Efficient Scheduling Techniques for ROBDD Construction

Rajeev Murgai   Jawahar Jain   Masahiro Fujita

Fujitsu Laboratories of America, Inc.,   Sunnyvale, CA
{murgai,jawahar,fujita}@fla.fujitsu.com

Abstract

The most common way to build the reduced ordered binary decision diagram (ROBDD) of a complex gate (or function) f of a network is bottom-up, i.e., by first building the ROBDDs of the sub-expressions of f and then suitably combining them. Such a method, however, has been found to suffer from memory explosion, even when the ROBDD of f is not large. This leads to the following fundamental question: *Given an arbitrary boolean expression $f(x_1, x_2, \ldots, x_n)$ and the ROBDDs of x_is (in terms of circuit inputs), how should the ROBDD of f be constructed so that the intermediate memory required to build the ROBDD is minimized, and a heavy time penalty is not incurred?* In this paper, we address this question for a restricted f: a multi-way AND or OR operation.[1] We propose various schemes for scheduling the binary operations of the expression f. These schemes are based on an analysis of the sizes and support-sets of the intermediate ROBDDs. One of our main contributions is to prove that under certain conditions, these schemes provide the optimum solution. We tested the proposed schemes on complex functions present within ISCAS85 as well as large industrial circuits. On average, our best scheme (which is based on size as well as support-set of the component ROBDDs) yields a 25% reduction in ROBDD sizes as compared to the technique implemented in sis [15]. In some cases, a reduction of up to 4 orders of magnitude was seen. Since ROBDDs are a key technology in various synthesis and verification tasks, our work can be of immediate use in all these applications.

1 Introduction

Reduced Ordered Binary Decision Diagrams (ROBDDs) [3] are frequently used in various combinatorial as well as CAD problems such as synthesis, digital-system verification, protocol validation and testing [4]. Unfortunately, ROBDDs suffer from the memory explosion problem – in many cases, they just require too much memory. There are two possibilities:
1. The final ROBDD is too large to fit in the main memory, or
2. the final ROBDD is reasonable, but some intermediate BDD is too large.

The first case can arise for functions, such as multipliers [3], where any ROBDD representation must require space which is exponential in the number of primary inputs (PIs). In such cases, a monolithic ROBDD representation is impractical and one may need to switch to other representations, for example, gBDDs [2], IBDDs [8], structural BDDs [11], or partitioned BDDs [9, 1], etc.

The second case can manifest itself in at least two ways, as discussed below. Suppose we want to build ROBDDs of the primary outputs of a circuit that consists of complex gates. This is usually done bottom-up – starting from the primary inputs, visit the gates in a topological order and build the ROBDD of a gate f, ROBDD(f), from the ROBDDs of its fanin gates, x_is. This is shown in Figure 1. The triangle rooted at x_i denotes ROBDD(x_i). Given a boolean expression for f (e.g., as a sum-of-products or a factored form), ROBDD(f) is constructed by repeatedly invoking the *apply* procedure [3] which carries out Boolean operations between different ROBDDs as required. It can happen that ROBDDs of the primary outputs have reasonable sizes, but ROBDD(f) does not. The following example illustrates this.

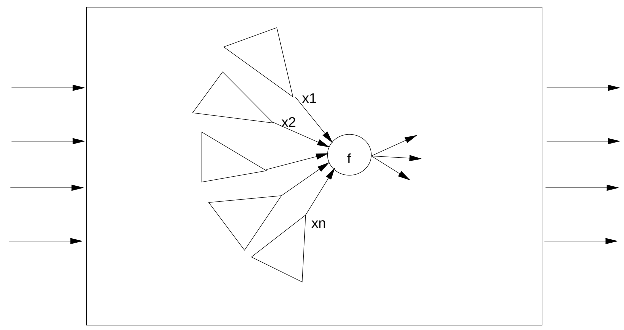

Figure 1: Building ROBDD of a complex gate f in a network

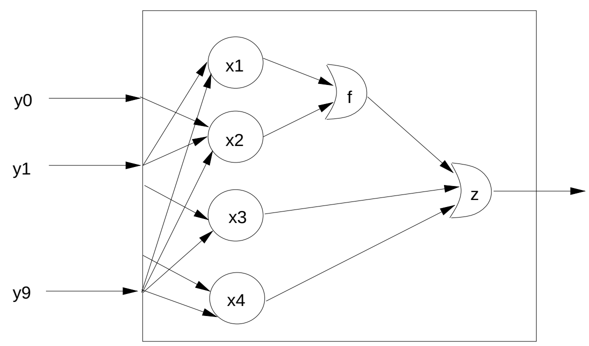

Figure 2: An example network

Example 1.1 *Consider the circuit of Figure 2. It has one primary output z, ten primary inputs y_0 through y_9, and six intermediate gates x_1, x_2, x_3, x_4, f, and z as follows:*

$x_1 = (\overline{y_1}\,\overline{y_7} + \overline{y_2}\,y_4)(y_6 y_8 y_9)$;
$x_2 = (y_0 y_1 y_2 + y_3 y_4)(y_5 y_6 y_8 y_9)$;
$x_3 = y_6 y_8 y_9$;
$x_4 = y_9\,\overline{y_8}$;
$f = x_1 + x_2$;
$z = f + x_3 + x_4$.

Suppose, we want to construct ROBDD(z) in terms of the primary inputs y_0 through y_9. Let us fix the ROBDD variable ordering

[1] One can build ROBDD of an arbitrary expression by using these two operations, along with inversion.

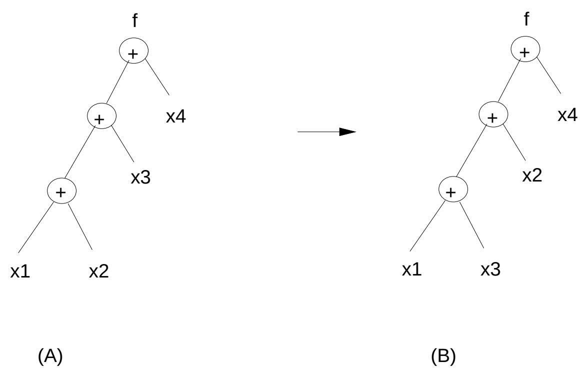

Figure 3: Rescheduling the BDD construction

to $y_0 \prec y_1 \prec \ldots \prec y_9$. It can be verified that under this variable ordering ROBDD(x_1) has 10 nodes, ROBDD(x_2) 9, ROBDD(x_3) 3, ROBDD(x_4) 2, ROBDD(f) 20, and ROBDD(z) 3.[2] *Note that actually $z = (\overline{y_8} + y_4)y_9$.*

This example showed that ROBDD for an intermediate gate (f) can be much larger than that for the primary output (z). The intermediate memory explosion problem can also arise *while* the ROBDD of a complex intermediate function f is being built. The last example, slightly modified, illustrates this phenomenon.

Example 1.2 *Suppose we have a 4-input OR gate f with inputs x_1, x_2, x_3, x_4 buried in a circuit whose primary inputs are y_0 through y_9. Let the functionality be as follows:*

$x_1 = (\overline{y_1}\,\overline{y_7} + \overline{y_2}\,y_4)(y_6 y_8 y_9)$;
$x_2 = (y_0 y_1 y_2 + y_3 y_4)(y_5 y_6 y_8 y_9)$;
$x_3 = y_6 y_8 y_9$;
$x_4 = y_9\,\overline{y_8}$;
$f = x_1 + x_2 + x_3 + x_4$.

Assume that for all x_is, ROBDD(x_i) have already been built. Now we wish to construct ROBDD(f). The variable ordering is the same as in Example 1.1. As mentioned earlier, an ROBDD is typically constructed by using binary operations, say using the apply *procedure. Since f is a 4-input OR function of x_is, the overall computation has to be split into binary computations. Consider the following order of computations: $(((x_1\ x_2)\ x_3)\ x_4)$, as shown in Figure 3 (A). This order implies that first ROBDD($x_1 + x_2$) should be built, which should then be ORed with ROBDD(x_3). Finally, ROBDD($x_1 + x_2 + x_3$) should be ORed with ROBDD(x_4) generating ROBDD(f). It can be checked that ROBDD($x_1 + x_2$) has 20 nodes. ROBDD($(x_1 + x_2) + x_3$) = ROBDD($y_6 y_8 y_9$) has 3 nodes. Finally, ROBDD(f) also has 3 nodes, since $f = (\overline{y_8} + y_4)y_9$. This is much smaller than the largest intermediate ROBDD – ROBDD($x_1 + x_2$), which has 20 nodes.*

In literature, various techniques have been proposed to address the memory explosion problem of ROBDDs.

1. **Variable reordering**: It is well-known that size of an ROBDD is sensitive to the variable ordering [3]. If an ROBDD becomes large, it may be possible to reduce its size by reordering variables. The main problem with this approach is that a lot of CPU time may be spent in reordering.
2. **Techniques based on decomposition**: Here the restriction of building a single monolithic representation for a function is relaxed. One could build either a set of **decomposed decision diagrams**, and then compose them back [10], or use alternate, space efficient, BDD representations such as **partitioned ROBDDs** [9, 1]. For instance, in Example 1.2, if after building ROBDD($x_1 + x_2$) (which is represented in terms of y_0 through y_9), one realizes that it is large (relatively speaking), one can introduce decomposition points w_1 and w_2 at x_1 and x_2 respectively. w_1 and w_2 correspond to new intermediate variables. One can then build ROBDD(f) in terms of w_1, w_2, and other primary inputs y_6, y_8, y_9, instead of y_0 through y_9. For instance, with the same schedule as above, ROBDD($x_1 + x_2$) is now built in terms of w_1 and w_2, i.e., ROBDD($x_1 + x_2$) = ROBDD($w_1 + w_2$) has two nodes. ROBDD($(x_1 + x_2) + x_3$) = ROBDD($w_1 + w_2 + y_6 y_8 y_9$) has 5 nodes (assuming w_1, w_2 are placed at the end of the variable ordering). Eventually, ROBDD(f) = ROBDD($w_1 + w_2 + y_9(y_6 + \overline{y_8})$) has 5 nodes. However, we also need ROBDD(w_1) = ROBDD(x_1) and ROBDD(w_2) = ROBDD(x_2). This yields a decomposed representation of f using three ROBDDs. Note that we avoided building ROBDD($x_1 + x_2$) – the 20-node BDD – in terms of primary inputs. Now, by composing BDDs of w_1, w_2 in f, we can obtain the required canonical OBDD of f in terms of primary input variables. Often such a decomposition/composition based approach can avoid intermediate peak explosion of BDD sizes [10]. Partitioned ROBDDs try to solve the problem of memory explosion by partitioning the truth table of f into disjoint parts and then building an ROBDD for each part separately. Different variable orderings can be chosen for different ROBDDs, further reducing the ROBDD sizes.

[2] We do not count the terminal 1 and 0 nodes in all cases.

In this paper, we attempt to reduce the intermediate memory explosion by **exploiting operator commutativity and associativity**.

Example 1.3 *In Example 1.2, consider an alternate evaluation schedule for f obtained by switching x_2 and x_3 in the schedule (A) of Figure 3. The new schedule is $(((x_1\ x_3)\ x_2)\ x_4)$, as shown in Figure 3 (B). $x_3 + x_1 = y_6 y_8 y_9$, whose ROBDD has 3 nodes. ROBDD($(x_3 + x_1) + x_2$) = ROBDD($y_6 y_8 y_9$) also has 3 nodes. Finally, ROBDD(f) has 3 nodes as well. The largest intermediate ROBDD in this case is that of $x_3 + x_1$, requiring only 3 nodes, as compared to 20 nodes for the first schedule. Even if we take into account sizes of the ROBDD(x_i), the largest ROBDD has 10 nodes (for x_1). Note that we used associativity and commutativity of the OR operation as the basis for coming up with the schedule. Also, in this example, the schedule was a simple chain: $(((x_1\ x_3)\ x_2)\ x_4)$. In general, it could be a tree such as $((x_2\ x_3)\ (x_1\ x_4))$.*

This example illustrates that by altering the order of computation, it is possible to reduce sizes of the intermediate ROBDDs. Such a possibility of being able to reduce the intermediate ROBDD sizes in *apply*-based BDD construction/manipulation is prevalent in almost all ROBDD-based applications. Upon inquiry we found that in several ROBDD-based packages available in academia as well as in industry, explosion in the intermediate sizes has not been adequately addressed. Thus, a reasonable solution to this problem will be a quite useful contribution.

We address the following problem in the paper:

Given a set of ROBDDs to be ANDed (ORed) two at a time, compute an appropriate schedule of the binary AND (OR) operations such that the maximum (or sum of) intermediate ROBDD size(s) is minimized. Further, determining the schedule itself should not cause a significant time penalty.

We propose various scheduling schemes and prove that under certain situations, they generate optimum schedules.

For an arbitrary intermediate function f in a network, we will apply our schemes as follows:

1. Starting with a sum-of-products representation of f, determine a good schedule for each product term,

2. use these schedules to build ROBDD for each product term, using *apply* for each binary AND,
3. determine a good schedule for the final sum term, and
4. build the ROBDD of f using the schedule for the sum term.

This is how ROBDDs are built for complex functions, for instance, in sis [15]. However, sis does not address the problem of generating good schedules for the AND and the OR operations to minimize the intermediate ROBDD sizes; it arbitrarily uses a chain-like schedule determined by the serial order in which literals appear in the input file of the design.

The paper is organized as follows. In Section 2, we briefly review related work on scheduling. In Section 3 we propose various scheduling schemes. Section 4 provides experimental results on various benchmarks. We provide concluding remarks in Section 5.

2 Related Work

[5] and [12] recognized ROBDD-scheduling as an important problem in the context of reachability computation using partitioned transition relations. They schedule ROBDDs based on common variables in their support sets. Given a set B of ROBDDs, $B = \{B_1, \ldots, B_k\}$, the members of B are arranged greedily such that each successive member has maximum support common with the preceding ROBDD. Some variants of this heuristic were discussed in [5]. As we will show in Section 4, such approaches can be improved.

We are also aware of the works by Shiple [16] and Hett *et al.* [6], which instead of multiple invocations of binary *apply*, use a *generalized multi-way apply* to build an ROBDD. In our work, however, we will focus on using *binary apply* as the building block, since it is the technology in almost all ROBDD packages. Also, some of these methods such as [16] suffer from enormous run-times and are not practical.

3 Proposed Scheduling Schemes

Assume $f = x_1 \odot x_2 \odot \ldots \odot x_n$, where $\odot$ = AND or OR. Let b_i = ROBDD(x_i), and $S = \{b_1, b_2, \ldots, b_n\}$. We wish to compute ROBDD(f) by $\odot$ing ROBDDs in S, two at a time.

We investigate various schemes for scheduling the $\odot$ operations. All the schemes are greedy in that at each step, they select two ROBDDs B_i and B_j from S such that the resulting ROBDD $B(i,j) = B_i \odot B_j$ is small. B_i and B_j are then deleted from S and $B(i,j)$ is added to S. This step is repeated until $|S| = 1$.

3.1 Size-based Analysis

Given a pair of ROBDDs B_i, B_j, and a Boolean operation $\odot$, it is well-known that the worst-case size of the resulting ROBDD $B_i \odot B_j$ is $O(|B_i||B_j|)$ [3]. Based on this observation, we propose the heuristic `min-size`, which selects two smallest ROBDDs B_i and B_j at each step, with the hope that the resulting ROBDD will be small as well.

As the following theorem shows, `min-size` can be optimum in certain situations. For this theorem, the optimum order is defined to be the one that minimizes the sum of the intermediate ROBDD sizes.

Theorem 3.1 *Given an initial set S of ROBDDs, $S = \{b_1, b_2, \ldots b_n\}$, such that any two ROBDDs B_i and B_j derived from ROBDDs in S by a sequence of $\odot$ operations obey*

$$|B_i \odot B_j| = |B_i| + |B_j|, \tag{1}$$

`min-size` *(i.e., $\odot$ing two minimum-sized ROBDDs at each step) yields an optimum schedule for computing f, the cost function being the sum of the sizes of the intermediate ROBDDs.*

Proof A schedule corresponds to a weighted binary tree, whose leaves are original ROBDDs $b_1, b_2, \ldots, b_n$, with weights $|b_1|, |b_2|, \ldots, |b_n|$ respectively. Performing $\odot$ operation on two ROBDDs B_i and B_j generates a tree node $V(i,j)$, which corresponds to the resulting ROBDD $B(i,j)$. The weight associated with $V(i,j)$ is $|B(i,j)| = |B_i| + |B_j|$ (from (1)), the sum of the sizes of the children ROBDDs. The sum of the intermediate ROBDD sizes is then the sum of the weights of the (non-leaf) tree nodes. By Huffman's Theorem [7], this sum is minimized in a tree which is obtained by combining two smallest-weight nodes at each step. This is the same as the strategy in `min-size`. ■

The theorem shows that the optimum schedule for *well-behaved* ROBDDs is to evaluate them in increasing size. Interestingly, if we change the optimality criterion from minimizing the sum of the intermediate ROBDD sizes to minimizing the maximum intermediate ROBDD size, the problem becomes NP-hard, even for well-behaved ROBDDs.

Theorem 3.2 *Given an initial set S of ROBDDs, $S = \{b_1, b_2, \ldots b_n\}$, such that any two ROBDDs B_i and B_j derived from ROBDDs in S by a sequence of $\odot$ operations obey*

$$|B_i \odot B_j| = |B_i| + |B_j|, \tag{2}$$

finding a schedule that minimizes the maximum intermediate ROBDD size is NP-hard.

Proof Note that size of an ROBDD at any node in the scheduling tree is sum of the weights $|b_i|$ of the leaf-nodes in th sub-tree rooted at that node. Given (2), the maximum intermediate ROBDD size would be that of one of the children c_1 or c_2 of the root of the complete tree. Since minimizing the larger of c_1 and c_2 is equivalent to minimizing the difference between $|\text{ROBDD}(c_1)|$ and $|\text{ROBDD}(c_2)|$, the problem becomes that of partitioning the weights b_is into two disjoint sets such that the difference in the sum of the weights of the two is minimized. This can be restated as the MINIMUM DIFFERENCE problem:

INSTANCE: *Finite set B, a weight $s(b) \in \mathcal{Z}^+$ for each $b \in B$, and K.*
QUESTION: *Is there a subset $\widetilde{B} \subseteq B$ such that*

$$|\sum_{b \in \widetilde{B}} s(b) - \sum_{b \in B - \widetilde{B}} s(b)| \leq K \tag{3}$$

We prove that MINIMUM DIFFERENCE is NP-complete by transforming the NP-complete problem PARTITION to it. Consider PARTITION:

INSTANCE: *Finite set B and a weight $s(b) \in \mathcal{Z}^+$ for each $b \in B$.*
QUESTION: *Is there a subset $\widetilde{B} \subseteq B$ such that*

$$\sum_{b \in \widetilde{B}} s(b) = \sum_{b \in B - \widetilde{B}} s(b) \tag{4}$$

Clearly, MINIMUM DIFFERENCE is NP-complete, since for $K = 0$, it reduces to PARTITION.

■

3.2 Support-based Analysis

Under certain situations, scheduling ROBDDs based only on sizes is not sufficient. It can happen that the two smallest ROBDDs B_i and B_j are such that $|B_i \odot B_j| = |B_i||B_j|$. Instead had we chosen ROBDDs B_ℓ and B_m that are slightly larger but have disjoint support sets, we could have obtained a much smaller intermediate ROBDD, with size $|B_\ell| + |B_m|$. The following theorem makes this precise. Let $sup(b)$ denotes the support set of ROBDD b.

Theorem 3.3 *[13] Given ROBDDs B_ℓ and B_m such that $sup(B_\ell) \cap sup(B_m) = \phi$. Also, assume that the variable ordering π is such that the first $|sup(B_\ell)|$ positions in π are occupied by the variables of B_ℓ. Then, ROBDD $B_\ell \odot B_m$ can be obtained by just appropriately concatenating the ROBDDs B_ℓ and B_m.*

This theorem underscores the importance of a support-based analysis. The simplest support-based heuristic, `min-support`, ignores sizes completely and at each step selects two ROBDDs that have minimum supports. This is similar to the support-based heuristics of [5, 12], in which the first ROBDD is the minimum-support ROBDD, and the second ROBDD is the one that introduces fewest extra variables after the operation is carried out. We call this scheme `support_extra-support`, since the first ROBDD (B_i) has minimum support and the second ROBDD (B_j) has, among all the remaining ROBDDs, the minimum *extra support* from B_i. Extra support is the number of additional variables introduced in the support of $B(i,j) = B_i \odot B_j$ as compared to B_i. It is equal to $|sup(B_j) - \text{sup}(B_i)|$.

3.3 Size- and Support-based Analysis

We now present a scenario in which both size and support-set are needed for an optimum schedule (here also, the optimum schedule is the one that minimizes the sum of the sizes of the intermediate ROBDDs). If a set of ROBDDs can be partitioned into subsets of disjoint support-set ROBDDs, and the ROBDDs within each subset obey a certain cost function during $\odot$ing, the following theorem states that the optimum schedule is to evaluate the ROBDDs within each subset using `min-size` and then apply `min-size` on the resulting disjoint support-set ROBDDs.

Theorem 3.4 *Consider a set of* ROBDD*s $S = \{b_1, b_2 \ldots, b_n\}$ such that*

1. *either $sup(b_i) = sup(b_j)$ or $sup(b_i) \cap sup(b_j) = \phi$ for all i, j.*
2. *whenever $sup(b_i) = sup(b_j)$,*
$$|b_i \odot b_j| = \min\{|b_i|, |b_j|\} \qquad (5)$$
Moreover, (5) holds for any two ROBDD*s derived from same-support* ROBDD*s of S.*

Let m be the total number of distinct ROBDD *support sets in S. So S can be partitioned into sets $S_1, S_2, \ldots, S_m$, where* ROBDD*s in a set S_i have identical supports and a* ROBDD *in S_i has disjoint support with any* ROBDD *in $S_j, j \neq i$. Let B_i be the* ROBDD *obtained after $\odot$ing all the* ROBDD*s in S_i. Then, given that for each i variables of S_i are contiguous in the global* ROBDD *variable ordering, an optimum schedule for computing $b_1 \odot b_2 \odot \ldots b_n$ is to evaluate each S_i using* `min-size` *to get B_i, and then apply* `min-size` *on $\{B_1, B_2, \ldots, B_m\}$.*

Proof Omitted due to lack of space. Please see [14]. ■

This theorem used both size and support-set information to derive the optimum schedule. The natural step therefore is to propose a scheme that combines both size and support-set information. In fact, we propose various such schemes. The first ROBDD B_i is always the minimum-sized ROBDD.

1. `size_common-support`: The second ROBDD B_j is the one that shares maximum support with B_i.
2. `size_extra-support`: B_j is the one that has minimum extra support with respect to B_i.
3. `size_support`: To decide on B_j, we rank separately the remaining ROBDDs of the set S by size and extra support in L_{size} and L_{sup}. ROBDDs with minimum rank (size or extra support) are earlier in the lists. Then, we pick a very small number of ROBDDs (such as 2) from the head of L_{size} and L_{sup}. We perform an explicit AND operation of each of these ROBDDs with B_i. The ROBDD that results in the smallest size is the desired B_j.

3.4 Partial Traversal

Given ROBDDs B_i and B_j, this method estimates the size $W(i,j)$ of the resulting ROBDD $B(i,j)$ by exploring each ROBDD partially. For B_i, all possible paths p of length up to k (where k is a small constant) starting at the root of B_i are traversed. Assume that a path p ends at vertex v_i. v_i may be terminal (0 or 1) or non-terminal. B_j is also traversed for this path p. Let the vertex reached at the end of path p in B_j be v_j (a terminal vertex may be reached before p is completely traversed, in which case we stop at the terminal vertex). This corresponds to taking cofactor of B_j with respect to the cube corresponding to p. Let $|v_i|$ denote the size of the ROBDD rooted at v_i. We initialize $W(i,j) = 0$. We have the following cases:

- $v_i = 0$: do nothing.
- $v_i = 1$: if v_j is non-terminal, $W(i,j) = W(i,j) + |v_j|$.
- otherwise (v_i is non-terminal): if v_j is non-zero, $W(i,j) = W(i,j) + |v_i| \cdot |v_j|$.

We repeat this analysis for all paths p in B_i of length at most k.

This analysis is done for all pairs (B_i, B_j) and $W(i,j)$ is computed. Finally, choose B_i and B_j such that $W(i,j)$ is minimized.

The `partial-traversal` heuristic attempts to do a more accurate size estimation than is possible with `min-size`. For instance, if a path p in B_i ends at $v_i = 0$, the corresponding vertex in the resulting ROBDD $B(i,j)$ will be 0, irrespective of the kind of vertex v_j in B_j. Thus, the contribution of such a path to the size is 0. If there are many paths in the top part of B_i or B_j ending in 0, $B(i,j)$ will be small. `partial-traversal` examines the relationship between corresponding paths in B_i and B_j; such a functional analysis is not possible in `min-size`. Interestingly, `partial-traversal` reduces to `min-size` for $k = 0$.

However, `partial-traversal` has the following drawbacks. First, it does not consider size of $B(i,j)$ for the first k levels. $W(i,j)$ is an estimate of $B(i,j)$ below k levels. This is reasonable only if k is small. Secondly, when computing $W(i,j)$, `partial-traversal` does not take into account possible node sharing between ROBDDs rooted at v_i and v_j. Thus, it overestimates the size of $B(i,j)$ (below k levels). Also, if the number of ROBDDs to be ANDed, n, is large, such an analysis can be computationally expensive (the complexity is $O(n^2 2^k)$). So, in our implementation, at each step, we fix B_i to be the minimum-sized ROBDD. B_j is then determined by carrying out the foregoing analysis for the remaining ROBDDs. This reduces the run-time by about a factor of 2.

4 Experimental Results

In the following we analyze the experimental performance of our scheduling algorithm, and prove that they can indeed make significant improvements to the state of the art methods.

Experimental Setup: The proposed scheduling computation algorithms of Section 3 have been implemented within sis [15] environment. Our test circuits include ISCAS85 combinational benchmark circuits as well as the combinational parts of various designs from Fujitsu such as data transfer buffers and a controller for a parallel processor. Our experiments were carried out on a Sun SPARC 20 with 512 MBytes of RAM and more than 2GB swap space. The run-times are reported in seconds. The goal is to build the ROBDDs for these circuits. Each node of the circuit can have arbitrary logic function associated with it. The ROBDDs are built topologically from inputs to outputs. Therefore, at any node, the ROBDDs for the fanins of the node have already been built. Each benchmark is pre-processed such that each node is either an AND or an OR, with unbounded number of inputs. Note, if we use dynamic reordering then the final ROBDD need not have the same variable order as all the intermediate BDDs. Since time required in reordering is directly proportional to the size of graphs, thus for simplicity as well as for giving only a conservative estimate of the benefits of our techniques, we will ignore the effect of reordering the intermediate graphs.

Description of Tables: Table 1 lists sum of the sizes of the intermediate ROBDDs generated by various scheduling schemes for interesting nodes of each benchmark. We say a node is interesting if its ROBDD has at least 1000 nodes.[3] The sum of the intermediate ROBDD sizes is a useful metric since it captures different intermediate ROBDD sizes in a single number, and is also a measure of the total time taken in building the final ROBDD. Table 2 lists the maximum ROBDD size during scheduling and compares it with the final ROBDD size for the node.

The algorithm currently implemented in sis to build the network ROBDD, to be called sis from now on, constructs the product and sum ROBDDs in a serial order, determined by how the literals appear in the input file. We do not report the results for `size_common-support`, since they were not much different from `size_extra-support`. For `partial-traversal`, k was set to 5. Also, heuristic names have been shortened in the tables. For instance, `size-esup` is the same as `size_extra-support`.

Each row in the table corresponds to an interesting node of the benchmark, and contains the following information: name of the benchmark, the type of node – AND or OR, the number of immediate fanins, and the sum of the sizes of intermediate ROBDDs (including the final ROBDD for the node) for various scheduling schemes. In each row, the minimum sum is highlighted in bold. The row *total* summarizes the performance of each scheme vis-a-vis sis. It shows average relative sum of the intermediate ROBDD sizes for each scheme with respect to that of sis.

Analysis of Table 1: It can be safely concluded that `size_support` is the best heuristic; it is about 25% better than sis in terms of the ROBDD size sum. Also, it gives the minimum sum 29 times out of 35. For the 4^{th} node of C3540 (a 4-input AND gate), `size_support` is more economical than all other scheduling schemes by a factor of two. `min-size` is the next best – it is about 20% better than sis and gives minimum sum 15 times. Disappointingly, the most elegant scheme, `partial-traversal`, does not live up to its expectations. Also, sis is the worst of them all. That is as expected, since sis processes literals in the product and sum terms in the same order as typed in the input blif file, and this order may not be good at all.

Analysis of Table 2: In comparing the maximum ROBDD size during scheduling with the final ROBDD size for the node, we again find that `size_support` offers the best performance. In some examples, such as the second node of `C1355` and the last node of `C432`, the intermediate ROBDD size in sis is almost twice the final size, whereas other schemes avoid this *explosion*.

We noticed that on certain logic functions in the industrial benchmark `ut` (Figure 4), `size_support` and other scheduling schemes yield an improvement of as much as 3 to 4 orders of magnitude over the sis heuristic in the intermediate size. Since `support_extra-support` is essentially the technique proposed by [5, 12], our results demonstrate that it is possible to do much better than a purely support-based scheme. In general, support-based heuristics `min-support` and `support_extra-support` do not perform well. Although there are instances where they outperform others, there are far more instances where they end up at the bottom. In fact, as we can notice in Figure 5, which lists intermediate sizes of the ROBDDs for each scheduling scheme for two interesting cases of circuit mswcn, such schemes can be impractical for many real-life circuits.

Run-time Performance: The run-times of various heuristics are given in Table 3. sis takes long time to create ROBDDs, since the intermediate sizes can be huge, and operating on them can be costly. Clearly, `size_extra-support` is the fastest, closely followed by `min-size`. `partial-traversal` is the slowest, which is understandable, since it is traversing the ROBDDs up to k levels. It is quite slow on C5315 and C2670, but on the rest, takes time comparable to sis. `size_support` is overall a little faster than sis, but is about twice as slow as `min-support` and `min-size`. However, since in terms of solution quality `size_support` is much better than other schemes, the hit in run-time should be acceptable.

[3] This filter reduces the amount of information analyzed.

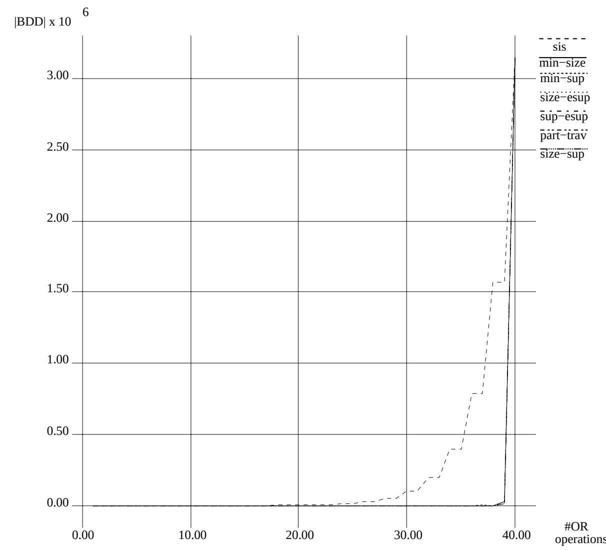

Figure 4: Interesting case in `ut`: a 41-input OR gate. Note that except sis, most scheduling algorithms had an identical performance.

5 Conclusions

In this paper, we addressed the following problem: given a multi-input AND (also OR) function $f(x_1, x_2, \ldots, x_n)$ and ROBDDs of x_is (in terms of circuit inputs), construct ROBDD of f using binary AND (or OR) as the basic operation such that the intermediate memory requirements are minimized and a heavy time penalty is not incurred. We proposed various schemes for scheduling the binary operations. The best scheme was found to be the one based on an analysis of both sizes and support-sets of the intermediate ROBDDs. One of our main contributions was to prove that under certain conditions, these schemes provide the optimum solution. We proposed a simple method to use these schemes for building ROBDDs of arbitrary functions. We also presented experimental results for complex functions present within ISCAS85 as well as large industrial circuits. On average, a 25% reduction in ROBDD sizes was obtained over the technique implemented in sis [15]. In some cases, a reduction of up to 4 orders of magnitude was seen. Our future work on scheduling can address two interesting problems.

1. Our schemes can also be applied to the general network scheduling problem. To build the ROBDDs for the outputs of a network, ROBDDs of the intermediate gates need to be built. In what order should these gates be traversed? Although many topological orderings are possible, we would like to pick one that minimizes intermediate memory. An important related problem is that of *restructuring the network* so that intermediate memory can be minimized. For instance, if the network contains a sub-network consisting only of 2-input AND gates, the sub-network can be collapsed into one single

Bench	type	inputs	sis	min-size	min-sup	size-esup	sup-esup	part-trav	size-sup
C5315	OR	6	14549	14549	**11096**	14700	**11096**	14549	14549
C5315	OR	4	15655	**6054**	**6054**	**6054**	**6054**	**6054**	**6054**
C5315	OR	16	83987	44204	44543	**44156**	**44156**	44192	44172
C3540	AND	5	16363	**7767**	**7767**	**7767**	**7767**	7842	**7767**
C3540	AND	5	11233	11233	9947	9947	9947	11233	**9799**
C3540	OR	15	130149	57373	57343	53877	57476	57623	**48792**
C3540	AND	4	3091	3091	2846	3329	2846	3091	**1501**
C3540	AND	3	**5113**	**5113**	6195	6655	6195	**5113**	**5113**
C3540	AND	5	**4187**	**4187**	4657	4218	4381	**4187**	4218
C3540	AND	4	3623	3623	3482	**3130**	3482	3623	**3130**
C3540	OR	9	9143	7693	9536	7497	7565	9143	**7186**
C3540	AND	4	28946	**25319**	27421	26637	26637	28946	**25319**
C3540	OR	6	49354	**40158**	51056	40840	45639	49354	**40158**
C3540	OR	7	47067	34565	40148	35837	35300	47067	**34418**
C3540	AND	3	19029	**16623**	19029	**16623**	**16623**	19029	**16623**
C3540	OR	5	37770	27462	33288	27462	34934	37770	**27351**
C3540	OR	8	60677	**46606**	82700	47430	65008	65100	**46606**
C3540	AND	3	**3137**	**3137**	5170	5170	5170	**3137**	**3137**
C2670	OR	16	17698341	**6817897**	8584118	7936566	8584118	8410650	6843633
C1908	OR	5	6249	**2715**	**2715**	2927	2990	3418	**2715**
C1908	AND	10	9568	9568	7612	7612	7612	9568	**6225**
C1908	OR	4	5367	**3863**	**3863**	5367	5367	**3863**	**3863**
C1355	OR	3	6264	5034	5034	5034	5034	5034	**5014**
C1355	OR	3	7446	5018	5018	5018	5018	5018	**4996**
C432	OR	13	24119	18329	21725	21442	19219	24119	**16333**
C432	OR	13	10382	10357	11602	**10317**	11571	10382	10331
C432	AND	9	7408	7518	12262	8190	12142	7408	**7368**
C432	OR	11	49331	55298	50740	45190	45305	49331	**42906**
C432	OR	14	76664	79212	110630	86751	110630	76664	**67653**
C432	AND	4	**8726**	**8726**	13547	10958	13547	**8726**	**8726**
C432	OR	7	42541	15149	15149	15149	15149	13921	**8511**
mswcn	AND	4	**786469**	**786469**	1048538	1048538	1048538	**786469**	**786469**
mswcn	AND	4	**2097198**	**2097198**	3014593	3014593	3014593	**2097198**	**2097198**
mswcn	AND	4	5005951	5005951	**4842035**	**4842035**	**4842035**	5005951	5005951
ut	OR	41	9437157	3173571	3173749	3159986	3159986	3173485	**3159568**
total			100.0	80.3	92.8	86.3	90.5	85.1	74.8

Table 1: Comparison of various schemes – sum of ROBDD sizes

Bench	type	final size	sis	min-size	min-sup	size-esup	sup-esup	part-trav	size-sup
C5315	OR	4964	7718	7718	4964	7882	4964	7718	7718
C5315	OR	5693	5693	5693	5693	5693	5693	5693	5693
C5315	OR	5697	5697	5697	5697	5697	5697	5697	5697
C3540	AND	6180	6180	6180	6180	6180	6180	6180	6180
C3540	AND	3523	3825	3825	3671	3671	3671	3825	3523
C3540	OR	10349	10349	10349	10349	10349	10349	10349	10349
C3540	AND	1055	1748	1748	1055	1344	1055	1748	1055
C3540	AND	3817	3817	3817	3817	3817	3817	3817	3817
C3540	AND	1258	1810	1810	1307	1396	1258	1810	1396
C3540	AND	1556	1810	1810	1556	1556	1556	1810	1556
C3540	OR	1223	2060	1223	2170	1272	1374	2060	1223
C3540	AND	8621	11512	8813	11512	10131	10131	11512	8813
C3540	OR	9600	10930	9600	14431	10210	14311	10930	9600
C3540	OR	11313	11570	11570	16519	12107	11570	11570	11423
C3540	AND	8636	10393	8636	10393	8636	8636	10393	8636
C3540	OR	8612	10921	8612	11784	8612	15963	10921	8612
C3540	OR	13886	13886	13886	23340	13886	18314	17779	13886
C3540	AND	2374	2374	2374	2796	2796	2796	2374	2374
C2670	OR	2346311	2346311	2346311	2346311	2346311	2346311	2346311	2346311
C1908	OR	1201	1916	1201	1201	1397	1460	1244	1201
C1908	AND	2013	2013	2013	2013	2013	2013	2013	2013
C1908	OR	1922	3425	1922	1922	3425	3425	1922	1922
C1355	OR	2507	3757	2527	2527	2527	2527	2527	2507
C1355	OR	2506	4940	2512	2512	2512	2512	2512	2506
C432	OR	3600	3848	3600	3600	3600	3600	3848	3600
C432	OR	2979	2979	2979	3606	2979	3606	2979	2979
C432	AND	2139	2139	2139	3928	2139	3621	2139	2139
C432	OR	7688	7688	9837	7688	7688	7688	7688	7688
C432	OR	7349	9389	10060	10120	10308	10120	9389	7811
C432	AND	3098	3406	3406	6327	4122	6327	3406	3406
C432	OR	4147	8809	4147	4147	4147	4147	4147	4147
mswcn	AND	786391	786391	786391	786391	786391	786391	786391	786391
mswcn	AND	2097092	2097092	2097092	2097092	2097092	2097092	2097092	2097092
mswcn	AND	3957343	3957343	3957343	3957343	3957343	3957343	3957343	3957343
ut	OR	3145727	3145727	3145728	3145728	3145728	3145728	3145728	3145728

Table 2: Comparison of various schemes – max ROBDD size

Benchmark	sis	min-size	size-esup	sup-esup	min-sup	part-trav	size-sup
C5315	53.16	16.06	13.18	21.17	17.22	265.5	35.89
C3540	155.05	100.59	94.17	107.65	130.07	120.32	172.61
C2670	823.62	453.73	355.44	609.16	599.83	2337.46	899.25
C1908	76.26	8.42	6.74	11.77	8.74	59.64	13.89
C1355	34.07	12.97	10.67	19.58	11.74	58.26	16.73
C432	61.56	24.19	19.73	43.75	31.69	47.46	61.65

Table 3: Run-time comparison of various heuristics

```
++++++++++++++++++++++++++++++++++++++++++++++++
/* min-sup, sup-esup, size-esup are the worst */
sis       AND   12 94 2097092
min-size  AND   12 94 2097092
min-sup   AND   12 917489 2097092
size-esup AND   12 917489 2097092
sup-esup  AND   12 917489 2097092
part-trav AND   12 94 2097092
size-sup  AND   12 94 2097092
++++++++++++++++++++++++++++++++++++++++++++++++
/* min-sup, sup-esup, size-esup are the worst */
sis       AND   77 1048531 3957343
min-size  AND   77 1048531 3957343
min-sup   AND   360444 524248 3957343
size-esup AND   360444 524248 3957343
sup-esup  AND   360444 524248 3957343
part-trav AND   77 1048531 3957343
size-sup  AND   77 1048531 3957343
++++++++++++++++++++++++++++++++++++++++++++++++
```

Figure 5: Interesting cases for `mswcn`

multi-input AND gate, which can then be scheduled using our schemes.

2. Scheduling is a central issue in computing the order in which partitioned transition relations should be ANDed during reachability analysis of a finite state machine. Previously, researchers have used support-set based analysis [5, 12]. Since our scheduling algorithm is superior than support-set based analysis, we can apply our techniques to schedule partitioned transition relations.

References

[1] A. Narayan, S. P. Khatri, J. Jain, M. Fujita, R. K. Brayton, and A. Sangiovanni-Vincentelli. A Study of Composition Schemes for Mixed Apply/Compose Based Construction of ROBDDs. In *Proc. of the Intl. Conf. on VLSI Design*, 1996.

[2] P. Ashar, S. Devadas, and A. Ghosh. Boolean Satisfiability and Equivalence Checking Using General Binary Decision Diagrams. In *Proceedings of the International Conference on Computer Design*, pages 259–264, October 1991.

[3] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[4] R. Bryant. Symbolic Boolean Manipulation With Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24::293–318, September 1992.

[5] D. Geist *et al.* Efficient Model Checking by Automated Ordering of Transition Relation Partitions. In *CAV*, 1994.

[6] A. Hett, R. Drechsler, and B. Becker. MORE: An Alternative Implementation of BDD Packages by Multi-Operand Synthesis. In *Proceedings of the European Design Automation Conference*, pages 164–169, 1996.

[7] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proceedings of the IRE*, volume 40, pages 1098–1101, September 1952.

[8] J. Jain, M. Abadir, J. Bitner, D. S. Fussell, and J. A. Abraham. Indexed BDDs: Algorithmic advances in techniques to represent and verify Boolean functions. *IEEE Trans. Comp.*, November 1997.

[9] J. Jain, J. Bitner, D. Fussell, and J. Abraham. Functional Partitioning for Verification and Related Problems. In *Proceedings of the Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*, pages 210–226, March 1992.

[10] J. Jain, A. Narayan, C. Coelho, S. Khatri, A. Sangiovanni-Vincentelli, R. Brayton, and M. Fujita. Decomposition Techniques for Efficient ROBDD Construction. In *Formal Methods in CAD 96*, LNCS. Springer-Verlag, 1996.

[11] S-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi. Extended BDDs: Trading Off Canonicity for Structure in Verification Algorithms. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 464–467, November 1991.

[12] S. Krishnan and R. Hojati. Early Quantification and Partitioned Transition Relations. In *Proceedings of the International Conference on Computer Design*, 1996.

[13] 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.

[14] R. Murgai, J. Jain, and M. Fujita. Efficient Scheduling Techniques for ROBDD Construction. In *Fujitsu Labs of America Internal Report*, 1997.

[15] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. *SIS: A System for Sequential Circuit Synthesis.* Memorandum No. UCB/ERL M92/41, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[16] T. Shiple, B. Brayton, and A.S. Vincentelli. Computing Boolean Expressions with OBDDs. In *UCB/ERL M93/84 Internal Report, University of California, Berkeley*, 1993.