BDS: A BDD-Based Logic Optimization System

Congguang Yang Maciej Ciesielski
Dept. of Electrical & Computer Engineering
University of Massachusetts, Amherst, MA
cyang,ciesiel@ecs.umass.edu

Vigyan Singhal
Tempus Fugit, Inc.
Albany, CA 94706
vigyan@home.com

Abstract
This paper describes a new BDD-based logic optimization system, *BDS*. It is based on a recently developed theory for BDD-based logic decomposition, which supports both algebraic and Boolean factorization. New techniques, which are crucial to the manipulation of BDDs in a partitioned Boolean network environment, are described in detail. The experimental results show that *BDS* has a capability to handle very large circuits. It offers a superior runtime advantage over SIS, with comparable results in terms of circuit area and often improved delay.

1. INTRODUCTION

Through the continuously intensive research and development in logic synthesis area for the last twenty years, the general framework for logic synthesis has been well established. While the space for further improvement of the synthesis flow seems to be limited, there is still potential for significant improvement in many procedures in the synthesis process [1]. This is especially true when more efficient ways to represent Boolean functions become available.

The history of logic synthesis demonstrates a simple, yet clear fact that the representation of Boolean logic plays a central role in the evolution of synthesis methods. It seems natural that logic synthesis methods will continue evolving with the advent of newer and more efficient Boolean logic representations. We believe that this evolution will be further enabled by the accumulation of expertise in Binary Decision Diagrams (BDDs). Our research is trying to address this new opportunity.

BDDs were first proposed by Akers [2], and popularized by Bryant [3] and Brace *et al* [4]. Due to their *implicit* power to represent Boolean functions, BDD's are considered the most efficient Boolean representation known so far. The synthesis approaches based on decomposition and manipulation of BDD's have been found promising in FPGA synthesis [5, 6, 7], PTL synthesis [8, 9, 10], multi-level synthesis [11, 12, 13, 14, 15], and mixed CMOS/PTL synthesis [16, 17]. All these BDD-based synthesis methods have a great potential to outperform traditional logic synthesis approaches.

In [18] we proposed a new BDD-based logic decomposition method based on *dominators*. Our work was based on the observation that the *structure* of a properly ordered BDD reveals important information about its *functional* decomposition. Different structures, called *dominators* can be identified in a BDD, leading to efficient AND, OR, XOR and MUX decompositions. This method offers a capability to efficiently perform both algebraic and Boolean factorizations. In terms of the area and runtime, the synthesis results obtained with this method for AND/OR-intensive functions are comparable to that of SIS, while results for XOR-intensive functions are significantly better than SIS. In both cases, our program demonstrates great runtime advantage over SIS. However, the method is limited to the decomposition of monolithic (global) BDDs, and as such it can only be used to optimize circuits for which BDDs can be constructed.

In this paper, a complete BDD-based logic optimization system, *BDS*, designed to handle arbitrarily large circuits, is described. While retaining the BDD decomposition capability of [18], a more general framework which incorporates a typical logic synthesis procedures has been implemented. Several techniques which proved very efficient in manipulating BDDs in the partitioned Boolean network environment are discussed in detail.

2. SYNTHESIS FLOW

A very important feature of a logic synthesis system is its scalability. The scalability requires that the size of the *representation* of a *problem* be proportional to the size of the problem itself. In our case, the size of a BDD should be proportional to the size of a circuit (which is commonly measured by the number of gates). However, the size of a global BDD for a given Boolean network is completely unpredictable. It strongly depends on the type of the circuit, and not necessarily on the total number of gates. Therefore, a proper partitioning of Boolean network is required prior to performing the BDD decomposition. Fortunately, the same problem appeared in traditional logic synthesis, where it has been handled properly.

Current multi-level logic synthesis flow exemplified by SIS has drawn from twenty years of intensive research. We believe it has the capability to handle very large circuits and it does grasp the essence of logic synthesis in general. Therefore, *BDS* adopts the general synthesis flow of SIS. Fig. 1 compares the synthesis flow of SIS and BDS. The similarity between the two systems is obvious. The fundamental difference between SIS and BDS is the way in which they represent Boolean nodes and carry out all individual synthesis procedures. In *BDS*, after a Boolean network has been built all Boolean nodes are represented as *local* BDDs (using only immediate fanins for each node). All the subsequent procedures are carried out based on the local BDDs.

It should be mentioned that all procedures in the synthesis flow are heavily influenced by the underlying Boolean representation. Therefore, while retaining a similar synthesis flow, new algorithms specially tailored for BDDs should be developed for all the procedures. In the following section, the BDD decomposition techniques introduced in [18] are briefly reviewed. These techniques are central to *BDS*. Other essential procedures in the synthesis flow are presented next.

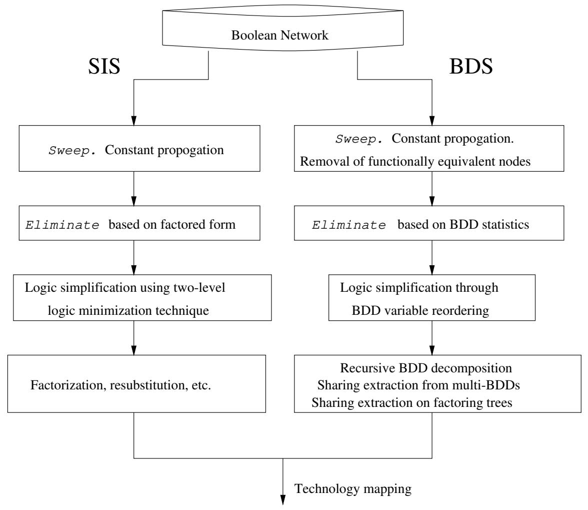

Figure 1: Synthesis flow of SIS and BDS

3. REVIEW OF BDD DECOMPOSITION

The BDD-based logic decomposition techniques used by *BDS* are summarized in Table 1. Detailed explanation of the terminology and examples can be found in [18]. All types of atomic decompositions and their corresponding BDD structures (*dominators*) have been identified. These dominators form the central BDD decomposition engine of *BDS*. The decomposition begins with the BDD structure scan. The BDD structural information obtained by the scan is used to guide the various decomposition types.

Type	BDD Structure	Decomposition
1	*1-dominator*	algebraic AND
2	*0-dominator*	algebraic OR
3	*x-dominator*	algebraic XNOR
4	*generalized dominator*	Boolean AND/OR
5	*generalized x-dominator*	Boolean XNOR
6	*cof. w.r.t. single node*	simple MUX
7	*cof. w.r.t. super node*	complex MUX

Table 1: Various *dominators* used in BDD decomposition

The BDD decomposition process is recorded by constructing a set of *factoring trees* for each function. A factoring tree will keep growing until the BDD decomposition is completed. Subsequently, several steps can be applied to the factoring trees to further improve the decomposition results. In particular, sharing between different factoring trees can be efficiently detected. For this purpose, BDDs are constructed for all factoring trees in a bottom-up fashion, and canonicity property of a BDD is used to identify functionally equivalent sub-trees.

3.1 BDD Decomposition - Review

The following example, for function $F = e + bd$, shown in Fig. 2(a), will illustrate the idea of Boolean AND/OR decomposition based on a *general dominator*. First, a cut is performed on the BDD of F. The generalized dominator is the portion of the BDD above the cut. The edges of the generalized dominator that that are connected to terminal nodes in the original BDD remain connected to those terminals, while the remaining edges (in this case a 1-edge incident to node d) are redirected to constant **1**. This results in a Boolean divisor $D = e + d$. The quotient Q is obtained from F by minimizing F using the off-set $\{e'd'\}$ of D as a don't care. This leads to $Q = e + b$. The final decomposition is $F = (e + d)(e + b)$. It can be shown that all cuts form a set of compatible classes, each class leading to a unique decomposition (division). This feature drastically reduces the number of cuts to be considered.

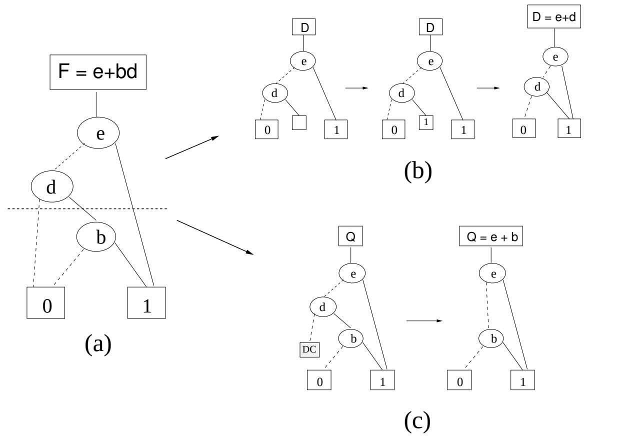

Figure 2: A simple example of Boolean division.

The following example illustrates additional BDD decomposition capabilities of our tool, namely the XOR and MUX decompositions. Given an initial global BDD representation of a two-output function, $\{g, h\}$, the BDD for each primary output is decomposed independently of each other, and the factoring trees constructed for each function. Finally, logic sharing is identified in the resulting factoring trees. Fig. 3 shows the decomposition of BDD g. First, an *x-dominator* of g is found, leading to an algebraic XNOR decomposition. *X-dominator* is a BDD structure which contains a node (in our example node d) with the property that all paths from root to terminal nodes **1** and **0** pass through that node. In this case F can be decomposed into $F = u\overline{\oplus}f$, where f is a function rooted at d, and u is obtained from the original BDD by replacing f with 1. A factoring tree node, with operator XNOR, is created to record this decomposition.

Fig. 4 shows the decomposition of function h based on the extraction of complex MUX. Such a MUX decomposition exists if the BDD contains two nodes that cover all paths from

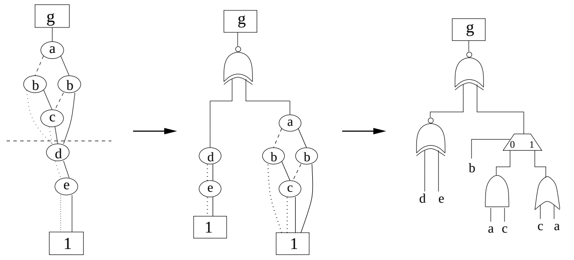

Figure 3: Decomposition of BDD g.

root to terminal nodes. In our case these are the d nodes below the cut. The function can then be decomposed into $F = v'(de) + v(d+e)$, where v is derived from the portion of the BDD above the cut, by redirecting the two d nodes to 0 and 1, respectively. Additional MUX decomposition is possible for function v. A factoring tree node, with operator MUX, is created to record this decomposition.

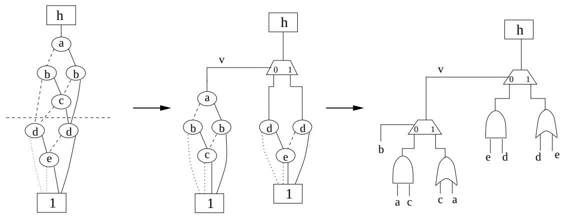

Figure 4: Decomposition of BDD h.

Finally, logic sharing between functions g and h is efficiently identified by constructing BDDs in a bottom-up fashion on the factoring trees for both functions. Fig. 5 shows the process of extracting the shared logic. The shadowed subtrees of g and h are functionally equivalent. This equivalence can be identified by constructing BDDs on both factoring trees.

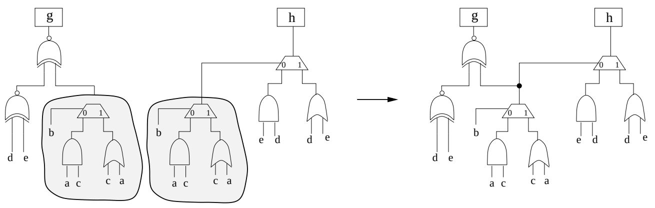

Figure 5: Sharing extraction on the factoring trees of g and h.

3.2 BDD Store/Load Mechanism

In our BDD-based logic optimization scheme [18], BDD variable reordering algorithm serves as an implicit logic simplification. It should be emphasized that variables are reordered with respect to a BDD manager, not w.r.t. a specific BDD. Hence, if there is more than one BDD in the manager, variable reordering may not result in the desired simplification. In order to achieve maximum logic simplification of a Boolean function (BDD), all other BDDs must be freed from this BDD manager before performing variable reordering. However, those "freed" BDDs must be present in the BDD manager when they are needed for decomposition at a later time. Therefore, an efficient store/load mechanism must be developed.

A naive way to store BDDs is to represent them in a SOP form. The advantage of SOP form is that the BDD can be reconstructed under a variable order which is different from the order when the BDD is stored. This offers some flexibility for the implementation. However, since the number of SOP terms of a BDD could be exponential in the number of BDD nodes, storing BDDs in SOP form is not a feasible solution.

A new data structure, `bddPool`, has been devised to perform BDD store/load operations. Basically, a `bddPool` is a DAG which is graphically isomorphic to the BDD it represents. A BDD is copied to a `bddPool` before it is freed from the BDD manager. The BDD can be reconstructed later by applying an *ite* operation $|V|$ times, where $|V|$ is the number of BDD nodes. Since an *ite* operation takes constant time, the overall complexity of our BDD store/load algorithm is $O(|V|)$. The disadvantage of `bddPool` is that the variable order of the BDD manager into which a BDD is loaded must be the same as the order in which a BDD is stored. Forcing a BDD manager to a certain variable order could result in an exponential increase in the BDD size if the manager is not empty. However, in our application, when a BDD is loaded, the BDD manager is always empty. A complete process of store/load is shown in Fig 6.

Another important feature of our `bddPool` mechanism is to allow the variable substitution during the process of BDD reconstruction. This can be accomplished easily by modifying the *ite* operator as $f = ite(\mathcal{M}(x), g, h)$, while $\mathcal{M}$ is the mapping of variables. This feature plays a crucial role in our efficient *iterative eliminate paradigm* (Section 4.2).

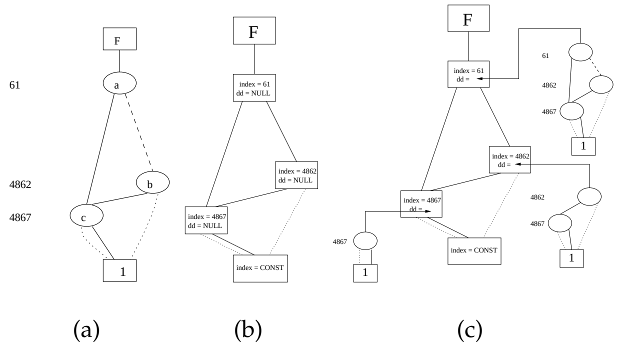

Figure 6: BDD store/load mechanism. (a) A BDD. (b) BDD stored in `bddPool`. (c) Re-construction of BDD from `bddPool` in a bottom-up fashion.

4. IMPLEMENTATION ISSUES

4.1 Sweeping Boolean Networks

Procedure `sweep` is the first step in the proposed synthesis flow. It removes some obvious redundancy from the Boolean networks. There is no real logic optimization involved in this procedure. However, for some multi-level Boolean functions, `sweep` plays a very important role in removing redundancy from the networks.

4.1.1 Constant and Single-Variable Nodes Removal

A logic optimization program should take advantage of constant and single-variable nodes to reduce the complexity of a Boolean network. It should be noted that the removal of one constant or single-variable node may create another, and such nodes may be produced during the process of logic optimization. Therefore, `sweep` is iterative, and it is invoked many times in a complete logic optimization process.

4.1.2 Removal of Functionally Equivalent Nodes

Traditionally, when a multi-level Boolean function is represented as a SOP form, only constant and single-variable nodes can be identified and removed from the Boolean network during `sweep`. When a Boolean network is represented in the BDD form, there is an added bonus. Since BDD is canonical, the functional equivalence between different Boolean nodes can be detected easily. Therefore, in addition to removing constant and single-variable nodes, all functionally equivalent nodes can be also be removed from a Boolean network during `sweep`.

Although the functionally equivalent Boolean nodes in a Boolean network can also be removed by later optimization procedures (*e.g. eliminate, re-substitution*) in the traditional synthesis flow, it is always beneficial to remove this type of redundancy before the actual logic optimization. Shown in Table 2 is the number of functionally duplicated nodes for some famous testcases. We were surprised to find so much redundancy in those testcases. This is the first time ever that functionally duplicated Boolean nodes can be removed before actual logic optimization.

Circuit	Total Nodes	Duplicated Nodes
C1908	441	118
C2670	787	72
C3540	956	247
C5315	1467	197
C6228	2353	30
C7552	2165	355
C880	302	10
dalu	985	249
i8	1183	186
i9	329	22
i10	1634	84
pair	830	16
vda	123	3

Table 2: Number of functionally duplicated nodes in a Boolean network

Since the removal of one functionally equivalent node may create another, the duplication removal in *BDS* is iterative. The numbers shown in Table 2 are just the number of duplicated nodes in the first iteration. Removing functionally duplicated Boolean nodes helps *BDS* to reach the final optimized netlist. This also contributes to the runtime advantage over traditional approaches, because logic optimization algorithms are generally more expensive than `sweep`.

4.2 Node Elimination

To comply with the scalability requirement, a Boolean network should be allowed to be partially collapsed into a set of *super* Boolean nodes; then logic optimization algorithms can be applied on each super node. Partial collapsing also helps to remove logic redundancy which is embedded in the multi-level configuration. The most frequent cause of redundancy is the so-called *re-convergence*. Partial collapsing is very efficient at removing it.

To carry out partial collapsing, procedure such as `eliminate` comes into play. `eliminate` attempts to find a partially collapsed Boolean network such that Boolean nodes are not too large for logic optimization algorithms. On the other hand, Boolean nodes should not be too fine, otherwise some redundancy may remain in the network. A properly designed `eliminate` scheme provides a better starting point for logic optimization algorithms.

`eliminate` has been successfully implemented in SIS [19]. Cost estimation based on the factored forms was used to guide the process of elimination. Two approaches have been proposed for `eliminate` using BDDs. The first one is based on *progressive* elimination [9]. In this approach, BDDs are constructed from primary inputs to primary outputs. At any point, if the size of a BDD is larger than a predefined threshold, an intermediate variable is introduced and the BDD construction process continues until primary outputs are reached. This approach, however, ignores the specific structure of a Boolean network. As a result, the elimination often stops at boundaries which are not natural to the specific Boolean network. This approach may also cause memory blow-up. The second approach is based on *iterative* elimination [10] which is quite similar to the `eliminate` procedure in SIS [19]. In the process, BDD node count is used as the cost function to guide the elimination.

To comply with the mainstream synthesis flow, an approach similar to that of [19, 10] has been adopted in it BDS. However, due to the efficiency of described techniques for BDD manipulations, our `eliminate` is orders of magnitude faster than [10].

A generic algorithm for iterative elimination is shown in Algorithm 1. In `collect` all Boolean nodes which are eligible for collapsing into its fanout are collected. In `execute` the Boolean network is modified to lock the recent changes. The BDD manager is reordered before the next `collect`. The whole process is iterated until no Boolean node can be eliminated.

```
candidate = collect(bddmgr,network);
while(candidate) {
  execute(bddmgr,candidate);
  reorder(bddmgr);
  candidate = collect(bddmgr,network); }
```

Algorithm 1: Eliminate

In practice, a straightforward implementation of this process does not work. This is mainly because of the abuse of BDD variable reordering. When local BDDs are constructed for a Boolean network, an intermediate variable is created for each Boolean node. Therefore, in addition to all primary inputs, a BDD manager also contains all intermediate variables. The number of such variables could be huge even for a medium-size circuits. Since the complexity of variable reordering could be exponential, reordering a BDD manager with large number of variables will severely degrade the overall runtime performance.

```
candidate = collect(bddmgr,network);
while(candidate) {
  execute(bddmgr,candidate);
  newbddmgr = bddMapping(bddmgr,network);
  reorder(newbddmgr);
  free(bddmgr);
  bddmgr = newbddmgr;
  candidate = collect(bddmgr,network); }
```

Algorithm 2: Efficient Iterative Eliminate Paradigm

In *BDS*, new BDD manipulation techniques are developed to make the approach feasible in practice. Algorithm 2 shows the modified version of Algorithm 1. In this algorithm, instead of using a single BDD manager for all the operations, a new BDD manager is initialized in each iteration. All BDDs after one iteration are *mapped* (to be explained later) into the new BDD manager. A variable reordering is performed for the new BDD manager. The old BDD manager is abandoned. The process is iterated until no Boolean nodes can be eliminated.

The need for a new BDD manager and BDD *mapping* technique can be justified as follows. A typical approach for variable reordering is based on adjacent variable swapping. To find the optimal position for a variable, a bulk of *unique table* will be traversed. During the process of *eliminate*, the removal of one Boolean node from the Boolean network corresponds to the demise of one variable in the BDD manager, so the variable will not be used again. Let us refer these variables to as *unused* variables. After the termination of one iteration many Boolean nodes have been removed, and the BDD manager contains large number of *unused* variables. Table 3 shows the reduction of Boolean nodes after first iteration. It can be found that about 63% variables in the BDD manager become *unused*. It is obvious that performing variable reordering in a BDD manager with large number of *unused* variables is very inefficient.

Circuit	Before	After	Reduction
C1355	474	60	88 %
C1908	325	94	72 %
C2670	656	281	58 %
C3540	793	344	57 %
C432	123	63	49 %
C499	162	57	65 %
C5315	1228	387	69 %
C6288	2338	704	70 %
C7552	1829	455	76 %
C880	296	122	59 %
dalu	764	241	69 %
des	681	294	57 %
mult32	5507	2467	56 %
pair	818	450	45 %
Total	15994	6019	63 %

Table 3: Number of Boolean node reduction after first *eliminate*

Instead of reordering the BDD manager with large number of *unused* variables, a new BDD manager with set of *used* variables is initialized. BDDs are then transfered into the new BDD manager using our `bddPool` mechanism (see Section 3.2). During the process, variables are substituted according to $\mathcal{M}$, where $\mathcal{M}$ is the mapping of variables between the old and new BDD managers. When all BDDs are reconstructed in the new BDD manager, a set of BDDs which are graphically isomorphic to the original ones, but much more compact in the range of indexes, are obtained. This process is referred to as a BDD *mapping*.

Table 4 shows the results of our iterative eliminate paradigm. The results of [10] are also listed for comparison. On average, our `eliminate` is 85 times faster than [10]. The runtime advantage of *BDS* becomes stronger for larger test cases. Although *BDS* is targeting multi-level implementation, it is obvious that the iterative eliminate paradigm in *BDS* will also be useful for PTL synthesis as well. Due to the efficient way of handling BDD variable reordering, our iterative eliminate paradigm has the capability to handle very large circuits.

Circuit	Chaudhry *et al* [10]		BDS	
	BDD nodes	CPU (s)	BDD nodes	CPU (s)
C1355	211	270	207	0.3
C1908	310	25.4	276	0.6
C2670	615	197	527	1.7
C3540	974	101.7	901	3.2
C432	181	4.5	183	0.5
C499	196	2.4	228	0.2
C5315	1008	307.6	918	4.0
C6288	1677	540.7	1507	4.4
C7552	1592	382.1	1227	6.4
C880	298	7.5	300	0.4
Total	7066	1838.9	6274	21.7

Table 4: Results of iterative eliminate paradigm. Both experiments are carried out on Pentium-200.

5. EXPERIMENTAL RESULTS

The experiments have been conducted on a Pentium-III/500 machine running Linux. Most large combinational circuits in *LGSynth91* test case suite are covered in the experiment. All test cases are synthesized by both *BDS* and SIS (*script.rugged*). The results are then mapped by the SIS mapper.

Table 5 shows the experimental results. Two testcases, *dalu* and *vda*, have been singled out from this table for special illustration. In summary, *BDS* uses about 3% more area than SIS. The delay of circuits synthesized by *BDS* is 13% smaller. The memory used by *BDS* is 30% less than SIS. It should be noted that the memory usages reported for SIS in Table 5 only include the memory used by logic optimization procedures in *script.rugged*. Memory used by `full_simplify` is not included. Since global BDDs are constructed during `full_simplify`, which is not the case for *BDS*, it is unfair to compare the total memory used by *script.rugged* with *BDS*. In terms of runtime, *BDS* demonstrates superior advantage over SIS, it is more than 8 times faster than SIS. We must mention that compared with real industrial circuits, all the test cases used in this experiment are very small. We run both *BDS* and SIS on a set of circuits with more than 40K gates. *BDS* was over 100 times faster than SIS. Since those circuits are not in the public domain, the experimental results are not published

Circuit	SIS				BDS-1.3			
	Area	Delay	CPU (s)	Mem (M)	Area	Delay	CPU (s)	Mem (M)
C1355	689	39.40	6.6	1.2	711	45.60	0.4	1.0
C1908	695	68.60	8.1	1.2	730	65.00	0.8	1.0
C3540	1695	81.40	16.1	3.3	1713	81.20	3.6	1.9
C432	290	75.90	46.1	0.7	357	78.40	0.2	0.5
C499	689	39.40	6.8	0.9	708	43.60	0.6	0.5
C5315	2286	68.60	10.2	3.1	2402	70.50	5.3	3.0
C6288	4631	237.8	21.8	4.1	4677	178.3	3.8	1.1
C7552	3038	115.70	54.2	4.9	3112	83.30	4.2	4.8
C880	567	56.10	1.9	1.0	563	43.20	0.7	0.8
pair	2274	74.30	16.1	2.5	2466	52.60	2.1	2.0
rot	965	51.60	4.5	2.0	1025	51.90	1.0	0.9
Total	17819	908.8	192.4	24.9	18464	793.6	22.7	17.5

Table 5: Comparison between BDS-1.3 and SIS. The memory reported for SIS does not include the memory used by `full_simplify`.

here. However, one conclusion we can reach is that the overall complexity of *BDS* is lower than that of SIS.

5.1 Analysis

There are two causes which contribute to the larger circuit area obtained with *BDS*. First, *BDS* has a capability to perform XOR and MUX decompositions. XORs and MUXes have been represented explicitly on the factoring trees and in the final BLIF format generated by *BDS*. However, due to the weak capability to identify XORs and MUXes by the tree-based technology mapper, only a small fraction of XORs and MUXes synthesized by *BDS* can be mapped to XOR and MUX gates. The same problem has been observed in our previous experiment [18].

Second, currently *BDS* does not have the capability to perform *don't care* minimization. If the redundancy can not be removed by `eliminate`, it will most likely remain in the final synthesized circuits. Lack of such a capability is the major hold-back for the current version of *BDS*. Shown in Table 6 are the synthesis results for *dalu* and *vda*. The results can be greatly improved by applying `full_simplify` on the circuits synthesized by *BDS*. However, the area and delay of *dalu* is still 50% more than SIS. Extensive comparison between *BDS* and SIS should be done for *dalu*.

Circuit	SIS		BDS-1.3			
			no full_simplify		full_simplify	
	Area	Delay	Area	Delay	Area	Delay
dalu	1307	58.40	2680	103.5	1927	93.3
vda	837	39.8	1380	32.6	1049	43.2

Table 6: The effect of `full_simplify`.

6. CONCLUSION

In this paper, a new BDD-based logic optimization system has been presented. The experimental results show that BDD-based logic optimization is a promising alternative to the existing logic optimization approaches. Currently, *BDS* is still a very dynamic program. We anticipate the performance of *BDS* will be promoted to a higher level in the near future. A preliminary version of *BDS* can be downloaded from [20].

Compared with the state-of-the-art logic synthesis methodology, BDD-based logic synthesis is new but much less mature. It is too soon to conclude that it will be a practical alternative to the widely accepted methodology. Extensively fundamental research has to be done to make this approach a truly successful synthesis method. We hope our research can initiate a new round of research in logic synthesis area in the years to come.

Acknowledgment:

This work has been supported by a grant from NSF under contract No. MIP-9613864.

References

[1] R. Rudell, "Tutorial : Design of a logic synthesis system," in *Proc. 33rd Design Automation Conference*, 1996, pp. 191–196.

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

[3] Randal E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computer*, vol. 35, no. 8, pp. 677–691, August 1986.

[4] K. Brace, R. Rudell, and R. Bryant, "Efficient Implementation of a BDD Package," in *Proc. Design Automation Conference*, 1990, pp. 40–45.

[5] Yung-Te Lai, Kuo-Rueih Pan, and Massoud Pedram, "OBDD-Based Function Decomposition: Algorithms and Implementattion," *IEEE Trans. on CAD*, vol. 15, no. 8, pp. 977–990, August 1996.

[6] Shih-Chieh Chang, M. Marek-Sadowska, and T. Hwang, "Technology Mapping for TLU FPGA's Based on Decomposition of Binary Decision Diagrams," *IEEE Trans. on CAD*, vol. 15, no. 10, pp. 1226–1235, October 1996.

[7] Jason Cong and Yuzheng Ding, "Beyond the Combinational Limit in Depth Minimization for LUT-Based FPGA Designs," in *IEEE International Conference on Computer-Aided Design*, 1993, pp. 110–114.

[8] K. Yano, Y. Sasaki, K. Rikino, and K. Seki, "Top-Down Pass Transistor Logic Design," *IEEE J. Solid-State Circuits*, vol. 31, no. 6, pp. 792–803, June 1996.

[9] P. Buch, A. Narayan, R. Newton, and A. Sangiovanni-Vincentelli, "On Synthesizing Pass Transistor Logic," in *Intl. Workshop on Logic Synthesis*, 1997.

[10] R. Chaudhry, T. Liu, A. Aziz, and J. Burns, "Area-Oriented Synthesis for Pass-Transistor Logic," in *International Conference on Computer Design*, 1998, pp. 160–167.

[11] Kevin Karplus, "Using if-then-else DAGs for Multi-Level Logic Minimization," Tech. Rep. UCSC-CRL-88-29, University of California Santa Cruz, 1988.

[12] V. Bertacco and M. Damiani, "The Disjunctive Decomposition of Logic Functions," in *IEEE International Conference on Computer-Aided Design*, 1997, pp. 78–82.

[13] Ted Stanion and Carl Sechen, "Boolean Division and Factorization Using Binary Decision Diagrams," *IEEE Trans. on CAD*, vol. 13, no. 9, pp. 1179–1184, September 1994.

[14] S. Minato, "Fast Factorization Method for Implicit Cube Set Representation," *IEEE Trans. on CAD*, vol. 15, no. 4, pp. 377–384, April 1996.

[15] M. A. Thornton and V. S. S. Nair, "Behavioral Synthesis of Combinational Logic Using Spectral Based Heuristics," *ACM Transactions on Design Automation of Electronic Systems*, vol. 4, no. 2, pp. 219–230, April 1999.

[16] S. Yamashita, K. Yano, Y. Sasaki, Y. Akita, H Chikata, K. Rikino, and K Seki, "Pass-Transistor/CMOS Collaborated Logic: The Best of Both Worlds," in *Symposium on VLSI Circuits Digest of Technical Papers*, 1997, pp. 31–32.

[17] C. Yang and M. Ciesielski, "Synthesis for Mixed CMOS/PTL Logic: Preliminary Results," in *International Workshop on Logic Synthesis*, 1999.

[18] C. Yang, V. Singhal, and M. Ciesielski, "BDD Decomposition for Efficient Logic Synthesis," in *International Conference on Computer Design*, 1999, pp. 626–631.

[19] E. Sentovich *et al.*, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, ERL, Dept. of EECS, Univ. of California, Berkeley., 1992.

[20] `http://www.ecs.umass.edu/ece/labs/vlsicad/ciesielski.html`.