Compositional Techniques for Mixed Bottom-Up/Top-Down Construction of ROBDDs

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)

[1]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA 94720
tel: (510)-642-5048
fax: (510)-643-5052
[2]Fujitsu Laboratories of America, San Jose, CA 95134

Abstract

Reduced Ordered Binary Decision Diagrams (ROBDDs) have traditionally been built in a bottom-up fashion, through the recursive use of Bryant's *apply* procedure [4], or the *ITE* [2] procedure. With these methods, the intermediate peak memory utilization is often larger than the final ROBDD size. This peak memory requirement limits the complexity of the circuits which can be processed using ROBDDS.

Recently it was shown in [9] that for a large number of applications, the peak memory requirement can be substantially reduced by a suitable combination of bottom-up (decomposition based) and top-down (composition based) approaches of building ROBDDs. This approach consists of selecting suitable decomposition points during the construction of the ROBDD using the *apply* procedure, followed by a symbolic composition to obtain the final ROBDD.

In this paper, we focus on the composition process. We detail four heuristic algorithms for finding good composition orders, and compare their utility on a set of standard benchmark circuits. Our schemes offer a matrix of time-memory tradeoff points.

1 Introduction

In the last decade, CAD tools have permeated the field of digital design. Reduced Ordered Binary Decision Diagrams (ROBDDs) [4] are frequently used to solve various CAD problems such us synthesis, digital-system verification and testing. Many solution techniques for combinational and sequential optimization problems use ROBDDs as the Boolean representation of choice.

Since the construction of ROBDDs is often a time and memory intensive process, techniques that can help in a quicker construction of ROBDDs are of practical significance. ROBDDs are canonical given a variable ordering, and hence the final size of the representation is a constant, regardless of the sequence of boolean operations by which the ROBDD was constructed. In the traditional (*apply* based) scheme of constructing ROBDDs, the peak intermediate memory requirement often far exceeds the final (canonical) representation size of the given function. Although we are solely interested in obtaining the ROBDD of the output, the intermediate peak memory requirement often limits our ability to construct the output ROBDD. Hence the peak memory requirement places a limit on the complexity of circuits that can be processed using ROBDDs, and also usually dictates the time required for ROBDD construction. It would therefore be very desirable to construct ROBDDs in a fashion which reduces the intermediate peak memory requirement. This has an attendant benefit of a reduction in the time required to construct the ROBDD.

The presence of an intermediate peak followed by a subsequent reduction in the memory requirements indicates that Boolean simplification has occurred in the circuit. We attempt to capture this Boolean simplification so as to circumvent the peak memory requirement. To achieve this goal, we follow the general procedure outlined in [9]. We introduce suitable decomposition points, and perform all computations on smaller decomposed graphs. Thus, we get a decomposed representation of the output function, which captures the Boolean simplification occurring in the circuit. Finally, we compose the decomposition functions into the ROBDD of the output, to obtain the canonical output ROBDD.

In this scheme, we are able to build ROBDDs even when the intermediate memory requirement using traditional techniques is larger than can be handled by existing ROBDD packages. Keeping the peak memory requirements low has the added advantage that the ROBDD is usually built in less time, since operations on large ROBDDs are avoided.

In the first phase of this method, a bottom-up construction of the ROBDD is attempted. Decomposition points are introduced when the output ROBDD and / or the ROBDD manager grows beyond a user-defined threshold. Just as in [9], we introduce decomposition points "functionally", in the sense that the decomposition points do not necessarily correspond to structural nodes in the circuit. Thus, 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 the second phase, the decomposition functions are composed back into the output ROBDD, to get the monolithic ROBDD of the output in terms of the primary inputs. In this paper, we focus our attention on this phase. We discuss and demonstrate various heuristic algorithms to perform this composition step; therein lies our contribution.

In Section 3, we discuss the basic definitions and terminology that we use throughout this paper. In Section 4 we discuss the decomposition procedure. We describe our composition techniques in Section 5, and discuss the results obtained in Section 7. We conclude with Section 8 where we summarize our contributions, and give directions for future research in this area.

2 Previous Work

Though ROBDDs have been researched for about four decades [12, 1], they found widespread use only after Bryant [4] showed that such graphs, under some conditions, can be easily manipulated. These conditions 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 ROBDD is called a Reduced Ordered BDD. Bryant introduced two symbolic manipulation procedures - *apply* and *compose*, which are used to combine two identically ordered ROBDDs. *Apply* allows ROBDDs to be combined under some Boolean operation, and *compose* allows the composition of one ROBDD into another. Both algorithms require the ROBDD variables to have an identical order.

Another important development in this area is the work on Dynamical Variable Ordering [16]. Local alterations are performed on the existing variable order, and a change in the variable order is accepted if it results in a reduced ROBDD manager size. This scheme was shown to require reduced amounts of memory for most circuits, at the cost of increased execution time.

ROBDDs are typically constructed using some variant of Bryant's *apply* procedure [4]. The gates of the circuit are processed in a depth-first manner until the ROBDDs of the desired output gate(s) are constructed.

However, an exclusively bottom-up approach is often not the most computationally efficient technique for constructing ROBDDs, as discussed in [9]. The primary focus of [9] was on finding good decomposition techniques. A purely structural method was discussed and a functional method proposed for finding these decomposition points. The functional method was shown to have more flexibility in choosing decomposition points, encompassed the structural method in its scope, and obtained better results. Accordingly, we use the functional decomposition method in our work.

In our opinion, while the overall procedure of [9] is sound, insufficient attention was given to the problem of composing the decomposed variables back into the ROBDD of the decomposed output function. This is where we focus our attention. We propose various composition schemes and compare their effectiveness.

3 Preliminaries

In this section we provide the definitions and terminology for rest of the paper. In the sequel, we restrict our discussion to Reduced Ordered Binary Decision Diagrams (ROBDDs) alone. However, the work is equally well applicable to other graph based representation schemes which are an improvement over ROBDDs [13, 10, 3] as well as other representation schemes such as FDDs [11], OKFDDs [6] and Typed-Free BDDs [7]. For details on ROBDDs, please refer to [3, 4, 5]. For an efficient implementation of the ROBDD package, [3] is an excellent reference.

Assume we are given a circuit representing a boolean function $F \equiv F : B^n \rightarrow B^o$, with n primary inputs $x_1, \ldots, x_n$, and o primary outputs. To simplify the discussion, we will focus our attention on a single output G. Let $G_d(\Psi)$ represent the ROBDD of G expressed in terms of a variable set $\Psi = \{\psi_1, \psi_2, \ldots, \psi_m\}$. Each $\psi_i \in \Psi$ has a corresponding ROBDD, $\psi_{i_{bdd}}$, 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 $i < j$. These $\psi_i \in \Psi$ are called *Decomposition points* of G. $G_d(\Psi)$ is the decomposed ROBDD of G in terms of the decomposition points.

The composition [4] of ψ_i in $G_d(\Psi)$ is denoted by $G_d(\Psi).(\psi_i \leftarrow \psi_{i_{bdd}})$ where $G_d(\Psi).(\psi_i \leftarrow \psi_{1_{bdd}}) = \overline{\psi_{i_{bdd}}}.G_d(\Psi)_{\overline{\psi_i}} + \psi_{i_{bdd}}.G_d(\Psi)_{\psi_i}$.

$G_d(\Psi)_{\psi_i}$ represents the *restriction* of $G_d(\Psi)$ 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. $G_d(\Psi)_{\psi_i}$ is also referred to as the ψ_i-cofactor of $G_d(\Psi)$.

Other techniques for ROBDD composition have been proposed [15, 8]; for our purpose we will consider the approach described above.

4 Decomposition Strategy

We use the functional decomposition technique reported in [9].

We introduce a decomposition variable at points where the ROBDD size and / or the ROBDD manager size increases by a large measure. ROBDDs of the remaining circuit are now built in terms of primary inputs, previously introduced decomposition variables as well as the newly introduced decomposition variable. Accordingly, the ROBDD of the output is built in a decomposed manner. In this scheme no decomposition is performed when the function can be processed without any memory explosion.

In our implementation, we decompose a function when the total ROBDD manager size increases by more than a user-specified threshold, as a result of some Boolean operation. This check is only done if the manager size is larger than a user-specified minimum. Additionally, a decomposition point is added when an individual ROBDD grows beyond another threshold value, also provided by the user.

Note that this decomposition scheme may introduce decomposition variables that do not physically correspond to any circuit node, hence it has a large flexibility in the choice of decomposition points. Results on the ISCAS85 benchmarks showed a great improvement in time and memory, over the traditional bottom-up method of building the ROBDDs [9]. In numerous cases, the output ROBDD could be built only by the new scheme. The improvements were especially significant for circuits for which the ROBDD is known to be difficult to build. A significant improvement was demonstrated for different variable ordering schemes, including Dynamic Variable Reordering [16].

5 Our Approach

In this section we describe our algorithms for finding good composition orders. As we have already discussed, decomposition points are introduced purely based on the growth of the ROBDDs. No topological analysis of the circuit is required to identify these points. As such, every new decomposition point ($\psi_{i_{bdd}}$) may have in its support set some ψ_j's, where $j < i$. We store this dependency information in the form of a *dependency graph*. After analyzing this graph, we choose the best ψ_i to be composed. The choice of the ψ_i to compose is made on the basis of some cost function, which tries to estimate the resulting graph size after composition. Once a variable ψ_i is composed in G_d, we update the dependency graph for subsequent compositions. So the algorithm consists of three main parts:

- Initializing the dependency graph
- Selection of the ψ_i to be composed
- Updating the dependency graph after composition.

In the following section we describe each of these steps in some detail.

5.1 Initializing the dependency graph

The dependency graph is a directed acyclic graph. For every decomposition point (say ψ_i), there is a corresponding node in the dependency graph ($node_{\psi_i}$). There is one additional node corresponding to G_d, called $node_{G_d}$. There is an edge from $node_{\psi_i}$ to $node_{\psi_j}$ if and only if ψ_i is present in the support of $\psi_{j_{bdd}}$. Similarly an edge from $node_{\psi_i}$ to $node_{G_d}$ represents that G_d depends on ψ_i. This graph is guaranteed to be acyclic as ψ_i is present in ψ_j only if $j > i$.

If an edge goes from $node_i$ to $node_j$ then $node_j$ is called the parent of $node_i$ and $node_i$ is called the child of $node_j$. A node $node_i$ is called the immediate child of $node_j$ if $node_i$ is a child of $node_j$ and no other path exists between $node_i$ and $node_j$.

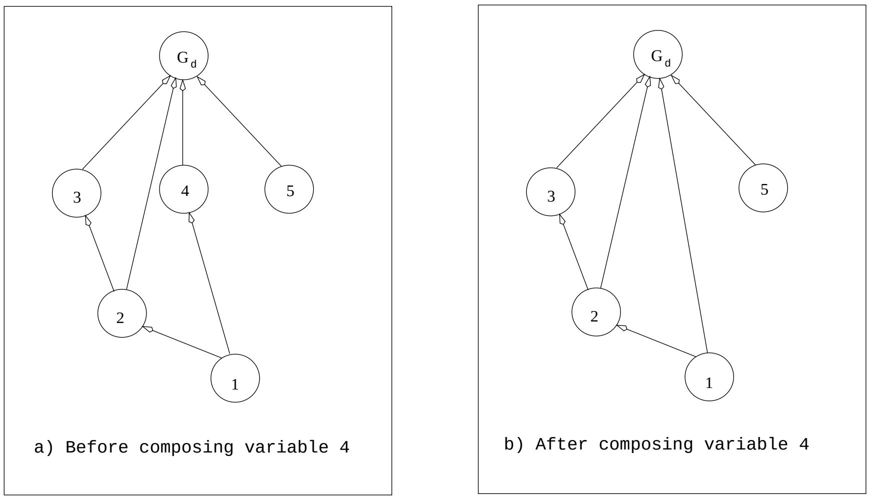

Figure 1: *Dependency Graph, showing changes after an update*

Figure 1(a) shows a dependency graph. Note that nodes 3, 4 and 5 are immediate children of the G_d node, while 2 is not, even though it is a child of G_d.

5.2 Cost Function Heuristics

We implement four separate heuristics for cost function evaluation. For every candidate variable that can be composed into G_d, we assign a cost which estimates the size of the resulting composed ROBDD. The variable with the lowest cost estimate is composed.

In all our heuristics, the candidate nodes for composition are the immediate children of G_d. This choice ensures that we never need more compositions than the cardinality of the set of decomposition variables. Without this restriction, the composition process was found to take an inordinately long time.

Our four heuristics are detailed below.

5.2.1 Bound of the Size of the Composed ROBDD

For any boolean operation on ROBDDs f_1 and f_2 it is well known that the resulting ROBDD is bounded in size by $\mid f_1 \mid \times \mid f_2 \mid$, where $\mid f \mid$ represents the number of nodes in the ROBDD of f.

With this result, it is easy to show that the composed ROBDD $G_d(\Psi).(\psi_i \leftarrow \psi_{i_{bdd}})$ is bounded by

$| G_d(\Psi) |^2 \times | \psi_{i_{bdd}} |$. Bryant conjectures in [4] that this bound is actually $| G_d(\Psi) | \times | \psi_{i_{bdd}} |$. The size of the composed ROBDD $| G_d(\Psi).(\psi_i \leftarrow \psi_{i_{bdd}}) |$ is equal to $| \overline{\psi_i} G_d(\Psi)_{\overline{\psi_i}} + \psi_i G_d(\Psi)_{\psi_i} |$. The size of $G_d(\Psi)_{\overline{\psi_i}} \overline{\psi_i}$ and $G_d(\Psi)_{\psi_i} \psi_i$ are bounded by $| G_d | \times | \psi_{i_{bdd}} |$. This bound is strong for the product operation. If Bryant's conjecture is true, then the reason why the resulting composition doesn't blow up is that the OR operation doesn't blow up. If this is the case then the bound for the size of the composed ROBDD would be $(| G_d(\Psi)_{\overline{\psi_i}} | + | G_d(\Psi)_{\psi_i} |) \times | \psi_{i_{bdd}} |$. We use this bound as our first cost function to select the composition variable.

5.2.2 Support Set Size of the Composed ROBDD

A good rule of thumb for the size of an ROBDD is the size of its support set. As our second cost function, we use the increase in the support size of the composed ROBDD to select the decomposition point to be composed. We choose that decomposition variable which leads to the smallest increase in the size of the support of the composed ROBDD. This method works well, as we will see in the results section.

5.2.3 Sum of the sizes of $G_d(\Psi)_{\overline{\psi_i}}$ $G_d(\Psi)_{\psi_i}$ and ψ_i ROBDD

In most cases, the product of sizes of two functions as an estimate for the size of their resulting ROBDD is very loose. Instead of taking the products as the cost function, we take the sum of the sizes of the ROBDDs as a measure of cost of composing a decomposition point. So we compose a variable ψ_i whose estimate of $| G_d(\Psi)_{\overline{\psi_i}} | + | G_d(\Psi)_{\psi_i} | + | \psi_{i_{bdd}} |$ is minimum. In essence this scheme favors variables ψ_i which have smaller $\psi_{i_{bdd}}$s and result in smaller cofactors $G_d(\Psi)_{\overline{\psi_i}}$ and $G_d(\Psi)_{\psi_i}$

5.2.4 Size and Number of Occurrences of ψ

Our final cost function counts the number of nodes in G_d that correspond to a variable ψ. This is a measure of the number of times each instance of the composition will have to be done on the graph, and is a reasonable estimate of the complexity of composing ψ into G_d. This count is multiplied by the size of $\psi_{i_{bdd}}$, which serves as an estimate of the complexity of each instance of composition. It can be shown that if the supports of $\psi_{i_{bdd}}$ and G_d are disjoint, then $\psi_{i_{bdd}}$ can be composed in G_d by replacing all occurrences of ψ_i in G_d by

$\psi_{i_{bdd}}$. The resulting graph size, in this case, would be the same as the estimate.

5.3 Updating the dependency graph

Suppose a decomposition point, ψ_i, is composed in $\psi_{k_{bdd}}$. Now the dependency graph needs to be updated. This is done by removing the edge from $node_{\psi_i}$ to $node_{\psi_k}$ and adding new edges from the children of node i to the node k. Also, if node i (corresponding to ψ_i) does not fan out to any other node after composition then we remove node i and all the edges coming into node i. Figure 1(b) shows the result of composing $\psi_{4_{bdd}}$ in G_d.

In Section 6 we discuss the algorithm and implementation details, including the data structure used for the dependency graph.

6 Algorithm and Implementation Details

Figure 2 shows the complete algorithm schematically.

```
Make G_d and ψ-array
Initialize the dependency graph
while (no. of decomp. points > 0)
    Select the minimum cost immediate child of node_{G_d}
    Compose the selected ψ in G_d
    Update the dependency graph
repeat
```

Figure 2: **Algorithm:** *Functional Decomposition and Composition*

First the decomposed version of the output node is built in a bottom-up fashion. Starting from the primary inputs, the nodes are visited in depth-first order and their ROBDDs built in terms of primary inputs. Once the manager size becomes bigger than a user-specified threshold, a decomposition point is introduced. A new variable is created and subsequent ROBDDs are created in terms of this variable.

The ROBDD corresponding to this decomposition point is stored in an array denoted by ψ-array. At the end of the decomposition procedure, we get the ROBDD of the output in terms of primary inputs and intermediate variables ψ_is. We also get an array containing the $\psi_{i_{bdd}}$s.

The dependency graph is maintained in the form of an *adjacency matrix*. This matrix has n rows and $(n+1)$ columns where n is the number of decomposition points introduced during the bottom-up phase of the algorithm. This is a $0-1$ matrix. A "1" bit present in i^{th} row and j^{th} column, where $1 \leq i, j \leq n$ indicates that $\psi_{j_{bdd}}$ depends on ψ_i. A "1" entry in the i^{th} row and $(n+1)^{th}$ column indicates that G_d depends on ψ_i. By looking at the support set of $\psi_{i_{bdd}}$'s and G_d we can easily initialize the dependency graph.

We select the node from the immediate children of $node_{G_d}$ that has the minimum cost of composition. The strategy is greedy in nature as it picks the best solution at each stage. By restricting the choice of decomposition points to immediate children of $node_{G_d}$ we make sure that a decomposition point has to be composed only once in the final graph. This results in minimum number of compositions to get the monolithic representation of the target function. The while loop has to be executed only n times. If we do not restrict the choice to immediate children, then in the worst case we will need $O(n^2)$ compositions. Since composition is the most time-consuming operation in the entire algorithm, this reduction in the algorithm complexity is of significance.

To check whether a given ψ is the immediate child of G_d, we need to check if the $(i,(n+1))^{th}$ entry is a "1" and all the other entries in the i^{th} row are 0. To update the graph after ψ_i is composed in G_d, we change the $(i, n+1)$ entry to a "0", and for rows having a non-zero i^{th} entry, we change the i^{th} column to "0" and $(n+1)^{th}$ column to "1".

7 Results

We implemented the algorithms in the C language, under the SIS [17] programming environment. Our results were run on a DECsystem 5900/260, with 440MB memory. We ran our experiments on the ISCAS85 benchmark circuits, for outputs which are considered hard for traditional ROBDD packages [17].

In the tables below, COMP-SIZE corresponds to the method which estimates cost by the bound on the size of the composed ROBDD shown in Section 5.2.1. SUP-SIZE refers to the support set heuristic of Section 5.2.2, while SUM-SIZE uses the size estimate of Section 5.2.3 for the composed ROBDD. Finally, NUM-PSI uses the number of occurrences of ψ_i variable in G_d times the size of $\psi_{i_{bdd}}$ as the cost function

(Section 5.2.4).

For each scheme, we have two sets of experiments. In the first set, we use Natural Ordering with and without Dynamic Variable Reordering [16]. These results are summarized in Tables 1, 3, 5 and 7. The results with Dynamic Variable Reordering are shown under the heading 'DR'. The second set of experiments uses Malik's Ordering [14]. These results are presented in Tables 2, 4, 6 and 8.

Tables 2 and 1 report the size of the maximum ROBDD encountered during the run. Methods COMP-SIZE and SUM-SIZE do very well under Natural Ordering both with and without DR but not so well under Malik's Ordering. SUP-SIZE and NUM-PSI do better under Malik's Ordering as compared to the other two schemes. SUP-SIZE and NUM-PSI give similar results without DR. With DR, NUM-PSI slightly outperforms SUP-SIZE in the results shown here.

Ckt	**Out**	**COMP-SIZE**		**SUP-SIZE**		**SUM-SIZE**		**NUM-PSI**	
	#	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR
C880	26	42623	2395	42623	2395	42623	2395	42623	2395
C1908	24	8519	4032	8519	4032	8519	4032	8519	4032
C1908	25	4032	3978	4032	3978	4032	3978	4032	3978
C2670	140	–	30063	–	19341	–	48689	–	30421
C3540	22	127916	28755	259859	40969	126297	28755	267589	37079
C6288	12	56267	29954	61685	30126	61685	26849	61685	26353
C6288	13	122797	48954	150788	71705	150788	69135	150788	79841
C6288	14	302716	183314	367730	183298	367730	152002	367730	130860

Table 1: Max. ROBDD Size, Natural Ordering, 1M limit

Ckt	**Out**	**COMP-SIZE**		**SUP-SIZE**		**SUM-SIZE**		**NUM-PSI**	
	#	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR
C880	26	1081	1081	1081	1081	1081	1081	1081	1081
C1908	24	1735	1735	1735	1735	1735	1735	1735	1735
C1908	25	1044	1044	1044	1044	1044	1044	1044	1044
C2670	140	78523	37542	12183	5494	78523	37542	78523	26199
C3540	22	–	71278	–	44418	–	87604	–	41212
C6288	12	68709	22788	69856	38467	68709	52265	69856	43499
C6288	13	239991	115439	195175	99536	195175	98056	195175	72673
C6288	14	–	300231	483822	212648	352499	184150	483822	183675

Table 2: Max. ROBDD Size, Malik Ordering, 1M limit

Tables 4 and 3 report the maximum size of the ROBDD manager at any given point in the computation.

In these tables, the SUP-SIZE heuristic does well in almost all cases on average. For some examples, NUM-PSI outperforms SUP-SIZE under Malik's Ordering and DR. The reason for this is that SUP-SIZE requires just two support set computations, and NUM-PSI requires the number of nodes of the variable ψ_i present in G_d. No cofactor operations are needed for calculating the cost, as is the case in COMP-SIZE and SUM-SIZE, where the computation of positive and negative cofactors of G_d with the variable under consideration increases the total memory usage.

Ckt	**Out**	**COMP-SIZE**		**SUP-SIZE**		**SUM-SIZE**		**NUM-PSI**	
	#	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR
C880	26	49405	10002	49405	10002	49405	10002	49405	10002
C1908	24	45072	11237	44082	10888	45072	11237	39168	10886
C1908	25	44448	11970	28282	11884	44448	11970	28282	11884
C2670	140	–	59441	–	35230	–	99889	–	51486
C3540	22	442478	130819	684133	93160	413262	140495	435012	103270
C6288	12	191047	131515	111809	76881	159144	87911	111809	70649
C6288	13	460666	100312	324514	135923	362562	155589	324514	189991
C6288	14	933533	341926	764686	216267	838597	391903	764686	261911

Table 3: Max. ROBDD Manager Size, Natural Ordering, 1M limit

Ckt	**Out**	**COMP-SIZE**		**SUP-SIZE**		**SUM-SIZE**		**NUM-PSI**	
	#	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR
C880	26	6468	6468	6468	6468	6468	6468	6468	6468
C1908	24	13800	13800	10628	10628	13800	13800	10628	10628
C1908	25	16669	9992	13091	9795	16669	9992	13091	9795
C2670	140	160625	79082	15590	9578	160625	79082	106165	32215
C3540	22	–	217532	–	159691	–	207927	–	118776
C6288	12	301873	66350	167242	77790	301873	88657	167242	72343
C6288	13	719791	242289	378995	202737	511168	202903	378995	136456
C6288	14	–	610310	878639	410622	992463	335467	878639	424327

Table 4: Max. ROBDD Manager Size, Malik Ordering, 1M limit

Tables 5 and 6 report the total runtime for each example, while tables 7 and 8 report the time spent in the cost estimation routines. Without DR, SUP-SIZE and NUM-PSI are about as fast and are much faster than the other two schemes. This is primarily due to the difference in the time taken for cost estimation. In COMP-SIZE and NUM-PSI, we actually take the cofactors of G_d for estimating the cost. This is a time consuming process, requiring the allocation of new ROBDD nodes. With DR, the time taken by reordering is the dominating factor, hence none of the scheme performs consistently better in the presence of DR although the time taken for cost estimation is still more for COMP-SIZE and SUM-SIZE.

Ckt	**Out**	**COMP-SIZE**		**SUP-SIZE**		**SUM-SIZE**		**NUM-PSI**	
	#	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR
C880	26	2.28	2.20	2.00	2.18	2.30	2.22	2.09	2.12
C1908	24	13.36	10.10	6.73	11.13	13.10	10.24	6.64	11.28
C1908	25	12.94	4.25	12.22	4.26	12.98	4.13	12.75	4.26
C2670	140	–	882.08	–	345.09	–	2293.22	–	572.12
C3540	22	536.61	417.72	144.26	504.34	574.35	384.74	178.27	375.60
C6288	12	111.48	183.05	13.32	243.10	65.56	263.59	13.38	594.58
C6288	13	337.33	775.39	36.14	1088.12	99.00	1458.55	36.10	816.77
C6288	14	877.60	3849.23	89.32	3787.94	270.35	3313.25	94.41	2279.89

Table 5: Total Time (sec), Natural Ordering, 1M limit

Ckt	**Out**	**COMP-SIZE**		**SUP-SIZE**		**SUM-SIZE**		**NUM-PSI**	
	#	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR
C880	26	0.18	0.16	0.16	0.15	0.13	0.17	0.15	0.15
C1908	24	3.40	3.10	1.16	1.17	3.07	3.15	1.25	1.24
C1908	25	3.51	4.05	2.68	3.49	3.44	4.06	2.68	3.45
C2670	140	13.84	50.94	1.41	21.45	13.38	51.46	5.85	46.96
C3540	22	–	783.06	–	826.49	–	986.51	–	849.76
C6288	12	112.50	339.33	20.44	483.42	105.94	384.53	20.15	431.65
C6288	13	367.39	1054.64	48.05	861.05	418.49	557.62	50.90	961.91
C6288	14	–	4443.96	139.14	2099.95	1523.63	3331.66	132.74	2974.57

Table 6: Total Time (sec), Malik Ordering, 1M limit

Ckt	**Out**	**COMP-SIZE**		**SUP-SIZE**		**SUM-SIZE**		**NUM-PSI**	
	#	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR
C880	26	0.13	1.74	0.01	0.01	0.13	1.76	0.01	0.01
C1908	24	7.52	0.24	0.53	0.03	7.27	0.24	0.52	0.03
C1908	25	2.09	0.09	1.13	0.02	2.11	0.09	1.11	0.01
C2670	140	–	388.34	–	0.35	–	1738.51	–	0.59
C3540	22	451.69	178.36	21.84	4.33	492.72	173.28	35.72	4.90
C6288	12	101.53	134.71	0.88	0.93	54.35	217.77	0.84	1.75
C6288	13	312.96	296.42	2.76	4.35	70.06	106.09	2.64	3.92
C6288	14	830.12	2045.58	8.79	7.80	200.21	1951.14	8.42	13.72

Table 7: Time for cost estimation (sec), Natural Ordering, 1M limit

Ckt	Out	COMP-SIZE		SUP-SIZE		SUM-SIZE		NUM-PSI	
	#	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR	w/o DR	with DR
C880	26	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C1908	24	2.42	2.16	0.11	0.11	2.13	2.21	0.11	0.12
C1908	25	1.16	0.90	0.26	0.25	1.13	0.88	0.26	0.24
C2670	140	8.48	2.96	0.05	0.03	8.20	3.03	0.29	0.08
C3540	22	–	437.56	–	10.37	–	817.31	–	18.77
C6288	12	93.24	8.05	1.21	1.34	86.10	134.35	1.22	4.55
C6288	13	322.55	58.08	4.14	7.93	370.47	39.88	4.25	8.27
C6288	14	–	674.32	14.52	21.94	1358.28	324.22	15.33	23.55

Table 8: Time for cost estimation (sec), Malik Ordering, 1M limit

8 Conclusions

In this paper, we have focussed on the composition process of a mixed apply/compose based scheme for building ROBDDs. Given a set of decomposition points and the decomposed version of the 'target' ROBDD, we tried different heuristics to determine the order in which the decomposition points should be composed to avoid intermediate memory blow-up. The main conclusions of our work are as follows:

- We showed that the intermediate memory requirement is very sensitive to the order of composition. A bad order of composition may utilize up to twice the memory of a good order.
- We showed that SUP-SIZE and NUM-SIZE are reasonable heuristics for guiding the composition process for cases where the maximum manager size is of concern.
- In cases where the maximum ROBDD size is of importance, COMP-SIZE and SUM-SIZE often perform as well as or better than the other two methods when Natural Ordering is used.
- COMP-SIZE and SUM-SIZE require significantly larger cost estimation time, which sometimes offsets the composition time gains.
- Under Dynamic Variable Reordering, none of the schemes performs consistently better than the others in runtime. This is because Dynamic Variable Reordering is the dominating factor in the runtime.

References

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

[2] 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.

[3] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. *27th Design Automation Conference*, pages 40–45, 1990.

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

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

[6] 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.

[7] J Gergov and Ch. Meinel. Efficient analysis and manipulation of typed free bdds can be extended to read-once-only branching programs. *Tech. Report 92-10, University of Trier. Also to appear in IEEE Transaction on Computers, June 1995*, 1992.

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

[9] 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.

[10] Kevin Karplus. Using if–then–else dag's for multi–level minimization. *Decennial Caltech Conference on VLSI*, May 1989.

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

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

[13] J. C. Madre and J. P. Billon. Proving circuit correctness using formal comparison between expected and extracted behavior. *25th Design Automation Conference*, pages 205–210, 1988.

[14] 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.

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

[16] 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.

[17] 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.