OVERCOMING MEMORY CONSTRAINTS IN ROBDD CONSTRUCTION BY FUNCTIONAL DECOMPOSITION AND PARTITIONING

by

Amit Narayan, Sunil P. Khatri, Jawahar Jain, Masahiro Fujita,
Robert K. Brayton, and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M95/91

6 November 1995

OVERCOMING MEMORY CONSTRAINTS IN ROBDD CONSTRUCTION BY FUNCTIONAL DECOMPOSITION AND PARTITIONING

by

Amit Narayan, Sunil P. Khatri, Jawahar Jain, Masahiro Fujita,
Robert K. Brayton, and Alberto Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M95/91

6 November 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Overcoming Memory Constraints in ROBDD Construction by Functional Decomposition and Partitioning

Amit Narayan[1] (anarayan@ic.eecs.berkeley.edu)

Sunil P. Khatri[1] (linus@ic.eecs.berkeley.edu)

Jawahar Jain[2] (jawahar@fla.fujitsu.com)

Masahiro Fujita[2] (masahiro@fla.fujitsu.com)

Robert K. Brayton[1] (brayton@ic.eecs.berkeley.edu)

Alberto Sangiovanni-Vincentelli[1] (alberto@ic.eecs.berkeley.edu)

November 6, 1995

[1]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720

[2]Fujitsu Laboratories of America, San Jose, CA 95134

Overcoming Memory Constraints in ROBDD Construction by Functional Decomposition and Partitioning

Abstract

In this paper, we address the issue of memory explosion in ROBDD based Boolean function manipulation methods. To reduce the intermediate peak memory requirement, we select suitable decomposition points and follow it by a symbolic composition process. In cases where the final memory requirement itself is very large or where the intermediate explosion cannot be avoided by decomposition, we create ROBDDs representing orthogonal partitions of the function. Since these partitions are orthogonal they can be manipulated and verified independently. This results in a more efficient utilization of memory resources. We discuss two partitioning approaches, one in which the partitions are chosen apriori, and another in which this choice is made dynamically as the composition proceeds.

We demonstrate the utility of our schemes on the ISCAS85 benchmark circuits as well as some industrial circuits. We are able to significantly reduce the memory requirements in most cases without paying a large time penalty. Additionally, we are often able to build the ROBDDs of circuits for which conventional methods fail. We experiment with many variable ordering schemes and get impressive memory reductions in all cases.

1 Introduction

Reduced Ordered Binary Decision Diagrams (ROBDDs) [7] are frequently used as the Boolean representation of choice to solve various VLSI-CAD problems such us synthesis, digital system verification and testing. ROBDDs are canonical for a given variable ordering and can require exponential memory resources for construction. This large memory requirement places a limit on the complexity of circuits that can be processed using ROBDDs. Hence techniques that can help in a more efficient construction of ROBDDs are of practical significance.

In the traditional method of ROBDD manipulation, when a sequence of ROBDD operations has to be performed, all intermediate ROBDDs are represented in terms of the primary inputs. This restriction of building the ROBDDs of all the intermediate results in terms of primary inputs often results in a large intermediate memory requirement even when the final (canonical) representation is small. On the other hand, the final representation size is often itself very large. This is because a good variable order cannot be found or the function is inherently intractable and has an exponential ROBDD under any variable order.

In this paper, we address both these issues. When the intermediate memory requirement is large compared to the final memory requirement, it indicates that there is Boolean simplification occurring in the circuit. We try to capture this simplification by introducing suitable decomposition points. We construct a decomposed ROBDD of the output in terms of these decomposition points. The decomposition points are then composed into the output ROBDD to obtain the canonical output ROBDD over input variables. In this way, we are able to significantly reduce the peak memory utilization in a large number of cases.

In cases where the final ROBDD memory requirement is large or the intermediate requirement can not be reduced, we divide the Boolean function into orthogonal partitions. The partitions are obtained from the decomposed representation of the function in a way which ensures that the memory required at each of the intermediate stages of ROBDD construction for any partition is less than that for the monolithic representation. Since we create orthogonal partitions, each of the partitions can be processed independently. Boolean manipulation and verification of two functions can be carried out on their corresponding partitions and the results can be combined at the end. Different partitions can be assigned a hash code and probabilistic

verification can be performed by combining the hash codes for different partitions [19]. Also, we are free to choose a different variable order for each partition. In this manner, we eliminate the constraint of having a common variable order for the entire ROBDD, thus extending the class of circuits which can be processed by ROBDDs while consuming polynomial resources. We claim that this way of partitioning is more flexible than FBDDs. We have two partitioning procedures: a static procedure to determine good partitions before the composition step, as well as a dynamic one, which invokes partitioning whenever a memory explosion occurs during the composition.

We combine all these ideas in an overall procedure for an ROBDD based tautology checker. Extensive experiments are performed, both on the ISCAS85 benchmark circuits as well as some industrial circuits. Our results show an impressive reduction in the overall memory requirements without a significant time penalty. We are able to build ROBDDs for many outputs for which the conventional methods fail.

We test the procedure under a number of different variable ordering schemes including dynamic variable reordering and consistently obtain good results. This shows that the procedure is robust and explores a complementary aspect of memory explosion problem. None of the procedures use any structural information at any stage. As a result, the overall algorithm is broad in scope and can be applied to any arbitrary sequence of ROBDD operations. Specifically, it can be used to build transition relations for sequential circuits and to check for language containment in formal design verification of digital circuits. The entire procedure can be built as a shell around any existing ROBDD package. In this way, it is transparent to the end-user of the ROBDD package. This gives our approach a considerable advantage over methods which use alternate representations like FDDs, OKFDDs, IBDDs etc. to represent Boolean functions as many highly efficient ROBDD packages already exist.

In Section 2, we look at the previous work that addresses the problem of memory explosion in ROBDD construction and contrast it with our approach. In Section 3, we define the terminology used in the sequel. We describe our approach in Section 4, and discuss the results obtained in Section 5. We conclude with Section 6 where we discuss the possible extensions for our techniques.

2 Previous Work

Though BDDs have been researched for about four decades [24, 1], they found widespread use only after Bryant [7] showed that such graphs, under some restrictions, can be easily manipulated. The two restrictions imposed are that the graph is reduced (i.e. no two nodes have identical subgraphs), and that a total ordering of the variables is enforced. The resulting BDD is an ROBDD. The important symbolic manipulation procedures introduced by Bryant were *apply* and *compose*; these techniques operate on two identically ordered ROBDDs. *Apply* allows two ROBDDs to be combined under some Boolean operation, and *compose* allows the substitution of an ROBDD variable with a function.

ROBDDs are typically constructed using some variant of Bryant's *apply* procedure [7]; the ROBDD for a gate g is synthesized by a symbolic manipulation of the ROBDDs of its inputs, based on the functionality of g. The gates of the circuit are processed in a depth-first manner until the ROBDDs of the desired output gate(s) are constructed. The *ITE* method [5] for constructing ROBDDs is similar; there are equivalent ITE operators for *apply* and *compose*. For details on ROBDDs, and the implementation of a typical ROBDD package, please refer to [5, 7, 9].

The problem of memory explosion during ROBDD construction has received a great deal of attention in the past. In this section we briefly discuss the previous approaches and contrast them with our work.

The size of an ROBDD is strongly dependent on the ordering of its variables. Much of the prior research has focused on finding good variable orders to reduce the memory requirements needed in the ROBDD construction [25, 15, 31, 29] Good static as well as dynamic variable ordering algorithms are known which perform very well is a large number of cases. However, the problem of finding optimum variable orders in co-NP complete and there are many instances where these schemes do not perform satisfactorily. As we will see later in the results section, our schemes provide significant additional memory reductions for all the variable ordering schemes that we tried. In addition, in our methods, a Boolean function is not constrained to have the same variable order; different partitions can be ordered independently. In this way our method not only takes advantage from the advances made in the variable ordering research, but also increases the benefits available from them.

In [28, 2], it was shown that by manipulating ROBDDs in a breadth-first fashion much larger ROBDDs can be processed. This is achieved by keeping only a few levels of the ROBDD in the main memory at any given time and storing the rest in the secondary memory which is typically much larger. As only a few levels are kept in the main memory at a time, it is difficult to dynamically reorder the ROBDD during an operation. Presently, to the best of our knowledge, no breadth-first manipulation based ROBDD package is available which incorporates dynamic variable reordering. Also, the present implementations of breadth-first manipulation based ROBDD packages are less efficient than the conventional depth-first manipulation based packages, making them less practical. Again, we are focusing on an orthogonal dimension of the memory explosion problem. Each of the partitions that we create can be manipulated in a breadth-first manner if an efficient package becomes available in future.

Parallel algorithms for constructing ROBDDs were investigated in [23, 16, 30]. Large communication requirements between different partitions limits their applicability. In [16, 30] the ROBDD nodes are distributed among machines in a breadth-first manner. This practically rules out the possibility of dynamic variable ordering across partitions. In our method we create orthogonal partitions of the Boolean function represented by the ROBDD. Hence, each of the partitions can be scheduled on a different processor with minimal communication overhead and without any restrictions on variable ordering.

Some novel non-canonical methods for analyzing Boolean functions have also been developed, such as gBDDs[3], Extended-BDDs [21], IBDDs [4], and FDDs [22]. These techniques can more easily relax the ordering constraints, and can handle some functions intractable for ROBDDs. However, they need further development and thus lack widespread acceptance. Also, they lose the advantages of the canonicity of the final representation.

Canonical but fundamentally different data structures such as Typed Free BDDs [17] and OKFDDs [13] have also been proposed to extend the set of functions that can be efficiently symbolically manipulated. In practice OKFDDs provide only a small improvement over ROBDDs and can in some cases require exponential resources for manipulation. As we will discuss later, Typed Free BDDs are a special case of our approach.

Many of the functions discussed in [3, 8, 14] are represented in space polynomially bounded in the number of inputs when subfunctions are ordered independently [18]. Circuits were verified by partitioning them into

these subfunctions. Most of the partitioning techniques suggested in [18] exploit specialized structural knowledge about the circuit and are difficult to automate. No heuristics for partitioning a decomposed representation are provided.

In [12], the notion of partitioning was considered for the analysis and verification of finite state machines. In [12], the terms range partitioning and domain partitioning are used in the context of recursively calculating the range of a vector of Boolean functions by dividing it into two smaller subproblems and later combining them into one ROBDD. They do not discuss the notion of processing different partitions independently.

In [10], the notion of partitioned transition relations is developed. The transition relation of a given finite state machine is expressed as either a disjunction (using an interleaved model) or a conjunction (using a synchronous model) of ROBDDs representing individual outputs and latches, and algorithms for model checking are presented. Partitioning is restricted to building the ROBDDs of the outputs and latches separately.

3 Preliminaries

In this section we establish the terminology for the rest of this paper.

Assume we are given a circuit representing a boolean function $F \equiv F : B^n \rightarrow B^o$, with n primary inputs $X \equiv \{x_1, \ldots, x_n\}$, and o primary outputs. Let $\Psi \equiv \{\psi_1, \ldots, \psi_k\}$ be a set of variables corresponding to a *decomposition set* of the circuit. Here each ψ_i corresponds to a *decomposition point* or a *decomposition variable.*

To simplify the discussion, we will focus on a single output G. Let $G_d(\Psi, X)$ represent the decomposed ROBDD of G. Let $\Psi_{bdd} \equiv \{\psi_{i_{bdd}}, \ldots, \psi_{k_{bdd}}\}$ represent the array containing the ROBDDs of decomposition points in term of previously introduced decomposition points and PIs i.e., each $\psi_i \in \Psi$ has a corresponding ROBDD, $\psi_{i_{bdd}} \in \Psi_{bdd}$, in terms of primary input variables as well as (possibly) other $\psi_j \in \Psi$, where $\psi_j \neq \psi_i$. Elements of Ψ can be ordered such that $\psi_{j_{bdd}}$ depends on ψ_i only if $i < j$.

The composition [7] of ψ_i in $G_d(\Psi, X)$ is denoted by $G_d(\Psi, X).(\psi_i \leftarrow \psi_{i_{bdd}})$ where,

$$G_d(\Psi, X).(\psi_i \leftarrow \psi_{i_{bdd}}) = \overline{\psi_{i_{bdd}}}.G_d(\Psi, X)_{\overline{\psi_i}} + \psi_{i_{bdd}}.G_d(\Psi, X)_{\psi_i} \quad (1)$$

Here, $G_d(\Psi, X)_{\psi_i}$ represents the *restriction* of $G_d(\Psi, X)$ at $\psi_i = 1$, and is obtained by directing all the incoming edges to the node with variable id ψ_i to its $\psi_i = 1$ branch and reducing the resulting graph. Other techniques for ROBDD composition have been proposed [26]; for our purpose we will consider the approach described above.

The vector composition of the Ψ in $G_d(\Psi, X)$ is denoted as $G_d(\Psi, X).(\Psi \overset{\pi}{\leftarrow} \Psi_{bdd})$ where

$$G_d(\Psi, X).(\Psi \overset{\pi}{\leftarrow} \Psi_{bdd}) = (\ldots((G_d(\Psi, X).(\psi_{\pi(1)} \leftarrow \psi_{\pi(1)_{bdd}})).(\psi_{\pi(2)} \leftarrow \psi_{\pi(2)_{bdd}}))\ldots).(\psi_{\pi(k)} \leftarrow \psi_{\pi(k)_{bdd}}) \quad (2)$$

Here, $\pi : N \rightarrow N$ is a one to one and onto mapping from the set of Natural numbers to itself representing the order in which ψ_is are composed in G_d. $G_d(\Psi, X).(\Psi_i \overset{\pi}{\leftarrow} \Psi_{bdd})$ represents the successive composition of the ψ_is into $G_d(\Psi, X)$, in the order specified by π.

If G is the monolithic ROBDD of the output in terms of PIs, and $G_d(\Psi, X)$ is the corresponding decomposed version with Ψ_{bdd} as the array of decomposition variables, then G can be obtained from G_d and Ψ_{bdd} by successive composition of Ψ in G_d i.e.

$$G = G_d(\Psi, X).(\Psi \overset{\pi}{\leftarrow} \Psi_{bdd}) \quad (3)$$

The problem of decomposition is to find a set Ψ with the corresponding Ψ_{bdd}.

We define a *window function*, w, such that $w : B^n \rightarrow B$. The window function is defined over PIs and represents a part of the truth table on which G is defined. For a Boolean function f, $|f|$ denotes the size of its ROBDD under a given variable ordering.

4 Our Approach

During a typical bottom-up ROBDD construction procedure there is frequent functional simplification due to Boolean Absorption: $f \vee (f \wedge g) = f$, and Boolean Cancellation: $f \wedge (\overline{f} \wedge g) = 0$. Here, f and g represent intermediate results in a sequence of ROBDD operations. Suppose g is an inherently complex function having an exponential ROBDD. In the traditional approach of building ROBDDs of all the intermediate points in

terms of PIs, we would have to create the ROBDD for g. On the other hand, if we represent the ROBDDs of f and g in a decomposed form, we can detect the Boolean simplifications without having to ever build the canonical ROBDD of g. In this way we avoid the intermediate memory explosion. In addition, since we work on smaller graphs, we often gain in time as well. For cases where the final memory requirement is large or where decomposition cannot capture Boolean simplification, we partition the function into disjoint parts so that the total memory requirement in processing each of these partitions is significantly lower than that required for the monolithic representation.

So the overall strategy consists of the following parts:

- Building the decomposed representation of the final result.
- Determining a good order of composition of the Ψs into G_d.
- Partitioning the function so that the ROBDD for each partition is smaller than the ROBDD of the entire function.

We will now discuss each of these parts in some detail.

4.1 Decomposition Approach:

We employ a 'functional' decomposition approach. In this approach decomposition points are introduced based on the increase in the ROBDD sizes during the intermediate stages of ROBDD construction. During a sequence of ROBDD operations, whenever the total number of nodes in a ROBDD manager increases by a *disproportionate measure* due to some operation, we introduce a decomposition point. By doing this we postpone the instances of difficult functional manipulations to a later stage. Due to Boolean cancellation and absorption many of these cases will never occur in the final result, especially if the final memory requirement is much less than the peak intermediate requirement. If we are trying to build the ROBDD for an output node of a Boolean network, a decomposition point may be introduced when some operation on the cubes within a node causes the threshold to be exceeded. In this case, there is no physical correspondence between decomposition points and circuit nodes.

In our current implementation, the check for memory explosion is done only if the manager size is larger

than a predetermined minimum. Also, decomposition points are added when the individual ROBDD grows beyond another threshold value, to ensure that the decomposition points themselves do not become very large.

When the target function is represented as a Boolean netlist, this purely functional scheme can be augmented with structural decomposition methods [11, 20].

4.2 Order of Composition

For every candidate variable that can be composed into G_d, we assign a cost which estimates the size of resulting composed ROBDD. The variable with the lowest cost estimate is composed. This problem of determining a good order of composition was studied in considerable detail in [27]. Various cost functions were tried, based on the worst case complexity of ROBDD manipulation under compose. It was shown that a cost function based on support set size performs well in practice. Accordingly, we choose that decomposition variable which leads to the smallest increase in the size of the support set of the ROBDD after composition.

In the functional decomposition approach, a decomposition point is frequently nested inside other decomposition points. For example, $\psi_{j_{bdd}}$ corresponding to the j^{th} decomposition point can have variable ψ_i in its support set if the i^{th} decomposition point is introduced before the j^{th} decomposition point i.e., $i < j$. This dependency information is stored in the form of a *dependency graph*. The worst case complexity for a general order π of composing Ψ in $G_d(\Psi, X)$, is $O(n^2)$. At each step, we restrict the candidate ψs for composition to those decomposition points which are not present in any of the other ψ_{bdd}s. This guarantees that a decomposition variable needs to be composed only once in G_d and we never need more compositions than the cardinality of the set of decomposition variables. In this way we reduce the complexity of composition process from $O(n^2)$ to $O(n)$. It can be easily shown that this is the minimum number of compositions to get the monolithic representation of the target function from a decomposed representation in terms of n decomposition points. Since composition is the most time consuming operation in the entire algorithm, this reduction in the algorithm complexity is of significance.

4.3 Methods of Partitioning

The problem of orthogonal partitioning can be defined as follows: given a Boolean function f, we want to find k window functions $w_1, w_2, ...w_k$ and k partition functions $f_1, f_2, ...f_k$ such that $f = w_1f_1 + w_2f_2 + ... + w_kf_k$ and $w_iw_j = 0$ for $i \neq j$ and $w_1 + w_2 + ...w_k = 1$.

This method of dividing a boolean function into orthogonal partitions has a number of applications.

1. **Boolean manipulation:** If f and g be two functions which have been partitioned using the window functions $w_1, w_2, \ldots, w_k$ with $f_1, f_2, \ldots f_k$ and $g_1, g_2, \ldots g_k$ as the corresponding partitions. Any Boolean operation between f and g can be expressed as follows:

$$f \circ g = (\sum_{i=1}^{k} w_i f_i) \circ (\sum_{i=1}^{k} w_i g_i) \tag{4}$$

   Using the fact that $w_iw_j = 0$ we get,

$$f \circ g = \sum_{i=1}^{k} w_i (f_i \circ g_i) \tag{5}$$

   The above equation implies that these partitions can be manipulated completely independent of the each other.

2. *Formal Design and Implementation Verification:* To check the equivalence of two functions f and g we can check if $f \oplus g = 0$. Let $G = f \oplus g$. If $G_1, G_2, \ldots, G_k$ are the partitions of G then $G = 0$ if and only if $G_i = 0$ for $\forall i$. If any one of the partitions $G_i \neq 0$ then we can conclude that the functions are not equivalent. This procedure can be used to check the correctness of a design according to its specification or for comparing equivalence of two implementations described at different levels of abstraction.

3. Probabilistic verification: In probabilistic verification [19] every true minterm of a function F is converted into an integer value under some random integer assignment ρ to the input variables. All the integer values are then arithmetically added to get the hash code $H_\rho(F)$ for F. One can assert, with a negligible probability of error, that $F \equiv G$ *iff* $H_\rho(F) = H_\rho(G)$. After a function F is orthogonally

partitioned, no two partitions share any common minterm. Hence, we can hash each partition separately, and just add their hash codes to obtain $H_\rho(F)$. This implies that to check if $H_\rho(F) = H_\rho(G)$, we can partition and hash both F and G independently. It is not necessary that both F and G have corresponding partitions with the same window functions.

Now we outline our partitioning method. First, given a window function w_i, a decomposed representation $G_d(\Psi, X)$ and Ψ_{bdd} of f, we want to find f_i such that the ROBDD representing f_i is smaller than f. Here we make the following observation:

Observation: Let $f_i = G_{d_{w_i}}(\Psi, X)(\Psi \leftarrow \Psi_{bdd_{w_i}})$ and $f = G_d(\Psi, X)(\Psi \leftarrow \Psi_{bdd})$. If w_i is a cube on PIs then $|f_i| \leq |f|$ for any given variable order for f and f_i.

Proof: Given,

$$f_i = G_{d_{w_i}}(\Psi, X)(\Psi \leftarrow \Psi_{bdd_{w_i}}) \tag{6}$$

If w_i depends only on PIs the order of cofactoring and composition can be changed,

$$f_i = [G_d(\Psi, X)(\Psi \leftarrow \Psi_{bdd})]_{w_i} \tag{7}$$

This gives,

$$f_i = f_{w_i} \tag{8}$$

If w_i is a cube, then $|f_{w_i}| \leq |f|$ and hence $|f_i| < |f|$.

Therefore, given G_d, Ψ and w_is which are cubes, we can create the cofactors Ψ_{w_i} in $G_{d_{w_i}}$. Then by composing Ψ_{w_i} we get partition function f_i which is guaranteed to have a smaller size than the monolithic representation f. Even if the window function is a more complex function of PIs than a cube, $f_i = f_{w_i}$ where f_{w_i} is the generalized cofactor of f on w_i. The generalized cofactor of f on w_i is generally much smaller than f. Also, as the size of the result of each composition while building f_i will be less than that while building f, the intermediate peak memory requirement is also reduced. Note that the above result doesn't hold if f_i and f can have different variable orderings, which typically is the case with dynamic variable reordering.

But in practice, since dynamic variable reordering works on smaller graphs in the case of partitions it is more effective in finding smaller representations for the partitions resulting in even higher gains.

After deciding how to construct the partition function from a given window function we examine methods to obtain good window functions. These methods can be divided into two categories: apriori selection and 'explosion' based selection. As the names suggest, in apriori selection we select the window functions based on G_d and Ψ at the beginning of composition process and compose each of the partitions separately. In the 'explosion' based method, we try to compose a ψ_i into the G_d and if the graph size increases disproportionately then we select a window function based on G_d and ψ_i and recursively call the routine on each of the partitions.

4.3.1 Apriori Partitioning

In this method we select a predetermined number of PIs to partition. If we decide to partition on 'k' PIs then we create 2^k partitions corresponding to all the binary assignments of these variables. For example, if we decide to partition on say x_1 and x_2 then we create four partitions x_1x_2, $x_1\overline{x_2}$, $\overline{x_1}x_2$ and $\overline{x_1}\overline{x_2}$. From the observation made in the previous section, we know that this way of partitioning on PIs *guarantees* that the memory requirements of these partitions will be less than the memory requirement of the monolithic function. The question that remains to be answered is how to select the PIs on which to partition. The goal is to maximize the partitioning achieved while minimizing the redundancy that may arise in creating different partitions independently. For this purpose we define the cost of partitioning a function f on variable x as

$$cost_x(f) = \alpha[partition\_factor_x(f)] + \beta[redundancy\_factor_x(f)] \tag{9}$$

where,

$$partition\_factor_x(f) = max(\frac{|f_x|}{|f|}, \frac{|f_{\overline{x}}|}{|f|}) \tag{10}$$

and,

$$redundancy\_factor_x(f) = \frac{|f_x| + |f_{\overline{x}}|}{|f|} \tag{11}$$

Notice that a lower partition_factor is good as it implies that the worse of the two partitions is small and similarly a lower redundancy_factor is good since it implies that the total work involved in creating the two partitions is less. The variable x which has the lower overall cost is chosen for partitioning.

Similarly, for a given vector of functions $\mathcal{F} = f_1, f_2, ..., f_k$ and a variable x, the cost of partitioning is defined as:

$$cost_x(\mathcal{F}) = \sum_{i=1}^{k} cost_x(f_i) \tag{12}$$

We order all the PIs in increasing order of their cost of partitioning G_d and Ψ and select the best 'k' (where 'k' is a predetermined number specified by the user). This type of selection, where all the PIs are ranked according to their cost of partitioning G_d and Ψ, is called *static* partition selection. On the other hand, we can have a *dynamic* partitioning strategy in which the best PI (say x) is selected based on G_d and Ψ and then the subsequent PI is recursively selected based on G_{d_x} and Ψ_x in one partition and in $G_{d_{\overline{x}}}$ and $\Psi_{\overline{x}}$ in the other partition. The dynamic partitioning method will require an exponential number of cofactors and can be expensive. This cost can somewhat be reduced by exploiting the fact that the only values that we are interested in are the sizes of the cofactors of G_d and $\psi_{i_{bdd}}$s. An upper bound of the value of $|G_{d_x}|$ can be calculated by traversing the ROBDD of G_d and taking the $x = 1$ branch whenever the node with variable id corresponding to x is encountered. This method doesn't give the exact count as the BDD obtained by traversing the ROBDD in this manner is not reduced. The advantage is that no new nodes need to be created and the traversal is fast.

4.3.2 Explosion based partitioning

In this method we successively compose the $\psi_{i_{bdd}}$s in G_d until the graph size increases drastically for some composition (say ψ_j). When this explosion in graph size occurs, we select a window function based on the current G_d and $\psi_{j_{bdd}}$. The window function is either a PI and its complement or a $\psi_{k_{bdd}}$ which is expressed in terms of PIs only. The cost function for selecting the window function is defined in a manner analogous to the previous section, except that cofactor operations become generalized cofactor operations for window functions which are non-cubes. Once the window function w, is obtained, we create two partitions (G_{d_w}, Ψ_w) and $(G_{d_{\overline{w}}}, \Psi_{\overline{w}})$ and recursively call the routine on each of the partitions.

4.3.3 Functional Partitioning and Typed Free BDDs

Typed Free BDDs are a canonical, manipulable, and compact BDD scheme where a BDD-like structure, referred to as a *type*, is generated on a subset of variables [17]. Each of the k leafs in this structure roots a canonical BDD such that no variable is repeated from the root of the resulting graph to its terminals. A path from the common root to the leaf is similar to the window function and the subgraph rooted from the leaf is analogous to the partition function. Hence, a typed Free BDD is a special case of our partitioned ROBDD where all the partitions are represented simultaneously and are merged with a common root. Partitioned ROBDDs are more general because each partition can exist independently and the window functions do not need to share a common structure. The window function and the partition function for each partition can have an independent variable order and we believe that this flexibility can make them exponentially more compact that Typed Free BDDs in some cases.

5 Results

We have implemented an ROBDD based tautology checker in the SIS [32] environment. For our experiments, we use the ISCAS85 benchmark circuits as well as some industrial circuits. Results are reported for those outputs which are considered 'hard' from the standpoint of building ROBDDs. We modify each of the circuits using *script.rugged* in SIS and check the equivalence of the original and modified circuits.

Results for the ISCAS85 benchmarks were run on a DECstation 5000/260 with 128MB of memory. Results on industrial circuits were run on a DEC 3000/500 Alpha server, with 160MB of memory.

In tables 1, 3, 5, and 7 the "Reference" column refers to the traditional depth-first method of constructing ROBDDs. In this method the nodes are traversed in a depth first manner and ROBDD of each intermediate node is created in terms of PIs before the ROBDD of the output is created. In our implementation of this method, we changed the code in the *ntbdd_node_to_bdd* function of SIS, so that if an ROBDD of a node is not required in subsequent computations, it is freed, to reduce memory utilization. This results in a 1.5X improvement over the *ntbdd_node_to_bdd* function as released in SIS. "Decomposition" uses functional decomposition followed by composition, but without any partitioning.

Results on partitioning are reported in tables 2, 4, 6, 8, and 9. In order to demonstrate the generality of our methods, we tested them using different variable ordering schemes. Table 2 uses natural input ordering (NO) without dynamic variable reordering (DR) [29]. Table 4 uses Malik's ordering (MO) [25] without DR. Table 6 uses NO with DR, while Table 8 uses MO along with DR. These tables report results using the apriori partitioning heuristic. Table 9 reports our experiments on industrial circuits.

In all the tables, the "size" column represents the peak ROBDD manager size. All experiments were run with a million node limit on the size of the ROBDD manager, and a time limit of three hours.

Table 1 compares the "Reference" and the "Decomposition" methods. We observe that the latter method performs better in memory utilization, in almost all cases. An average gain of 1.5X is obtained in memory utilization over the "Reference". A "*" against an entry indicates that this example was run with different memory explosion thresholds than the rest of the examples. As shown in Table 2, the use of partitioning results in a significant reduction (between 1.5X and 2.5X) in memory utilization over the "Decomposition" method. This gain is accompanied by an increase in the time taken. In two examples, C2670 (output 140) and C6288 (output 14), the partitioning methods could build the BDD while the "Decomposition" method and the "Reference" failed. We also observe that among the partitioning methods, as the number of partitions is increased, there is a decreasing trend in the memory utilization along with a gradual increasing trend in the time taken. This is as anticipated. In this table, we notice a 30% time penalty between 4 partitions and 16 partitions, for a 1.5X reduction in memory utilization.

Ckt	Out	Reference		Decomposition	
	#	Time	Size	Time	Size
C880	26	14.91	200157	4.35	80157
C1355	32	1.43	30562	0.45	9891
C1908	24	5.11	47791	5.01	47164
C1908	25	6.76	33859	6.26	70800
C2670	140	–	Spaceout	Timeout	–
C3540	22	283.02	988587	391.11*	401291*
C5315	123	–	Spaceout	–	Spaceout
C6288	12	58.77	375189	55.40	361434
C6288	13	169.57	980583	173.37	654850
Total		539.57	2656728	635.95	1625587

Table 1: Natural Ordering without Dynamic Reordering

Ckt	Out	4 Partitions		8 Partitions		16 Partitions		32 Partitions	
	#	Time	Size	Time	Size	Time	Size	Time	Size
C880	26	27.82	22930	27.15	21801	30.54	20926	35.12	20264
C1355	32	6.14	8056	6.23	8001	6.98	7234	8.85	7074
C1908	24	47.04	58291	32.07	50115	36.39	42536	43.57	40399
C1908	25	44.67	39887	43.71	34667	47.39	29394	56.29	29050
C2670	140	10332.68	263479	1676.74	247641	1719.84	245165	2669.90	245165
C3540	22	423.34*	151662*	467.09*	123449*	500.46*	111506*	524.08*	84471*
C5315	123	–	Spaceout	–	Spaceout	–	Spaceout	–	Spaceout
C6288	12	163.92	264325	163.31	222342	210.91	180174	192.80	164368
C6288	13	337.70	418668	369.25	381981	388.28	344703	492.25	325001
C6288	14	945.03	829218	1030.12	764307	1353.00	682015	1512.75	579619
Total		1050.63	963819	1108.81	842355	1220.95	736473	1352.96	670627

Table 2: Natural Ordering, without Dynamic Reordering and Partitioning

In Table 3, we once again observe that the "Decomposition" algorithm has better memory performance (by about 1.3X) over the "Reference" method. Examples C880, C1355, C1908, and C5315 could be built without any intermediate memory explosion, hence decomposition was not invoked. Accordingly, the memory utilization of both algorithms is identical for these examples. For this reason, these examples are not included in Table 4, even though the time and memory requirements were very small. This table once again shows an impressive reduction in the memory utilization. With 32 partitions, example C3540 was verified in a little under an hour. Further, output 14 of C6288 was verified by most of the partitioning methods while neither the "Reference" nor "Decomposition" schemes could do so. Once again the same trends in time and memory utilization are observed with an increasing number of partitions.

Tables 5, 6, 7 and 8 exhibit the same trends as before, validating the claim that our partitioning schemes work well even when Dynamic Reordering is used. In Tables 5 and 6, we obtain great memory improvements over the "Reference" and "Decomposition" algorithms (about 1.75X) with very little time penalty. In fact, the partitioning methods perform consistently better than the "Decomposition" method.

Once again, in Tables 7 and 8, we obtain vast time and memory improvements by using partitioning. In Tables 7, we once again observe that examples C880, C1355, C1908 and C5315 were built without the decomposition step being invoked. As a result, these examples are not included in Table 8.

We were able to obtain a couple of large industrial examples to run our examples on. The results of

Ckt	Out	Reference		Decomposition	
	#	Time	Size	Time	Size
C880	26	0.27	9944	0.33	9944
C1355	32	0.85	12671	0.88	12671
C1908	24	1.40	14039	1.48	14039
C1908	25	1.71	13790	1.88	13790
C2670	140	26.16	234562	1.55	28002
C3540	22	–	Spaceout	–	Spaceout
C5315	123	0.97	10008	1.15	10008
C6288	12	35.94	366237	25.25	197341
C6288	13	97.91	861241	207.98	932092
Total		160.01	1462040	234.78	1157435

Table 3: Malik Ordering without Dynamic Reordering

Ckt	Out	4 Partitions		8 Partitions		16 Partitions		32 Partitions	
	#	Time	Size	Time	Size	Time	Size	Time	Size
C2670	140	137.94	35301	141.04	26108	148.41	23166	153.00	21632
C3540	22	–	Spaceout	–	Spaceout	–	Spaceout	3268.42	998070
C6288	12	116.06	136114	116.54	111020	125.64	101727	139.89	96903
C6288	13	344.80	566148	347.08	410485	418.70	376649	489.15	324906
C6288	14	–	Spaceout	1970.08	963159	2152.01	989251	3013.71	895904
Total		598.80	737563	604.66	547613	692.75	501542	782.04	443441

Table 4: Malik Ordering, without Dynamic Reordering and Partitioning

Ckt	Out	Reference		Decomposition	
	#	Time	Size	Time	Size
C880	26	6.84	9809	7.54	12723
C1355	32	15.95	8745	0.46	9891
C1908	24	6.28	13975	9.42	12901
C1908	25	4.16	10065	4.41	10523
C2670	140	118.09	12579	172.99	11051
C3540	22	466.26	117210	943.37*	127598*
C5315	123	16.96	11107	11.54	9874
C6288	12	855.35	413792	4289.16	343331
Total		1489.89	597282	5438.89	537892

Table 5: Natural Ordering with Dynamic Reordering

Ckt	Out	4 Partitions		8 Partitions	
	#	Time	Size	Time	Size
C880	26	9.83	1926	9.96	1777
C1355	32	12.20	5744	13.07	5745
C1908	24	12.80	4339	13.28	4156
C1908	25	6.03	1484	6.20	1802
C2670	140	204.19	8947	133.80	9203
C3540	22	525.68*	82857*	704.06*	88552*
C5315	123	13.13	572	13.02	579
C6288	12	1508.14	221099	1091.36	174987
Total		2292.00	326968	1984.75	286801

Table 6: Natural Ordering, with Dynamic Reordering and Partitioning

Ckt	Out	Reference		Decomposition	
	#	Time	Size	Time	Size
C880	26	0.74	9944	0.67	9944
C1355	32	1.97	12671	1.93	12671
C1908	24	12.64	11307	12.77	11307
C1908	25	11.48	10679	11.70	10679
C2670	140	12.70	10041	10.56	9191
C3540	22	2984.75	553817	3166.14	453807
C5315	123	2.23	10008	2.36	10008
C6288	12	1890.81	293221	1190.55	184685
C6288	13	7822.49	751922	–	Spaceout
Total		12710.75	1609001	4367.25	647683

Table 7: Malik Ordering with Dynamic Reordering

Ckt	Out	4 Partitions		8 Partitions	
	#	Time	Size	Time	Size
C2670	140	14.54	576	14.89	582
C3540	22	2083.47	307335	2465.80	400511
C6288	12	801.49	107576	481.12	49700
C6288	13	4297.38	502645	5618.27	361018
Total		2899.50	415487	2961.81	450793

Table 8: Malik Ordering, with Dynamic Reordering and Partitioning

these runs are shown Table 9. Both these circuits have more than 200 inputs and 2000 complex multi-level gates. They could not be verified using a non-ROBDD based scheme [6]. The "Reference" algorithm was not able to build the ROBDDs for either 'Ind1' or 'Ind2', using Malik Ordering with DR. For 'Ind1', the "Decomposition" method was able to build the ROBDD. Our partitioning method shows further gains in memory on this example. For 'Ind2', the partitioning method timed out in 5 hours. The "Decomposition" method was able to build the ROBDD in just under 5 hours.

Ckt	Reference		Decomposition		4 Partitions	
	Time	Size	Time	Size	Time	Size
Ind1	–	Spaceout	2547.77	100852	2551.22	74760
Ind2	–	Spaceout	17754.99	993345	Timeout	–

Table 9: Industrial Circuits, Malik Ordering, with Dynamic Reordering

6 Conclusions

The main conclusions of our work are as follows:

- We have implemented a general ROBDD construction procedure. Using decomposition and partitioning we are able to effectively reduce the memory requirements in the ROBDD construction process by trading off memory with time.
- Two heuristics for generating orthogonal partitions are presented. One is based on an apriori analysis of the decomposed output function, while the other relies on an analysis of the growth of the decomposed output function, as the composition proceeds.
- Our decomposition and partitioning strategies are based on purely functional considerations, and as a result, our overall scheme can be applied to any general sequence of ROBDD operations. More specifically, the method can also be used in the construction of transition relations of finite state machines and for language containment and model checking in formal design verification.

- Results were run on known hard circuits from the ISCAS85 benchmark suite, as well as on some hard industrial circuits. We show significant improvements over the conventional bottom-up procedures of verifying ROBDDs. In some cases, our methods were able to build ROBDDs where the existing schemes failed. All the experiments are done using only the default values for memory explosion thresholds, unless where otherwise noted. This shows that the methods are robust. Much better results can be obtained by some experimentation with the thresholds.

- Results are reported for different variable ordering schemes including dynamic ordering and are consistently better than the conventional methods. These results show that our approach complements the research on variable ordering and can be used in conjunction with any ordering scheme to get further reductions in memory utilization.

- Since different partitions can have different variable orders our approach extends the class of functions that can be manipulated with polynomial resources using ROBDDs. We show that our partitioned representation is more general than Typed Free BDD.

- Our scheme is flexible in that if the output ROBDD can be built without any memory explosion, no decomposition variables are introduced. Also, different schemes for decomposition and composition can be used, resulting in a powerful array of ROBDD construction techniques.

Future research is directed towards identifying other ways of partitioning the ROBDDs. We also plan to augment the functional decomposition scheme with structural methods for the purpose of combinational verification. We plan to use this approach to reduce the resources required in computing transition relations, reachability and for model checking and language containment in the context of formal design verification.

References

[1] Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27:509–516, June 1978.

[2] P. Ashar and M. Cheong. Efficient breadth-first manipulation of binary decision diagrams. *ICCAD*, pages 622–627, 1994.

[3] P. Ashar, A. Ghosh, and S. Devadas. Boolean satisfiability and equivalence checking using general binary decision diagrams. *ICCD*, pages 259–264, October 1991.

[4] J. Bitner, J. Jain, D. S. Fussell, J. A. Abraham, and M. Abadir. Efficient algorithmic circuit verification using indexed bdds. *International Symposium on Fault Tolerant Computing*, pages 266–275, 1994.

[5] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient Implementation of a BDD Package. In *Proc. of the Design Automation Conf.*, pages 40–45, June 1990.

[6] D. Brand. Verification of Large Synthesized Designs. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 534–537, November 1993.

[7] R. E. Bryant. Graph based algorithms for Boolean function representation. *IEEE Transactions on Computers*, C-35:677–690, August 1986.

[8] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, C-40:206–213, February 1991.

[9] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24:293–318, September 1992.

[10] J. R. Burch, E. M. Clarke, D. E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(4):401–424, April 1994.

[11] G. Cabodi, P. Camurati, and Stefano Quer. Auxillary variables for extending symbolic traversal techniques to data paths. *31st Design Automation Conference*, pages 289–293, 1994.

[12] O Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines based on symbolic execution. *International Workshop on Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science*, 407:365–373, 1989.

[13] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. *31st Design Automation Conference*, pages 415–419, 1994.

[14] L. Fortune, J. Hopcroft, and E. M. Schmidt. The complexity of equivalence and containment for free single variable program schemes. *Goos, Hartmanis, Ausiello and Bohm, Eds., Lecture Notes in Computer Science 62, Springer-Verlag*, pages 227–240, 1978.

[15] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Boolean Decision Diagrams. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 2–5, November 1988.

[16] M. Rebaudengo G. Cabodi, S. Gai and M. Sonza Reorda. A data parallel approach to Boolean function manipulation using BDDs. In *First International Conference on Massively Parallel Computing Systems (MPCS)*, pages 163–75, May 1994.

[17] J. Gergov and C. Meinel. Efficient analysis and manipulation of OBDDs can be extended to read-once-only bracnhing programs. *IEEE Transcations on Computers*, C-43:1197–1209, October 1994.

[18] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham. Functional partitioning for verification and related problems. *Brown/MIT VLSI Conference*, March 1992.

[19] J. Jain, J. Bitner, D. S. Fussell, and J. A. Abraham. Probabilistic verification of Boolean functions. *Formal Methods in System Design*, 1, 1992.

[20] J. Jain, A. Narayan, C. Coelho, S. Khatri, A. Sangiovanni-Vincentelli, R. Brayton, and M. Fujita. Combining Top-down and Bottom-up Approaches for ROBDD Construction. Technical Report UCB/ERL M95/30, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, April 1995.

[21] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Structural BDDs: Trading canonicity for structure in verification algorithms. *ICCAD*, 1991.

[22] U. Kebschull, E. Schubert, and W. Rosentiel. Multilevel logic based on functional decision diagrams. *European Design Automation Conference*, pages 43–47, 1992.

[23] S. Kimura and E. M. Clarke. A parallel algorithm for constructing binary decision diagrams. *ICCD*, pages 220–223, 1990.

[24] C. Y. Lee. Representation of switching circuits by binary-decision programs. *Bell Syst. Tech. J.*, 38:985–999, 1959.

[25] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 6–9, November 1988.

[26] K. L. McMillan. Symbolic model checking: An approach to the state explosion problem. *Ph.D Thesis, Dept. of Computer Sciences, Carnegie Mellon University*, 1992.

[27] A. Narayan, S. P. Khatri, J. Jain, M. Fujita, R. K. Brayton, and A. Sangiovanni-Vincentelli. Compositional Techniques for Mixed Bottom-Up / Top-Down Construction of ROBDDs . Technical Report UCB/ERL M95/51, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, June 1995.

[28] H. Ochi, K. Yasuoka, and S Yajima. Breadth-first manipulation of very large binary decision diagrams. *ICCAD*, pages 48–55, 1993.

[29] R. L. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams . In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 42–47, November 1993.

[30] M Rebaudengo S. Gai and M. Sonza Reorda. An improved data parallel algorithm for Boolean function manipulation using BDDs. In *Euromicro Workshop on Parallel and Distributed Processing*, pages 33–9, January 1995.

[31] F. Somenzi S. Panda and B. F. Plessier. Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. In *Proc. of the Intl. Conf. on Computer-Aided Design*, pages 628–631, November 1994.

[32] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.