Faster SAT and Smaller BDDs via Common Function Structure

Fadi A. Aloul
Igor L. Markov
Karem A. Sakallah

Technical Report

December 10, 2001

THE UNIVERSITY OF MICHIGAN

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan, 48109-2122
USA

Abstract

The increasing popularity of SAT and BDD techniques in verification and synthesis encourages the search for additional speed-ups. Since typical SAT and BDD algorithms are exponential in the worst-case, the structure of real-world instances is a natural source of improvements. While SAT and BDD techniques are often presented as mutually exclusive alternatives, our work points out that both can be improved via the use of the same structural properties of instances. Our proposed methods are based on efficient problem partitioning and can be easily applied as pre-processing with arbitrary SAT solvers and BDD packages without source code modifications.

Finding a better variable-ordering is a well recognized problem for both SAT solvers and BDD packages. Currently, all leading edge variable-ordering algorithms are dynamic, in the sense that they are invoked many times in the course of the "host" algorithm that solves SAT or manipulates BDDs. Examples include the DLCS ordering for SAT solvers and variable-sifting during BDD manipulations. In this work we propose a universal variable-ordering MINCE (MIN Cut Etc.) that pre-processes a given Boolean formula in CNF. MINCE is completely independent from target algorithms and outperforms both DLCS for SAT and variable sifting for BDDs. We argue that MINCE tends to capture structural properties of Boolean functions arising from real-world applications. Our contribution is validated on the ISCAS circuits and the DIMACS benchmarks. Empirically, our technique often outperforms existing techniques by a factor of two or more. Our results motivate search for stronger dynamic ordering heuristics and combined static/dynamic techniques.

1 Introduction

Algorithms that efficiently manipulate Boolean functions arising in real-world applications are becoming increasingly popular in several areas of computer-aided design and verification. In this work we focus on two classes of these algorithms: complete Boolean satisfiability (SAT) solvers [21, 27, 30, 35] and algorithms for manipulating Binary Decision Diagrams (BDDs) [6, 9, 18]. A generic complete SAT solver must correctly determine whether a given Boolean function represented in *conjunctive normal form* (CNF) evaluates to *false* for all input combinations. Aside from its pivotal role in complexity theory, the SAT problem has been widely applied in electronic design automation. Such applications include ATPG [17, 31], formal verification [3], timing verification [28] and routing of field-programmable gate arrays [22], among others. While no exact polynomial-time algorithms are known for the general case, many exact algorithms [21, 27, 30, 35] manage to complete very quickly for problems of practical interest. Such algorithms are available in the public domain and are typically based on “elementary steps” that consider one variable at a time (e.g. branch-and-bound algorithms select the next variable for branching.) Previously published results [21, 27, 30, 35], as well as our empirical data, clearly imply that the order of these steps critically affects the runtime of leading edge SAT algorithms. This order of steps depends on the order of variables used to represent the input function, but can also be controlled dynamically based on the results of previous steps.

BDDs* [6, 9] are commonly used to implicitly represent large solution spaces in combinatorial problems that arise in synthesis and verification. A BDD is a directed acyclic graph constructed in such a way that its directed paths represent combinatorial objects of interest (such as subsets, clauses, minterms, etc.). An exponential compression rate is achieved by BDDs whose number of paths is exponential in the number of vertices and edges (graph size). BDDs can be transformed by algorithms that visit all vertices and edges of the directed graph in some order and therefore take polynomial time in the “current” size of the graph. However, when new BDDs are created, some of these algorithms tend to significantly increase the number of vertices, potentially leading to exponential memory and runtime requirements. Several BDD ordering techniques have been proposed to overcome this problem. These include static [11, 19] and dynamic approaches [23, 25]. Just as for SAT solvers, the order of “elementary steps” is critically important. This order can either be chosen *statically*, i.e. by pre-processing the input formula, or *dynamically*, based on the outcome of previous steps during the search process. The construction order of the BDD can also have a significant effect on the intermediate sizes of the BDD.

A reliable and fast variable-ordering heuristic for a given application can dramatically affect its competitiveness and is often considered an important part of implementation. For example, the leading-edge SAT solver GRASP [27] is typically used with the dynamic variable-ordering heuristic DLIS, and the renowned CUDD package [29] for BDD manipulation incorporates the dynamic variable-sifting heuristic which is applied many times in the course of BDD transformations. Variable sifting is affected by the initial order, but can also be completely turned off to improve runtime. Sifting for BDDs is relatively more expensive than most dynamic ordering heuristics for SAT. However, the effect of ordering heuristics on total runtime is highly instance-specific.

* Only Reduced Ordered Binary Decision Diagrams (ROBDDs) are considered in this work.

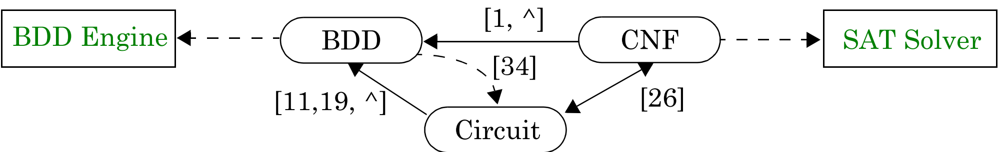


Fig. 1. Conversions between Compact Representations of Boolean Functions.
(^ stands for this work)

We noticed that, for some CNF formulae in Table II (such as hole-9 and par16-2-c), turning off sifting for BDD manipulations and turning off DLIS in SAT resulted in significantly smaller runtimes. For BDDs, this also led to memory savings, especially for circuit benchmarks from the ISCAS89 set. In other words, using a good order of variables when encoding problems into a CNF formula was, by itself, superior to using the best known dynamic heuristic with a poor static order of variables (note that static and dynamic can be trivially combined). In practice, static variable- orderings are easier to work with because they do not require modifying the source code of the host algorithm. In particular, the same variable-ordering implementation can be used for SAT solvers and BDD manipulations if it, indeed, improves both classes of algorithms. However, an application-specific encoding procedure may overlook superior static variable-orderings. Therefore, we propose a domain-independent algorithm to automatically find good "static" variable-orderings that capture global properties of CNF formulae and circuits.

This work involves three types of Boolean function representations: CNF formulas, Boolean circuits, and BDDs. While BDDs are the most flexible of popular representations, they often need to be constructed from other representations, such as CNFs, circuits, or *disjunctive normal form* formulae. We address the construction of BDDs from circuits and CNFs as shown in Figure 1.

The remainder of the paper is structured as follows. In Section 2, we review existing work on solving CNF instances using SAT solvers and constructing BDDs from CNF and circuits. Section 3 motivates our reliance on circuit/CNF partitioning and placement and reviews recent progress in that area. Section 4 describes applications to SAT and BDDs and shows examples of hypergraph partitioning. Section 5 provides experimental evidence of the effectiveness of partitioning-based variable-ordering. Section 6 concludes the paper and provides perspective on future work.

2 Background

2.1 Solving CNF problems using SAT

A *conjunctive normal form* (CNF) formula φ on n binary variables $x_1, \ldots, x_n$ is the conjunction (AND) of m *clauses* $\omega_1, \ldots, \omega_m$ each of which is the disjunction (OR) of one or more literals, where a literal is the occurrence of a variable or its complement. The size of a clause is the number of its literals. A formula φ denotes a unique n-variable Boolean function $f(x_1, \ldots, x_n)$ and each of its clauses corresponds to an implicate of f [13]. The satisfiability problem (SAT) is concerned with finding an assignment to the arguments of $f(x_1, \ldots, x_n)$ that makes the function equal to 1 or proving that the function is equal to the constant 0.

Gate Type	Gate Function	Gate Formula
	$z = BUF(x_1)$	$(\overline{x_1} + z) \cdot (x_1 + \bar{z})$
	$z = NOT(x_1)$	$(x_1 + z) \cdot (\overline{x_1} + \bar{z})$
	$z = NOR(x_1, \ldots, x_j)$	$\left[\prod_{i=1}^{j} (\overline{x_i} + \bar{z})\right] \cdot \left(\sum_{i=1}^{j} x_i + z\right)$
	$z = NAND(x_1, \ldots, x_j)$	$\left[\prod_{i=1}^{j} (x_i + z)\right] \cdot \left(\sum_{i=1}^{j} \overline{x_i} + \bar{z}\right)$
	$z = OR(x_1, \ldots, x_j)$	$\left[\prod_{i=1}^{j} (\overline{x_i} + z)\right] \cdot \left(\sum_{i=1}^{j} x_i + \bar{z}\right)$
	$z = AND(x_1, \ldots, x_j)$	$\left[\prod_{i=1}^{j} (x_i + \bar{z})\right] \cdot \left(\sum_{i=1}^{j} \overline{x_i} + z\right)$

Fig. 2. CNF formulas representing simple gates

Backtrack search algorithms implicitly traverse the space of 2^n possible binary assignments to the problem variables looking for a satisfying assignment. A typical backtrack search algorithm consists of three main engines:

- *Decision* engine: that makes *elective* assignments to the variables
- *Deduction* engine: that determines the consequences of these assignments, typically yielding additional *forced* assignments to, i.e. implications of, other variables
- *Diagnosis* engine: that handles the occurrence of conflicts (i.e. assignments that cause the formula to become unsatisfiable) and backtracks appropriately.

Several techniques have been proposed to improve the above three engines. Nevertheless, selecting an intelligent variable decision order remains a challenge. Many decision heuristics have been proposed. Some are based on an analysis of the number of variables and clauses in the problem, such as DLCS (select the variable that appears in the maximum number of unresolved clauses) or DLIS (select the literal that appears in the maximum number of unresolved clauses). On the other hand, others are based on randomized algorithms.

2.2 Construction of BDDs from CNFs

CNF formulae can be viewed as a two-level logic circuit, in which each clause is represented by an OR gate whose fanins are equal to the number of literals in the clause. The outputs of all OR gates are ANDed together to produce the function f. In contrast,

circuit consistency functions can also be represented in CNF in linear time [26]. Each gate is represented using a CNF formula that denotes a valid input-output assignment to the gate. The CNF formula for the circuit consists of the conjunction of the formulae representing each gate. Figure 2 shows the CNF formulae for simple gates.

In general, dynamic sifting is the main variable ordering heuristic used in constructing BDDs from CNFs. In this report, we show that BDD variable ordering in addition to the order in which clauses are processed can be very effective in reducing the execution runtime and the size of the BDDs.

2.3 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* [11]: 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* [19]: 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* [19]: 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 [25], that is now considered an integral part of every BDD package [29] and entails pairwise swaps of variables.

In addition to ordering the BDD variables (PIs in circuit), the order in which gates are processed can also be varied. After 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†. In general, option 1 shows the best performance.

3 Problem Partitioning

We first observe that Boolean functions arising in many applications represent spacial, logical or causal dependencies/connections among variables. Therefore, processing "connected" variables together seems intuitively justified. For example, if a large SAT instance is not satisfiable because of a small group of inconsistent variables, the variables in this group must be "connected" by some clauses. If we can partition all variables into, say, two largely independent groups, then such a function is likely to be represented by a BDD with a small cut, i.e. there will be relatively few edges between these two groups. BDDs with many small cuts tend to have fewer edges, and therefore fewer vertices (since every vertex is a source of exactly two edges). This intuition suggests that we interpret CNF formulae as hypergraphs by representing variables by vertices and clauses by edges. Two vertices share an edge if the two corresponding variables share a clause in the formula. Applying balanced min-cut partitioning to such hypergraphs separates the original CNF formula into relatively independent subformulae. Ordering the variables in each part together would be a step towards ordering "connected" variables next to each other, as advocated earlier. Once the first partitioning is performed, the parts can be partitioned recursively. This process can provide a complete variable-ordering. We note that cuts of CNF formulae have been studied in [24], and instances having small cuts were theoretically shown to be "easy" for SAT. Our work seeks constructive and efficient ways to amplify the "easiness" of CNF instances with small cuts by finding good variable-orderings.

Additionally, circuit cutwidth has been correlated with the size of BDDs [2]. Figure 7(a-b) shows 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 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 [7]. 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.

† Different tie-breaking strategies lead to different topological orderings. We experimented with Malik's *level* and *fanin* options as gate tie-breakers. The results were similar to the *index* tie-breaking approach.

3.1 Recursive Bisection and Hypergraph Placement

Recursive min-cut bisection of hypergraphs has been intensively studied in the context of VLSI placement for at least 30 years. In particular, the recursive bisection procedure described earlier for CNF formulae corresponds to the *linear placement problem* [14], where hypergraph vertices are placed in one, rather than in two, dimensions. It is well-known that placement by recursive bisection leads to small "half-perimeter wire-length" that translates back to small average clause span in CNF formulae. Here we define the *span* of a clause with respect to a variable-ordering as *the difference between the greatest and the smallest variables in this clause* (so that the span exactly corresponds to the half-perimeter wirelength of a hyperedge). We can also define the *i-th cut* with respect to a given ordering as the number of clauses including variables with numbers both less than and greater than *i+0.5*.

Observation: Given a variable-ordering, the total clause span equals the sum of all cuts (E and V denote the set of hyperedges and vertices, respectively, in the hypergraph)

$$TotalSpan = \sum_{e \in E} span(e) = \sum_{i=0}^{|V|-1} cut(i)$$

The average clause span is proportional to the average cut,

$$AverageSpan = \frac{\sum_{e \in E} span(e)}{|E|}$$

$$AverageCut = \frac{\sum_{i=0}^{|V|-1} cut(i)}{|V|-1} = \frac{|E|}{|V|-1} = \frac{\sum_{e \in E} span(e)}{|E|} = \frac{|E|}{|V|} \cdot \frac{|V|}{|V|-1} \cdot AverageSpan$$

and the coefficient is approximately equal to the clause-to-variable ratio of the CNF formula. (Since the total number of clauses and variables equals the total number of hyperedges and vertices, respectively)

$$AverageCut \approx \frac{|Clauses|}{|Variables|} \cdot AverageSpan$$

It is known from VLSI placement that recursive min-cut bisection of hypergraphs produces placements with small total net-length. Since the total net-length of hypergraphs corresponds to the total clause span of CNF formulae, we will use the leading-edge hypergraph placer CAPO [7] based on recursive min-cut bisection [8, 16] to minimize the average clause spans and cuts of CNF formulae. CAPO implements several improvements to classical recursive bisection, reducing the total clause span. Such techniques include bisection with high balance tolerance and adaptive cut-line selection, which allows greater freedom in partition sizes in order to improve the cut. The underlying multi-level hypergraph partitioner MLPart [8] outperforms the well-known hMetis [16], while both rely on Multi-Level Fiduccia-Mattheyses (MLFM) partitioning heuristics. Since the MLFM heuristic is randomized, it returns different solutions on every call (we call it a *start*). On every call, MLPart executes two

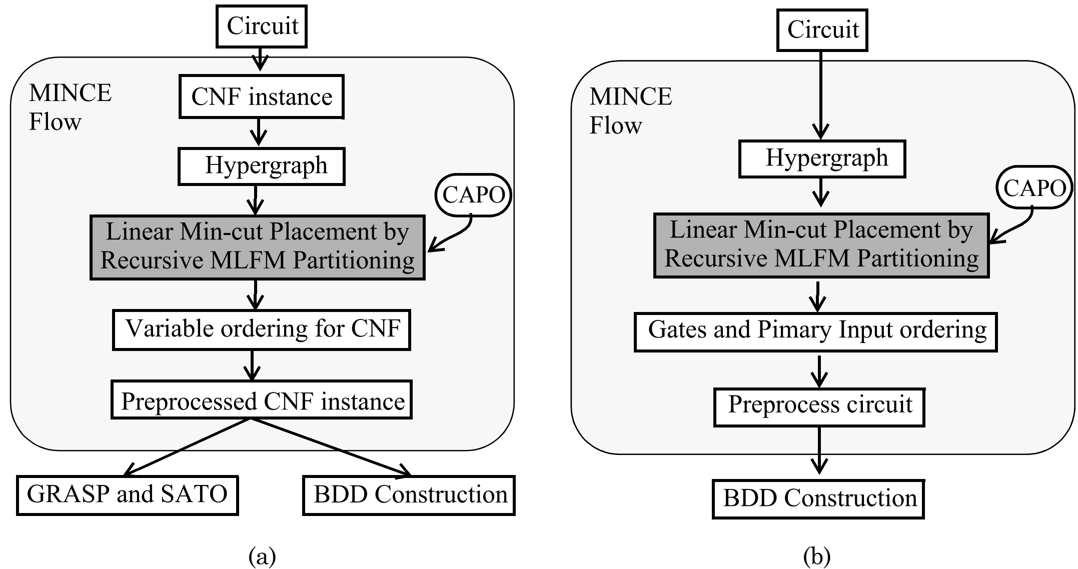


Fig. 3. The MINCE heuristic based on Multi-Level Fiduccia-Mattheyses (MLFM) partitioning [7, 8, 16] for (a) CNF problems (b) circuits

independent *starts* and applies one V-cycle [8, 16] to further improve the better solution.

Wood and Rutenbar have already used linear hypergraph placement as a variable-ordering technique for BDD minimization in 1998 [33]. However, they used spectral methods which entail converting hyperedges to edges and then minimizing quadratic edge length, rather than the half-perimeter (linear) edge length. Spectral placement methods used in [33] do not appear to have direct connection to cut minimization. As of 2001, spectral methods for partitioning and placement are practically abandoned due to their unacceptable runtime on large instances and poor solution quality as measured by half-perimeter edge length. This can be contrasted with min-cut placement that is among the fastest known approaches, provides good solutions and is obviously related to cut minimization.

4 Proposed Techniques

4.1 Ordering Variables in CNFs

We propose the following heuristic that orders variables in CNF formulae (see Figure 3(a)). An initial CNF formula (that may originate from circuits or other applications) is converted into a hypergraph (see Figure 4). An ordering of hypergraph vertices is then found via min-cut linear placement and translated back into an ordering of CNF variables. The original CNF formula is reordered and used (i) as input to an arbitrary SAT solver, or (ii) to construct a BDD representation of the Boolean function it represents. The results produced by SAT solvers and BDD manipulations are then translated back into the original variable order.

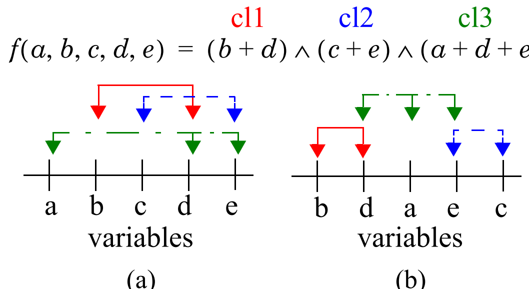


Fig. 4. Example of (a) default vertex-ordering (b) improved vertex-ordering

Note that this approach does not require modifications in SAT solvers, BDD manipulation software or the min-cut placer.‡ We call this heuristic MINCE (MIN-Cut, Etc.) and implemented it by chaining publicly available software with PERL scripts.

To enable black-box reuse of publicly available software (CAPO), we ignore polarities of literals in CNF formulae. We note that the oriented version of min-cut bisection has been extensively studied in the context of timing-driven placement. In particular, a small unoriented cut can be interpreted as an oriented cut which is not greater. Vice versa, in most real-world examples, near-optimal oriented cuts can be found by unoriented partitioning.

On the empirical side, our results with BDD minimization presented below show that MINCE, by itself, outperforms variable-sifting (used without static ordering) in both runtime and memory. According to [13], as of 2000, variable sifting is the best published dynamic variable reordering heuristic for BDDs with near-linear performance.** From this, we conclude that our proposed technique outperforms all other published scalable approaches to BDD minimization. Of course, dynamic variable reordering techniques can be applied on top of MINCE or can use MINCE order as a tie-breaker.

Applications that entail several BDD operations or solve similar SAT problems can reuse the same static ordering for all runs. On the other hand, since MINCE is randomized and returns different solutions every time it is called, it can also be used to perform random restarts of SAT solvers [12].

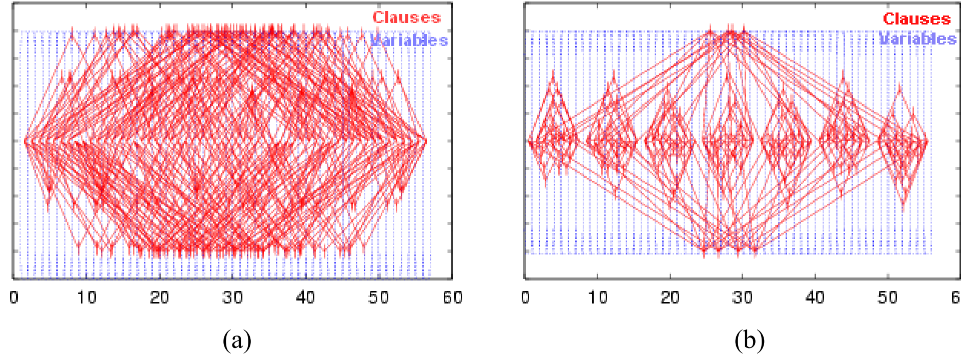
4.1.1 Example

Figure 4 illustrates the difference between a good and a bad variable order for a CNF formula. We use the CAPO placer to find an ordering of vertices, i.e. variables, that produces a small total (equivalent average) clause span. Figure 4(b) shows a sample order returned by MINCE for the example described. The total span of all clauses in this CNF formula is reduced from 8 to 4 by this better variable order. In addition, the number of edges crossing each variable (cut) is reduced. The original problem has a maximum *variable cut* (at variable c) of 3 which is reduced to 1 in the MINCE order.

In general, *structured* problems such as the *hole-n* series of benchmarks (e.g., hole-10, hole-11, etc.) are divided by MINCE into several partitions. Figure 5 shows such an example. The initial variable order has average *clause span* and *variable cut* equal

‡ Commercial EDA software can be used, e.g. Cadence QPlace.

** Some generic or simulated annealing reordering algorithms can gene-rate smaller BDDs but may incur longer runtimes.



(a) (b)

Fig. 5. Sample hypergraph representing the structure of the *hole-7* instance using (a) default vertex-ordering (b) improved vertex-ordering.
Variables are represented by points on the x-axis and clauses are represented by stars of edges that connect those points. The center-point of each star is elevated proportionally to the span of the clause with the given ordering of the variables (i.e., the distance between the right-most and the left-most variables in the clause)

to 74 and 20, respectively. In comparison, the new variable order, has average *clause span* and *variable cut* equal to 17 and 4.7, respectively. As shown in Figure 5(b), this reduction exposes the problem's structure. Our experiments show that such MINCE variable-ordering generally speeds up SAT solvers and improves runtime/memory of BDD manipulations.

Similar techniques and intuitions apply in related contexts. For example, one can apply MINCE to DNF formulae rather than CNF formulae. In this and related cases, one starts with a description of a Boolean function that is sparse, i.e., connects very few groups of variables (by clauses, minterms, in terms of circuit connectivity, etc.). Recursive partitioning orders the "connected" variables close to each other. Since connections between variables often imply logical dependencies, min-cut orderings allow SAT solvers and BDD engines to track fewer variables beyond their neighborhoods.

4.2 Ordering Clauses in CNFs

In Section 4.1, we used linear hypergraph placement to generate a static variable ordering technique to speedup SAT and minimize BDDs. We propose to further minimize the BDD and speed up the BDD construction runtime by ordering the clauses using linear hypergraph placement. If we can partition the clauses into several groups, in which clauses in each group share common variables, then such a function is expected to have a faster runtime and smaller intermediate BDD sizes, since we are likely to traverse a specific part of the BDD that only involves the common shared variables. On the other hand, constructing BDDs for a random order of clauses, could require traversing the BDD between its highest and lowest index node each time a clause is added.

We propose the following heuristic that orders the clauses in CNF formulae. An initial CNF formula is converted to a hypergraph, in which clauses are represented by vertices. For each variable x, a hyperedge is created that connects all clauses including the variable x. Applying balanced min-cut partitioning to such hypergraphs separates the CNF formula into relatively independent subformulae. Constructing the

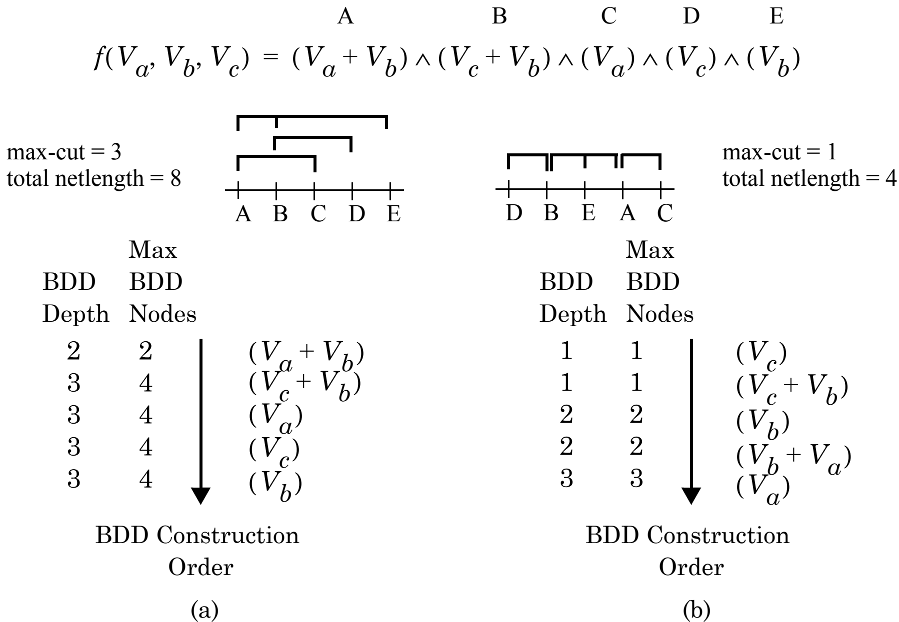


Fig. 6. Example of (a) default clause-ordering (b) improved min-cut clause-ordering

BDD for the clauses in each part would be a step towards manipulating given parts of the BDD, as advocated earlier. Figure 6 shows an example. When building the clauses using the original clause order, the BDD will have a maximum size of 4 after constructing the second clause. In comparison, if the improved clause order is used, the BDD will have a maximum size of 3 levels after constructing the fifth clause.

In addition to the min-cut clause ordering approach, we propose to order the clauses according to their literals. Each clause *level* is computed as $level = min(literals)$. Clauses with the highest levels are constructed first, since they include BDD variables with the highest index or at the bottom of the BDD. Such an approach allows for a bottom-up construction of the BDD and permits the manipulation of variables at specific levels in the BDD. We will refer to this as the *Bottom-Up* clause ordering approach.

4.3 Ordering Primary Inputs in Circuits

Similarly, recursive min-cut bisection can also be applied to circuits to identify tightly connected clusters of gates. Processing such clusters should help in reducing the construction runtime and the size of BDDs. We use the min-cut circuit placer CAPO [7], based on multi-level Fiduccia-Mattheyses min-cut partitioner MLPart [8]. 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 hyperdges, 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 7(a). After CAPO is applied to this hypergraph and returns an ordering of gates, the ordering of PIs is derived from the gate ordering.

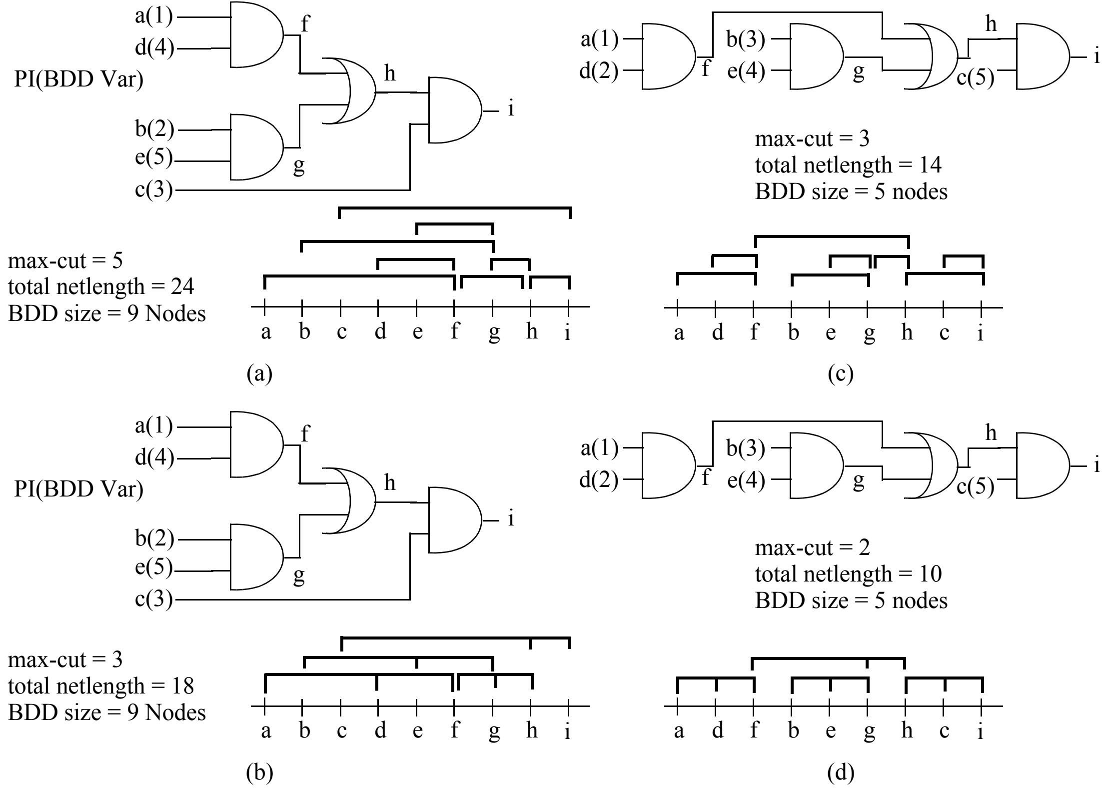


Fig. 7. Example using (a) default variable ordering with Circuit hypergraph (b) Dual hypergraph (c) min-cut variable ordering with Circuit hypergraph (d) Dual hypergraph

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 7(b) shows an example of a *Dual HG*. *Dual HG*s 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 7(c-d), show an example of the hypergraph generated by CAPO for the given circuit using the *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.

5 Empirical Results

In this section, we present experimental evidence of the improvements obtained by MINCE. We used GRASP as our SAT solver [27], CUDD as our BDD engine [29], and CAPO as our min-cut circuit placer. Empirical results are given for the DIMACS [10] and the *n-queens* CNF benchmarks, as well as flat versions of the ISCAS89 circuit benchmarks [4] expressed in CNF and the ISCAS85 circuit benchmarks. Experiments were conducted on a Pentium-II 333 MHz, running Linux with 512 MB RAM. For all experiments, the CPU time was set to 10K and 1K seconds for the SAT and BDD experiments, respectively. The memory limit was set to 500MB for all experiments.

TABLE I: Summary of GRASP runtimes for the DIMACS set (winning and total runtimes are in bold).

Bench mark	**#I**	**MSTS**		**MSOS**		**DLCS**		**DLIS**		**Original**		**MINCE**				**Avg Var Cut**	
		#I	**Time**	**#I**	**Time**	**#I**	**Time**	**#I**	**Time**	**#I**	**Time**	**#I**	**Order +**	**Solve =**	**Total**	**Fix**	**New**
aim	72	72	**2.61**	72	3.16	72	3.81	72	6.71	72	2.72	72	148	2.94	150	11676	6542
bf	4	4	2.63	4	4.97	4	2.56	4	**2.3**	3	10019	4	50.5	2.1	53	2853	440
dub	13	13	29.06	13	18.12	13	2.15	13	2.73	13	**0.71**	13	6.91	0.69	7.6	1717	106
hanoi	2	1	10005	1	12267	0	20000	0	20000	2	**83.13**	2	42.8	83.8	127	408	321
hole	5	3	26956	2	30193	4	11705	5	9466	5	6287	5	**4.07**	**660**	**664**	581	108
ii16	10	10	5407	10	6189	8	20259	9	10321	10	17685	10	**543.2**	**1832**	**2375**	76466	7935
ii32	17	16	11063	16	11187	17	9492.6	17	**4.94**	15	20598	16	399.8	10028	10428	49616	11531
ii8	14	14	2.98	14	2.75	14	8.79	14	7.99	14	**1.04**	14	207.9	0.74	209	25396	2749
jnh	50	50	**5.08**	20	6.58	50	6.48	50	8.51	50	27.62	50	395.3	31.9	427	25952	22701
par16	10	10	21652	10	20470	8	27708	9	21855	10	2536	10	**65.8**	**1477**	**1543**	4789	879
par8	10	10	**0.19**	10	0.21	10	0.22	10	0.22	10	0.21	10	11.8	0.22	12	1613	436
pret	8	8	0.72	8	**0.68**	8	0.7	8	0.66	8	0.59	8	3.94	0.52	4.5	865	138
ssa	8	8	97.33	8	12.63	8	3.73	8	**2.44**	6	20001	8	161	5.38	166	6104	768
Total	**223**	**219**	**75224**	**218**	**80355**	**216**	**89193**	**219**	**61679**	**218**	**77242**	**222**	**2040**	**14124**	**16164**	**208036**	**54654**

SAT Experiment: Table I and Table II show runtime in seconds for MINCE versus the dynamic MSTS, MSOS, DLCS and DLIS orderings, as well as the static *Original* variable-ordering [27]. "#I" denotes the number of instances solved by each decision heuristic. The average *variable cut* is included for the original and the MINCE variable orders. As the data clearly illustrate, deciding on closely-connected variables leads to a reduction in search time. Since "connected" variables are ordered next to each other, this approach allows the solver to quickly identify and avoid unpromising partial solutions. In other words, instead of deciding on variables from separate partitions, one partition is considered at a time. This approach is more effective on *structured* problems, such as the *hole-n* or the *n-queens* problem, which consist of multiple partitions. On these problems, MINCE finds variable orders compatible with the problem's structure, and this speeds up SAT solvers and BDD engines. For example, a speedup of 16, 16, 16, 14, and 9, was obtained for the *hole-10* benchmark over the MSTS, MSOS, DLCS, DLIS, and *Original* decision heuristics, respectively. MINCE also achieved significant speed-ups over other decision heuristics for large instances from the *n-queens* set. Particularly, none of the dynamic or *Original* decision heuristic were able to solve the *nqueens-35* instance in 10K seconds, but it was solved in less than 320 seconds using MINCE.

In general, GRASP run time is almost always reduced when the recursive bisection ordering is used. However, for particularly easy†† SAT instances recursive bisection itself requires more time than GRASP‡‡ with either *Original*, MSTS, MSOS, DLCS, or DLIS ordering. Observe that MINCE has a worst- and best-case performance of $\Theta(N\log^2 N)$. On the other hand, SAT problems have an exponential worst-case and a best-case of $\Theta(N)$, where N is the number of variables and clauses in the problem. Hence, MINCE *should be useful in solving instances that otherwise require exponential runtime,* but unhelpful in solving easy instances.

To explore the variability of orders returned by independent random starts of MINCE, we applied GRASP to three different orders of benchmark *ii32d3.cnf*, generated by MINCE. Two cases time out in 10K seconds and the third was solved in 14.8 seconds. This empirically confirms the heavy-tail distribution theory for SAT instanc-

†† We define *easy* instances as those that can be solved in near-linear time.

‡‡ Same is expected with comparable and faster solvers, e.g. Chaff [21].

TABLE II: GRASP runtimes for selected benchmarks from the DIMACS set and the *n-queens* problem.

Selected Instances	MSTS Time	MSOS Time	DLCS Time	DLIS Time	Original Time	MINCE Order	+Solve	=Total
aim100-2_0-no-2	0.04	0.02	**0.01**	**0.01**	**0.01**	0.72	0.01	0.73
bf0432-007	1.72	3.85	1.74	**1.48**	10K	7.34	1.6	8.94
hanoi4	4.54	2267	10K	10K	**1.75**	8.8	1.66	10.5
hole8	6879	10K	140	70.3	61	**0.44**	**6.44**	**6.88**
hole9	10K	10K	1556	752	623	**0.53**	**52.8**	**53.3**
hole10	10K	10K	10K	8637	5597	**1.94**	**599**	**601**
ii16b1	174	217	10K	10K	4840	**90.7**	**0.47**	**91.2**
ii16b2	133	153	71.3	238	5507	**53.2**	**1.38**	**54.6**
ii32c4	650	696	24.9	**1.24**	10K	84.3	6.11	90.4
par16-2-c	1321	1325	2469	3570	184	**3.16**	**110**	**113**
par16-5	7329	7348	315	10K	111	**11.3**	**14.4**	**25.7**
pret150_25	0.15	0.13	0.14	**0.12**	**0.12**	0.62	0.14	0.76
ssa0432-003	**0.04**	0.04	0.06	0.05	0.06	1.91	0.03	1.94
ssa2670-141	95.2	9.78	2.15	**1.1**	10K	6.9	1.41	8.3
Nqueens-20	482	1485	**23**	24.9	3160	40	0.31	40.3
Nqueens-25	10K	10K	178	183	94.9	**93**	**0.79**	**93.4**
Nqueens-30	10K	10K	5233	5402	10K	**217**	**2.27**	**219**
Nqueens-35	10K	10K	10K	10K	10K	**317**	**1.06**	**318**

es [12] and suggests that a solution can be produced in 60 seconds if multiple starts of the SAT solver are launched with a 20-sec time-out (MINCE took 55 seconds per order on that instance, which is negligible compared to a 10K time-out).

Although not presented in the tables of results, we tested the given benchmarks using the SATO SAT solver [35]. SATO implements an intelligent dynamic decision heuristic and was able to solve the given DIMACS benchmarks in approximately 45,000 seconds (4 instances timed-out after 10,000 seconds) as opposed to 16,200 seconds using GRASP with recursive bisection ordering. However, for some instances, SATO was faster. MINCE failed to generate effective variable-orderings for these instances, since most of them were not structured.

Our preliminary experiments with the recently published Chaff SAT solver [21] indicate that MINCE is not helpful on most standard benchmarks. This, in part, is due to the highly optimized implementation of Chaff, but is also explained by the relative simplicity of the instances. Indeed, if an instance of an NP-complete problem is solved in near-linear time by a generic algorithm, this instance must be easy. Note, however, that while the worst-case complexity of both GRASP and Chaff is exponential, MINCE always runs in near-linear time, perhaps with a greater constant. Finally, even when MINCE's runtime makes it prohibitively expensive for a particular SAT instance where it reduces a solver's runtime, capturing the instance structure may lead to a better understanding and be useful for practical purposes.

CNF to BDD Experiment: Table III and Table IV show the BDD construction runtimes for circuit consistency functions of the ISCAS89 circuit benchmarks. Note that this is not representative of symbolic state traversal, but is a standard experimental procedure for evaluating BDD packages [15]. The table shows runtimes (sec) and the BDD sizes, which represent the maximum number of seen nodes (K) at any point during the construction of the BDD, using the *original*, *original* with *sifting*, MINCE, and MINCE with *sifting* variable orderings, respectively. In addition, for each variable ordering, three clause orderings heuristics are used: *original*, *bottom-up*, MINCE min-cut clause order. Clearly, the MINCE variable ordering leads to faster and smaller BDDs. In terms of circuits, this can be explained by MINCE ordering the

TABLE III: Statistics for constructing the BDDs of the ISCAS89 CNF Benchmarks without Sifting

Var. Order ->	Original						MINCE						MINCE	
Clause Order ->	Original		Bottom-Up		MINCE		Original		Bottom-Up		MINCE		VAR	CL
Instance	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Time
s27	0.08	181	0.13	256	0.12	181	0.09	73	0.06	89	0.13	73	0.26	0.25
s208.1	time-out		out-of-mem		148	1.1M	0.54	2388	0.18	1289	0.29	1313	0.59	0.86
s298	time-out		out-of-mem		646	6.9M	1.27	8213	1.16	13K	3.44	9574	0.61	1.27
s344	out-of-mem		out-of-mem		time-out		11.6	25K	1.28	18K	1.43	11K	0.92	1.57
s349	time-out		out-of-mem		time-out		6.14	17K	1.2	16K	4.75	11K	0.89	1.5
s382	out-of-mem		out-of-mem		time-out		6.71	29K	1.12	19K	6.59	15K	0.8	1.65
s386	out-of-mem		out-of-mem		116.72	365K	79.6	353K	11.6	109K	5.48	24K	1.23	1.64
s400	out-of-mem		out-of-mem		out-of-mem		4.02	12K	1.47	8981	2.28	9229	1.12	1.81
s420	out-of-mem		out-of-mem		time-out		4.32	18K	1.08	7345	4.05	7553	0.92	1.87
s444	out-of-mem		out-of-mem		time-out		3.76	11K	1.84	13K	2.28	10K	0.9	1.7
s510	time-out		out-of-mem		time-out		time-out		8.95	50K	321	2.3M	2.95	2.13
s526	out-of-mem		out-of-mem		time-out		29.5	59K	4.63	34K	11.4	58K	1.29	2.44
s526n	out-of-mem		out-of-mem		time-out		9.88	24K	4.51	31K	18.4	24K	1.66	2.11
s641	out-of-mem		time-out		time-out		186	260K	42.4	228K	60.6	390K	1.63	3.02
s713	out-of-mem		time-out		time-out		107	267K	32	161K	181	167K	1.79	3.2
s832	out-of-mem		out-of-mem		time-out		time-out		time-out		120	172K	6.71	3.64
s838	out-of-mem		out-of-mem		time-out		25.7	54K	19	88K	36	45K	1.96	3.77
s838.1	out-of-mem		out-of-mem		out-of-mem		85.4	93K	3.67	14K	38	15K	2.29	4.17
s953	time-out		out-of-mem		time-out		time-out		408	1.8M	time-out		2.65	3.88
s1196	out-of-mem		out-of-mem		time-out		time-out		358	2M	time-out		4.7	5.9
s1238	out-of-mem		out-of-mem		time-out		time-out		556	2M	time-out		4.9	5.6
Total	**0.08**	**181**	**0.13**	**256**	**911**	**8.4M**	**562**	**1.2M**	**1457**	**6.5M**	**818**	**3.3M**	**40.5**	**53.7**
#Built	**1**		**1**		**4**		**16**		**20**		**18**			

TABLE IV: Statistics for constructing the BDDs of the ISCAS89 CNF Benchmarks with Sifting

Var. Order ->	Original						MINCE						MINCE	
Clause Order ->	Original		Bottom-Up		MINCE		Original		Bottom-Up		MINCE		VAR	CL
Instance	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Time
s27	0.07	181	0.13	256	0.07	181	0.08	73	0.07	89	0.08	73	0.26	0.25
s208.1	14.7	6420	24	8137	2.75	2185	1.18	2385	0.18	1289	0.24	1313	0.59	0.86
s298	47	28K	565	295K	5.66	5007	22	10K	7.76	9981	13.9	16K	0.61	1.27
s344	524	130K	time-out		7.8	6454	479	88K	9.9	8720	8.09	8437	0.92	1.57
s349	268	83K	time-out		8.25	5913	215	75K	6.05	9241	9.97	9738	0.89	1.5
s382	158	88K	272	118K	9.85	7564	82	37K	11.2	8303	3.79	4436	0.8	1.65
s386	256	97K	403	131K	19.1	11K	232	99K	47.2	22K	21.9	14K	1.23	1.64
s400	564	194K	330	68K	8.24	6858	76	24K	7.64	6845	33.1	18K	1.12	1.81
s420	361	94K	time-out		12.5	7419	876	420K	6.63	6902	18.9	12K	0.92	1.87
s444	252	85K	time-out		10.5	7515	500	188K	9.45	6557	10	12K	0.9	1.7
s510	time-out		time-out		220	141K	time-out		51.1	18K	89	46K	2.95	2.13
s526	time-out		time-out		28.3	17K	time-out		26.5	19K	43.8	18K	1.29	2.44
s526n	time-out		time-out		19.6	11K	time-out		26	14K	22.3	10K	1.66	2.11
s641	time-out		time-out		283	78K	time-out		386	96K	time-out		1.63	3.02
s713	time-out		time-out		568	132K	time-out		484	127K	467	127K	1.79	3.2
s832	time-out		time-out		time-out		time-out		time-out		498	115K	6.71	3.64
s838	time-out		time-out		258	64K	time-out		181	35K	269	73K	1.96	3.77
s838.1	time-out		time-out		97.2	17K	time-out		78	14K	70	19K	2.29	4.17
s953	time-out		time-out		time-out		time-out		time-out		time-out		2.65	3.88
s1196	time-out		time-out		time-out		time-out		time-out		time-out		4.7	5.9
s1238	time-out		time-out		time-out		time-out		time-out		time-out		4.9	5.6
Total	**2445**	**806K**	**1594**	**620K**	**1559**	**521K**	**2487**	**944K**	**1340**	**405K**	**1580**	**506K**	**40.5**	**53.7**
#Built	**10**		**6**		**17**		**10**		**17**		**17**			

gates to minimize the "total length of wires". Using the *original* clause ordering, MINCE enabled the BDD construction for all 16 ISCAS89 circuits as opposed to only 10 with sifting and 1 with a *original* variable-ordering. MINCE's variable ordering time is negligible in most cases. MINCE reduced the average *variable cut* for the ISCAS89 circuits from 200 to 26.

In addition, combining the *bottom-up* or MINCE clause ordering heuristics with MINCE variable ordering allows the construction of more circuits as opposed to using the *original* clause ordering. This is justified by the fact that, sorting the clauses helps in localizing the BDD manipulations to specific levels of the BDD and keeps the BDD depth as small as possible which helps in achieving better BDD construction runtime and memory results. The tables also clearly show that enabling sifting will, in general, produce BDDs with a fewer number of nodes, but will require an extra overhead in runtime. Despite the fact that our tables show a higher number of instances solved without sifting, we believe most instances will be solved with sifting given a longer runtime limit. When comparing the bottom-up with MINCE clause ordering heuristic, bottom-up is successful in constructing more circuits than MINCE, however for some instances, such as the $s832$, MINCE is able to build the BDD whereas the bottom-up approach fails. We should note that MINCE is a randomized algorithm. Different runs can produce different clause orderings which can lead to better solutions. On average, the new clause ordering heuristics combined with MINCE variable ordering are able to obtain significant performance improvement in comparison with the original variable and clause orderings. The technique is simple and easy to use in practice. Its static nature allows for a variety of applications where dynamic approaches fail.

Circuit to BDD Experiment: Table V and Table VI summarize the runtime and memory results for the third set of experiments which construct 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* [11], *Malik-level* [19], *Malik-fanin* [19], and MINCE orderings using the *Circuit HG* and the *Dual HG*, respectively. The tables also include the runtime needed by CAPO to generate the gate orderings. As the data clearly illustrate, FUJ and MAL-fan successfully construct the most number of circuits among the previous 6 PI ordering heuristics. However, in the non-sifting case, 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. As discussed earlier, building BDDs with sifting generally uses fewer BDD nodes but requires more runtime. This can be illustrated by our results with the Dual HG model, where all 10 instances were solved in 21 seconds without sifting as opposed to 140 seconds with sifting. On the other hand, the total BDD size is only 33K nodes for the sifting experiment, whereas it needs 211K nodes in the non-sifting experiment. We believe the proposed static ordering should be very effective with applications that do not allow dynamic sifting. We are currently looking into further improving the performance by running multiple independent starts of MINCE. The main

TABLE V: Statistics for constructing the BDDs of the ISCAS85 Circuits without Sifting

Inst-ance	Original		BFS		DFS		FUJ		MAL-Lev		MAL-Fan		MINCE				
													CAPO	Circuit HG		Dual HG	
	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Time	Size	Time	Size
c17	0	7	0	7	0	7	0	7	0	7	0	5	0.21	0	6	0.01	6
c432	0.03	523	1.4	9199	0.75	6625	0.76	6625	0.76	6625	0.83	6880	0.66	0.04	733	0.04	777
c499	0.42	4945	0.72	7203	0.4	6100	0.4	6092	0.4	6100	0.35	5804	1.72	0.5	3405	0.35	4317
c880	6.37	111K	10.5	368K	9.15	245K	9.11	245K	8.47	212K	9.76	220K	1.48	2.86	83K	0.17	5696
c1355	1.12	4945	2.43	6255	1.1	6100	1.09	6092	1.1	6100	0.99	5804	1.89	1.65	4581	1.12	3390
c1908	0.84	8519	0.35	2437	0.51	4808	0.52	4808	0.44	4790	0.5	4837	3	0.73	3459	0.9	7905
c2670	out-of-mem		44.9	5.1M	42.1	4.9M	37.6	4.4M	31.5	3.7M	27.7	2.9M	4.99	2.63	101K	1	23K
c3540	24.6	329K	out-of-mem		205	2M	197	2M	191	1.8M	162	1.6M	7.72	out-of-mem		14.4	118K
c5315	out-of-mem		out-of-mem		time-out		time-out		out-of-mem		time-out		11.3	2.13	15K	2.05	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	4.8	39K	1.71	6682
Total	**33.4**	**459K**	**60.4**	**5.5M**	**258**	**7M**	**247**	**6.6M**	**233**	**5.6M**	**202**	**4.7M**	**58.8**	**15.3**	**251K**	**21**	**211K**
#Built	**7**		**7**		**8**		**8**		**8**		**8**			**9**		**10**	

TABLE VI: Statistics for constructing the BDDs of the ISCAS85 Circuits with Sifting

Inst-ance	Original		BFS		DFS		FUJ		MAL-Lev		MAL-Fan		MINCE				
													CAPO	Circuit HG		Dual HG	
	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Size	Time	Time	Size	Time	Size
c17	0	7	0	7	0	7	0	7	0	7	0	5	0.21	0.01	6	0.01	6
c432	0.26	386	1.08	1715	0.8	450	0.81	450	0.8	450	1.16	566	0.66	0.27	411	0.25	541
c499	12.1	3957	12.1	5030	12.4	4483	13.5	4131	12.5	4483	12.2	4836	1.72	20.4	3661	10.2	2609
c880	4.43	8176	11.7	10K	4.07	3213	4.08	3213	2.37	3433	3.84	2852	1.48	12.6	36K	1.96	2411
c1355	45.5	4649	112	6862	64.5	4098	42.5	4019	64.4	4098	34.3	4451	1.89	46.8	3285	58.8	2397
c1908	5.17	1835	6.67	1721	7.51	2327	7.54	2327	8.95	3147	5.43	1645	3	6.36	1669	9.1	2444
c2670	7.51	2142	7.13	3079	11.4	7099	8.08	10K	13.7	12K	8.89	8178	4.99	7.43	4212	6	2385
c3540	55.3	15K	25	14K	25	14K	25.1	14K	25.7	14K	27.1	11K	7.72	50	16K	27.2	11K
c5315	3.34	791	3.68	2518	4.4	1821	3.94	2077	3.86	2077	4.21	823	11.3	2.87	2391	2.96	1627
c6288	time-out		time-out		time-out		time-out		time-out		time-out		9.53	time-out		time-out	
c7552	26	3450	23.8	6178	24.3	6267	24.3	6267	28	5890	32	4133	16.3	14.8	2160	23.6	8132
Total	**160**	**40K**	**203**	**51K**	**154**	**44K**	**130**	**47K**	**160**	**49K**	**129**	**38K**	**58.8**	**162**	**70K**	**140**	**33K**
#Built	**10**		**10**		**10**		**10**		**10**		**10**			**10**		**10**	

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

6 Conclusions and Future Work

Our work proposes a static variable-ordering heuristic MINCE for CNF formulae with applications to SAT and BDDs. The main advantage of this heuristic is its very good performance on standard benchmarks in terms of implied runtime of SAT solvers as well as memory/runtime of BDD primitives. We believe that this is due to the fact that the proposed variable-ordering is *global* and relies on high-performance hypergraph partitioning and placement (MLPart [7] and CAPO [8]). Unlike problem-specific dynamic variable-ordering heuristics, such as DLCS, DLIS, and variable-sifting, MINCE can be implemented once and used for different applications without modifying the application code. Given that MINCE shows strong improvements in seemingly unrelated applications (SAT and BDD) and for a wide variety of standard benchmarks, we believe that it is able to capture structural properties of CNF instances and cir-

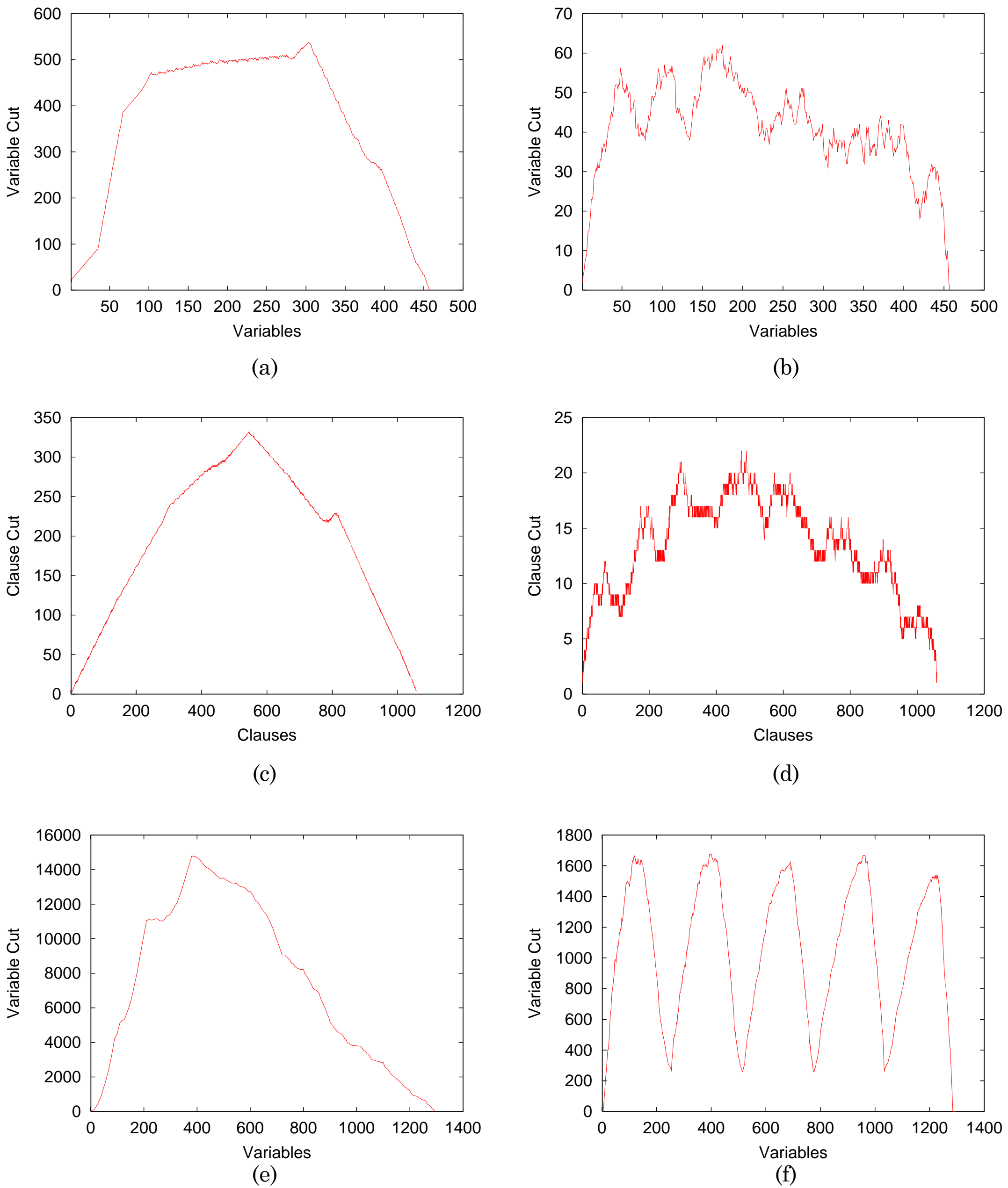


Fig. 8. Cutwidth profiles for the circuit hypergraph of the ISCAS89 *s838* instance using the Original and MINCE variable order for the (a-b) CNF hypergraph (c-d) Dual CNF hypergraph. Cutwidth profile for the *9symml_gr_rcs_w5* FPGA routing instance using the Original and MINCE variable order for the CNF hypergraph (e-f)

cuits. We show that when "connected" variables, clauses, or gates are ordered next to each other, SAT and BDD operations can achieve better performance. For example, when a CNF formula is created from a circuit, it is not difficult to see that MINCE essentially performs recursive partitioning and linear placement of this circuit, and then orders variables so that respective circuit elements are located near each other on average. One particular example is shown in Figure 8 where cut-profiles of a particular

circuit-derived CNF instance with the original and MINCE variable orderings are compared (a clause is “cut” by all variables ordered between its left- and right-most variables). More significantly, MINCE reduces all cuts and exposes design hierarchy of the original circuit. Figure 8 also shows the cutwidth profiles for the original and MINCE variable orderings of an FPGA routing instance. Interestengly, the five variable clusters in Figure 8(f) represent a one-to-one correspondence with the routing tracks in the FPGA interconnect fabric. In general, this technique should have better impact on BDDs, since they are more sensitive to variable-ordering than SAT. SAT solvers can reduce the damage incurred by a bad variable-ordering using the addition of conflict-induced clauses (a conflict clause connects literals of related variables even if they are very far from each other in the ordering).

We note that our use of a finely-tuned standard-cell placer CAPO results in better average cuts and clause spans than one expects from a “vanilla” recursive bisection (e.g., as commonly implemented with hMetis). This black-box software reuse is enabled by the pure preprocessing nature of the proposed techniques (we use GRASP as a black-box too). We hope that this will also enable its easy evaluation and adoption in the industry.

Our on-going work addresses additional types of benchmarks, better justifications of the MINCE heuristic and also analyses of the cases when it fails to produce near-best variable-orderings. An important research question is to account for polarities of literals. We are aware of work conducted in [32] which is similar to ours. Our colleagues use hMetis, modify the source-code of GRASP and attempt to account for polarities of literals by post-processing. Comparisons of preliminary results show that MINCE is surprisingly successful without using polarities of literals. We are also looking into further improving the runtimes by detecting symmetries in the problem’s structure. A public-domain implementation of MINCE is available at *http://andante.eecs.umich.edu/mince*.

7 Acknowledgments

This work is funded by the DARPA/MARCO Gigascale Silicon Research Center. Preliminary results of this work were reported at IWLS 2001 and ICCAD 2001. We thank IWLS participants and anonymous reviewers at ICCAD for many valuable suggestions that helped improve this paper.

8 References

[1] F. Aloul, I Markov, and K. Sakallah, “Faster SAT and Smaller BDDs via Common Function Structure,” *in Proc. International Conference on Computer Aided Design* (ICCAD), 2001.

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

[3] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, “Symbolic Model Checking using SAT procedures instead of BDDs,” *in Proc. Design Automation Conference* (DAC), 1999.

[4] F. Brglez, D. Bryan, and K. Kozminski, “Combinational problems of sequential benchmark circuits,” *in Proc. International Symp. on Circuits and Systems,* 1989.

[5] F. Brglez, and H. Fujiwara, “A neutral netlist of 10 combinational benchmark cir-

cuits and a target translator in FORTRAN," *in Proc. International Symposium on Circuits and Systems,* 1985.

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

[7] A. Caldwell, A. Kahng, and I. Markov, "Can Recursive Bisection Produce Routable Placements?" *in Proc. Design Automation Conference* (DAC), 2000.

[8] 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*, 2000.

[9] R. Drechsler and B. Becker, "Binary Decision Diagrams, Theory and Implementation," *Kluwer Academic Publishers*, 1998.

[10] DIMACS Challenge benchmarks in *ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf.*

[11] M. Fujita, H. Fujisawa, and N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams," *in Proc. International Conference on Computer Aided Design* (ICCAD), 1988.

[12] C. Gomes, B. Selman, and H. Kautz, "Boosting Combinatorial Search Through Randomization," *in Proc. National Conference on Artificial Intelligence* (AAAI), 1998.

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

[14] S. Hur and J. Lillis, "Relaxation and clustering in a local search framework: application to linear placement," *in Proc. Design Automation Conference* (DAC), 1999.

[15] G. Janssen, "Design of a Pointerless BDD Package," *in International Workshop on Logic Synthesis* (IWLS)*,* 2001.

[16] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design," *in Proc. Design Automation Conf.* (DAC), 1997.

[17] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, 11(1), 1992.

[18] Y. Lu, J. Jain, and K. Takayama, "BDD Variable Ordering Using Window-based Sampling," *in International Workshop on Logic Synthesis* (IWLS)*,* 2000.

[19] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment," *in Proc. International Conference on Computer Aided Desig*n (ICCAD), 1988.

[20] S. Minato, N. Ishiura, and S.Yajima, "Shared binary decision diagrams with attributed edges for efficient Boolean function manipulation," *in Proc. Design Automation Conf.* (DAC), 1990.

[21] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *in Proc. Design Automation Conference* (DAC), 2001.

[22] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," *in Proc. International Symposium on Physical Design* (ISPD), 2001.

[23] S. Panda and F. Somenzi, "Who are the variables in your neighborhood," *in Proc. International Conference on Computer Aided Design* (ICCAD), 1995.

[24] M. Prasad, P. Chong, and K. Keutzer, "Why is ATPG easy?" *in Proc. Design Automation Conference* (DAC), 1999.

[25]R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” *in Proc. International Conference on Computer Aided Design* (ICCAD), 1993.

[26]J. Marques-Silva and K. Sakallah, “Robust Search Algorithms for Test Pattern Generation,” *in Proc. IEEE Fault-Tolerant Computing Symposium*, 1997.

[27]J. Silva and K. Sakallah, “GRASP-A New Search Algorithm for Satisfiability,” *in Proc. International Conference on Computer Aided Design* (ICCAD), 1996.

[28]L. Silva, J. Silva, L. Silveira, and K. Sakallah, “Timing Analysis Using Propositional Satisfiability,” *in IEEE International Conference on Electronics, Circuits and Systems*, 1998.

[29]F. Somenzi, “Colorado University Decision Diagram package,” *http://vlsi.colorado.edu/~fabio/CUDD*, 1997.

[30]G. Stalmarck, “System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from Boolean Formula,” *United States Patent no.* 5,276,897, 1994.

[31]P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Combinational Test Generation Using Satisfiability,” *in IEEE Transactions on Computer-Aided Design*, 1996.

[32]D. Wang and E. Clarke, “Efficient Formal Verification through Cutwidth,” in *IEEE International High Level Design Validation and Test Workshop* (HLDVT), 2001.

[33]R. Wood and R. Rutenbar, “FPGA Routing and Routability Estimation Via Boolean Satisfiability,” *in IEEE Transactions on VLSI*, 6(2), 1998.

[34]C. Yang and M. Ciesielski, “BDS: A BDD-Based Logic Optimization System,” *in Proc. Design Automation Conference* (DAC), 2000.

[35]H. Zhang, “SATO: An Efficient Propositional Prover,” *in International Conference on Automated Deduction*, 1997.