Improving the Efficiency of Circuit-to-BDD Conversion by Gate and Input Ordering

Fadi A. Aloul, Igor L. Markov, Karem A. Sakallah
Department of Electrical Engineering and Computer Science
University of Michigan
{faloul, imarkov, karem}@eecs.umich.edu

Abstract

Boolean functions are fundamental to synthesis and verification of digital logic, and compact representations of Boolean functions have great practical significance. Popular representations, such as CNF, DNF, circuits and ROBDDs [4], offer different advantages and are preferred for different tasks. Conversion between those representations is common, especially when one is used to represent the input and another speeds up relevant algorithms.

Our work addresses the construction of ROBDDs that represent outputs of a given Boolean circuit. It is used in synthesis and verification [8]. Earlier works [7, 10] proposed ordering circuit inputs and gates by graph traversals. We contribute orderings based on circuit partitioning and placement, leveraging the progress in recursive bisection and multi-level min-cut partitioning achieved in late 1990s. Our empirical results show that the proposed orderings based on circuit partitioning and placement are more successful than straightforward DFS and BFS, as well as related heuristics proposed in [7, 10, 12].

1 Introduction

Due to the sheer number of n-variable Boolean functions, their explicit specification, e.g., by truth tables, is of limited practical use. Compact representations tend to be implicit, and their power is determined by the efficiency of operations they enable. Important operations include basic logic connectives and satisfiability checking. Subsets of astronomically-sized sets as well as large collections of such subsets are often represented by related techniques via characteristic functions.

In the last 15 years, a number of applications based on efficient manipulation of Boolean functions gained industrial significance, notably in automated design and verification of logic circuits [2, 8]. Today these applications encourage research in efficient operations on Boolean functions, and provide a number of sizable benchmarks.

Asymptotic worst-case complexity analysis often leave no hope as many important operations with Boolean functions are NP-complete, NP-hard or tend to require unaffordable amounts of memory [8]. However, actual performance on typical inputs and asymptotic best-case performance are equally important in applications. For example, if one explicitly stores all 2^n values of a Boolean function, the best-case will be exponential, and such representation will never scale, however good the input may be.

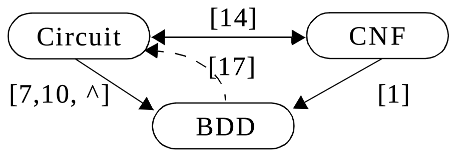


Figure 1. Conversions between Compact Representations of Boolean Functions. (^ stands for this work)

Minimal requirements for a compact representation of Boolean functions include *good best-case performance, reasonable performance on instances arising in applications and compatibility with efficient algorithms*. Other intuitive requirements, such as *irredundancy*, can be traded off. Given an application, the choice of a compact representation can be determined by the type of instances and relevant efficient algorithms. Since the efficient evaluation of Boolean functions is common to most applications, all popular representations are built around evaluation algorithms and can be viewed as "programs" to evaluate a given Boolean function.

Conjunctive Normal Form (CNF) [8] and Disjunctive Normal Form (DNF) [8] are special types of Boolean formulae whose variables represent inputs of a given single-output Boolean function. The value of the formula on particular input values represents the value of the function. Many N-input functions arising in applications can be represented with $O(N)$-bit CNFs or DNFs, furthermore, such representations naturally arise in many contexts. Boolean formulae can be evaluated in linear time and allow several other fast operations, such as co-factoring. In many practical cases neither CNF nor DNF allow both efficient union and intersection operations. Determining whether two formulae denote the same function is often very difficult.

Another compact representation is based on Boolean circuits, where inputs of the function are represented by primary inputs and the value is represented by a unique primary output. Circuit evaluation takes linear time and entails evaluating gates in topological order. Both intersection and union can be performed in linear time or faster, as well as co-factoring. However, the result of multiple unions and intersections will grow with the number of operations, even if a smaller representation is possible. Determining whether two circuits denote the same function is often very difficult. Boolean circuits are often sought as the final result in the synthesis of digital logic and given as input in circuit verification algorithms.

Binary Decision Diagrams (BDDs)[1] [4, 8] are levelized directed graphs, where nodes on every level represent one input of a given Boolean function, plus one additional level with two nodes: 0 and 1. Every node, except for 0 and 1 has two outgoing edges connecting it to nodes on lower levels, and potentially many incoming edges connecting to upper levels. A set of input values corresponds to a path leading from the single node at the top level down to 0 or 1. Thus, every set of input values evaluates to 0 or 1 in linear number of steps. The uniqueness of BDD representations is based on fixing the ordering of variables and hashing that efficiently prunes redundant nodes.

When used to represent Boolean functions, BDDs offer best-case exponential compression, linear-time evaluation, reasonably efficient unions and intersections, as well as uniqueness. The latter implies that if the result of multiple unions and intersections can be compactly represented, then such a representation will be automatically found. The equality of two BDDs can be tested in linear time or less. In most applications of BDDs, the main limiting factor is memory and not the runtime.

While BDDs are the most flexible of popular representations, they often need to be constructed from other representations, such as CNFs, DNFs and circuits. Our work addresses the construction of BDDs from circuits and draws upon recent work [1] where the construction of BDDs from CNFs was considered.

As a consequence, the size of BDDs dramatically depends on the chosen order of variables. Finding a better variable order is often worth spending considerable computational effort because this implies savings for further operations on the constructed BDD. Another parameter critical during the construction of BDDs is the maximal memory required, which may be greater than the memory required by the final BDD. We note that for a given variable order, a poorly implemented construction order may run into an intermediate BDD that does not fit into memory. Therefore, our work addresses both the variable ordering in the final BDD and the details of the construction process. The remainder of this paper is organized as follows. In Section 2, we review existing work on construction of BDDs from circuits, describe our motivation to use circuit partitioning and placement. Section 3 presents our algorithms and Section 4 shows empirical results. Conclusions and ongoing work are given in Section 5.

2 Background

We discuss prior work on the construction of BDDs from circuits. Circuit partitioning and placement concepts which form the basis of the proposed approach are also described.

2.1 Construction of BDDs from Circuits

Algorithms that construct a BDD for a single-output function given by a Boolean circuit are typically recursive. They start by constructing a BDD for each primary input (PI) and finish by constructing a BDD for the primary output (PO). The gates are traversed in a topological order, and at every step a BDD is computed for a new gate using BDDs for its fanin gates. As mentioned earlier, the size of the BDD and its execution time is dependent on the ordering of its variables. A good ordering can lead to a smaller BDD and faster runtime, whereas a bad ordering can lead to an exponential growth in the size of BDD and hence can exceed the available memory. Several heuristics have been proposed to order the BDD variables based on the given circuit input information. In the following, we describe some of the common variable ordering techniques:

- **Original**: Each PI is appended to the BDD variable ordering according to its original index in the circuit.
- **DFS**: A depth-first search (DFS) is performed starting from the PO. A PI is appended to the ordering as soon as its traversed.
- **BFS**: A breadth-first search (BFS) is performed starting from the PO. A PI is appended to the ordering as soon as its traversed.
- **Fujita** [7]: A DFS is performed starting from the PO. PIs with multiple fanouts are appended first to the ordering followed by PIs with single fanouts.
- **Malik-level** [10]: POs are assigned level 0. The level for each node in the circuit is computed by $level(g) = max(level(go) + 1)$, where go corresponds to the fanouts of node g. PIs with the maximum levels are appended to the ordering first.
- **Malik-fanin** [10]: A DFS is performed starting from the PO. However, unlike previous approaches, in which ties are broken between gate fanins by selecting the fanin with the smallest index, the transitive fanin (TFI) depth size is used a tie-breaker. The TFI-depth of a node j is defined as the maximum level of any node in the fanin cone of node j. Fanins with larger TFI-depths are visited first. A PI is appended to the ordering list as soon as its traversed.

The last three heuristics have been shown to provide the best performance when applied to circuits. Fujita's heuristic aims to minimize the number of crosspoints of nets in the circuit diagram. On the other hand, Malik's heuristics prioritize PIs that are far away from the POs in the circuit, since these PIs are expected to greatly influence the circuit behavior. The order of BDD variables can be further improved during the BDD construction by the dynamic sifting heuristic[2] [13], that is now considered an integral part of every BDD package [15] and entails pair-wise swaps of variables.

In addition to ordering the BDD variables (PIs in circuit), the order in which gates are processed can also be varied. It has been shown that different gate traversals orders can lead to large intermediate BDD sizes [12]. In our approach, after

1. Only Reduced Ordered Binary Decision Diagrams are considered in this work.

2. We use sifting from the latest 2.3.1 version of CUDD. We expect CUDD to include the best stable configuration of sifting.

the BDD variables are ordered as explained above, we consider three ways to order gates: (1) use the gate order from the DFS traversal from POs, (2) use the gate order from the BFS traversal from POs, (3) perform a BFS from PIs. In case of a tie, the gate with the smallest index is selected[3]. In general, option 1 shows the best performance. The results for options 2 and 3 are omitted due to limited space.

Recently, Murgai et al. [12], proposed various ways of ordering binary operations for multi-input gates. Specifically, for gates with more than two inputs. Their best scheme, referred to as *size_support*, used an analysis of the size and support sets of the intermediate BDDs to minimize their size. We compare their scheme to our techniques in Section 4.

2.2 Motivating Examples

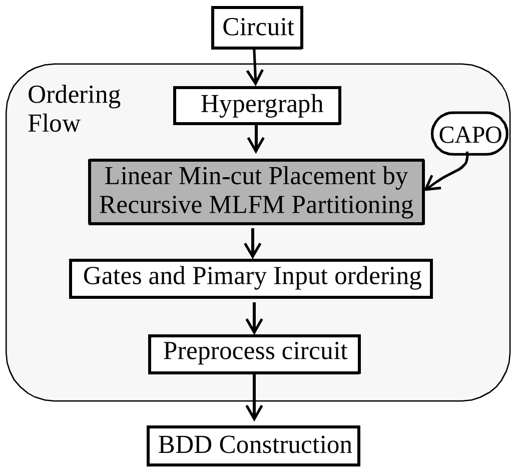
Figure 3(a,c) show two topological orderings of a small circuit that lead to BDDs of different sizes. For a given ordering, we define the *netlength* of a given signal net as the maximal difference in indices of gates[4] on this net. We observe that smaller total netlengths tend to co-exist with smaller BDDs. This connection can be explained as follows. It is known from VLSI placement, that smaller netlengths correlate with smaller cuts, which is used in min-cut placement [5]. Smaller cut-width in circuits have been related to smaller BDDs in [2]. Therefore, we will attempt to produce topological orderings that minimize total netlength, by using min-cut placement.

2.3 Circuit Partitioning and Placement

The expected size of BDDs for randomly chosen Boolean function is exponential in the number of variables, however in many applications BDDs have only polynomial size. This is made possible by the *automatic discovery and use of structure in Boolean functions* during the construction of BDDs. Therefore, it is natural to automatically select a variable ordering and modify the process of constructing BDDs, based on detected structure. Earlier proposed heuristics based on graph traversals (DFS and BFS) [7, 10] attempt to do that, but one expects that additional structure can be discovered by more sophisticated algorithms.

Motivated by the example in Figure 3, we consider the cut-structure of circuits. Intuitively, tightly connected clusters of gates should be processed together, and this can be achieved if gates are ordered by recursive min-cut bisection. Recursive min-cut bisection was studied in (one- and two-dimensional) VLSI placement for at least 30 years and is known to be very successful in minimizing "half-perimeter wire-length" which translates back to smaller netlength [5]. For example, given a gate-ordering, the total netlength equals the sum of all cuts (N and G denote the set of nets and gates -including PIs- in the circuit, respectively):

3. Different tie-breaking strategies lead to different topological orderings. We explored with Malik's *level* and *fanin* options as gate tie-breakers. The results were similar to the *index* tie-breaking approach.
4. Label assigned to the gate's output.



Circuit
Ordering Flow
Hypergraph
CAPO
Linear Min-cut Placement by Recursive MLFM Partitioning
Gates and Pimary Input ordering
Preprocess circuit
BDD Construction

Figure 2. Proposed Ordering Flow.

$$TotalNetLength = \sum_{n \in N} NetLength(n) = \sum_{i=0}^{|G|-1} cut(i)$$

$$AverageNetLength = \frac{\sum_{n \in N} NetLength(n)}{|N|}$$

$$AverageCut = \frac{\sum_{i=0}^{|G|-1} cut(i)}{|G|-1}$$

$$AverageCut = \frac{|N|}{|G|-1} \cdot \frac{\sum_{n \in N} NetLength(n)}{|N|}$$

$$AverageCut = \frac{|N|}{|G|-1} \cdot AverageNetLength$$

Since the total number of nets N and gates G are fixed, the average netlength is proportional to the average cut.

Each partitioning is performed subject to approximate equality of partition sizes, which is critical to performing placement in $\Theta(M(log M)^2)$ time, where M is the size of the input. This efficiency is due to a 1997 breakthrough in min-cut partitioning that brought us multi-level Fiduccia-Mattheyses methods [9]. The Fiduccia-Mattheyses heuristic by itself is vastly inferior to multi-level methods by both runtime and solution quality. Multi-level partitioning consists of a clustering stage and refinement stage. During clustering, randomly chosen pairs of gates connected by wires are merged, creating several clustered circuits. During refining, the iteratively-improving pass-based Fiduccia-Mattheyses heuristic is applied to clustered circuits from the most clustered to the original. At every refining step, the best seen solution is propagated to the next circuit, and therefore the cut cannot become worse.

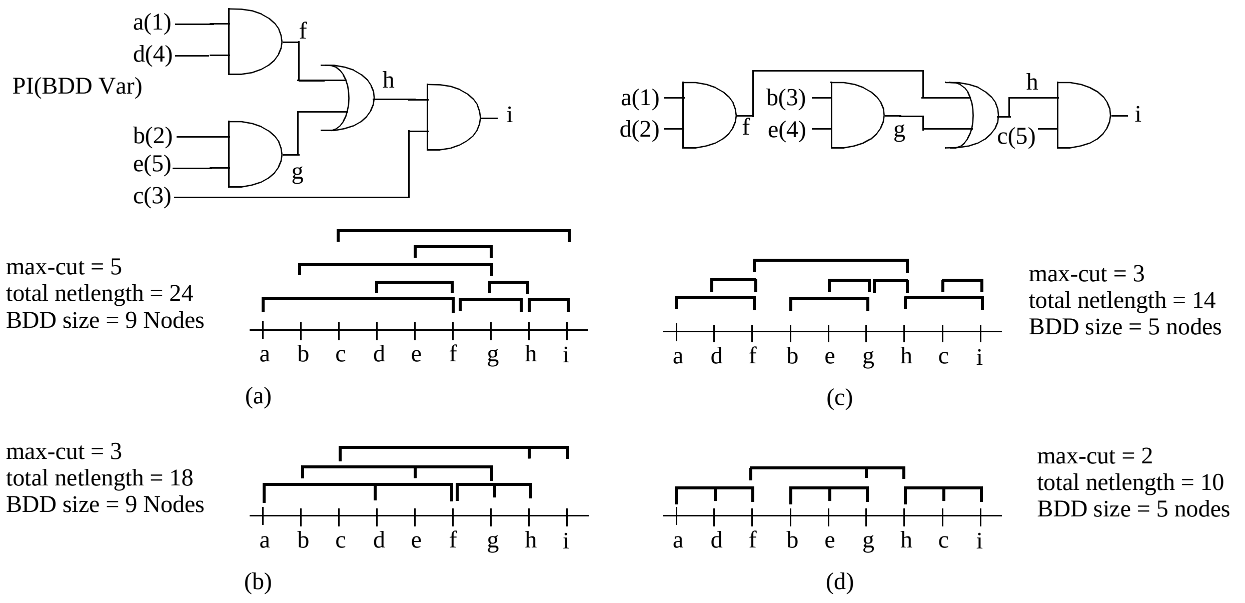


Figure 3. Example using (a) default variable ordering with Circuit hypergraph (b) Dual hypergraph (c) min-cut variable ordering with Circuit hypergraph (d) Dual hypergraph.

The minimization of netlength in one dimension fits our needs by providing a linear ordering heuristic with good asymptotic performance. While in some of our experiments such min-cut placement consumes more time than the BDD construction, its runtime is limited by a near-linear function and is very predictable in practice. Therefore, min-cut placement is a reasonable insurance policy against bad variable orderings that make the construction of BDD impossible.

Aloul et al. [1] have previously used linear hypergraph placement as a static variable ordering technique, *Mince*, for constructing BDDs from CNFs rather than arbitrary combinational circuits.

3 Proposed Techniques

We use the min-cut circuit placer CAPO [5], based on multi-level Fiduccia-Mattheyses min-cut partitioner MLPart [6]. Since circuit partitioning and placement are typically performed on hypergraph representations of circuits, we distinguish two such hypergraph models: the circuit hypergraph (*Circuit HG*) and the dual hypergraph (*Dual HG*).

A *Circuit HG* models circuits by representing each gate with a hypergraph node and each signal net driven by a gate with a hyperedge. PIs and signal nets driven by PIs are also included as hypergraph nodes and hyperedges, respectively. Each hyperedge connects the fanout of a gate to the fanins of the gates that its connected to. An example is shown in Figure 3(a). After CAPO is applied to this hypergraph and returns an ordering of gates, the ordering of PIs is derived from the gate ordering.

A *Dual HG* can also be generated by replacing the above hyperedges, with new hyperedges that connect the fanout of each gate to its fanins. Figure 3(b) shows an example of a *Dual HG*. *Dual HGs* are more likely to produce better PI ordering than the *Circuit HG* approach, since the inputs of each gate are ordered closely to the output of the gate. Figure 3(c-d), show an example of the hypergraph generated by CAPO for the given circuit using the *Circuit HG* and the *Dual HG* models. Clearly, the total netlength and the max-cut were reduced for both cases. The original ordering of the *Dual HG* model implied a total netlength, max-cut, and BDD size of 5, 24, and 9 nodes, respectively. In comparison, the new PI-ordering for the *Dual HG* model reflected a total netlength, max-cut, and BDD size of 3, 14, and 5 nodes, respectively. We conjecture that such PI ordering should yield better BDD runtime and memory results.

4 Empirical Results

In this section, we present experimental evidence of the improvements obtained by using min-cut hypergraph partitioning to represent Boolean functions in BDDs. Empirical results are given for the ISCAS85 circuit benchmarks [3]. All algorithms are implemented in C++ and use the CUDD [15] package to build the BDDs. We used CAPO [5] as our min-cut circuit placer. The experiments were conducted on a Pentium-II 333 MHz, running Linux and equipped with 512 MB of RAM. The runtime and memory limit were set to 1,000 seconds and 500MB, respectively.

Tables 1 and 2 summarize the runtime and memory results for constructing single output BDDs for the PO functions of the ISCAS85 circuits in terms of their PIs. In both tables, the columns represent the *original, BFS, DFS, Fujita* [7], *Malik-level* [10], *Malik-fanin* [10], and Mince orderings using the *Circuit HG* and the *Dual HG,* respectively. In each circuit, the same variable ordering was chosen for all its outputs. The ordering was identified using the output with the largest TFI-depth. The tables also include the runtime needed by CAPO

Table 1. Statistics for constructing the BDDs of the ISCAS85 Circuits without Sifting.
(Node = Maximum number of BDD nodes seen during the construction of the BDD)

Inst-ance	Original		BFS		DFS		FUJ		MAL-Lev		MAL-Fan		Mince				
													CAPO	Circuit HG		Dual HG	
	Time	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time	Time	Node	Time	Node
c17	0.57	7	0.56	7	0.56	7	0.57	7	0.56	7	0.06	**5**	0.21	0.06	6	0.06	6
c432	0.59	**523**	2.84	9199	1.74	6625	1.78	6625	1.76	6625	1.06	6880	0.66	0.11	822	0.11	777
c499	1.04	4945	1.31	7203	1.03	6100	1.02	6092	1.01	6100	0.49	5804	1.72	0.64	**3405**	0.52	4317
c880	6.93	111K	11.9	368K	9.68	245K	9.69	245K	9.05	213K	11.4	220K	1.48	3.41	83K	0.28	**5696**
c1355	1.72	4945	3.05	6255	1.71	6100	1.71	6092	1.71	6100	1.41	5804	1.89	2.26	4581	1.62	**3390**
c1908	2.48	8519	1.36	**2437**	1.64	4808	1.61	4808	1.43	4790	0.74	4837	3	0.98	3459	1.25	7905
c2670	out-of-mem		46.3	5.1M	43.9	4.9M	39.1	4.4M	32.9	3.7M	30.6	3M	4.99	3.47	101K	1.33	**24K**
c3540	26	329K	out-of-mem		256	2M	253	2M	231	1.8M	212	1.6M	7.72	out-of-mem		20.3	**118K**
c5315	out-of-mem		out-of-mem		time-out		time-out		time-out		time-out		11.3	2.54	**15K**	2.77	40K
c6288	out-of-mem		out-of-mem		out-of-mem		out-of-mem		out-of-mem		out-of-mem		9.53	out-of-mem		out-of-mem	
c7552	out-of-mem		out-of-mem		out-of-mem		out-of-mem		out-of-mem		out-of-mem		16.3	5.76	39K	2.22	**6682**
Total	**39**	**459K**	**67**	**5.5M**	**316**	**7.1M**	**308**	**6.7M**	**280**	**5.7M**	**258**	**4.8M**	**58.8**	**19**	**250K**	**30**	**210K**
#Built	**7**		**7**		**8**		**8**		**8**		**8**			**9**		**10**	

Table 2. Statistics for constructing the BDDs of the ISCAS85 Circuits with Sifting.
(Node = Maximum number of BDD nodes seen during the construction of the BDD)

Inst-ance	Original		BFS		DFS		FUJ		MAL-Lev		MAL-Fan		Mince				
													CAPO	Circuit HG		Dual HG	
	Time	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time	Node	Time	Time	Node	Time	Node
c17	0.07	7	0.06	7	0.06	7	0.07	7	0.06	7	0.06	**5**	0.21	0.07	6	0.06	6
c432	0.32	**386**	1.06	1715	0.8	1770	0.81	1770	0.8	1770	1.11	2025	0.66	0.31	444	0.3	541
c499	11.6	3957	11.6	5030	11.9	4483	12.8	4131	11.9	4483	11.6	4836	1.72	19.3	3661	9.77	**2609**
c880	4.11	8176	11.1	10K	3.77	3213	3.76	3397	2.22	3433	3.48	2852	1.48	15.6	23K	1.86	**2411**
c1355	44	4649	107	6862	61.9	4098	41	4019	62.1	4098	33	4451	1.89	44.8	3477	56	**2397**
c1908	5.06	1835	6.41	1721	7.17	2327	7.22	2327	8.55	3147	5.19	**1645**	3	6.1	1741	8.68	2444
c2670	22.3	21489	7.26	9782	10.1	9670	15.3	11K	11.1	13K	8.95	8694	4.99	5	3036	5.98	**2589**
c3540	53.7	14904	24.1	14K	24.2	14K	24.3	14K	24.5	14K	25	**11K**	7.72	47.8	16K	41	12K
c5315	3.13	2297	3.26	2518	3.84	1821	3.56	2077	3.29	2077	3.77	**677**	11.3	2.29	865	3.22	3073
c6288	time-out		time-out		time-out		time-out		time-out		time-out		9.53	time-out		time-out	
c7552	22.1	3450	20	6178	20.7	6267	21	6267	23.5	5890	27.3	4133	16.3	12.9	**3099**	16.1	3918
Total	**166**	**61K**	**192**	**58K**	**144**	**48K**	**129**	**49K**	**148**	**51K**	**120**	**40K**	**58.8**	**154**	**55K**	**143**	**32K**
#Built	**10**		**10**		**10**		**10**		**10**		**10**			**10**		**10**	

to generate the gate orderings. The runtimes are reported in units of seconds, *Node* represent the maximum number of nodes seen during the construction of the final BDD from the circuits, and *Total* represents the sum for successfully built circuits only. The BDDs were constructed by repeatedly calling the *apply* procedure and traversing the circuit using a DFS approach from the POs. A BDD is constructed for a gate as soon as all the BDDs of its fanins are constructed.

As the data illustrate, FUJ and MAL-fan successfully construct the greatest number of circuits compared to the previous 6 PI ordering heuristics. However, when sifting is disabled, *circuit HG* and *Dual HG* orderings are yet able to construct more BDDs than all other approaches. Out of 11 ISCAS85 benchmarks, *circuit HG* and *Dual HG* constructed 9 and 10 BDDs, respectively, as opposed to 8 BDDs by FUJ and MAL-fan. Furthermore, the Dual HG model was successful in solving more instances, using smaller runtimes and BDD nodes, than the *Circuit HG* model. This can be attributed to fact that constructing the BDD for a gate's output is heavily dependent on the gate's inputs which are ordered more closely using the *Dual HG* model. When comparing the results with sifting, the *Dual HG* model does not perform as fast as the MAL-fan, but it does utilize fewer BDD nodes. Building BDDs with sifting generally uses fewer BDD nodes but requires an extra runtime overhead. This is especially noticeable for the *Dual HG* model, which solves all 10 instances in 30 seconds without sifting as opposed to 143 seconds with sifting. On the other hand, the total BDD size is only 32K nodes with sifting, whereas 210K nodes are needed without sifting. The proposed static ordering is very effective in applications that do not allow dynamic sifting.

Tables 3 and 4 show the final BDD sizes (the size of the largest BDD among all BDDs representing the POs) using all 8 variable ordering heuristics with and without sifting, respectively. The *Dual HG* model outperforms all other heuristics. Additionally, we tested Murgai's algorithm [12] (described in Section 2.1) with the *Dual HG* model to minimize the size of the intermediate BDD nodes, overall, the approach is slower due to the overhead in analyzing the size and support sets of the intermediate BDDs and the memory improvements are minimal.

If minimizing the final size is the goal, multiple independent random starts of Mince can be used, and the best result is selected from among all starts.

To summarize, the main advantage of the proposed approach is the use of circuit structure detected by global min-cut partitioning and placement algorithms with near-linear worst-case runtime.

Table 3. Final number of nodes for constructing the BDDs of the ISCAS85 Circuits without Sifting.

Inst-ance	**Original**	**BFS**	**DFS**	**FUJ**	**MAL-Lev**	**MAL-Fan**	**Mince**	
							Circuit HG	**Dual HG**
c17	7	7	7	7	7	**5**	6	6
c432	**523**	9199	6625	6625	6625	6880	733	777
c499	4773	5507	3395	3387	3395	**3131**	3390	4023
c880	111K	368K	245K	245K	212K	220K	83K	**5696**
c1355	4773	5584	3395	3387	3395	**3131**	4430	3326
c1908	8519	2437	4591	4591	4359	3386	**2832**	3573
c2670	n/a	5.1M	5M	4.4M	3.7M	3M	101K	**24K**
c3540	305K	n/a	1.7M	1.7M	1.6M	1.4M	n/a	**79K**
c5315	n/a	n/a	n/a	n/a	n/a	n/a	**15K**	20K
c6288	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
c7552	n/a	n/a	n/a	n/a	n/a	n/a	18K	**6682**
Total	**435K**	**5.5M**	**6.8M**	**6.3M**	**5.5M**	**4.5M**	**228K**	**147K**
#Built	**7**	**7**	**8**	**8**	**8**	**8**	**9**	**10**

Table 4. Final number of nodes for constructing the BDDs of the ISCAS85 Circuits with Sifting.

Inst-ance	**Original**	**BFS**	**DFS**	**FUJ**	**MAL-Lev**	**MAL-Fan**	**Mince**	
							Circuit HG	**Dual HG**
c17	7	7	7	7	7	**5**	6	6
c432	**386**	385	450	450	450	384	411	541
c499	2139	2610	2387	3187	2387	2755	2478	**1979**
c880	5209	4060	2820	2957	3433	**1886**	15080	2050
c1355	2163	3410	2672	3226	2672	2667	2430	**1998**
c1908	1793	1721	2327	2327	1803	1800	**1113**	1987
c2670	21K	9782	9670	9126	13K	8694	**1671**	2589
c3540	6741	8633	8661	8661	8633	7377	7127	**6732**
c5315	645	**625**	777	802	804	677	653	734
c6288	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
c7552	2002	5079	5395	5395	4902	3367	2031	**1004**
Total	**43K**	**36K**	**35K**	**36K**	**38K**	**30K**	**33K**	**19K**
#Built	**10**	**10**	**10**	**10**	**10**	**10**	**10**	**10**

5 Conclusions

We propose a new approach for constructing BDDs from Boolean circuits, based on min-cut circuit partitioning and placement, applied to the orderings of the BDD variables and the circuit gates. We empirically validate our heuristics on ISCAS circuit benchmarks and achieve competitive results. In particular, our best results (for heuristics *Circuit HG* and the *Dual HG*) are typically better than those published in [7, 10]. Related work by Wang et al. [16], uses the hMetis min-cut partitioner [9] instead of MLPart [6], and does not explicitly use a circuit placer. While we compare the efficiency of different circuit models in terms of undirected hypergraphs, Wang et al. [16] considers directed hypergraph models and post-processes the results of classical min-cut partitioning to improve oriented cuts. Although our approach does not pursue oriented cuts, our empirical results are very competitive.

Currently, we are working on an open source release of our implementations.

Acknowledgments

This work is funded in part by the DARPA/MARCO Gigascale Silicon Research Center and an Agere Systems/SRC Research fellowship.

References

[1] F. Aloul, I Markov, and K. Sakallah, "Faster SAT and Smaller BDDs via Common Function Structure," *in Proc. of the International Conference on Computer Aided Design*, pp. 443-489, 2001.

[2] C. Berman, "Circuit Width, Register Allocation, and Ordered Binary Decision Diagrams," *in IEEE Transactions on Computer Aided Design*, 10(8), pp. 1059-1066, 1991.

[3] F. Brglez, and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN," *in Proc. of the International Symposium on Circuits and Systems*, pp. 677-692, 1985.

[4] R. Bryant, "Graph-based algorithms for Boolean function manipulation," *in IEEE Transactions on Computers*, 35(8), pp. 677-691, 1986.

[5] A. Caldwell, A. Kahng, and I. Markov, "Can Recursive Bisection Produce Routable Placements?" *in Proc. of the Design Automation Conference*, pp. 477-482, 2000.

[6] A. Caldwell, A. Kahng, and I. Markov, "Improved Algorithms for Hypergraph Bipartitioning," *in Proc. of the IEEE ACM Asia and South Pacific Design Automation Conference*, pp. 661-666, 2000.

[7] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams," *in Proc. of the International Conference on Computer Aided Design*, pp. 2-5, 1988.

[8] G. Hachtel and F. Somenzi, "Logic Synthesis and Verification Algorithms," *Kluwer Academic Publishers*, 3rd ed., 2000.

[9] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design," *in Proc. Design Automation Conference*, pp. 526-529, 1997.

[10] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," *in Proc. of the International Conference on Computer Aided Design*, pp. 6-9, 1988.

[11] S. Minato, N. Ishiura, and S.Yajima, "Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation," *in the Proc. of the Design Automation Conference*, pp. 52-57, 1990.

[12] R. Murgai, J. Jain, M. Fujita, "Efficient Scheduling Techniques for ROBDD Construction," *in Proc. of the International Conference on VLSI Design*, pp. 394-401, 1999.

[13] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *in Proc. of the International Conference on Computer Aided Design*, pp. 42-47, 1993.

[14] J. Marques-Silva and K. Sakallah, "Robust Search Algorithms for Test Pattern Generation," *Proc. of the IEEE Fault-Tolerant Computing Symposium*, 1997.

[15] F. Somenzi, "Colorado University Decision Diagram Package," *http://vlsi.colorado.edu/~fabio/CUDD*, 1997.

[16] D. Wang, E. Clarke, Y. Zhu, and J. Kukula, "Using Cutwidth to Improve Symbolic Simulation and Boolean Satisfiability," *in the International High Level Design Validation and Test Workshop*, 2001.

[17] C. Yang and M. Ciesielski, "BDS: A BDD-Based Logic Optimization System," *in Proc. of the Design Automation Conference*, pp. 92-97, 2000.