Boolean Satisfiability and Equivalence Checking Using General Binary Decision Diagrams

Pranav Ashar, Abhijit Ghosh
Department of EECS
U. C. Berkeley

Srinivas Devadas
Department of EECS
MIT, Cambridge

Abstract

We show how general Binary Decision Diagrams (BDDs), *i.e.*, BDDs where input variables are allowed to appear multiple times along any path in the BDD, can be used to check for Boolean satisfiability. Our satisfiability checking strategy is based on an *input smoothing* operation on general BDDs. We develop various input smoothing strategies for general BDDs. In order to verify the equivalence of two functions f_1 and f_2, we check $f_1 \oplus f_2$ for satisfiability.

Using general BDDs we were able to verify different implementations of a 16×16 multiplier, a modified Achilles-heel function and a complex add-shift function. It was not possible to construct OBDDs for any of the three functions.

1 Introduction

Reduced, ordered Binary Decision Diagrams (OBDDs) [4] have gained widespread use in the areas of combinational and sequential logic verification (*e.g.* [11] [7]) due to their canonicality and easy manipulability.

Since OBDDs are a canonical form, verifying the equivalence of two combinational logic functions involves selecting a common input ordering for the two circuits, constructing OBDDs for each of the circuits, and checking to see if the two OBDDs are isomorphic. Satisfiability checking simply corresponds to comparing the OBDD of a given circuit, under any input ordering to the constant 0 OBDD.

Finding a good input ordering that produces OBDDs of manageable size is a difficult problem and has received considerable attention (*e.g.* [11, 10, 1]). However, there are classes of combinational circuits, notably multipliers, for which OBDDs under any possible input ordering have a provably exponential size [4]. Circuits other than multipliers have also been encountered where finding an input ordering to obtain an OBDD of manageable size has not been possible so far. In fact, it is easy to construct simple examples where OBDD sizes grow exponentially with the number of inputs to the example (*cf.* Section 5).

There has been some work in the generalization of BDDs to verify larger classes of circuits. The method of Friedman [9] uses pBDDs, where variables can appear more than once along a path from the root to a leaf. There are no guarantees about the size of the pBDDs for a multiplier and equivalence checking is NP-complete. Simonis [12] uses constraint logic programming to verify classes of multipliers but this method cannot be easily generalized to circuits containing multipliers. Burch, in [6], showed that the replication of inputs to a $n \times n$ multiplier to obtain a circuit with $2n^2$ inputs would result in a OBDD of $O(n^3)$ size under a high-to-low input ordering. However, correspondence between the replicated inputs of the logic-level implementations of the circuits that are being checked for equivalence is required. This is not generally possible in a synthesis scenario, where the logic may be restructured dramatically.

In this paper, we show how general BDDs, *i.e.*, BDDs where input variables are allowed to appear multiple times along any path in the BDD, can be used to check for Boolean satisfiability. We construct general BDDs in the following manner. The inputs in a given circuit η are first replicated to obtain a new circuit η'. After choosing an appropriate ordering for the inputs to η', we use OBDD construction algorithms to obtain a general BDD for η.

Our satisfiability checking strategy is based on an *input smoothing* operation. After all the inputs have been smoothed away, if the function reduces to a constant 1, then it means that the original function was satisfiable. In order to verify the equivalence of two functions, f_1 and f_2, we check $f_1 \oplus f_2$ for satisfiability. Various input smoothing strategies are presented in this paper.

General BDDs are not a canonical form and are a much more powerful representation than OBDDs. For example, it has been shown that a general BDD of $O(n^3)$ size can be constructed for a $n \times n$ multiplier [6]. Using general BDDs we were able to verify different implementations of a 16×16 multiplier, a modified Achilles heel function and a complex add-shift function. It was not possible to construct OBDDs for any of the three functions. The verification was carried out *without requiring any additional information*, other than the given logic-level descriptions. The use of general BDDs, as opposed to OBDDs, also dramatically reduces the memory requirements to verify other classes of circuits.

For basic logic synthesis and BDD terminology, the reader is referred to [3, 4]. Definitions of BDD operations can be found in [13]. In Section 2, we outline the overall strategy toward satisfiability checking and combinational logic verification using general BDDs. Input smoothing algorithms for general BDDs are presented in Section 3. Input replication and ordering algorithms, and some implementation details are discussed in Section 4. Experimental results are presented in Section 5.

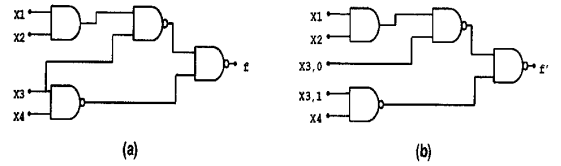

Figure 1: **Example of Input Replication**

2 Satisfiability and Equivalence Checking

2.1 Satisfiability Checking

Given a logic circuit f to check for satisfiability, assume that we cannot construct a manageable OBDD for f due to memory or CPU time restrictions. In such a case we replicate the inputs to f to obtain a different circuit f', keeping track of what inputs to f' are derived from the same input to f. Replicating the inputs to an example circuit is shown in Figure 1 where the input x_3 in Figure 1(a) has been replicated to $x_{3,0}$ and $x_{3,1}$ in Figure 1(b).

If we replicate enough inputs and find a good ordering of replicated inputs, we can use OBDD construction algorithms [4] to construct an OBDD for f', which can be viewed as a general BDD, G, for the circuit f (if we coalesce the replicated inputs). Note that the general BDD is *not* a canonical form for the function f. For example, function f may not be satisfiable, but G may be large.

We now sequentially smooth away the inputs to f using its general BDD (the OBDD for f') as a base representation. We pick an input to f, say x_i. Assume that x_i has been replicated n times in f', and say the replicated inputs correspond to $x_{i,0}$, ..., $x_{i,n-1}$. We have to smooth away the $x_{i,0}$, ..., $x_{i,n-1}$ inputs to f' making sure that they have the same values (0 or 1). The smoothing of f by x_i (whose replicated instances are $x_{i,0}$, ... $x_{i,n-1}$), under the f' representation, is defined as:

$$S_{x_i} f' = f'_{x_{i,0} \cdot x_{i,1} \cdots x_{i,n-1}} + f'_{\overline{x_{i,0}} \cdot \overline{x_{i,1}} \cdots \overline{x_{i,n-1}}}$$

Essentially, we are cofactoring f' with respect to the cube corresponding to all the $x_{i,j}$'s set to 1, and with the cube corresponding to all the $x_{i,j}$'s set to 0 and OR'ing the results.

As in [13] we can define:

$$S_x f' = S_{x_0} \ldots S_{x_{k-1}} f'$$

to sequentially smooth away the k sets of inputs to f', corresponding to the distinct inputs to f.

Our smoothing strategy can use highly efficient OBDD manipulation algorithms. Input smoothing in OBDDs can in general be performed efficiently [13]. Input smoothing under general BDD representations is more complicated, and we discuss several strategies in Section 3.

2.2 Equivalence Checking

In order to check for the equivalence of two logic functions f_1 and f_2, we construct $f_1 \oplus f_2$ as shown in Figure 2(a). We treat this composite function as our f function above, and replicate inputs as needed to construct a general BDD for f. If necessary, we can break up the inputs to f_1 and f_2 so they have disjoint support in f' (Figure 2(b)). If OBDDs are constructible for f_1 and f_2, there is no need to use our technique to verify the equivalence of f_1 and f_2. Our technique, however, will allow the further replication of the inputs to f_1' and f_2' so a manageable-sized general BDD can be constructed for f', even in the case where OBDDs cannot be constructed for f_1 or f_2.

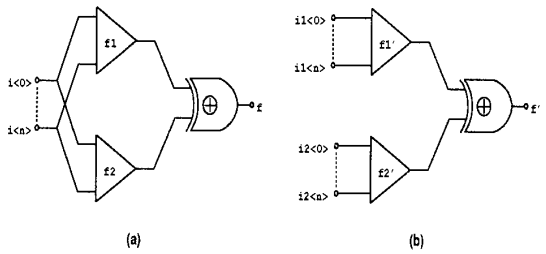

Figure 2: **Equivalence Checking**

3 Input Smoothing

3.1 A Branching Strategy for Smoothing Replicated Inputs

While input smoothing in an OBDD is not guaranteed to reduce the size of the OBDD, efficient algorithms exist [13] that can smooth away a set of inputs in a OBDD by making a single pass over the OBDD. Input smoothing in general BDDs is more complicated, and can increase the size of the general BDD.

In our experiments we found that, given a set of replicated inputs to be smoothed, the order in which the inputs are smoothed can have a significant effect on the resulting general BDD. Further, the size of a general BDD that is obtained after smoothing away even a single set of replicated inputs can be substantially larger than the original BDD. In general, a massive replication of inputs in a general BDD makes smoothing more difficult. This is intuitive, since the smoothed BDD will be substantially different from the original general BDD. We therefore devised the general branching algorithm shown below, that operates in a depth-first manner.

In the routine **satisfiable**(), G is the general BDD, and X is the set of inputs to be smoothed away in order to check for satisfiability. Each $x_i \in X$ consists of $x_{i,0}$, ..., x_{i,n_i} replicated inputs.

```
satisfiable( G, X ):
{
    if ( X = φ ) {
        if ( G ≡ 0 ) return(FALSE) ;
        else return(TRUE) ;
    }
    x_p = select-input ( G, X ) ;
    Compute G_{x_p} = G_{x_{p,0} · x_{p,1} ··· x_{p,n_p}} ;
    Compute G_{¬x_p} = G_{¬x_{p,0} · ¬x_{p,1} ··· ¬x_{p,n_p}} ;
    if ( G' = bdd-or ( G_{x_p}, G_{¬x_p}, param1 ) ) {
        return ( satisfiable( G', X − x_p ) ) ;
```

```
    }
    else {
        if ( satisfiable( G_{x_p}, X - x_p ) ) return(TRUE) ;
        else if ( satisfiable( G_{\overline{x_p}}, X - x_p ) ) return(TRUE) ;
        else return(FALSE) ;
    }
}
```

The parameter `param1` controls the amount of branching that can take place versus the size of the smoothed BDDs, and can be set based on the amount of memory available. During the **bdd-or**() operation the size of the resulting BDD is monitored, and if its size exceeds `param1`, the **bdd-or**() operation is terminated. If the **bdd-or**() has to be terminated, branching on the cofactors takes place. This ability to branch gives our approach the capacity to trade off CPU time for memory usage.

The routine **select-input**() selects a set of replicated inputs in the general BDD to be smoothed. Currently, we use a simple heuristic that has to do with the fraction of the general BDD that is affected by the smoothing. Smoothing with respect to x_i only affects the portion of the general BDD below $x_{i,min}$, where $x_{i,min}$ corresponds to the replicated instance of x_i that has the lowest index (is closest to the root of the general BDD) among all the replicated instances of x_i. We select the input x_p such that $x_{p,min}$ has the maximum index among all $x_{i,min}$.

3.2 Smoothing Inputs Using Circuit Transformations

3.2.1 Smoothing Unreplicated Inputs

In the case where unreplicated inputs exist in the general BDD, it is possible to use the algorithm of [13] to simultaneously smooth away these inputs. However, the strategy may require too much CPU time or memory, especially if the other inputs have been extensively replicated (*cf.* discussion at the end of this section).

A more efficient circuit-transformation-based strategy is as follows. Given a general BDD, G, we derive a multiplexor-based circuit η corresponding to G, by replacing all nodes in G by 2-input multiplexors whose control input is the decision variable corresponding to the node. In η, we replace all the multiplexors corresponding to the unreplicated inputs by OR [1] gates to obtain η'. Now, we can choose an ordering for the remaining inputs to η', (and coalesce replicated inputs if necessary) to construct a new general BDD for η'.

In a general BDD, we cannot simply replace each node corresponding to an instance of a replicated input by an OR gate. Doing so may make paths in the general BDD corresponding to conflicting values for different instances of a replicated variable sensitizable in the derived circuit, thus destroying functionality. However, a replicated input which does not appear more than once along any path in the general BDD can be smoothed away by replacing all its nodes by OR gates in the derived circuit.

This technique can perform significantly better than a straightforward smoothing algorithm because of the degree of freedom in choosing a different replication and ordering for the general BDD of η'. For example, assume that the variable x (with f being the function associated with its BDD node) is being smoothed. Straightforward smoothing within the BDD forces the BDD for $f_x + f_{\overline{x}}$ to be constructed under a global variable-ordering that is suitable for f_x and $f_{\overline{x}}$ but not necessarily for $f_x + f_{\overline{x}}$, leading to a much larger BDD after smoothing. While this problem is usually not encountered in the case where no replicated inputs are present, it assumes great significance when they are.

We can use the above technique iteratively. When constructing the general BDD for η', we can leave some inputs unreplicated, and then obtain a circuit η'' with these inputs smoothed away using the OR-gate transformation. If a large number of inputs are smoothed away, it is possible that a OBDD will be constructible for η' or η'', simply by coalescing all the replicated inputs.

3.2.2 Smoothing Replicated Inputs

Smoothing replicated inputs in a general BDD via a circuit transformational method is more complicated. In the case of replicated inputs appearing several times along a path we can use the following strategy. Assume we have a general BDD G, and the input x_i has been replicated k times, the replicated instances being $x_{i,0}, x_{i,1}, \ldots, x_{i,k-1}$. We now wish to transform G into G' so that all paths in G' that correspond to conflicting values for the different instances of the replicated variable x_i have zero-terminal vertices. Other than this modification, G' has the same functionality as G. We first obtain G_1 from G by making all the paths in G that correspond to a *zero* value for any of the replicated instances of x_i have zero-terminal vertices. Similarly, we obtain G_2 from G by making all the paths in G that correspond to a *one* value for any of the replicated instances of x_i have zero-terminal vertices. The OR of $G1$ and $G2$ satisfies the property required of G'. We can now obtain a circuit from G' by replacing all the nodes in the G' corresponding to the replicated instances of the variable x_i by OR gates and all the other nodes by 2-input multiplexors. The replicated primary input x_i has effectively been smoothed away in this derived circuit. We can now reconstruct a (possibly general) BDD from the derived circuit after picking an ordering for the remaining variables.

Note that the above strategy requires path enumeration in the BDD and can rapidly become inefficient if the replicated instances of an input are widely separated in the global variable ordering.

3.3 Hybrid Strategies

At any given point while using the circuit transformational approach, the branching strategy can be invoked to check the current general BDD for satisfiability. Similarly, while using the branching method, after cofactoring some inputs, the circuit transformational method can be applied to smooth away the remaining inputs.

[1] The use of complemented nodes [2] in a BDD makes this transformation slightly more complicated. A process of pushing the inversions all the way back to the terminal vertices while deriving the circuit is necessary, and the derived circuit can be twice as large as the original BDD.

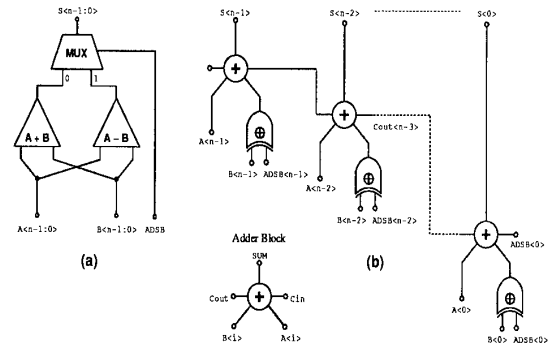

Figure 3: **Two Versions of the Adder-Subtractor**

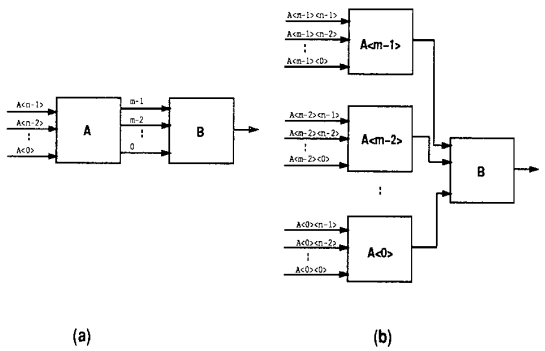

Figure 4: **A Circuit with a Block Structure and Its Replication**

4 Input Replication & Ordering

4.1 Static Input Replication and Ordering

4.1.1 Replication By Fanout Splitting

In many cases, OBDDs are completely different structurally and much larger than the circuit itself. In such cases, the use of appropriately constructed general BDDs can lead to graphs that are significantly smaller than the OBDDs and are structurally closer to the circuit.

The goal of attempting to make the BDD-based circuit structurally as close to the given circuit as possible guides the static replication and ordering of the inputs, under the fanout splitting method. This is best illustrated with an example.

Consider the **adder-subtractor** circuit shown in Figure 3. Figure 3(a) shows an area-inefficient implementation in which two distinct blocks are used for addition and subtraction. The desired output is then selected by the output multiplexor based on the value of the **ADSB** line. This is exactly the structure of the OBDD that would be obtained if the **ADSB** input were at the root of the OBDD. The area-optimum implementation of the **adder-subtractor** on the other hand is shown in Figure 3(b). This circuit is much smaller than and structurally different from the circuit in Figure 3(a). A general BDD whose structure is close to that of Figure 3(b) can be derived in the following manner:

1. **ADSB** is replicated n times, where n is the bit-width. The replicated **ADSB** lines are labeled so that **ADSB**$< i >$ corresponds to the fanout path from **ADSB** that is XOR'ed with **B**$< i >$ in Figure 3(b).

2. An OBDD is constructed with the ordering **B**$< n-1 >$, **ADSB**$< n-1 >$, **A**$< n-1 >$, **B**$< n-2 >$, **ADSB**$< n-2 >$, **A**$< n-2 >$, $\cdots$, **B**$< 0 >$, **ADSB**$< 0 >$, **A**$< 0 >$. The general BDD with the replicated inputs in the OBDD coalesced will have the same structure as the circuit of Figure 3(b).

The example of the **adder-subtractor** circuit leads us to a general procedure for replicating and ordering inputs. To begin with, a possibly area-optimized version of the circuit is obtained to guide the replication. In the optimized circuit, an input with a large depth (where depth is defined as the number of gates between the input and the primary output) and which fans out multiple times so that each fanout path is used at different depths in the circuit is a good candidate for replication (The **ADSB** input in the **adder-subtractor** satisfies these requirements). Such an input is replicated as many times as it fans out. Once all inputs deemed to be good candidates are replicated, either the ordering strategy of [11] or some other strategy dependent on the circuit structure can be used to construct the OBDD. When a large number of inputs have been replicated, the circuit tends to assume the form described in Lemma 1 of [11] wherein the output f can be expressed as $f = g(f_1, g_1(f_2, g_2(...g_{k-1}(f_k, f_{k+1})...)))$ and each f_i has a support that is disjoint from the others. In such a case, the optimum ordering for f is the concatenation of the optimum orderings for the f_i's. The orderings for the f_i's can either be concatenated in the order from $k+1$ to 1 or from 1 to $k+1$ depending on the circuit.

Burch in [6] used the fanout splitting method to obtain a general BDD of $O(n^3)$ size for a $n \times n$ parallel multiplier, under a particular ordering that required functional as well as topological information. Unfortunately, we have found that changing the ordering slightly can result in a substantially larger general BDD. Given 16×16 multiplier circuits, that were run through intensive logic optimization, we were unable to create general BDDs for these circuits using static fanout splitting and input ordering.

4.1.2 Replicating Logic Subfunctions

A strategy that works well for multiplier circuits and circuits which have a *block* structure is to replicate entire logic blocks in the circuit along with their inputs.

Consider the circuit structure of Figure 4(a). All the primary inputs to the circuit feed the first block A. The outputs of A feed block B. It is possible that OBDD representations are constructible for A and B in isolation, but not for $A \rightarrow B$. The fanout splitting method typically fails to construct a general BDD for $A \rightarrow B$, because it does not simplify the interactions between the outputs of A within block B. This will especially

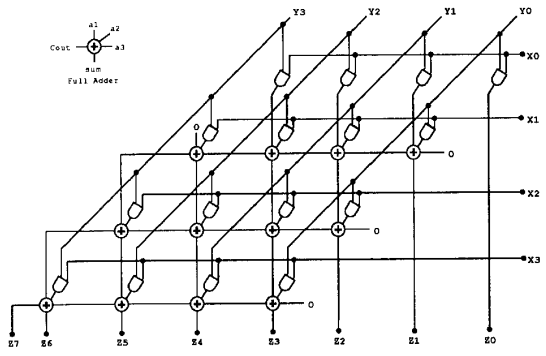

Figure 5: **A** 4×4 **Multiplier**

be the case when each input to A fans out to few gates.

It is possible to create a general BDD for the circuit above using the replication shown in Figure 4(b). Each output of A has been implemented as a separate function, with disjoint support from the other outputs of A. We can choose good orderings for each set of inputs to $A < 0 >$ through $A < m-1 >$, namely $A < 0 >< n-1:0 >$ through $A < m-1 >< n-1:0 >$. In order to obtain a small-sized general BDD for the replicated circuit of Figure 4(b), we have to keep each set of inputs $A < i >< n-1:0 >$ contiguous in the global ordering for the replicated circuit.

A 4×4 multiplier is shown in Figure 5. One way of creating a general BDD for a $n \times n$ multiplier is to partition the circuit into two circuits A and B, where A corresponds to the AND gate row plus the first $\frac{n}{2} - 1$ adder rows, and B to the remaining $\frac{n}{2}$ adder rows. It is possible to create a reasonable-sized general BDD by replicating each output of A to be a separate function with disjoint support from the other outputs (as illustrated in Figure 4), if $n \leq 16$. For larger multipliers it is necessary to partition the circuit into 3 or more subcircuits.

4.2 Dynamic Input Replication and Ordering

We briefly describe a dynamic input replication and ordering strategy that can perform both fanout splitting and logic subfunction replication in a uniform way.

We begin with a circuit where all the inputs are unreplicated. Beginning from the primary inputs of the circuit, we traverse the circuit in breadth-first manner constructing BDDs for each intermediate line or gate. At any given point of time, we have two BDDs for logic subfunctions f_1 and f_2, and we are constructing a BDD representation for $f_1 < op > f_2$ ($< op >$ could be an arbitrary Boolean operator over two variables). We monitor the size of $f_1 < op > f_2$ during the *apply* [4] operation. If $||f_1 < op > f_2|| \geq$ `param2` $\times$ ($||f_1|| + ||f_2||$), we replicate the inputs to f_1 and f_2 such that they are disjoint, and order the replicated inputs such that the inputs to f_2 follow those of f_1, or vice-versa. Under this replication and ordering, $||f_1' < op > f_2'|| = ||f_1'|| + ||f_2'||$.

The parameter `param2` trades off the amount of replication against the size of the resulting general BDD. If f_2 happens to be a primary input that is also an input to f_1, then fanout splitting could occur during this dynamic replication. In general, f_1 and f_2 could be arbitrary logic subfunctions, whose support is made disjoint by replication.

EXAMPLE	#I	#O	OP#	GBDD	
				Time[1]	Size
ach32	64	1	1	61s	7073
add-shift32	64	32	31	123s	10721
			15	80s	5089
ach-parity32	64	1	1	91s	7853
mult16	32	64	31	216m	15246
			15	51m	3894

[1] In minutes on a IBM 6000 Model 320

Table 1: **Equivalence Checking Applied to Combinational Circuits Not Amenable to OBDD Representation**

5 Experimental Results

The viability of our approach is illustrated by the results shown in Table 1 for several examples. The significance of the results lies in that a canonical representation in the form of an OBDD could not be obtained for any of the functions and that none of the circuits are collapsible into two levels of logic. Therefore, our approach is a viable approach for verifying these circuits, without requiring additional information other than the given logic-level descriptions.

In Table 1, **#I** and **#O** correspond to the total number of inputs and outputs in the circuits, respectively. **OP#** corresponds to the number of the output for the equivalence checking of which data is provided in the table. The CPU time (**Time**) involved in the equivalence checking are provided in the table for our approach (**GBDD**). The CPU time includes the time for creating the general BDD as well as for smoothing the variables. The size (in terms of the number of nodes in the graph) of the general BDD corresponding to the XOR of the two functions being verified is provided under the column **Size.**

ach32 in Table 1 is a single-output modified 32-bit Achilles-heel function. The following equation describes the circuit. $f = \overline{mux_0} \cdot \overline{mux_1} \cdot \overline{mux_2} \cdot \overline{mux_3} \cdot \overline{mux_4} \cdot f_0 + mux_0 \cdot \overline{mux_1} \cdot \overline{mux_2} \cdot \overline{mux_3} \cdot \overline{mux_4} \cdot f_1 + \ldots + mux_0 \cdot mux_1 \cdot mux_2 \cdot mux_3 \cdot mux_4 \cdot f_{31}$ where the f_i are defined as follows:

$$f_0 = x_0 \cdot y_0 + x_1 \cdot y_1 \cdots x_{31} \cdot y_{31}$$

$$f_i,\ 1 < i < 32 = \prod_{j=0}^{31} (x_j + y_{(j+i) \bmod 31})$$

It can be shown using the theory developed in [5] that any OBDD (under any possible ordering) for a n-bit modified Achilles heel function requires $O(2^{\frac{n}{2}})$ vertices [8]. This is because each f_i requires a different ordering for its inputs for a reasonable-sized OBDD representation. The replication of the primary inputs for this

function, on the other hand, makes the support for the various f_i disjoint and therefore allows the variable orderings for the f_i to be effectively independent. While equivalence checking using our approach can be done in about a minute, it cannot be done by any other currently available method. Note that f has been chosen so that neither f nor $\overline{f}$ is collapsible to a two-level sum-of-products representation. Both the circuit transformational and branching approach to smoothing successfully completed in about a minute, after a static replication and ordering of inputs using fanout splitting was applied to the circuits.

The second circuit in Table 1 is **add-shift32**. This circuit is similar in nature to **ach32**. It performs one of 32 functions based on the *mux* control signals. The output of the circuit is equal to the arithmetic addition of the input A and the the input B rotated by an amount given by the *mux* signals. For reasons similar to those for **ach32**, an OBDD cannot be built for this circuit. Also, this circuit is not collapsible to two levels of logic since it involves arithmetic operations. Our approach, on the other hand, is able to perform equivalence checking for the most complex output in 2 minutes. The remaining outputs all required fewer times. Again, after static replication using fanout splitting was applied, both the circuit transformational and branching approaches were successful (and comparable in CPU time usage).

The third circuit **ach-parity32** corresponds to the function $f_0 \oplus f_1 \oplus \cdots \oplus f_{31}$, with the f_i's the same as those in **ach32**.

We have been able to verify different implementations of a 16×16 multiplier, using static replication with logic subfunction replication (with either the circuit transformational or the branching method), as described in Section 4.1.2. We verified the **C6288** ISCAS-85 benchmark against a hand-crafted multiplier whose basic cell was implemented differently from **C6288**. We also verified an optimized version of **C6288** against the original – the run-time differences were negligible from those in Table 1.

We are currently implementing the dynamic replication and ordering strategy to provide a unified method of general BDD-based logic verification.

6 Conclusions

We have shown that a representation of logic functions, namely general Binary Decision Diagrams (BDDs), can be used in conjunction with efficient ordered Binary Decision Diagram (OBDD) manipulation algorithms, to check the satisfiability of, and verify combinational logic circuits that were not verifiable using previous techniques. In particular, we were able to verify large and complex arithmetic functions, for which sum-of-products or OBDD representations could not be constructed.

Future work will address improving the efficiency of the input smoothing operation on general BDDs, and the replication/ordering of inputs.

7 Acknowledgments

Discussions with Richard Newton on logic verification are acknowledged. This work was supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825, and in part by a grant from Analog Devices, Inc. An equipment grant from IBM corporation is gratefully acknowledged.

References

[1] C. L. Berman. Ordered Binary Decision Diagrams and Circuit Structure. In *Proceedings of the Int'l Conference on Computer Design: VLSI in Computers*, pages 392–395, October 1989.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proc. of 27^{th} Design Automation Conference*, pages 40–45, June 1990.

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

[4] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[5] R. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. In *IEEE Transactions on Computers*, pages 205–213, February 1991.

[6] J. Burch. Using BDDs to Verify Multipliers. In *Proceedings of 1991 International Workshop on Formal Methods in VLSI Design*, January 1991.

[7] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. In *IMEC-IFIP Int'l Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, November 1989.

[8] S. Devadas. Comparing Two-Level and Ordered Binary Decision Diagram Representations of Logic Functions. In *MIT Technical Report (available from the author)*, June 1991.

[9] S. Friedman. Data Structures for Formal Verification of Circuit Designs. In *Ph.D thesis, Department of Computer Science, Princeton University*, January 1990. Technical Report CS-TR-236-90.

[10] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 2–5, November 1988.

[11] S. Malik, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 6–9, November 1988.

[12] H. Simonis. Formal Verification of Multipliers. In *Proceedings of 1991 International Workshop on Formal Methods in VLSI Design*, January 1991.

[13] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines Using BDD's. In *Proc. of Int'l Conference on Computer-Aided Design*, pages 130–133, November 1990.