also showed the ROBDD is a canonical form for a logic function; that is, two functions are equivalent if, and only if, the ROBDD's for each function are isomorphic.

It is well known that the size of the ROBDD for a given function depends on the variable order chosen for the function. This paper is not concerned with the variable ordering problem. In practice, we have found that a simple topological based ordering heuristic, such as proposed by Malik *et al.* [1], is sufficient for many applications in CAD.

4 Implementation

4.1 Notation

The sink nodes of the ROBDD are written as **1** and **0**. A variable is denoted by a lowercase letter, such as v. The variables in the ROBDD are totally ordered. We say that v is smaller than w ($v < w$) if v comes before w in the variable order (higher up in the ROBDD).

At each node F there is a variable v and v is called the *top variable* of F. The top variable of a set of formulas is the smallest of the top variables of those formulas.

Each node in the ROBDD represents a Boolean function, and is written using a capital letter, such as F, and can be denoted by the triple (v, G, H), where v is the top variable of F, G is the node connected to the *1* (or *then*) edge of F, and H is the node connected to the *0* (or *else*) edge of F. A node in the ROBDD which represents a function the user is interested in is called a *formula*. Other nodes, which are needed to build a user's formula, are called *internal nodes*.

$|F|$ is the number of nodes below F in the ROBDD.

4.2 ITE Operator

The If-Then-Else or ITE operator forms the core of the package. ITE is a Boolean function defined for three inputs F, G, H which computes: If F then G else H. This is equivalent to:

$$ite(F, G, H) = F \cdot G + \overline{F} \cdot H.$$

It is well known that the ITE operation can be used to implement all two-variable Boolean operations as shown in Figure 1. Also, because ITE is the logical function performed at each node of the ROBDD, it is an efficient building block for many other operations on the ROBDD. The programming language function for the ITE operator will be written as *ite*.

4.3 Unique-Table

A hash table imposes a strong canonical form on the nodes in the ROBDD, so that each node in the ROBDD represents a unique logic function. Hence, this hash table is called the *unique-table*.

Table	Name	Expression	Equivalent form
0000	0	0	0
0001	AND(F,G)	$F \cdot G$	$ite(F, G, 0)$
0010	F > G	$F \cdot \overline{G}$	$ite(F, \overline{G}, 0)$
0011	F	F	F
0100	F < G	$\overline{F} \cdot G$	$ite(F, 0, G)$
0101	G	G	G
0110	XOR(F,G)	$F \oplus G$	$ite(F, \overline{G}, G)$
0111	OR(F,G)	$F + G$	$ite(F, 1, G)$
1000	NOR(F,G)	$\overline{F + G}$	$ite(F, 0, \overline{G})$
1001	XNOR(F,G)	$\overline{F \oplus G}$	$ite(F, G, \overline{G})$
1010	NOT(G)	$\overline{G}$	$ite(G, 0, 1)$
1011	F ≥ G	$F + \overline{G}$	$ite(F, 1, \overline{G})$
1100	NOT(F)	$\overline{F}$	$ite(F, 0, 1)$
1101	F ≤ G	$\overline{F} + G$	$ite(F, G, 1)$
1110	NAND(F,G)	$\overline{F \cdot G}$	$ite(F, \overline{G}, 1)$
1111	1	1	1

Figure 1: All two variable functions described using ITE.

The unique-table maps a triple (v, G, H) to an ROBDD node $F = (v, G, H)$. Each node in the ROBDD has an entry in the unique-table. Before a new node is added to the ROBDD, a lookup in the unique-table determines if a node for that function already exists. If so, the existing node is used. Otherwise, the new node is added to the ROBDD and a new unique-table entry is made. By assumption, when we create a new node F, the nodes G and H will already obey the strong canonical form. Hence, the function F exists in the ROBDD if, and only if, the triple (v, G, H) is already in the unique-table, thus maintaining the strong canonical form.

The unique table allows a single *multi-rooted* DAG to represent all of the user's formulae simultaneously.

4.4 Recursive Formulation of ITE

Shannon's decomposition theorem states that

$$F = v \cdot F_v + \overline{v} \cdot F_{\overline{v}}$$

where F_v and $F_{\overline{v}}$ are F evaluated at $v = 1$ and $v = 0$ respectively. Let $F = (w, T, E)$ and assume $v \leq w$. Finding the cofactors of F with respect to v is trivial: $F_v = F$ (if $v < w$) or T (if $v = w$), and $F_{\overline{v}} = F$ (if $v < w$) or E (if $v = w$).

The following recursive formulation is the key to computing $ite(F, G, H)$ for functions F, G, H represented in ROBDD form. Let $Z = ite(F, G, H)$ and let v be the top variable of F, G, H. Then,

$$\begin{aligned} Z &= v \cdot Z_v + \overline{v} \cdot Z_{\overline{v}} \\ &= v \cdot (F \cdot G + \overline{F} \cdot H)_v + \overline{v} \cdot (F \cdot G + \overline{F} \cdot H)_{\overline{v}} \\ &= v \cdot (F_v \cdot G_v + \overline{F_v} \cdot H_v) + \overline{v} \cdot (F_{\overline{v}} \cdot G_{\overline{v}} + \overline{F_{\overline{v}}} \cdot H_{\overline{v}}) \\ &= ite(v, ite(F_v, G_v, H_v), ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}})) \\ &= (v, ite(F_v, G_v, H_v), ite(F_{\overline{v}}, G_{\overline{v}}, H_{\overline{v}})) \end{aligned}$$

The terminal cases for this recursion are: $ite(1, F, G) = ite(0, G, F) = ite(F, 1, 0) = F$.

We note that this formulation is valid for any Boolean function of any number of variables. However, we use the ITE function for the reasons mentioned earlier.

4.5 Memory Function for ITE

We use a memory function to improve the performance of *ite*. Bryant mentioned the use of a memory function for operations

```
ite(F, G, H){
    if (terminal case) {
        return result;
    } else if (computed-table has entry {F, G, H}) {
        return result;
    } else {
        let v be the top variable of {F, G, H};
        T = ite(F_v, G_v, H_v);
        E = ite(F_v̄, G_v̄, H_v̄);
        if T equals E return T;
        R = find_or_add_unique_table(v, T, E);
        insert_computed_table({F, G, H}, R);
        return R;
    }
}
```

Figure 2: The *ite* algorithm.

on an ROBDD [13], and this idea has been used in other implementations [14, 15]. We call the memory function for *ite* the *computed-table*. The computed-table maps three nodes F, G, H to the result node $ite(F, G, H)$ once this result has been computed. Assume for now that the computed-table is implemented using a hash table.

4.6 ITE Algorithm

Figure 2 shows the outline of the complete *ite* algorithm. With the assumption of constant time lookup and insert in the computed and unique-tables, all operations in *ite* take constant time. Observe that *ite* can be called at most once for each combination of nodes in F, G, H, i.e., $O(|F| \cdot |G| \cdot |H|)$ times. So the time complexity is $O(|F| \cdot |G| \cdot |H|)$. In practice, the typical performance is closer to the size of the resulting function.

An example of *ite* is shown in Figure 3.

5 ROBDD Extensions

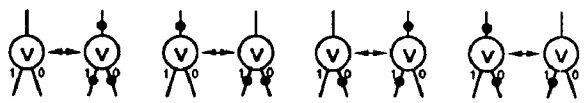
5.1 Complement Edges

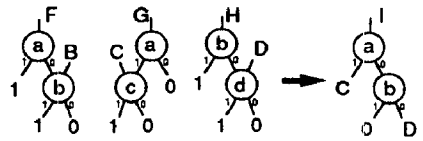
The first extension we consider is introducing complement edges into the ROBDD. Akers [16] describes using complement edges for hand-generated BDD's. Both Karplus [15] and Madre [17] formulated sets of rules to guarantee canonical ROBDD's using complement edges. Our implementation is similar to these.

Consider, for example, the ROBDD nodes for G and $\overline{G}$ which are similar except that their sink nodes **0** and **1** are interchanged. This similarity can be exploited by using *complement edges*. A complement edge is an ordinary edge with an extra bit (*complement bit*) set to indicate that the connected formula is to be interpreted as the complement of the ordinary formula. Therefore $\overline{G}$ could be represented by a complement edge to the node for G, saving intermediate nodes.

In our notation, when we say node F (or formula F), we are referring to a node referenced through either an ordinary or complement edge and $\overline{F}$ is the same node referenced through the other kind of edge. Note that we only need one constant node. We chose to keep **1**, allowing the function **0** to be represented by a complement edge to **1**.

To maintain a canonical form, we must constrain where complement edges are used. A dot on an edge indicates it is a complement edge. The following 4 pairs of functions are functionally equivalent:





$$
\begin{aligned}
I &= ite(F, G, H) \\
&= (a, ite(F_a, G_a, H_a), ite(F_{\overline{a}}, G_{\overline{a}}, H_{\overline{a}})) \\
&= (a, ite(1, C, H), ite(B, 0, H)) \\
&= (a, C, (b, ite(B_b, 0_b, H_b), ite(B_{\overline{b}}, 0_{\overline{b}}, H_{\overline{b}}))) \\
&= (a, C, (b, ite(1, 0, 1), ite(0, 0, D))) \\
&= (a, C, (b, 0, D))
\end{aligned}
$$

Figure 3: Example of *ite*()

The ROBDD must follow this rule: the *then* edge of every node must be a regular edge. Thus, we always choose the left member of each equivalent pair above. This guarantees a canonical form, as no function-preserving change to an ROBDD which follows this rule can yield a different ROBDD which also follows this rule.

Therefore G and $\overline{G}$ are represented by the same node, and the complement operation and the identification of complement functions takes constant time. Therefore we add another terminal case to *ite*: $ite(F, 0, 1) = \overline{F}$. Complement edges are realized at a negligible processing cost in *ite*. There is no added memory overhead because we use the low bit of each node pointer as the complement bit, although a separate bit could be used on a machine where this is not allowed.

For the set of 12 examples presented in Section 6, we find that the final DAG is 7% smaller when complement edges are used. However, the total run-time needed to form the DAG for these examples is decreased by almost a factor of 2. The large decrease in run-time is mostly due to the ability of the ROBDD package to support a constant-time complement operation.

5.2 Standard Triples

For the function and parameters $ite(F_1, F_2, F_3)$ there may exist parameters G_1, G_2, G_3 such that $ite(F_1, F_2, F_3) = ite(G_1, G_2, G_3)$ but $F_i \neq G_i$ for some i. We define an equivalence relation on sets of three functions F_1, F_2, F_3 based on the equivalence of the Boolean function $\mathrm{ITE}(F_1, F_2, F_3)$. We would like to choose a *standard* triple from each equivalence set where the result of the *ite* is stored. Therefore, on any call to $ite(F_1, F_2, F_3)$, the standard arguments G_1, G_2, G_3 are substituted first before any lookup or entries are made in the computed-table. This improves the efficiency of the computed-table by reducing the storage required in the computed-table and eliminating some recomputation which would yield an equivalent result.

Because of the strong canonical form and the use of complement edges, it is possible to recognize when two functions are equal or the complements of each other in constant time. Using only these two queries, we can easily detect when equivalent two-variable Boolean functions are computed. For example the following calls to *ite* are all functionally equivalent to $F + G$:

$$ite(F, F, G) = ite(F, 1, G) = ite(G, 1, F) = ite(G, G, F).$$

We choose the standard triple from this set as follows. First, the following simplifications are applied to the arguments of the *ite* where possible:

$$
\begin{aligned}
ite(F, F, G) &\Rightarrow ite(F, 1, G) \\
ite(F, G, F) &\Rightarrow ite(F, G, 0) \\
ite(F, G, \overline{F}) &\Rightarrow ite(F, G, 1) \\
ite(F, \overline{F}, G) &\Rightarrow ite(F, 0, G)
\end{aligned}
$$

As stated previously checking $F = G$ and $F = \overline{G}$ are constant time operations. Next, consider the following equivalent pairs:

$$\begin{aligned} ite(F,1,G) &= ite(G,1,F) \\ ite(F,G,0) &= ite(G,F,0) \\ ite(F,G,1) &= ite(\overline{G},\overline{F},1) \\ ite(F,0,G) &= ite(\overline{G},0,\overline{F}) \\ ite(F,G,\overline{G}) &= ite(G,F,\overline{F}) \end{aligned}$$

To choose the unique element, for example, between $ite(F,1,G)$ and $ite(G,1,F)$, the first argument of the *ite* is given the formula with the smallest top variable. In the case of a tie, the formulas are ordered based on their unique id (in C, the address of the node).

At this point, the simplified arguments to *ite* are F, G, H. Complement edges lead to the following equivalences:

$$ite(F,G,H) = ite(\overline{F},H,G) = \overline{ite(F,\overline{G},\overline{H})} = \overline{ite(\overline{F},\overline{H},\overline{G})}$$

A unique triplet is chosen from these four forms according to the rule that the first and second arguments to *ite* should not be complement edges. Given arbitrary values for F, G, and H, this condition is met by exactly one of the above forms. For the last two cases, the computation will yield the complement of the function, and then the function will be complemented before it is returned.

Note that these rules effectively detect equivalences according to DeMorgan's Laws. For example, suppose that A and B are both regular edges, and we first compute $A + B$, which will become $ite(A,1,B)$. If we later compute $\overline{A} \cdot \overline{B}$ as $ite(\overline{A},\overline{B},0)$, this will become $\overline{ite(A,1,B)}$. The computed table will have the result, which will only need to be complemented before being returned. Likewise, we can detect when redundant computation is performed, for example $F + \overline{F} = ite(F,1,\overline{F}) = ite(F,1,1) = 1$. Bryant's *apply* operation, which performs an arbitrary operation on two formulae [13], does not recognize these equivalences.

The complete set of terminal cases for the recursion are: $ite(F,1,0) = ite(1,F,G) = ite(0,G,F) = ite(G,F,F) = F$ and $ite(F,0,1) = \overline{F}$.

5.3 The ite_constant Algorithm

The *ite_constant* algorithm, outlined in Figure 4, is a modification of *ite* which returns a result only if it is a constant function; otherwise, it returns a failure value. *ite_constant* is useful for testing logical implication because $F \leq G$ (i.e., F implies G) is the same as $ite\_constant(F,G,1) = 1$. This can be done much more efficiently, on average, than computing the result of the *ite* and checking for a constant value because no intermediate nodes are constructed and the routine exits as soon as the result is found to be nonconstant.

5.4 Garbage Collection

The implementation described here is the first to include automatic garbage collection. Each node F has a reference count of the number of other nodes that reference it plus the number of user formulae that reference it. This count is maintained incrementally. References from the unique-table or from computed-table entries are not included. A node with a reference count of 0 is called *dead*.

When a user formula is freed, the reference count of the corresponding node $F = (v, G, H)$ is decremented. If the new reference count for F is 0, then the reference counts of the nodes G and H are recursively decremented. F cannot be freed at this time because it may be referenced by computed-table entries. The memory overhead of maintaining pointers from F to the elements of the computed-table which point to it is excessive, so we choose to use garbage collection instead.

```
ite_constant(F, G, H){
    if (trivial case) {
        return result (0, 1 or non_constant);
    } else if (computed-table has entry for (F,G,H)) {
        return non_constant;
    } else {
        let v be the top variable of F, G, H;
        T = ite_constant(F_v, G_v, H_v);
        if (T ≠ 1 and T ≠ 0) return non_constant;
        E = ite_constant(F_v̄, G_v̄, H_v̄);
        if (E ≠ T) return non_constant;
        insert_computed_table({F, G, H}, T);
        return T;
    }
}
```

Figure 4: The *ite_constant* algorithm.

If a lookup in the computed-table returns a dead node, a *reclaim* operation is performed. First, the reference count of the node is incremented. If it was dead (which is always true at the top level), the *then* and *else* nodes are recursively reclaimed. This brings the dead node and any dead nodes under it back into the ROBDD.

Reference counting is used to keep an accurate count of the the number of dead nodes in the DAG. The number of dead nodes influences the memory management strategy. If the load factor in the unique-table exceeds 4 after an insertion the following check is made. If 10% of all nodes are dead, then garbage collection is performed. Garbage collection consists of deleting all computed-table entries that reference dead nodes and then freeing all dead nodes. If there are not enough dead nodes, then the unique-table and computed-table are both increased in size and all of the elements are re-hashed into the larger tables. Garbage collection is done at very low cost during this resize.

Memory overflow is handled using a similar technique. When the memory usage for the unique-table and computed-table exceeds a user-specified memory limit, and 10% of the nodes are dead, a garbage collection is done to free enough memory to continue. Otherwise the package *gives up*, automatically freeing intermediates ($T = ite(F_v, G_v, H_v)$ in Figure 2) on its way out of the *ite* and returning the zero pointer to the user. It is up to the user to give up or free unneeded formulae and continue.

The reference count and variable index share a single word in a node and only 8 bits are allocated for the reference count, so *saturating* increment and decrement operations are used. If the count overflows, the node will never be freed, guaranteeing correct operation of the package. In practice few nodes (other than the constant node) hit a reference count as large as 255.

These techniques provide effective memory management at a very low cost. For the set of 12 examples presented in Section 6, on average only 3% of the run-time is spent in garbage collection and resizing and 7% in free and reclaim operations.

5.5 Management of the Computed Table

Another modification we use is to implement the computed table as a *hash-based cache*. We call this the *caching computed-table*.

A caching computed-table takes advantage of a high locality of reference. A computed-table entry is created for every non-trivial recursive call to *ite*, but newer entries overwrite the older ones if the hashing function is satisfactory. This decreases the frequency of garbage collection which improves the run-time of the package. The caching computed-table requires less memory because it is not

necessary to link the elements together in a collision chain. Also, by controlling the ratio of the number of unique-table entries to the number of caching computed-table entries, it is easy to control the memory and run-time trade-off for the memory function.

The use of a caching computed-table introduces the possibility of recalculating previous results and invalidates our previous time-complexity analysis. In fact, the worst-case complexity of the *ite* operation is now exponential in the unlikely event that all keys hash to the same value. Experimentally, however, we find that the gain both in terms of average space and time is enough to warrant the risk of using the cache.

Note that the computed-table remains valid even across top-level calls to *ite*. Therefore, we initialize the computed-table only once when the ROBDD is created rather than at each top level call to *ite*.

For the 12 circuits presented in Section 6, the caching computed-table requires 16% more recursive calls to *ite* than the hashing computed-table, but the run-time is increased by only 6%. The caching computed-table causes computation to be repeated for some nodes because of a miss in the computed-table; however, the hashing computed-table requires more processing for the operations of lookup and garbage collection. Because of the lower memory overhead, we feel the caching computed-table is the best solution for this application.

Another experiment was performed to measure the effect of computed-table lookups which come from old data which happens to be left in the table. Approximately 17% more *ite* recursive steps are required if the computed-table is purged at each top-level call to *ite* and the run-time is increased by 22%. Note that not purging the cache also avoids the linear time operation of allocating and deallocating the cache.

5.6 Merging the Unique-table and DAG

Instead of using separate data structures for the unique-table and the ROBDD DAG, we combine the unique-table and the ROBDD into a single data structure. Each node now has an additional field which is the collision chain link in the unique-table. Hence, seemingly random elements of the ROBDD are linked together in the collision chain which is required for fast lookup of a node in the ROBDD. This leads to an improvement in memory usage as well as a small decrease in the overhead for memory allocation.

To analyze the memory usage, we assume that the caching computed-table and the merged-DAG unique-table use the same number of bins. We find that a load factor of 4 for the unique-table provides a reasonable trade-off between the time to find an entry in the hash table and the memory overhead per entry. When the load factor is 4, the memory usage is only 22 bytes per node.[1] This comes from 4 words for each merged-DAG unique-table entry and 1/4 of a 4 word entry in the computed-table (on average per node). The hash-table bin overhead adds an extra 2 bytes per entry (on average). Note that this is the amortized cost per node for the entire ROBDD package and not just the ROBDD itself. This memory usage has been verified experimentally.

6 Experimental Results

6.1 Forming the BDD for Digital Circuits

The first data we present is the run-time and memory requirements for converting the combinational portion of several large digital circuits into the ROBDD representation. These circuits are standard benchmarks available in the public domain [18] and they are known to have large sum-of-products representations.

Circuit	#in	#out	Fixed Order Size (nodes)	Fixed Order CPU (sec.)	Variable Order Size (nodes)	Variable Order CPU (sec.)
C432	36	7	30,200	79.5	–	–
C499	41	32	49,786	85.0	–	–
C880	60	26	7,655	10.5	–	–
C1355	41	32	39,858	80.4	–	–
C1908	33	25	12,463	28.9	–	–
C2670	233	140	unable		18,153	60.4
C3540	50	22	[2]208,947	704.0	–	–
C5315	178	123	32,193	51.0	–	–
C6288	32	32	unable		unable	
C7552	207	108	unable		5,895	60.5
des	256	245	7,128	29.9	–	–
rot	135	107	7,405	8.0	–	–

Table 1: ROBDD Results for large digital circuits.

First, each combinational network was converted into an unbounded-fanin NOR representation. Then the primary inputs to the combinational logic were ordered using the topological ordering heuristic given in [1]. An attempt was made to create the ROBDD for each output function using the same variable ordering for all outputs. A limit of 2 megabytes (Mb) was placed on the memory usage of the ROBDD package. A hash-based cache was used for the computed-table, and a load factor of 4 was used for the unique-table. The ratio of the number of bins in the computed-table to the number of bins in the unique-table was 1. The data is presented in Table 1.

The CPU time is given in seconds on a Sun 3/60 with 8 Mb of memory. Shown in the table are the results for both a fixed input order (where one input variable ordering is used for all outputs simultaneously) and a variable input order (where different input orderings are used for each output). Example *C3540* required 12 Mb to complete; the rest finished within the 2 Mb limit. For *C2670*, *C6288*, and *C7552*, we were unable to form the ROBDD for all outputs using a fixed input order. Example *C6288* is a 32-bit multiplier for which it has been proven that the ROBDD always requires exponential size for some output [13] even for the optimum input ordering. We were able to form the ROBDD for only the first 12 outputs of this example with a 12 Mb memory limit placed on the package.

The results presented here are more than ten times faster than the times than reported in [1, 2], even when accounting for the difference in the machines.[3] This improvement comes only from the implementation of the ROBDD package as described here; the variable orders, although probably not identical, were created using similar algorithms.

6.2 Implication-Set Results

The F-sets of a digital circuit are defined as [19]:

$$F_{ij}(s) = \{x | s = i \Rightarrow x = j\}$$

[1] We assume a 32-bit machine.

[2] Required 12 mB to complete.

[3] For example, *C432* required between 529 and 1,423 seconds on a Sun 3/260 in [2] and a between 530 and 1232 seconds on a DEC VAX 8650 in [1] (forming each BDD twice) depending on the program options used. If we assume a ratio of 4:3 for Sun 3/260 to Sun 3/60, and a ratio of 2:1 for DEC VAX 8650 to Sun 3/60, the run-time ratios are in the range 8–22 and 6–14 respectively. Other examples show larger differences.

Example	C10 size	F10 size	CPU (sec.)
C432	2,470	2,534	168
C499	2,753	2,753	816
C880	2,900	2,916	81
C1355	20,530	20,530	1,425
C1908	15,178	15,332	477
C5315	41,742	43,130	1,020
rot	7,943	10,175	177

Table 2: F-set computation results.

That is, for example, the F_{10}-set of a signal s is the set of other signals x in the network such that $s = 1$ implies $x = 0$. Trevillyan and Berman show how to use the F-sets and subsets of the F-sets (called C-sets) in a logic optimization algorithm.

A trivial algorithm to compute the F-sets is to check if $s \leq \overline{x}$ for all pairs of signals s and x in the circuit. This check is done using the *ite_constant* algorithm described earlier. Table 2 shows the results of forming the F-set for each example from the previous section. Shown in the table is the total size of the C_{10}-set and F_{10}-set for each example. Note that we are unable to form the F-sets for *C2670*, *C3540*, *C6288*, and *C7552* because we are unable to simultaneously create the ROBDD for every node in these networks.

The surprising conclusion is that the C-set is a substantial percentage of the F-set for most circuits. Also, we were able to compute the F-sets for many of these large circuits.

7 Conclusions

This paper has presented an efficient implementation of a package for manipulating Boolean functions represented as ROBDDś. This implementation is substantially faster than straightforward implementations of the original algorithms reported in [13]. The use of the caching computed-table, and other improvements such as merging the unique-table and ROBDD, lead to a version of the ROBDD package which is faster and much more memory efficient than versions implemented at CMU, for which experimental results have not been reported.

The amortized cost for all memory used by the package is approximately 22 bytes per node, which is substantially below that reported for similar packages. The run-time on a set of standard circuits shows the superiority of this implementation compared to similar packages. We have presented data for computing the F-sets in a circuit, and are the first to show that the C-set approximation for the F-sets appears to be quite good.

References

[1] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," in *Proc. Int. Conf. CAD (ICCAD-88)*, Nov. 1988, pp. 6–9.

[2] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams," in *Proc. Int. Conf. CAD (ICCAD-88)*, Nov. 1988, pp. 2–5.

[3] O. Coudert, C. Berthet, and J. Madre, "Verification of Sequential Machines using Boolean Function Vectors," in *IMEC-IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, 1989.

[4] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney, "The Transduction Method," IEEE *Trans. Comp.*, vol. 38, pp. 1404–1424, Oct. 1989.

[5] M. A. Breuer and A. D. Friedman, *Diagnosis and Reliable Design of Digitial Systems*. Computer Science Press, 1976.

[6] P. McGeer and R. Brayton, "Efficient Algorithms for Computing the Longest Viable Path in a Combinational Network," in *Proc. 26th Design Automation Conference*, July 1989, pp. 561–567.

[7] K. Cho, *Test Pattern Generation for Combinational and Sequential MOS Circuits by Symbolic Fault Simulation*. PhD thesis, Carnegie Mellon University, 1988.

[8] M. R. Garey and D. S. Johnson, *Computers and Intractability*. W. H. Freeman and Company, 1979.

[9] R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[10] D. Bostick, G. D. Hachtel, R. Jacoby, M. R. Lightner, P. Moceyunas, C. R. Morrison, and D. Ravenscroft, "The Boulder Optimal Logic Design System," in *Proc. Int. Conf. CAD (ICCAD-87)*, Nov. 1987, pp. 62–65.

[11] C. Y. Lee, "Representation of Switching Circuits by Binary-Decision Programs," *Bell System Technical Journal*, vol. 38, pp. 985–999, July 1959.

[12] Fortune, J. Hopcroft, and E. Schmidt, *The Complexity of Equivalence and Containment for Free Single Variable Program Schemes*, pp. 227–240. Springer-Verlag, 1978.

[13] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," IEEE *Trans. Comp.*, vol. C-35, pp. 677–691, Aug. 1986.

[14] D. S. Reeves and M. J. Irwin, "Fast Methods for Switch-Level Verification of MOS Circuits," IEEE *Trans. Elec. Comp.*, vol. CAD-6, pp. 766–779, Sep. 1987.

[15] K. Karplus, "Representing Boolean Functions with If-Then-Else DAGs," Computer Engineering UCSC-CRL-88-28, UC Santa Cruz, Dec. 1988.

[16] S. B. Akers, "Functional Testing with Binary Decision Diagrams," in *Eighth Annual Conference on Fault-Tolerant Computing*, 1978, pp. 75–82.

[17] J.-C. Madre and J.-P. Billon, "Proving Circuit Correctness using Formal Comparison between Expected and Extracted Behavior," in *Proc. 25th Design Automation Conference*, June 1988, pp. 205–210.

[18] R. Lisanke, "Logic Synthesis Benchmarks," in *Proc. Int. Workshop on Logic Synthesis (IWLS-89)*, May 1989.

[19] L. Berman and L. Trevillyan, *A Global Approach to Circuit Size Reduction*, pp. 203–214. MIT Press, 1988.