Journal of Universal Computer Science, vol. 10, no. 12 (2004), 1562-1596
submitted: 9/10/04, accepted: 14/8/04, appeared: 28/12/04 © J.UCS

MINCE: A Static Global Variable-Ordering Heuristic for SAT Search and BDD Manipulation

Fadi A. Aloul
(American University of Sharjah, U.A.E.
faloul@ausharjah.edu)

Igor L. Markov
(University of Michigan, U.S.A.
imarkov@umich.edu)

Karem A. Sakallah
(University of Michigan, U.S.A.
karem@umich.edu)

Abstract: The increasing popularity of SAT and BDD techniques in formal hardware verification and automated synthesis of logic circuits encourages the search for additional speedups. 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 modifying the source code of SAT/BDD tools.

Finding a better variable ordering is a well recognized problem for both SAT solvers and BDD packages. Currently, the best 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 algorithm MINCE (MIN Cut Etc.) that pre-processes a given Boolean formula in CNF. MINCE is completely independent from target SAT algorithms and in some cases outperforms both the variable state independent decaying sum (VSIDS) decision heuristic 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 SAT/BDD techniques by a factor of two or more. Our results motivate the search for better dynamic ordering heuristics and combined static/dynamic techniques.

Keywords: SAT, CNF, BDDs, backtrack search, decision heuristics, variable ordering, formal verification, partitioning.

Category: I.1.2, I.2.8

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 [Goldberg and Novikov, 2002], [Moskewicz et al., 2001], [Silva and Sakallah, 1996], [Stålmarck, 1994], [Zhang, 1997] and algorithms for manipulating Binary Decision Diagrams (BDDs) [Bryant, 1986], [Drechsler and Becker, 1998], [Lu et al., 2000]. 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 [Larrabee, 1992], [Stephan et al., 1996], formal verification of circuit functions [Biere et al., 1999], timing verification of circuits [Silva et al., 1998], and routing of field-programmable gate arrays [Nam et al., 2001], among others. While no exact polynomial-time algorithms are known for the general case, many exact algorithms [Goldberg and Novikov, 2002], [Moskewicz et al., 2001], [Silva and Sakallah, 1996], [Stålmarck, 1994], [Zhang, 1997] 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 [Goldberg and Novikov, 2002], [Moskewicz et al., 2001], [Silva and Sakallah, 1996], [Stålmarck, 1994], [Zhang, 1997], as well as our empirical data, imply that the order of these steps critically affects the run time of state-of-the-art 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 [1] [Bryant, 1986], [Bryant, 1992], [Drechsler and Becker, 1998] 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 in the BDD, potentially leading to exponential memory and run time requirements. Several BDD ordering techniques have been proposed to overcome this problem. These include static [Fujita et al., 1988], [Malik et al., 1988] and dynamic [Panda and Somenzi, 1995], [Rudell, 1993], [Somenzi, 2001] approaches. Just as for SAT solvers, the order of BDD variables 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 BDD construction process.

[1] Only Reduced Ordered Binary Decision Diagrams (ROBDDs) are considered in this work.

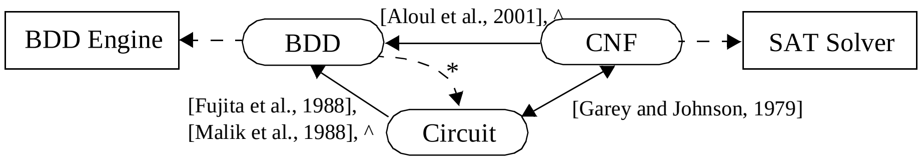


*Figure 1: Conversions between Compact Representations of Boolean Functions. (^ stands for this work and * stands for [Yang and Ciesielski, 2002])*

A reliable and fast variable-ordering heuristic for a given application can dramatically affect its competitiveness and is often considered an important part of its implementation. For example, the leading-edge SAT solver Chaff [Moskewicz et al., 2001] is typically used with the dynamic *variable-ordering heuristic variable state independent decaying sum* (VSIDS), and the renowned CUDD package [Somenzi, 1997] for BDD manipulation incorporates the dynamic variable-sifting heuristic [Rudell, 1993] 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 run time. Sifting for BDDs is typically more expensive than most dynamic ordering heuristics for SAT. However, the effect of ordering heuristics on total run time is highly instance-specific.

We noticed that, for some CNF formulae in [Tab. 2] and [Tab. 7] in [Section 5], turning off sifting for BDD manipulations and turning off VSIDS in SAT resulted in significantly smaller run times. For BDDs, this also led to memory savings, especially for circuit benchmarks from the ISCAS'89 set [Brglez et al., 1989]. In other words, using a good order of variables when encoding problems into a CNF formula is, by itself, superior to using the best known dynamic heuristic with a poor static order of variables (note that static and dynamic ordering heuristics can be used together). 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 problem 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 formulae, 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 CNFs and circuits in [Section 2.2] and [Section 2.3], respectively [see Fig. 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 the application of partitioning-based variable ordering 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 the formula's clauses corresponds to an implicate of *f* [Hachtel and Somenzi, 2000]. 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.

Backtrack search algorithms [Davis et al., 1962] 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:

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

Many 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 described [Goldberg and Novikov, 2002], [Moskewicz et al., 2001], [Shacham and Zarpas, 2003], [Silva and Sakallah, 1996], [Wang, 1997]. Some are based on an analysis of the number of variables and clauses in the problem, such as DLCS [Silva and Sakallah, 1996] (select the variable that appears in the maximum number of unresolved clauses) or DLIS [Silva and Sakallah, 1996] (select the literal that appears in the maximum number of unresolved clauses). Other decision heuristics are based on randomized algorithms.

2.2 Construction of BDDs from CNF

A CNF formula can be viewed as a two-level logic circuit, in which each clause is represented by an OR gate where the number of fanins (i.e., gate inputs) is equal to the number of literals in the clause. The outputs of all OR gates are ANDed together to produce the function *f*. Additionally, circuit consistency functions can be represented in CNF in linear time [Ryan], [Tseitin, 1983]. Each gate in the circuit is represented using a CNF formula that denotes the valid input-output assignments to the gate. The CNF formula for the circuit consists of the conjunction of the formulae representing each gate. [Tab. 1] shows the CNF formulae for simple gates.

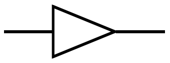
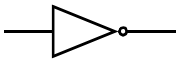
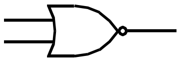
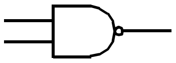
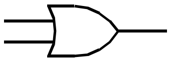
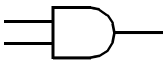
Gate Type	Gate Function	Equivalent CNF Formula
	$z = Buf(x_1)$	$(\bar{x}_1 + z) \cdot (x_1 + \bar{z})$
	$z = NOT(x_1)$	$(x_1 + z) \cdot (\bar{x}_1 + \bar{z})$
	$z = NOR(x_1, \ldots, x_j)$	$\left[\prod_{i=1}^{j} (\bar{x}_i + \bar{z})\right] \cdot \left(\sum_{i=1}^{j} x_i + z\right)$
	$z = NAND(x_1, \ldots, x_j)$	$\left[\prod_{i=1}^{j} (x_i + z)\right] \cdot \left(\sum_{i=1}^{j} \bar{x}_i + \bar{z}\right)$
	$z = OR(x_1, \ldots, x_j)$	$\left[\prod_{i=1}^{j} (\bar{x}_i + z)\right] \cdot \left(\sum_{i=1}^{j} x_i + \bar{z}\right)$
	$z = AND(x_1, \ldots, x_j)$	$\left[\prod_{i=1}^{j} (x_i + \bar{z})\right] \cdot \left(\sum_{i=1}^{j} \bar{x}_i + z\right)$

TABLE 1: CNF formulae representing simple gates.

In general, dynamic sifting [Rudell, 1993], [Somenzi, 1997] is the main variable ordering heuristic used in constructing BDDs from CNFs. In this paper, we show that BDD variable ordering, in addition to the order in which clauses are processed, can be very effective in reducing the run time 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 the execution time for building the BDD depend on the ordering of its variables. A good ordering can lead to a smaller BDD and faster run time, whereas a bad ordering can lead to an exponential growth in the size of the BDD and hence can exceed available memory. Several heuristics have been proposed to order the BDD variables based on the given circuit input information [Fujii et al., 1993], [Fujita et al., 1988], [Malik et al., 1988], [Minato et al., 1990]. 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 (i.e., ascending order of the original indices) 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 that PI is traversed.
- *BFS*: A breadth-first search (BFS) is performed starting from the PO. A PI is appended to the ordering as soon as that PI is traversed.
- *Fujita* [Fujita et al., 1988]: A DFS is performed starting from the PO. PIs with multiple fanouts are appended first to the ordering as soon as these PIs are traversed. Once that is complete PIs with single fanouts are appended to the ordering.
- *Malik-level* [Malik et al., 1988]: POs are assigned level 0. The level of each node g in the circuit is computed by $level(g) = max(level(go) + 1)$, where go represents the fanouts of node g. PIs are appended to the ordering in descending order of their levels. Ties are broken between PIs with the same level by selecting the PI with the smallest index first.
- *Malik-fanin* [Malik et al., 1988]: A DFS is performed starting from the PO. However, unlike previous heuristics, in which ties are broken between gate fanins by selecting the fanin with the smallest index, the transitive fanin (TFI) depth size is used as 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 that PI is 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 [Fujita et al., 1988]. On the other hand, the heuristics of [Malik et al., 1988] give priority to 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 [Rudell, 1993] that entails pairwise swaps of variables and is now considered an integral part of every BDD package [Somenzi, 1997].

In addition to ordering the BDD variables, which represent the PIs in a circuit, the order in which gates are traversed when constructing the BDD can also be varied. After the BDD variables are ordered by one of the ordering techniques explained above, we consider three strategies to order the 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 first [2]. In general, strategy 1 shows the best performance [3].

[2] 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 those from the *index* tie-breaking approach.

3 Instance Partitioning

We first observe that Boolean functions arising in many applications represent spacial, logical or causal 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 variables, the variables in this group are likely to 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 (considered as graphs). BDDs with many small cuts tend to have fewer edges, and therefore fewer vertices (since in decision diagrams every vertex is a source of exactly two edges). This suggests that we interpret CNF formulae as hypergraphs (a hypergraph is a graph whose edges, referred to as hyperedges, can connect two or more vertices) by representing variables by vertices (polarities are ignored) and clauses by edges. Two vertices share (are incident to) an edge if the two corresponding variables share a clause in the formula.

We now look for partitioning formulations that are addressed in existing literature and for which efficient [heuristic] algorithms are available. One such formulation is *balanced min-cut hypergraph partitioning*. It was studied for more than 30 years in the VLSI CAD and database communities; near-linear-time heuristics are available today with excellent empirical performance [Caldwell et al., 2000c], [Karypis et al., 1997]. In min-cut k-way partitioning ($k > 1$ is an integer), one looks to assign each vertex of a given hypergraph to one of k partitions so as to minimize the number of hyperedges (that can be connected to more than two vertices) that connect vertices in more than one partition; this objective is commonly called the *cut*. In this work, as in many other applications, only the case $k = 2$ (bi-partitioning) is important. What makes min-cut bi-partitioning NP-hard is the balance constraint, which requires that the numbers of vertices in the two partitions be approximately equal (precise definitions are given below). Without balance constraints, a reduction to max-flow computation on graphs gives an exact polynomial-time algorithm. Vertex-weighted versions of balanced min-cut partitioning require that the sums of vertex weights in all partitions be approximately equal (vertex weights are typically positive numbers), and hyperedge-weighted versions minimize weighted cut, i.e., the sum of weights of all hyperedges in the cut.

While hyperedge and vertex weights are typically defined in domain-specific terms, the balance constraint itself is motivated by the desire to improve the run time and accuracy of recursive bisection (see the theorems below). In recursive bisection every partition that results from a bisection is further partitioned by a recursive call. Recursion bottoms out when a partition contains one vertex. If implemented properly, (e.g., using terminal propagation), recursive bisection results in a linear ordering of hypergraph vertices and can be considered a heuristic that optimizes several important objectives at once.

[3] When computing the BDD variable orders, POs with the highest level (as computed in Malik-level) are traversed first. When constructing the BDDs, POs are traversed according to the order specified in the input file.

Applying balanced min-cut partitioning to a hypergraph derived from a CNF formula separates the original formula into relatively independent sub-formulae. 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 results in a linear variable ordering. We note that cuts of CNF formulae have been studied in [Prasad et al., 1999], and instances having small cuts were theoretically shown to be easy for SAT [Prasad et al., 1999]. Our work seeks constructive and efficient ways to amplify the easiness of CNF instances with small cuts by finding good variable orderings. Related attempts to leverage partitioning techniques in the context of theorem proving and Boolean satisfiability were made prior to our work [Amir and McIlraith, 2000], however with somewhat different partitioning formulations and, we believe, much weaker partitioning algorithms (based on max-flow) in terms of scalability. Our approach leverages recent high-performance partitioning heuristics that run in near-linear time with near-linear memory requirements. We show that the use of such heuristics with modern SAT solvers is justified in terms of total run time. Aside from using more scalable algorithms and implementations compared to work using network flows [Caldwell et al., 2000a], [Caldwell et al., 2000b], [Caldwell et al., 2000c], [Fiduccia and Mattheyses, 1982], [Karypis et al., 1997], our overall improvements in speed can be explained by (i) our focus on *edge cuts* rather than *vertex separators* (i.e., groups of vertices whose removal makes the graph disconnected) [Amir and McIlraith, 2000], and (ii) performing *balanced* partitioning recursively. The idea of finding and using *balanced vertex separators* was briefly mentioned in [Amir and McIlraith, 2000], but discarded because of the computational difficulties of this approach (similar conclusions were reached several years earlier in the literature on parallel numerical algorithms). Our approach has good empirical performance (no empirical results were reported in [Amir and McIlraith, 2000]) and is formulated as a pre-processing step followed by one call of an existing SAT solver. Given existing software tools for partitioning and Boolean satisfiability, very few changes of these tools is required.

Additionally, we show that min-cut partitioning is relevant to other representations of Boolean functions than CNF formulae. Prior works correlated circuit cutwidth with the size of BDDs [Berman, 1991], [Bryant, 1992]. [Fig. 6(a-c)] show two topological orderings of a small circuit that lead to BDDs of different sizes. For a given ordering of gates and PIs, 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 (see [Fig. 6]) [Bryant, 1992]. 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 [Caldwell et al., 2000b]. Smaller cuts in circuits have been related to smaller BDDs in [Berman, 1991]. Therefore, we will attempt to produce topological orderings that minimize total netlength, by using min-cut placement.

3.1 Recursive Bisection and Hypergraph Placement

Recursive min-cut bisection of hypergraphs has been intensively studied in the context of VLSI partitioning and placement. First heuristics for balanced graph bisection were published in [Kernighan and Lin, 1970] and then significantly improved in [Fiduccia and Mattheyses, 1982], followed by multi-level extensions [Karypis et al., 1997] which not only improve performance but also guarantee near-linear run time in practice. Recursive bisection has been used for circuit layout since the 1960s at IBM and elsewhere—an early account can be found in [Breuer, 1977]. The theory behind min-cut partitioning and especially recursive bisection has been developed more recently in terms of approximation algorithms. Below we follow definitions and arguments from [Vazirani, 2001], except that we ignore non-trivial vertex and edge weights for clarity (they are not used in our application) and only allow for unit-weighted and zero-weight vertices, which means either counting or not counting a given vertex towards partition balances.

Given an undirected hypergraph $G = (V, E)$ and a rational number $b(0 < b < 1/2)$, balanced min-cut bisection aims to assign each vertex in V to partition 0 or partition 1 to minimize the number of hyperedges connecting vertices in different partitions, subject to $b|V| < |P_0| < (1-b)|V|$, where P_0 is the set of vertices in partition 0. For $b = 1/3$, a polynomial-time approximation algorithm based on linear programming ([Vazirani, 2001], page 193) is guaranteed ([Vazirani, 2001], Claim 21.28) to find cuts within $O(\log|V|)$ factor of the minimum cut. While no run time analysis is available in [Vazirani, 2001], it is unlikely that this algorithm will run in near-linear time. In fact, it may be slower than $O(|V|^2)$, which would make it impractical for large-scale applications. Therefore, high-performance heuristics are used in practice [Caldwell et al., 2000a], [Caldwell et al., 2000c], [Fiduccia and Mattheyses, 1982], [Karypis et al., 1997], even though no worst-case analysis is available to describe their performance. However, they are believed to typically produce near-optimal solutions on instances important in common applications.

The min-cut linear arrangement problem [Vazirani, 2001] of an undirected hypergraph $G = (V, E)$ is concerned with finding vertex orderings, i.e., it assigns each vertex a unique integer index from $1 \ldots |V|$, that minimizes the total wirelength. For every i from $1 \ldots |V| - 1$, we define the i^{th} cut with respect to a given vertex ordering as the number of hyperedges that connect vertices with numbers both less than and greater than $i + 0.5$. The goal is to minimize the maximal cut (out of $|V| - 1$ cuts) by choosing a good vertex ordering. While NP-hard, this problem is approximable up to a factor of $O(\log^2|V|)$ in polynomial time, e.g., using approximation algorithms for the balanced min-cut partitioning problem [Vazirani, 2001] within the framework of recursive bisection.

The main algorithmic subtlety of recursive bisection for linear ordering is in the treatment of connections to outer partitions. For example, suppose we partition the original hypergraph into P_0 and P_1, then connections from P_0 to P_1 require special treatment when P_0 itself is partitioned. Vertices in P_1 that are incident to such connections are modelled by fake zero-weight vertices fixed in the sub-partition of P_0 that is closer to P_1 (fake vertices are removed after partitioning and are only used to influence the assignments of actual vertices). This technique is sometimes called “terminal propagation” [Dunlop and Kernighan, 1985]. Practically every known algorithm for min-cut partitioning can be adapted to accommodate fixed vertices.

The argument proving the $O(log^2|V|)$-approximation of recursive bisection assuming that each balanced bisection is performed with accuracy $O(log\ |V|)$ goes as follows (see [Vazirani, 2001], Claim 21.30 for details). Suppose the minimum possible max-cut of a linear arrangement is c. Then the number of newly-cut hyperedges after every balanced bisection is guaranteed to be within $O(log\ |V|)c$, regardless of the previous cuts. Because every bisection is approximately balanced, the depth of recursion is no more than $O(log\ |V|)$. The partition balance b itself does not matter for asymptotics [4], as long as it is bounded away from 0, but in our implementation we require that $b > 1/3$ and choose more balanced partitions among those with equal cuts (the specific ratio is determined empirically and may be changed, further adjusted for different applications). The two worst-case $O(log\ |V|)$ factors prove the $O(log^2|V|)$ approximation property of recursive bisection. In terms of run time, if each balanced min-cut partitioning within recursive bisection is performed in polynomial (near-linear) time, then recursive bisection as a whole also takes polynomial (near-linear, respectively) time. This is because the complexity of carefully implemented recursive bisection can be estimated as the complexity of balanced min-cut partitioning multiplied by a logarithm of the size of the input hypergraph (|E|+|V|).

The recursive bisection procedure described above for CNF formulae has been empirically successful in solving the *linear placement problem*, where hypergraph vertices are placed on a line, i.e., ordered, to minimize the total length of hyperedges (equivalently, the average net length). The length of a hyperedge is defined as the distance between the lowest and the highest numbered incident vertices. If a hypergraph is used to model a circuit (with hyperedges representing signal nets), recursive bisection leads to small “half-perimeter wire-length” (which is equal to the total length of hyperedges). This result has been used in VLSI placement for at least 30 years, and we will use a high-performance large-scale VLSI placer in our experiments. On the other hand, if a hypergraph represents a CNF instance, minimizing the total length of hyperedges can translate 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 $i+0.5$.

[4] When we partition N nodes with a given partition balance b, the larger partition may have up to $(1-b)N$ nodes, and $(1-b)N < N$ as long as $b > 0$. Furthermore, the height of a binary tree of partitions is upper-bounded by $\log_{(1/(1-b))}(N)$, where the base of the logarithm depends on the partition balance b. For example, for $b = 1/2$ we have $\log_2(N)$, for $b = 1/3$, we have $\log_{(3/2)}(N)$, which is greater. Since $\log_a(N) = \ln(N)/\ln(a)$, the base of the logarithm does not affect the asymptotic behavior.

Observation: Given an ordering of variables in a CNF formula, the total clause span equals the sum of all cuts

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

The average clause span is proportional to the average cut,

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

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

$$= \left(\frac{|E|}{|V|} \cdot \frac{|V|}{|V|-1} \cdot AverageSpan\right)$$

and the coefficient $\left(\frac{|E|}{|V|} \cdot \frac{|V|}{|V|-1}\right)$ 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).

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

Since the total net-length of hypergraphs corresponds to the total clause span of CNF formulae, we will use the leading-edge VLSI [hypergraph] placer CAPO [Caldwell et al., 2000b] based on recursive min-cut bisection [Caldwell et al., 2000c], [Karypis et al., 1997] 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 (e.g., $b < 1/3$) and adaptive cut-line selection according to the actual balance achieved, which allows greater freedom in partition sizes in order to improve the cut. The underlying multi-level hypergraph partitioner MLPart [Caldwell et al., 2000c] outperforms the well-known hMetis [Karypis et al., 1997], while both rely on Multi-Level Fiduccia-Mattheyses (MLFM) partitioning heuristics. Since the MLFM heuristic is randomized, it returns different solutions on every call (each call is a *start*). On every call, MLPart executes two independent *starts* and applies one V-cycle [Caldwell et al., 2000c], [Karypis et al., 1997] to further improve the better solution. In the next two paragraphs we briefly review the basic terminology and algorithms related to MLFM.

The Fiduccia-Mattheyses (FM) heuristic [Caldwell et al., 2000a], [Fiduccia and Mattheyses, 1982] starts with an initial solution (e.g., randomly chosen) and applies deterministic linear-time passes that are executed using the same algorithm. No pass can make the cut worse (by construction), and the passes are stopped once no improvement is achieved in a pass. Every pass contains single-vertex moves in which vertices are moved to a different partition. No vertex can be moved twice in a pass, and no move can violate the required partition balance constraints. Every next move is chosen to have the best possible effect on the total cut with the caveat that the best possible move may make the cut worse. A pass ends when no allowed moves are available, and the best partitioning seen during the pass is restored. Linear time per pass is guaranteed by a clever bucketing scheme [Fiduccia and Mattheyses, 1982] for choosing moves. From an optimization point of view, the FM heuristic behaves greedily when it is possible to directly improve the cut, but otherwise temporarily increases the cut in the hope of decreasing it later (performs hill climbing). It is particularly good in moving clusters of vertices from partition to partition, which requires hill climbing and improves performance on structured hypergraphs (e.g., those representing circuit netlists or logic theories), i.e., graphs that each connect very few groups of vertices with hyperedges.

The performance of the FM heuristic starts deteriorating at 500-1000 vertices for sparse hypergraphs, both in terms of run time and solution quality. The run time deterioration is due to numerous passes that improve cut by only several hyperedges, and the solution quality is due to local minima. However, multi-level (ML) extensions [Caldwell et al., 2000c], [Karypis et al., 1997] solve the problem. The main idea is to cluster the hypergraph and partition it instead of the original hypergraph at first. Not only this is a speed improvement, but with a reasonably good clustering it produces decent partitionings for the original hyper-graph, and those partitions can be further decreased by the FM heuristic. The multi-level FM heuristic consists of three phases. At the first phase, the hypergraph is progressively clustered using fast greedy algorithms until the number of clusters reaches a predetermined value, say, 200. At the second phase, the flat (i.e., not multi-level) FM heuristic is applied with a random initial solution to the top-level clustered hypergraph. At the third stage, clusters are refined in steps, and at every step the flat FM heuristic is applied to further decrease cuts (in practice, it performs 2-4 passes per level, which takes linear time). Multi-level partitioning is fast on large hypergraphs because the total number of Fiduccia-Mattheyses passes is limited by [a constant times] the number of levels, which is logarithmic in the number of original vertices. If clustering requires near-linear time (i.e., n times polylogarithm of n) in the size of the input hypergraph, per level, then the total run time of multi-level partitioning must be near-linear.

While the FM heuristic is iterative, its multi-level extension is not—it cannot start with a given partitioning and improve it. This is remedied by V-cycles [Karypis et al., 1997], which are performed given a partitioning. A V-cycle is a variation of MLFM where vertices in different partitions cannot be clustered. The top-level clustered hypergraph automatically comes with a partitioning whose cut equals the cut of the initial partitioning. This is true because bottom-up clustering during a V-cycle does not merge vertices that are assigned to different partitions in the initial solutions. Therefore, each cluster is assigned to a certain partition, the overall partitioning of clusters is balanced, and the cut nets correspond one-to-one to cut nets in the original solution. Since the top-level cut cannot increase during further stages, V-cycle is guaranteed not to make it worse. A V-cycle takes asymptotically as much time as regular multi-level partitioning, but is somewhat faster in terms of actual CPU time because fewer FM passes are required to refine already good solutions.

Wood and Rutenbar have already used linear hypergraph placement as a variable-ordering technique for BDD minimization in [Wood and Rutenbar, 1998]. However, they used spectral methods which entail converting hyperedges to edges and then minimiz-

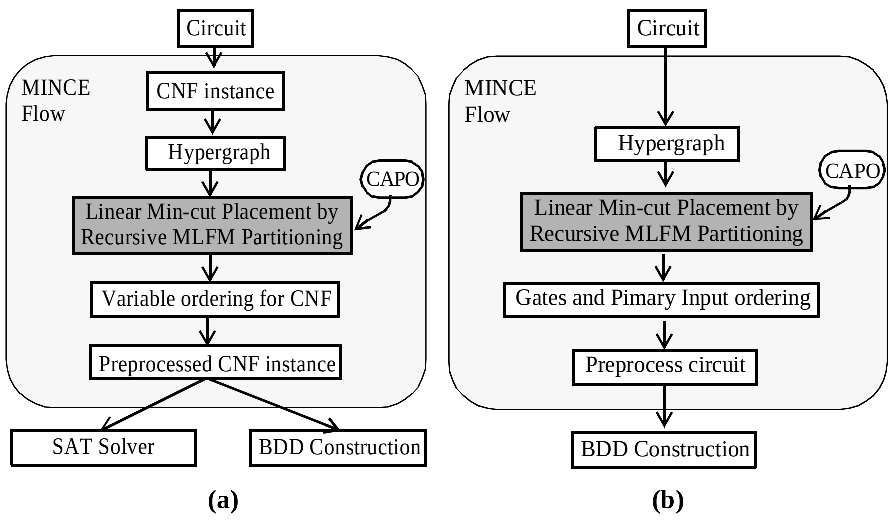

Circuit
MINCE Flow
CNF instance
Hypergraph
CAPO
Linear Min-cut Placement by Recursive MLFM Partitioning
Variable ordering for CNF
Preprocessed CNF instance
SAT Solver
BDD Construction
(a)
Circuit
MINCE Flow
Hypergraph
CAPO
Linear Min-cut Placement by Recursive MLFM Partitioning
Gates and Pimary Input ordering
Preprocess circuit
BDD Construction
(b)

Figure 2: The MINCE heuristic based on Multi-Level Fiduccia-Mattheyses (MLFM) partitioning [Caldwell et al., 2000b], [Caldwell et al., 2000c], [Karypis et al., 1997] *for (a) CNF problems (b) circuits.*

ing quadratic edge length, rather than the half-perimeter (linear) edge length. Spectral placement methods used in [Wood and Rutenbar, 1998] 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 run time 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 Using Partitioning-Based Variable Ordering

We propose to reorder variables in SAT and BDD in a particular way. The algorithm we use is empirically successful in lowering both the max-cut and the average cut.

4.1 Ordering Variables in CNF

We propose the following heuristic that orders variables in CNF formulae [see Fig. 2(a)]. An initial CNF formula (that may originate from circuits or other applications) is converted into a hypergraph [see Fig. 3]. An ordering of hypergraph vertices is then found via min-cut linear placement and translated back into an ordering of CNF variables. The variables of the original CNF formula are reordered and then the formula is used (i) as input to an arbitrary SAT solver, or (ii) to construct a BDD representation

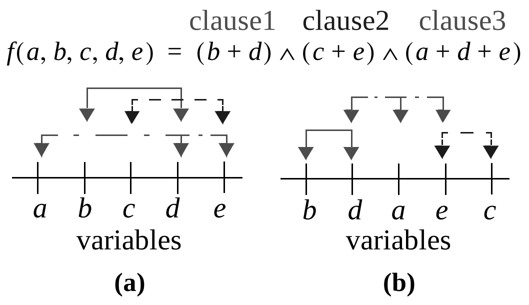


Figure 3: Example showing a hypergraph using (a) default vertex ordering (b) improved vertex ordering. The total span is reduced from 8 to 4 by the improved vertex-ordering. Arrows indicate hyperedges.

of the Boolean function it denotes. The results produced by SAT solvers and BDD manipulations are then translated back into the original variable order.

Note that this approach does not require modifications in SAT solvers, BDD manipulation software or the min-cut placer [5]. We call this heuristic MINCE (MIN-Cut, Etc.) and implemented it by chaining publicly-available software with PERL scripts.

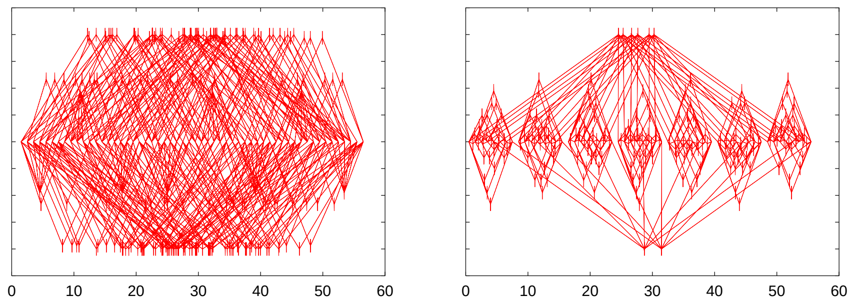
To enable black-box reuse of publicly-available software (CAPO), we ignore polarities of literals in CNF formulae (i.e., assume negative literals are positive). We note that the oriented version of min-cut bisection has been extensively studied in the context of timing-driven placement, i.e., circuit layout techniques that minimize circuit delay along signal paths. In particular, a small unoriented cut can be interpreted as an oriented cut of the same size or less. 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 outperforms variable sifting (used without static ordering) in both run time and memory usage. According to [Hachtel and Somenzi, 2000], as of 2000, variable sifting is the best published dynamic variable reordering heuristic for BDDs with near-linear performance [6]. 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 the 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 [Gomes et al., 1998] (e.g., a new ordering is created before each restart).

[5] Commercial software for Electronic Design Automation can be used, e.g., QPlace - a commercial software tool from Cadence Design Systems, Inc.

[6] Some generic or simulated annealing reordering algorithms can generate smaller BDDs but may incur longer run times.



(a) **(b)**

Figure 4: 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).

4.1.1 Motivating Example

[Fig. 3] shows 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. [Fig. 3(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 new 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. [Fig. 4] shows such an example. The initial variable order has average *clause span* and *variable cut* equal 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 [Fig. 4(b)], this reduction exposes the problem's structure. Our experiments show that such MINCE variable ordering generally speeds up SAT solvers and improves run time/memory of BDD manipulations.

Similar techniques and intuitions apply in related contexts. For example, one can apply MINCE to formulas in DNF (Disjunctive Normal Form) rather than in CNF (as in Boolean Satisfiability). DNF-based descriptions are useful in automatic synthesis of logic circuits. 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, 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 to speedup SAT and minimize BDD sizes. We propose to further minimize the intermediate BDD sizes and speed up the BDD construction run time by ordering the clauses using linear hypergraph placement. If we can partition the clauses, in a formula, into several groups, such that clauses in each group share common variables, then such a formula is expected to be built faster and to require smaller intermediate BDDs, 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 (regardless of the polarity of x). Applying balanced min-cut partitioning to such hypergraphs separates the CNF formula into relatively independent subformulae. Constructing the BDD for the clauses in each part would be a step towards manipulating given parts of the BDD, as advocated earlier. [Fig. 5] shows an example. When building the clauses using the original clause order, the BDD will have a maximum size of 4 nodes after constructing the second clause. In comparison, if the improved clause order is used, the BDD will have a maximum size of 3 nodes after constructing the fifth clause.

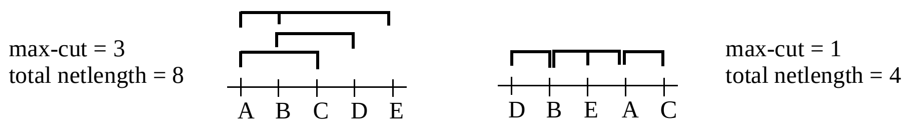
In addition to the min-cut clause ordering approach, we propose to order the clauses according to their literals. Each clause *level* is computed as $level = min(literal)$. Clauses with the highest levels are constructed first, since they include BDD variables with the highest index (i.e., at the bottom of the BDD). This 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 approach 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 run time and the size of BDDs. We use the min-cut circuit placer CAPO [Caldwell et al., 2000b]. 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 con-

$$f(V_a, V_b, V_c) = \overset{A}{(V_a + V_b)} \wedge \overset{B}{(V_c + V_b)} \wedge \overset{C}{(V_a)} \wedge \overset{D}{(V_c)} \wedge \overset{E}{(V_b)}$$



BDD Depth	Max BDD Nodes	
2	2	$(V_a + V_b)$
3	4	$(V_c + V_b)$
3	4	(V_a)
3	4	(V_c)
3	4	(V_b)

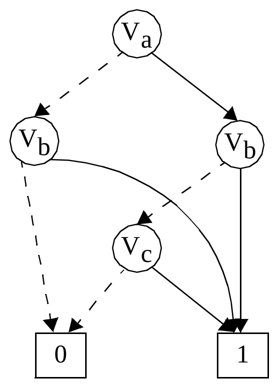
BDD Construction Order

(a)

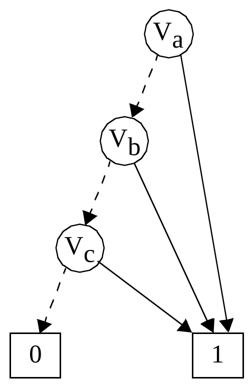
BDD Depth	Max BDD Nodes	
1	1	(V_c)
1	1	$(V_c + V_b)$
2	2	(V_b)
2	2	$(V_a + V_b)$
3	3	(V_a)

BDD Construction Order

(b)



(c)



(d)

Figure 5: Example of (a) default clause-ordering (b) improved min-cut clause-ordering. BDD for (a) after reading the first two clauses is shown in (c). BDD for (b) after reading all five clauses is shown in (d).

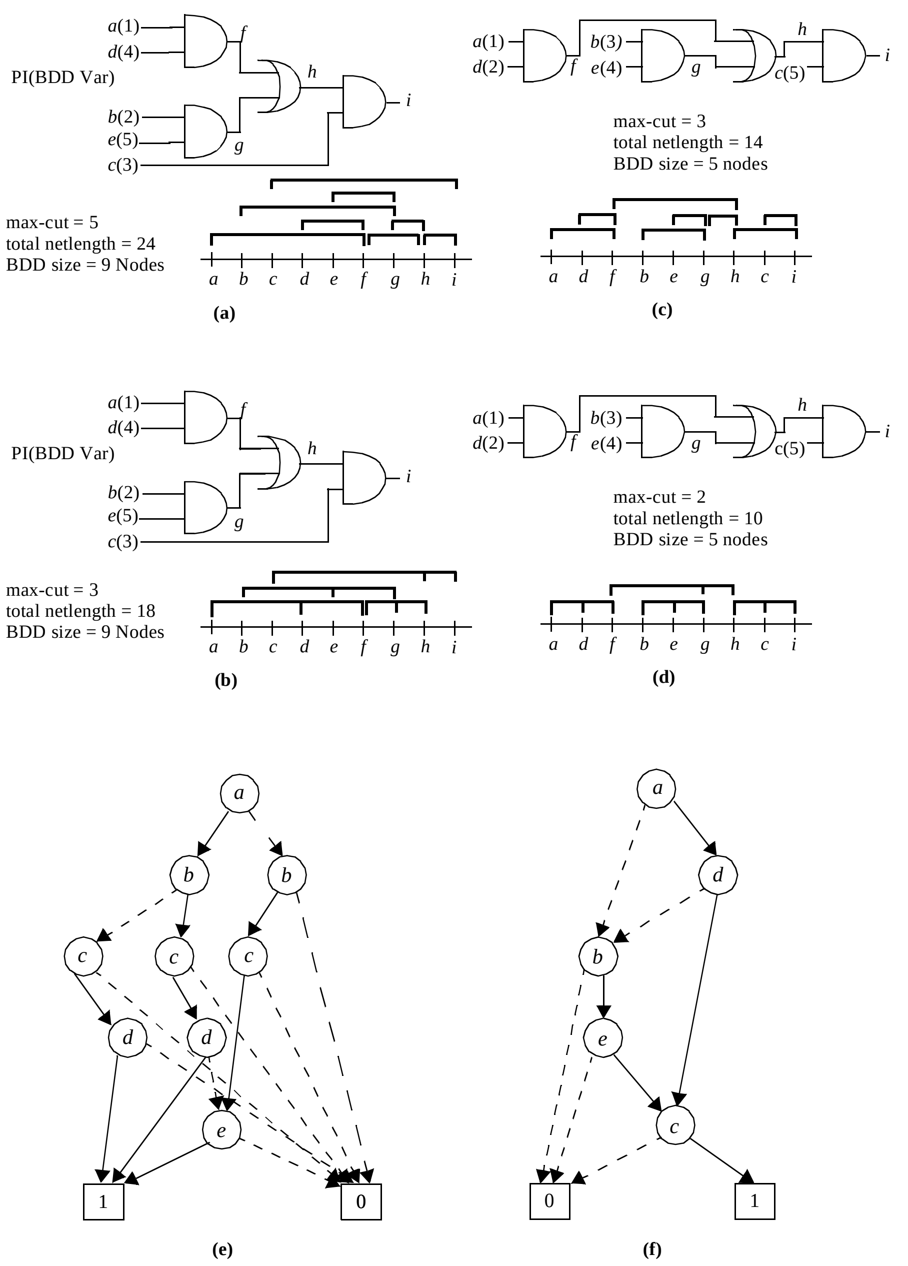


Figure 6: Example using (a) default variable ordering with Circuit hypergraph (b) Dual hypergraph (c) min-cut variable ordering with Circuit hypergraph (d) Dual hypergraph. BDDs representing (a) and (c) are shown in (e) and (f), respectively.

nects the fanout of a gate to the fanins of the gates that it is connected to. An example is shown in [Fig. 6(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 hyperedges in a *Circuit HG*, with new hyperedges that connect the fanout of each gate to its fanins. [Fig. 6(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. [Fig. 6(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 run time and memory results.

5 Empirical Results

In this section, we present experimental evidence of the improvements obtained by MINCE. We used zChaff [Moskewicz et al., 2001] as our SAT solver and CAPO [Caldwell et al., 2000c] as our min-cut circuit placer. Experimental results are given for 38 instances from the following benchmark families: pigeon-hole [DIMACS], randomized Urquhart [Urquhart, 1987], global routing [Aloul et al., 2002a], FPGA routing (*fpga* and *chnl*) [Aloul et al., 2002b], xor-chains [SAT 2004], microprocessor verification (*pipe*) [Velev and Bryant, 1999], bounded model checking (*barrel* and *longmult*) [Biere et al., 1999], *n*-queens [SATLIB], ISCAS'85 circuits [ISCAS, 1985], and flat versions of the ISCAS'89 circuits [Brglez et al., 1989] expressed in CNF. The experiments were performed on a Linux (Red Hat 9.0) workstation with a 2-GHz Xeon and 1 GB of RAM. The run time limit for all experiments was set to 1000 seconds.

SAT Experiments: We performed two sets of experiments. In the first set, we used the original instances. In the second set the variables were statically re-ordered using MINCE. For each set, we ran zChaff with three settings: (1) Fixed decision heuristic [Davis et al., 1962] (unassigned variable with smallest index is chosen first); (2) Variable state independent decaying sum (VSIDS) dynamic decision heuristic [Moskewicz et al., 2001]; and (3) VSIDS with random restarts (referred to as VRR). VSIDS selects the variable that appears in the highest number of clauses and gives some priority to variables that appear in recent conflict-induced clauses. It has been found to be significantly effective in a variety of EDA problem instances [Moskewicz et al., 2001].

[Tab. 2] shows zChaff's run times for each set of experiments and three settings. The table also shows the ordering run times of MINCE. [Tab. 3] shows the size of each instance (number of variables #V and CNF clauses #C) and the average variable cut for the original and MINCE variable orders. We observe the following from analyzing the data:

Instance	**zChaff run time (sec)**									**MINCE Order (sec)**
	Original (Search)			**MINCE (Search)**			**MINCE (Order + Search)**			
	Fixed	**VSIDS**	**VRR**	**Fixed**	**VSIDS**	**VRR**	**Fixed**	**VSIDS**	**VRR**	
hole9	**0.34**	0.95	0.89	0.31	1.31	1.89	0.45	1.45	2.03	0.14
hole10	2.16	13.66	15.24	1.49	20.68	12.29	**1.75**	20.94	12.55	0.26
hole11	15.6	128	68.08	8.15	162	84.7	**8.36**	162.2	84.9	0.21
Urq3_1	>1000	147	136	51.05	241	214	**51.1**	241	214	0.09
Urq3_5	>1000	>1000	>1000	659	>1000	>1000	**659**	>1000	>1000	0.15
Urq3_9	114	4.47	4.31	1.14	8.87	4.69	**1.22**	8.95	4.77	0.08
grout-3.3-3	72.14	37.89	**5.49**	0.46	1.48	7.37	9.35	10.37	16.26	8.89
grout-3.3-4	**0.04**	1	32.41	0.25	1.96	4.61	6.04	7.75	10.4	5.79
grout-3.3-8	156	**0.71**	2.44	0.56	0.4	3.79	6.85	6.69	10.08	6.29
grout-3.3-10	**0.05**	729	22.17	38.38	138	0.98	58.4	158	20.97	19.99
fpga10_8	58.58	20.65	160	2.57	29.87	565	**2.79**	30.1	565.2	0.22
fpga10_9	734	124	4.25	2.34	112	181	**2.7**	112.4	181.4	0.36
fpga12_8	499	316.9	974	1.86	403	486	**2.55**	403.7	486.7	0.69
fpga12_9	>1000	>1000	>1000	141	677	>1000	**141.9**	677.9	>1000	0.88
fpga12_11	>1000	>1000	>1000	15.62	33.66	147	**16.7**	34.7	148.1	1.07
fpga12_12	>1000	936	76.02	13.22	395	60.81	**13.6**	395.3	61.14	0.33
fpga13_9	>1000	>1000	>1000	8.86	>1000	>1000	**10.18**	>1000	>1000	1.32
fpga13_10	>1000	>1000	>1000	397	>1000	>1000	**398.3**	>1000	>1000	1.34
fpga13_12	>1000	>1000	>1000	12.38	422	>1000	**12.8**	422.4	>1000	0.41
chnl10_11	2.06	13.65	15.31	1.85	13.64	10.79	**2.3**	14.09	11.24	0.45
chnl10_12	2.02	14.06	11.86	1.54	16.47	15.79	**1.88**	16.81	16.13	0.34
chnl10_13	2.04	14.63	15.16	1.4	25.03	14.28	**1.84**	25.5	14.72	0.44
chnl11_12	14.7	128	68.51	8.18	182	63.59	**8.63**	182.5	64.04	0.45
chnl11_13	14.98	232	97.54	6.99	242	102	**7.48**	242.5	102.5	0.49
chnl11_20	**15.1**	>1000	627.1	41.05	397.9	117	48.7	405.6	124.7	7.68
x1_24	422	11.29	43.68	0.29	12.18	1.62	**0.72**	12.61	2.05	0.43
x1_32	>1000	>1000	>1000	2.82	13.7	22.44	**3**	13.88	22.62	0.18
x1_36	>1000	>1000	>1000	5.02	>1000	19.84	**5.18**	>1000	20	0.16
2pipe_1_o	>1000	**0.16**	0.17	>1000	0.15	0.19	>1000	8.25	8.29	8.1
2pipe_2_o	>1000	**0.22**	0.24	>1000	0.18	0.17	>1000	9.81	9.8	9.63
2pipe	>1000	**0.14**	**0.14**	26.74	0.13	0.11	33.96	7.35	7.33	7.22
barrel7	39.04	9.72	**9.57**	6.59	12.79	10.97	42.95	49.15	47.33	36.36
barrel8	79.47	**28.34**	38.51	26.72	33.08	25.52	73.35	79.71	72.15	46.63
longmult7	>1000	**13.39**	14.3	2.49	13.09	13.15	14.44	25.04	25.1	11.95
longmult8	>1000	67.7	118	12.76	56.72	141	**26.73**	70.69	155	13.97
NQueens20	57.86	114.3	**1.49**	0.01	197.6	1.57	8.54	206.1	10.1	8.53
NQueens22	>1000	>1000	**1.96**	0.64	>1000	1.99	14.15	>1000	15.5	13.51
NQueens24	91.1	>1000	**1.91**	0	>1000	2.44	17.48	>1000	19.92	17.48
Total	**18392**	**14108**	**10567**	**3500**	**9864**	**7338**	**3733**	**10097**	**7571**	**233**
# Solved	**22**	**27**	**30**	**36**	**32**	**33**	**36**	**32**	**33**	

TABLE 2: zChaff search run times (in seconds) for the CNF-SAT instances, using various static and dynamic decision heuristics.

Instance	#V	#C	Avg Var Cut		MINCE Order (sec)
			Original	MINCE	
hole9	90	415	149.4	25.4	0.14
hole10	110	561	200.9	29.9	0.26
hole11	132	738	263.1	34.7	0.21
Urq3_1	43	334	219.7	95.2	0.09
Urq3_5	46	470	351.9	142.3	0.15
Urq3_9	37	236	161.9	63.4	0.08
grout-3.3-3	960	9156	2281	273.9	8.89
grout-3.3-4	912	8356	2082	262.2	5.79
grout-3.3-8	912	8356	2082	258.8	6.29
grout-3.3-10	1056	10862	2707	311.2	19.99
fpga10_8	120	448	116.9	31.5	0.22
fpga10_9	135	549	140.9	36.2	0.36
fpga12_8	144	560	141.4	36.5	0.69
fpga12_9	162	684	170.3	41.5	0.88
fpga12_11	198	968	237.9	56.9	1.07
fpga12_12	216	1128	276.6	61.8	0.33
fpga13_9	176	759	185.5	45.1	1.32
fpga13_10	195	905	221.6	52.0	1.34
fpga13_12	234	1242	300.8	64.7	0.41
chnl10_11	220	1122	200.9	34.9	0.45
chnl10_12	240	1344	239.2	34.6	0.34
chnl10_13	260	1586	280.9	39.7	0.44
chnl11_12	264	1476	263.1	34.7	0.45
chnl11_13	286	1742	308.9	39.8	0.49
chnl11_20	440	4220	732.4	95.6	7.68
x1_24	70	186	66.5	27.0	0.43
x1_32	94	250	89.9	33.3	0.18
x1_36	106	282	105.8	33.2	0.16
2pipe_1_o	834	7026	2517	781.5	8.1
2pipe_2_o	925	8212	3054	882.9	9.63
2pipe	861	6695	2187	727.2	7.22
barrel7	3523	13765	767	358	36.36
barrel8	5106	20083	1021	505	46.63
longmult7	3319	10335	468	155	11.95
longmult8	3810	11877	472	157	13.97
NQueens20	400	12560	2728	2738	8.53
NQueens22	484	16808	3624	3649	13.51
NQueens24	576	21920	4697	4723	17.48
Total	**29627**	**202684**	**36353**	**17161**	**233**

TABLE 3: The size of CNF-SAT instances in terms of variables and clauses is shown. The table also includes ordering run times of the static decision heuristic MINCE and the average cutwidth for each instance.

Instance	BerkMin561 run time (sec)						MINCE Order (sec)
	Original (Search)		MINCE (Search)		MINCE (Order + Search)		
	Fixed	Dynamic	Fixed	Dynamic	Fixed	Dynamic	
hole9	3.15	2.83	0.16	3.17	**0.3**	3.31	0.14
hole10	17.13	50.81	0.83	53.01	**1.09**	53.27	0.26
hole11	127.04	>1000	10.5	1000	**10.71**	>1000	0.21
Urq3_1	>1000	**92.32**	>1000	637.6	>1000	637.7	0.09
Urq3_5	>1000	>1000	>1000	>1000	>1000	>1000	0.15
Urq3_9	3.77	0.84	0.18	0.64	**0.26**	0.72	0.08
grout-3.3-3	**4.92**	7.96	7.4	11.93	16.29	20.82	8.89
grout-3.3-4	**5.49**	4.18	1.59	2.14	7.38	7.93	5.79
grout-3.3-8	**6.17**	6.73	>1000	4.87	>1000	11.16	6.29
grout-3.3-10	26.29	**22.17**	7.11	8.98	27.1	28.97	19.99
fpga10_8	>1000	**0.03**	0.16	0.01	0.38	0.23	0.22
fpga10_9	>1000	**0.02**	0.13	0.03	0.49	0.39	0.36
fpga12_8	>1000	**0.02**	0.6	0.1	1.29	0.79	0.69
fpga12_9	>1000	**0.04**	>1000	0.03	>1000	0.91	0.88
fpga12_11	>1000	**0.06**	>1000	0.02	>1000	1.09	1.07
fpga12_12	>1000	**0.15**	>1000	0.13	>1000	0.46	0.33
fpga13_9	>1000	**0.07**	>1000	0.07	>1000	1.39	1.32
fpga13_10	>1000	**0.03**	>1000	0.03	>1000	1.37	1.34
fpga13_12	>1000	**0.06**	>1000	0.02	>1000	0.43	0.41
chnl10_11	18.49	60.04	0.88	55.49	**1.33**	55.94	0.45
chnl10_12	19.74	>1000	3.31	47.92	**3.65**	48.26	0.34
chnl10_13	20.48	>1000	8.09	53.91	**8.53**	54.35	0.44
chnl11_12	130.8	>1000	10.34	>1000	**10.79**	>1000	0.45
chnl11_13	130.4	>1000	30.3	>1000	**30.79**	>1000	0.49
chnl11_20	**130**	>1000	>1000	>1000	>1000	>1000	7.68
x1_24	629	3	0.1	15.84	**0.53**	16.27	0.43
x1_32	>1000	**10.95**	15.74	>1000	15.92	>1000	0.18
x1_36	>1000	>1000	10.89	>1000	**11.05**	>1000	0.16
2pipe_1_o	>1000	**0.07**	0.23	0.07	8.33	8.17	8.1
2pipe_2_o	>1000	**0.07**	0.98	0.09	10.61	9.72	9.63
2pipe	21.92	**0.05**	0.07	0.06	7.29	7.28	7.22
barrel7	**19.69**	30.78	477	17.75	513.6	54.1	36.36
barrel8	75.67	294.4	9.66	214.15	**56.3**	261	46.63
longmult7	5.19	**3.96**	>1000	5.48	>1000	17.43	11.95
longmult8	22.42	**19.1**	>1000	23.33	>1000	37.3	13.97
NQueens20	>1000	**0.03**	0	0	8.53	8.53	8.53
NQueens22	>1000	**0.03**	0	0	13.51	13.51	13.51
NQueens24	>1000	**0.03**	0	0	17.48	17.48	17.48
Total	**19418**	**8611**	**12596**	**8156**	**12829**	**8389**	**233**
# Solved	**20**	**30**	**26**	**31**	**26**	**31**	

TABLE 4: BerkMin561 search run times (in seconds) for the CNF-SAT instances, using static and dynamic decision heuristics.

Instance	**Seige_v4 run time (sec)**			**MINCE Order (sec)**
	Original (Search)	**MINCE (Search)**	**MINCE (Order + Search)**	
	Dynamic	**Dynamic**	**Dynamic**	
hole9	**27.48**	31.56	31.7	0.14
hole10	172	167.1	**167.3**	0.26
hole11	595.4	553.2	**553.4**	0.21
Urq3_1	**17.67**	17.95	18.04	0.09
Urq3_5	**357.1**	475.4	475.6	0.15
Urq3_9	0.95	0.44	**0.52**	0.08
grout-3.3-3	**4.6**	7.39	16.28	8.89
grout-3.3-4	**7.35**	4.49	10.28	5.79
grout-3.3-8	**10.15**	55.76	62.05	6.29
grout-3.3-10	**11.1**	8.57	28.56	19.99
fpga10_8	**0.01**	0.3	0.52	0.22
fpga10_9	**0.01**	0.34	0.7	0.36
fpga12_8	**0.01**	0.2	0.89	0.69
fpga12_9	**0.01**	0.46	1.34	0.88
fpga12_11	**0.01**	2.62	3.69	1.07
fpga12_12	**0.01**	0.52	0.85	0.33
fpga13_9	**0.01**	0.39	1.71	1.32
fpga13_10	**0.01**	1.56	2.9	1.34
fpga13_12	**0.01**	1.04	1.45	0.41
chnl10_11	**166.1**	172.7	173.2	0.45
chnl10_12	**310.6**	865	865.3	0.34
chnl10_13	**379.8**	927.4	927.9	0.44
chnl11_12	>1000	773.7	**774.1**	0.45
chnl11_13	>1000	>1000	>1000	0.49
chnl11_20	>1000	>1000	>1000	7.68
x1_24	3.42	0.75	**1.18**	0.43
x1_32	9.31	4.25	**4.43**	0.18
x1_36	**76.49**	100.3	95.16	0.16
2pipe_1_o	**0.13**	0.07	8.17	8.1
2pipe_2_o	**0.13**	0.11	9.74	9.63
2pipe	**0.11**	0.05	7.27	7.22
barrel7	**21.22**	24.44	60.8	36.36
barrel8	**127.1**	133.3	179.9	46.63
longmult7	**7.39**	7.76	19.71	11.95
longmult8	**36.57**	33.75	47.72	13.97
NQueens20	**0.01**	0.01	8.54	8.53
NQueens22	**0.01**	0.01	13.52	13.51
NQueens24	**0.01**	0.01	17.49	17.48
Total	**5342**	**6367**	**6600**	**233**
# Solved	**35**	**36**	**36**	

TABLE 5: Seige_v4 search run times (in seconds) for the CNF-SAT instances. Random seed 1 was used.

- For the pigeon-hole, unsatisfiable FPGA routing (*chnl*), satisfiable FPGA routing (*fpga*), urquhart, and xor-chain instances, the combination of MINCE/Fixed yields the best search run times.
- For the *n*-queens instances, the combination MINCE/Fixed (when not counting MINCE ordering run time) shows the best search run times in all cases.
- The results are mixed for the global routing, microprocessor verification, and bounded model checking instances. The combination Original/VSIDS yields the best search run times in six out of eleven cases. The two instances, *grout-3.3-3* and *grout-3.3-8*, are solved in a fraction of a second using the combinations MINCE/Fixed and MINCE/VSIDS (when not counting MINCE ordering run times) respectively. All microprocessor verification instances are solved in a fraction of a second using the combination MINCE/VSIDS/Restarts (when not counting MINCE ordering run times). Note that these instances have large average variable cuts.
- MINCE significantly reduces the average variable cut for most instances.
- Ordering run times are correlated with the size of the instance.

Deciding on closely-connected variables (i.e., variables that share many clauses) leads to a reduction in search run time. Since "connected" variables are ordered next to each other, this approach allows the SAT 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 was found to be more effective on instances with well-pronounced connectivity structure, such as the pigeonhole or FPGA routing instances, which consist of *multiple partitions*. On these problems, MINCE finds variable orders compatible with the problem's structure, which speeds up SAT solvers. For example, a speedup of 9 was obtained for the *hole10* instance over the VSIDS decision heuristic with random restarts.

Nevertheless, the dynamic decision heuristic, VSIDS (with or without Random Restarts), does outperform the use of MINCE with fixed decision heuristic on *general* structured EDA instances, such as the microprocessor verification instances. This, in part, is explained by the fact that the VSIDS decision heuristic accounts for the added conflict-induced clauses when selecting the next decision variable. Augmenting the CNF formula with a large number of conflict-induced clauses during the search process is likely to increase the formula's cutwidth and eliminate the advantage of the static orderings identified by MINCE. Observe that MINCE has a worst- and best-case performance of $\Theta((|C| + |V|)\log^2|V|)$ [Caldwell et al., 2000c], where V is the total number of variables and C is the number of clauses in a given CNF formula. MINCE's performance compares favorably with many SAT solvers, e.g., zChaff, that have exponential worst-case performance. Nevertheless, the table shows that pre-processing the instances with MINCE, even when the dynamic decision heuristic VSIDS is used, helps speed-up the search process. Finally, even when MINCE's run time makes it prohibitively expensive for a particular SAT instance where MINCE reduces a solver's run time, cap-

Instance	**MINCE Order (sec)**		**Avg Var Cut**	
	Average	**Median**	**Average**	**Median**
hole10	0.26	0.26	29.90	29.90
Urq3_9	0.09	0.08	64.59	63.57
grout-3.3-4	5.93	5.95	260.8	260.5
fpga10_8	0.18	0.17	31.91	32.00
chnl10_11	0.52	0.53	30.98	29.90
x1_24	0.36	0.42	27.31	27.14
2pipe	8.20	8.21	730.5	728.8

TABLE 6: Exploring the variability of orders returned by independent random starts of MINCE. 100 different variable orders were generated, using MINCE, for various instances in [Tab. 2]. The table shows the average and median of MINCE's ordering run times and the average cutwidth of the SAT instances.

turing the instance structure may lead to a better understanding and be useful for practical purposes.

In order to study MINCE's effect when using other state-of-the-art SAT solvers, we solved the instances using two of the best known SAT solvers: BerkMin561 [Goldberg and Novikov, 2002] and seige (variant 4) [Ryan]. BerkMin561 was run with two settings: (1) Fixed decision heuristic (BerkMin561 option - strategy 2); and (2) its default dynamic decision heuristic (BerkMin561 option - strategy 0). Seige can only be tested with its dynamic decision heuristic enabled. A random seed of 1 was used with Seige. Note that the source-code for both solvers is not public.

[Tab. 4] and [Tab. 5] show BerkMin561's and Seige's run times, respectively. The tables also show the ordering run times of MINCE. We observe the following from analyzing the data:

- For BerkMin561, pre-processing the instances with MINCE solves more instances (26 versus 20 with fixed ordering, and 31 versus 30 with dynamic ordering). Both for fixed and dynamic ordering, overall run times decrease (19418 sec versus 12829 sec with fixed ordering and 8611 sec versus 8389 sec with dynamic ordering).
- For Seige, pre-processing the instances with MINCE solves more instances (36 versus 35 with dynamic ordering). Note that some of the instances showed better results without pre-processing with MINCE. This is explained by the fact that Seige uses a dynamic decision heuristic only which can eliminate the advantage of the static ordering identified by MINCE.
- zChaff's total run times when using MINCE/Fixed outperform the best total run times for BerkMin561 and Seige (3733 sec for zChaff -includes MINCE's ordering run times- versus 8389 sec for BerkMin561 versus 5342 sec for Seige).
- For difficult instances (and overall), the time needed to reorder variables with MINCE does not dominate SAT-solving, either before or after variable ordering.

Var. Order:	**Original**						**MINCE**						**MINCE**	
Clause Order:	**Original**		**Bottom-Up**		**MINCE**		**Original**		**Bottom-Up**		**MINCE**		**VAR**	**CL**
Inst.	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Time**
s27	**0.09**	181	0.12	256	0.1	181	0.1	**60**	0.1	**60**	0.12	**60**	0.05	0.06
s298		o/m		o/m	307	6.9M	0.58	**8792**	**0.24**	**8792**	0.98	9K	0.2	0.37
s344		o/m		o/m		o/m	1.18	13K	**0.22**	9751	0.79	**9130**	0.33	0.41
s349		o/m		o/m		t/o	10.3	48K	**0.22**	**16K**	0.62	**16K**	0.29	0.4
s382		o/m		o/m		o/m	0.99	7492	**0.18**	**6748**	0.45	9051	0.31	0.45
s386		o/m		o/m	92	608K	32.6	492K	**2.2**	94K	3.2	**60K**	0.31	0.5
s400		o/m		o/m		o/m	0.98	6641	**0.18**	6253	0.36	**6177**	0.23	0.55
s420		o/m		o/m		t/o	1.04	17K	**0.22**	**7033**	0.46	7823	0.3	0.57
s444		o/m		o/m		o/m	0.64	**7000**	**0.3**	9092	0.77	7189	0.72	0.6
s510		o/m		o/m		t/o		o/m	**2.87**	**108K**	128	1.5M	0.86	0.64
s526		o/m		o/m		t/o	2.37	**25K**	**0.52**	**25K**	2.61	32K	0.45	0.68
s641		o/m		o/m		o/m	8.26	178K	**3.61**	106K	15.6	**89K**	0.54	0.84
s713		o/m		o/m		o/m	53.6	774K	**9.5**	450K	54.6	**295K**	0.56	0.98
s832		o/m		o/m		t/o		o/m	**34**	**310K**		t/o	1.29	1.09
s838		o/m		o/m		o/m	101	1M	**6.1**	**92K**	13.1	160K	0.61	1.17
s953		t/o		o/m		t/o		o/m	**134**	**4M**		t/o	0.77	1.13
s1196		t/o		o/m		t/o		o/m	**475**	11M	505	**3.4M**	1.27	1.59
s1238		o/m		o/m		t/o		o/m	**151**	**4M**		t/o	1.27	1.82
Total	**0.09**	**181**	**0.12**	**256**	**399**	**7.5M**	**214**	**2.6M**	**820**	**21M**	**727**	**5.7M**	**10.4**	**13.9**
#Built	**1**		**1**		**3**		**13**		**18**		**15**			

TABLE 7: Statistics for constructing the BDDs of the ISCAS89 CNF Benchmarks without Sifting. Size represents the maximum size during the construction of the BDD. o/m stands for out-of-memory. t/o stands for time-out.

To explore the variability of orders returned by independent random starts of MINCE, we generated 100 different variable orders, using MINCE, for various instances in [Tab. 2]. [Tab. 6] reports the results of those tests. Specifically, the table shows the average and median of MINCE's ordering run times and the average cutwidth of the SAT instances. MINCE's algorithm has near-linear run time, and empirically the run times on different starts are very close.

CNF-to-BDD Experiments: [Tab. 7] and [Tab. 8] show the BDD construction run times for circuit consistency functions (defined as a CNF formula consisting of the union of sets of clauses representing each gate in the circuit, i.e., internal variables are not quantified) of the ISCAS'89 circuit benchmarks. Note that this is not representative of symbolic state traversal, but is a standard experimental procedure for evaluating BDD packages [Janssen, 2001]. The table shows run times (sec) and BDD sizes, which represent the maximum number of seen nodes 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 or-

Var. Order:	**Original**						**MINCE**						**MINCE**	
Clause Order:	**Original**		**Bottom-Up**		**MINCE**		**Original**		**Bottom-Up**		**MINCE**		**VAR**	**CL**
Inst.	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Time**
s27	0.12	181	**0.1**	256	**0.1**	181	0.14	**60**	0.13	**60**	**0.1**	**60**	0.05	0.06
s298	8.84	28K	274	294K	1.4	6504	11.4	15K	**1.16**	**6445**	1.3	8513	0.2	0.37
s344	208	130K		t/o	1.6	**5423**	158	98K	**0.79**	9039	3.6	13K	0.33	0.41
s349	90	83K		t/o	1.9	8971	315	193K	1.87	11K	**1.2**	**6698**	0.29	0.4
s382	51.3	88K	87.2	118K	1.7	7688	19.2	28K	**0.46**	**4752**	1.9	12K	0.31	0.45
s386	99	96K	155	131K	**5.0**	**18K**	128	124K	8.37	**18K**	8.9	31K	0.31	0.5
s400	247	177K	106	68K	1.4	7665	97.6	107K	**0.71**	**4513**	1.2	4808	0.23	0.55
s420	136	94K	557	142K	2.7	8012	216	81K	**1.24**	**6536**	2.1	6116	0.3	0.57
s444	77.3	85K	943	508K	**1.5**	**5360**	166	136K	1.72	7684	21.4	40K	0.72	0.6
s510		t/o		t/o	24	41K		t/o	**13**	**18K**	18.4	66K	0.86	0.64
s526		t/o		t/o	3.6	14K	297	155K	**2.62**	**8959**	3.1	14K	0.45	0.68
s641		t/o		t/o	213	129K		t/o	**50.5**	**69K**	239	202K	0.54	0.84
s713		t/o		t/o	278	**144K**		t/o	**176**	223K	184	152K	0.56	0.98
s832		t/o		t/o		t/o		t/o	**102**	**83K**		t/o	1.29	1.09
s838		t/o		t/o	38.2	45K		t/o	**19.1**	**28K**	34.5	60K	0.61	1.17
s953		t/o		t/o		t/o		t/o		t/o		t/o	0.77	1.13
s1196		t/o		t/o		t/o		t/o		t/o		t/o	1.27	1.59
s1238		t/o		t/o		t/o		t/o		t/o		t/o	1.27	1.82
Total	**917**	**783K**	**2123**	**1.2M**	**574**	**441K**	**1408**	**939K**	**379**	**500K**	**521**	**617K**	**10.4**	**13.9**
#Built	**9**		**7**		**14**		**10**		**15**		**14**			

TABLE 8: Statistics for constructing the BDDs of the ISCAS89 CNF Benchmarks with Sifting. Size represents the maximum size during the construction of the BDD. o/m stands for out-of-memory. t/o stands for time-out.

dering heuristics are used: *original*, *bottom-up*, and MINCE min-cut clause order. The MINCE variable ordering leads to faster and smaller BDDs. In terms of circuits, this can be explained by MINCE ordering the gates to minimize the “total length of wires”. Using the *original* clause ordering, MINCE enabled the BDD construction for 16 ISCAS’89 circuits as opposed to only 10 with sifting and 1 with the *original* variable ordering. MINCE’s variable ordering time is negligible in most cases. MINCE reduced the average *variable cut* for the ISCAS’89 circuits from 250 to 49.

In addition, combining the *bottom-up* or MINCE clause ordering heuristics with the 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 run time and memory results. The tables also show that enabling sifting will, in general, produce BDDs with fewer nodes, but will require an overhead in run time. 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 run time limit. When comparing the

Instance	**Original**		**BFS**		**DFS**		**FUJ**		**MAL-Level**		**MAL-Fanin**		**MINCE**				
													CAPO	**Circuit HG**		**Dual HG**	
	Size	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Time**	**Size**	**Time**	**Size**	**Time**
c17	**17**	**0**	**17**	**0**	21	0	21	0	21	0	21	0	0.04	**17**	**0**	20	0
c432	6582	**0.01**	34K	0.03	149K	0.27	151K	0.28	161K	0.4	151K	0.28	0.23	7078	0.01	**5748**	**0.01**
c499	60K	0.07	141K	0.18	52K	0.06	61K	0.08	74K	0.1	63K	0.08	0.26	72K	0.11	**43K**	**0.05**
c880	1.2M	1.97	101K	0.15	**27K**	0.03	29K	0.02	29K	**0**	30K	0.02	0.48	257K	0.79	43K	0.04
c1355	184K	0.23	399K	0.57	**165K**	**0.19**	197K	0.24	251K	0.4	200K	0.25	0.62	188K	0.26	170K	0.22
c1908	90K	0.32	55K	0.08	**43K**	0.08	45K	0.09	127K	0.5	45K	**0.07**	1	59K	0.19	100K	0.22
c2670		o/m		o/m	11M	35.2	11.4M	37.5	12.8M	72	11.5M	37.4	1.61	158K	0.29	**86K**	**0.24**
c3540	2.5M	6.64	2.2M	5.49	916K	1.76	757K	**1.38**	**650K**	1.6	757K	1.44	2.08	920K	2.49	4.1M	11.48
c5315		o/m		o/m	74K	0.16	74K	0.15	50K	0.1	74K	0.16	3.15	245K	0.7	**35K**	**0.06**
c6288		o/m		o/m		o/m		o/m		o/m		o/m	(2.89)		o/m		o/m
c7552		o/m		o/m		o/m		o/m		o/m		o/m	4.83	**84K**	**0.16**	488K	0.91
Total	**4M**	**9.2**	**3M**	**6.5**	**12M**	**37.8**	**12.8M**	**39.8**	**14.1M**	**76**	**12.8M**	**40**	**14.3**	**1.9M**	**5.01**	**5.1M**	**13.2**
#Built	**7**		**7**		**9**		**9**		**9**		**9**			**10**		**10**	

TABLE 9: Statistics for constructing the BDDs of the ISCAS85 circuits without sifting using the nanotrav tool (from the CUDD distribution, version 2.4.0). o/m stands for out-of-memory.

bottom-up with MINCE clause ordering heuristic, bottom-up is successful in constructing more circuits than MINCE, however for some instances, such as the $s1196$, MINCE is able to build the BDD using less memory than the bottom-up approach. 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 Experiments: [Tab. 9] and [Tab. 10] summarize the run time and memory results for constructing the BDDs for the PO functions of the ISCAS'85 circuits in terms of their PIs (internal variables are quantified). We used the `nanotrav` tool (within the CUDD distribution, version 2.4.0) [Somenzi, 1997] to construct the BDDs. In both tables, the columns represent the *original*, *BFS*, *DFS*, *Fujita* [Fujita et al., 1988], *Malik-level* [Malik et al., 1988], *Malik-fanin* [Malik et al., 1988], and MINCE orderings using the *Circuit HG* and the *Dual HG*, respectively. The tables also include the run time needed by CAPO to generate the gate orderings. As the data clearly illustrate, DFS, Fujita, Malik-level, and Malik-fanin successfully construct more circuits than the original or BFS ordering heuristic. However, in the non-sifting case, circuit HG and Dual HG orderings are able to construct more BDDs than all other approaches. Out of 11 ISCAS'85 benchmarks, circuit HG and Dual HG constructed 10 BDDs as opposed to 9 BDDs by DFS, Fujita, Malik-level, and Malik-fanin. Furthermore, the Dual HG model was successful in solving more instances, using smaller run times and BDD nodes, than the Circuit HG model. This can be attributed to the fact that

Instance	**Original**		**BFS**		**DFS**		**FUJ**		**MAL-Level**		**MAL-Fanin**		**MINCE**				
													CAPO	**Circuit HG**		**Dual HG**	
	Size	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Size**	**Time**	**Time**	**Size**	**Time**	**Size**	**Time**
c17	**17**	**0**	**17**	**0**	21	0	21	0	21	0	21	0	0.04	**17**	**0**	20	0
c432	**4933**	**0.04**	4998	0.1	4856	0.13	5057	0.09	5213	0.1	5057	0.08	0.23	5284	0.05	5054	**0.04**
c499	39K	1.66	69K	2.37	42K	2.36	46K	2.98	50K	4.4	44K	1.77	0.26	54K	2.75	**34K**	**1.14**
c880	27K	1.01	41K	1.27	19K	**0.59**	20K	0.29	19K	0.7	20K	0.6	0.48	55K	5.72	**18K**	0.6
c1355	**121K**	17.7	144K	22.68	**121K**	18.1	124K	**10.4**	**121K**	22	140K	**10.4**	0.62	131K	12.2	132K	27.1
c1908	29K	**0.62**	25K	1.07	25K	1.04	27K	0.65	**21K**	1	28K	0.61	1	22K	0.92	24K	1.56
c2670	14K	1.07	11K	0.87	46K	6.23	27K	2.11	**10K**	**0.5**	40K	2.23	1.61	11K	0.94	**10K**	0.59
c3540	136K	**10.3**	267K	26.08	296K	58.2	292K	51	267K	27	292K	51.2	2.08	254K	40.7	**130K**	15.3
c5315	**10K**	0.47	11K	0.53	11K	0.52	12K	0.48	12K	0.4	12K	0.51	3.15	12K	0.41	**10K**	**0.37**
c6288		t/o		t/o		t/o		t/o		t/o		t/o	(2.89)		t/o		t/o
c7552	47K	3.29	46K	3.64	37K	3	37K	**2.88**	44K	3.1	38K	3.12	4.83	**27K**	3.71	45K	8.09
Total	**427K**	**36.1**	**619K**	**58.6**	**601K**	**90.2**	**590K**	**70.9**	**550K**	**60**	**618K**	**70.5**	**14.3**	**571K**	**67.4**	**409K**	**54.8**
#Built	**10**		**10**		**10**		**10**		**10**		**10**			**10**		**10**	

TABLE 10: Statistics for constructing the BDDs of the ISCAS85 circuits with sifting using the nanotrav tool (from the CUDD distribution, version 2.4.0). o/m stands for out-of-memory. t/o stands time-out.

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 outperforms Malik-level and utilizes fewer BDD nodes. As discussed earlier, building BDDs with sifting generally uses fewer BDD nodes but requires longer run time. This can be illustrated by our results with the Dual HG model, where all 10 instances were solved in 13 seconds without sifting as opposed to 55 seconds with sifting. On the other hand, the total BDD size is only 409K nodes for the sifting experiment, whereas it needs 5.1M nodes in the non-sifting experiment. We believe the proposed static ordering should be very effective with applications that do not allow dynamic sifting.

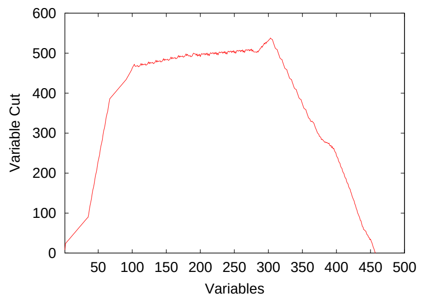
We are currently working on further improving the performance by running multiple independent starts of MINCE. The main advantage of our approach is the use of circuit structure detected by global min-cut partitioning and placement algorithms with near-linear worst-case run time. Note that as in the SAT experiments, MINCE is not expected to perform well on complex circuits that exhibit no structural properties.

6 Conclusions and Future Work

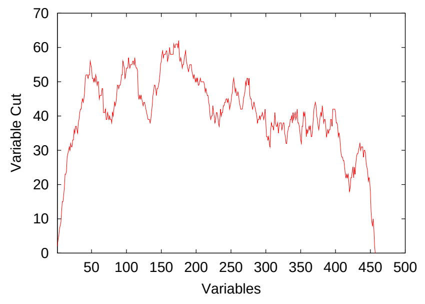
We proposed a static variable-ordering heuristic MINCE for CNF formulae with applications to SAT and BDDs. The main advantage of this heuristic is its very good performance on standard benchmarks in terms of run time of SAT solvers, as well as memory and run time of BDD construction. We believe that this is due to the fact that the proposed variable ordering is *global* and relies on high-performance hypergraph partitioning and placement (MLPart [Caldwell et al., 2000b] and CAPO [Caldwell et al., 2000c]). Unlike problem-specific dynamic variable-ordering heuristics, such as VSIDS 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 circuits. 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 [Fig. 7] 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. [Fig. 7] also shows the cutwidth profile for the original and MINCE variable orderings of an FPGA routing instance. Interestingly, the five variable clusters in [Fig. 7(f)] represent a one-to-one correspondence with the routing channels 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 Chaff as a black-box too). We hope that this will also enable its easy evaluation and adoption in the industry. The work in [Jin et al., 2002], published after our initial workshop paper, successfully used our proposed techniques as well as our implementation (available on the Web at *http://www.eecs.umich.edu/~faloul/Tools/mince)* in a different application—conjunction scheduling for reachability analysis.

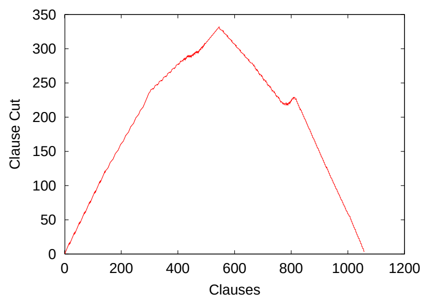
Our on-going work addresses additional types of benchmarks, better justifications of the MINCE heuristic and also analyses of the cases when it fails to produce near-best variable orderings. An important research question is to account for polarities of literals. We are aware of work conducted in [Wang and Clarke, 2001] 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. A comparison of results show that MINCE is surprisingly successful without using polarities of literals. Other work includes combining partitioning-based techniques like MINCE with very fast, entirely local techniques used in SAT solvers. While our work does offer some evidence to usefulness of this combination, a recent conference paper by [Huang and Darwiche, 2003] offers such a combination, with strong empirical results against the original Chaff. However, their techniques do not work on instances like pigeon-hole benchmarks (MINCE does).



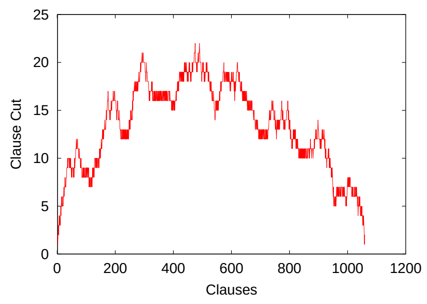
(a)



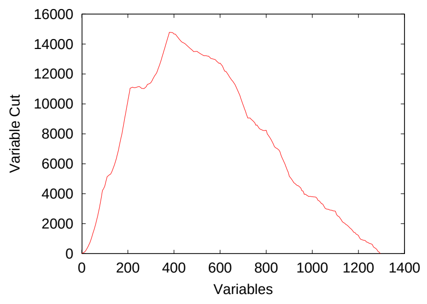
(b)



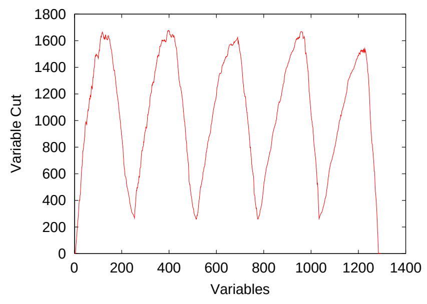
(c)



(d)



(e)



(f)

Figure 7: 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)

Acknowledgments

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

References

[Aloul *et al.*, 2001] F. A. Aloul, I. L. Markov, and K. Sakallah, "Faster SAT and Smaller BDDs via Common Function Structure," *in Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 443-448, 2001.

[Aloul *et al.*, 2002a] F. A. Aloul, A. Ramani, I. L. Markov, and K. Sakallah, "Generic ILP versus Specialized 0-1 ILP: an Update," in *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 450-457, 2002.

[Aloul *et al.*, 2002b] F. A. Aloul, A. Ramani, I. L. Markov, and K. Sakallah, "Solving Difficult SAT Instances in the Presence of Symmetry," in *Proceedings of the Design Automation Conference (DAC)*, pp. 731-736, 2002.

[Amir and McIlraith, 2000] E. Amir and S. McIlraith, "Improving the Efficiency of Reasoning Through Structure-Based Reformulation," in *Proceedings of the International Symposium on Abstractions, Reformulation, and Approximation (SARA)*, Lecture Notes in Artificial Intelligence, vol. 1864, pp. 247-259, Springer 2000.

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

[Biere *et al.*, 1999] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures Instead of BDDs," *in Proceedings of the Design Automation Conference (DAC)*, pp. 317-320, 1999.

[Brglez *et al.*, 1989] F. Brglez, D. Bryan, and K. Kozminski, "Combinational Problems of Sequential Benchmark Circuits," *in Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pp. 1929-1934, 1989.

[Brglez and Fujiwara, 1985] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in FORTRAN," *in Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pp. 785-794, 1985.

[Breuer, 1977] M. Breuer, "Min-cut Placement," *J. Design Automation Fault-Tolerant Computing*, 1(4), pp. 343-362, 1977.

[Bryant, 1986] R. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *in IEEE Transactions on Computers*, 35(8), pp. 677-691, 1986.

[Bryant, 1992] R. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *in ACM Computing Surveys*, 24(3), pp. 293-318, September 1992.

[Bryant and Meinel, 2001] R. Bryant and C. Meinel, "Ordered Binary Decision Diagrams," *in Logic Synthesis and Verification, S. Hassoun, and T. Sasao, eds., Kluwer Academic Publishers*, Boston/Dordrecht/London, 2001.

[Caldwell *et al.*, 2000a] A. Caldwell, A. Kahng, and I. L. Markov, "Design and Implementation of Move-Based Heuristics for VLSI Hypergraph Partitioning," *ACM Journal on Experimental Algorithms*, vol. 5, 2000. *http://www.jea.acm.org/volume5.html*

[Caldwell *et al.*, 2000b] A. Caldwell, A. Kahng, and I. L. Markov, "Can Recursive Bisection Produce Routable Placements?" *in Proceedings of the Design Automation Conference (DAC)*, pp. 477-482, 2000.

[Caldwell *et al.*, 2000c] A. Caldwell, A. Kahng, and I. L. Markov, "Improved Algorithms for Hypergraph Bipartitioning," *in Proceedings of the IEEE ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, pp. 661-666, 2000.

[Davis *et al*., 1962] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-Proving," in *Communications of the Association for Computing Machinery*, vol. 5, pp. 394-397, 1962.

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

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

[Dunlop and Kernighan, 1985] A. Dunlop and B. Kernighan, "A Procedure for Placement of Standard-Cell VLSI Circuits," *in IEEE Transactions on Computer-Aided Design,* 1(4), pp. 92-98, 1985.

[Fiduccia and Mattheyses, 1982] C. Fiduccia and R. Mattheyses, "A Linear-Time Heuristic for Improved Network Partitions," *in Proceedings of the Design Automation Conference (DAC)*, pp. 241-247, 1982.

[Fujii *et al.*, 1993] H. Fujii, G. Ootomo, and C. Hori, "Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams," *in Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 38-41, 1993.

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

[Garey and Johnson, 1979] M. Garey and D. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," *W. H. Freeman & Co.*, 1979.

[Goldberg and Novikov, 2002] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust SAT-Solver," in *Proceedings of the Design Automation and Test in Europe Conference (DATE)*, pp. 142-149, 2002.

[Gomes *et al.*, 1998] C. Gomes, B. Selman, and H. Kautz, "Boosting Combinatorial Search Through Randomization," *in Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 431-437, 1998.

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

[Huang and Darwiche, 2003] J. Huang and A. Darwiche, "A Structure-Based Ordering Heuristic for SAT," *in Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[ISCAS, 1985] ISCAS'85 Benchmark Information, *http://www.cbl.ncsu.edu/www/CBL_Docs/iscas85.html*

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

[Jin *et al.*, 2002] H. Jin, A. Kuehlman, and F. Somenzi, "Fine-grain Conjunction Scheduling for Symbolic Reachability Analysis", *in Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pp. 312-326, 2002.

[Karypis *et al.*, 1997] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel Hypergraph Partitioning: Applications in VLSI Design," *in Proceedings of the Design Automation Conference (DAC)*, pp. 526-529, 1997.

[Kernighan and Lin, 1970] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell System Technical Journal*, 49(1), pp. 291-307, 1970.

[Larrabee, 1992] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Transactions on Computer-Aided Design*, 11(1), pp. 4-15, 1992.

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

[Malik *et al.*, 1988] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment," *in Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 6-9, 1988.

[Minato *et al.*, 1990] S. Minato, N. Ishiura, and S. Yajima, "Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation," *in Proceedings of the Design Automation Conference (DAC)*, pp. 52-57, 1990.

[Moskewicz *et al.*, 2001] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *in Proc. of the Design Automation Conference (DAC)*, pp. 530-535, 2001.

[Nam *et al.*, 2001] G. Nam, F. A. Aloul, K. Sakallah, and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," *in Proceedings of the International Symposium on Physical Design (ISPD)*, pp. 222-227, 2001.

[Panda and Somenzi, 1995] S. Panda and F. Somenzi, "Who Are the Variables in Your Neighborhood," *in Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 74-77, 1995.

[Prasad *et al.*, 1999] M. Prasad, P. Chong, and K. Keutzer, "Why is ATPG Easy?" *in Proceedings of the Design Automation Conference (DAC)*, pp. 22-28, 1999.

[Rudell, 1993] R. Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams," *in Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 42-47, 1993.

[Ryan] L. Ryan, "Siege SAT Solver". *http://www.cs.sfu.ca/~loryan/personal/*

[SAT 2004] SAT 2004 Competition, *http://www.satlive.org/SATCompetition*

[SATLIB] Satlib SAT Benchmarks, *http://www.satlib.org*

[Shacham and Zarpas, 2003] O. Shacham and E. Zarpas, "Tuning the VSIDS Decision Heuristic for Bounded Model Checking," *in Microprocessor Test and Verification (MTV)*, pp. 75-82, May 2003.

[Silva and Sakallah, 1996] J. Silva and K. Sakallah, "GRASP-A New Search Algorithm for Satisfiability," *in Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pp. 220-227, 1996.

[Silva and Sakallah, 1997] J. Silva and K. Sakallah, "Robust Search Algorithms for Test Pattern Generation," *in Proceedings of the IEEE Fault-Tolerant Computing Symposium*, pp. 152-161, 1997.

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

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

[Somenzi, 2001] F. Somenzi, "Efficient Manipulation of Decision Diagrams," *in International Journal on Software Tools for Technology Transfer (STTT),* 3(2), pp. 171-181, 2001.

[Stålmarck, 1994] G. Stålmarck, "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.

[Stephan *et al.*, 1996] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *in IEEE Transactions on Computer-Aided Design,* 15(9), pp. 1167-1175, 1996.

[Tseitin, 1983] G. Tseitin, "On the Complexity of Derivation in Propositional Calculus," *in Studies in Constructive Mathematics and Mathematical Logic*, Part 2, pp. 115-125, 1968. *Reprinted in J. Siekmann, and G. Wrightson, eds., Automation of Reasoning*, Vol. 2, Springer-Verlag, pp. 466-483, 1983.

[Urquhart, 1987] A. Urquhart, "Hard Examples for Resolution," in *Journal of the ACM,* 34(1), pp. 209-219, 1987.

[Vazirani, 2001] V. Vazirani, "Appoximation Algorithms", Springer, pp. 378, 2001.

[Velev and Bryant, 1999] M. Velev and R. Bryant, "Superscalar Processor Verification Using Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic," in *Proceedings of the Conference on Correct Hardware Design and Verification Methods (CHARME),* LNCS 1703, pp. 37-53, 1999.

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

[Wang, 1997] J. Wang. "Branching Rules for Propositional Satisfiability Test." *in DIMACS series in Discrete Mathematics and Theoretical Computer Science*. vol. 35, 1997.

[Wood and Rutenbar, 1998] R. Wood and R. Rutenbar, "FPGA Routing and Routability Estimation Via Boolean Satisfiability," *in IEEE Transactions on VLSI*, 6(2), pp. 222-231, 1998.

[Yang and Ciesielski, 2002] C. Yang and M. Ciesielski, "BDS: A BDD-Based Logic Optimization System," *in IEEE Transactions on Computer-Aided Design*, 21(7), pp. 866-876, 2002.

[Zhang, 1997] H. Zhang, "SATO: An Efficient Propositional Prover," *in International Conference on Automated Deduction*, pp. 272-275, 1997.