Faster SAT and Smaller BDDs via Common Function Structure

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

Electrical Engineering and Computer Science
University of Michigan
{faloul, imarkov, karem}@umich.edu

Abstract

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

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

1 Introduction

Algorithms that efficiently manipulate Boolean functions arising in real-world applications are becoming increasingly popular in several areas of computer-aided design and verification. In this work we focus on two classes of these algorithms: complete Boolean satisfiability (SAT) solvers [18, 23, 26, 30] and algorithms for manipulating Binary Decision Diagrams (BDDs) [4, 7, 16]. 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 [15, 27], formal verification [2], timing verification [24] and routing of field-programmable gate arrays [19], among others. While no exact polynomial-time algorithms are known for the general case, many exact algorithms [18, 23, 26, 30] 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 [18, 23, 26, 30], as well as our empirical data, clearly imply that the order of these steps critically affects the runtime of leading edge SAT algorithms. This order of steps depends on the order of variables used to represent the input function, but can also be controlled dynamically based on the results of previous steps.

BDDs [4, 7] are commonly used to implicitly represent large solution spaces in combinatorial problems that arise in synthesis and verification. A BDD is a directed acyclic graph constructed in such a way that its directed paths represent combinatorial objects of interest (such as subsets, clauses, minterms, etc.). An exponential compression rate is achieved by BDDs whose number of paths is exponential in the number of vertices and edges (graph size). BDDs can be transformed by algorithms that visit all vertices and edges of the directed graph in some order and therefore take polynomial time in the "current" size of the graph. However, when new BDDs are created, some of these algorithms tend to significantly increase the number of vertices, potentially leading to exponential memory and runtime requirements. Several BDD ordering techniques have been proposed to overcome this problem. These include static [9, 17] and dynamic approaches [20, 22]. Just as for SAT solvers, the order of "elementary steps" is critically important. This order can either be chosen *statically*, i.e. by preprocessing the input formula, or *dynamically*, based on the outcome of previous steps during the search process.

A reliable and fast variable-ordering heuristic for a given application can dramatically affect its competitiveness and is often considered an important part of implementation. For example, the leading-edge SAT solver GRASP [23] is typically used with the dynamic variable-ordering heuristic DLCS (select the *variable* that appears in the maximum number of unresolved clauses) or DLIS (select the *literal* that appears in the maximum number of unresolved clauses), and the renowned CUDD package [25] for BDD manipulation incorporates the dynamic variable-sifting heuristic which is applied many times in the course of BDD transformations. Variable sifting is affected by the initial order, but can also be completely turned off to improve runtime. Sifting for BDDs is relatively more expensive than most dynamic ordering heuristics for SAT. However, the effect of ordering heuristics on total runtime is highly instance-specific.

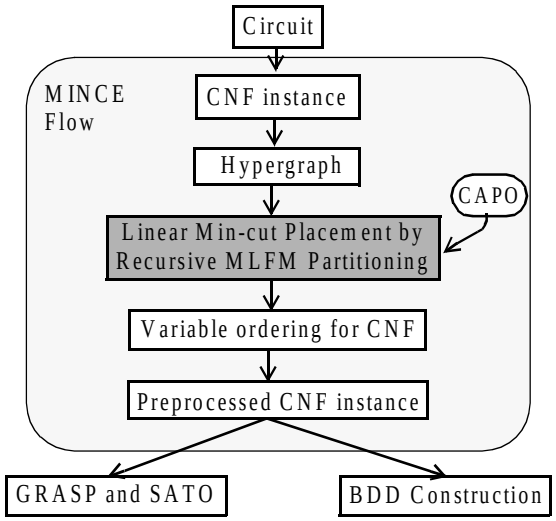

Fig. 1: The MINCE heuristic based on Multi-Level Fiduccia-Mattheyses (MLFM) partitioning [5, 6, 14]

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

The remainder of the paper is structured as follows. Section 2 motivates our reliance on hypergraph partitioning. Section 2.1 discusses the use of linear placement. Section 2.2 presents an example of hypergraph partitioning. Section 3 describes applications to SAT and BDDs and provides experimental evidence of the effectiveness of partitioning-based variable-ordering. Section 4 concludes the paper and provides perspective on future work.

2 Problem Partitioning

We first observe that Boolean functions arising in many applications represent spacial, logical or causal dependencies/connections among variables. Therefore, processing "connected" variables together seems intuitively justified. For example, if a large SAT instance is not satisfiable because of a small group of inconsistent variables, the variables in this group must be "connected" by some clauses. If we can partition all variables into, say, two largely independent groups, then such a function is likely to be represented by a BDD with a small cut, i.e. there will be relatively few edges between these two groups. BDDs with many small cuts tend to have fewer edges, and therefore fewer vertices (since every vertex is a source of exactly two edges). This intuition suggests that we interpret CNF formulae as hypergraphs by representing variables by vertices and clauses by edges. Two vertices share an edge if the two corresponding variables share a clause in the formula. Applying balanced min-cut partitioning to such hypergraphs separates the original CNF formula into relatively independent subformulae. Ordering the variables in each part together would be a step towards ordering "connected" variables next to each other, as advocated earlier. Once the first partitioning is performed, the parts can be partitioned recursively. This process can provide a complete variable-ordering. We note that cuts of CNF formulae have been studied in [21], and instances having small cuts were theoretically shown to be "easy" for SAT. Our work seeks constructive and efficient ways to amplify the "easiness" of CNF instances with small cuts by finding good variable-orderings. Additionally, Berman [1] related the size of BDDs to circuit width.

2.1 Recursive Bisection and Hypergraph Placement

Recursive min-cut bisection of hypergraphs has been intensively studied in the context of VLSI placement for at least 30 years. In particular, the recursive bisection procedure described earlier for CNF formulae corresponds to the *linear placement problem* [12], where hypergraph vertices are placed in one, rather than in two, dimensions. It is well-known that placement by recursive bisection leads to small "half-perimeter wire-length" that translates back to small average clause span in CNF formulae. Here we define the *span* of a clause with respect to a variable-ordering as *the difference between the greatest and the smallest variables in this clause* (so that the span exactly corresponds to the half-perimeter wirelength of a hyperedge). We can also define the *i-th cut* with respect to a given ordering as the number of clauses including variables with numbers both less than and greater than $i+0.5$.

Observation: Given a variable-ordering, the total clause span equals the sum of all cuts. The average clause span is proportional to the average cut, and the coefficient is approximately equal to the clause-to-variable ratio of the CNF formula.

Recursive bisection minimizes *both* average clause spans and cuts, therefore we will use the leading-edge hypergraph placer Capo [5] based on recursive min-cut bisection [6, 14]. Capo implements several improvements to classical recursive bisection, reducing the total clause span. Such techniques include bisection with high balance tolerance and adaptive cut-line selection, which allows greater freedom in partition sizes in order to improve the cut. The underlying multi-level hypergraph partitioner MLPart [6] outperforms the well-known hMetis [14], while both rely on Multi-Level Fiduccia-Mattheyses (MLFM) partitioning heuristics. Since the MLFM heuristic is randomized, it returns different solutions on every call (we call it a *start*). On every call, MLPart executes two independent *starts* and applies one V-cycle to further improve the better solution.

We propose the following heuristic that orders variables in CNF formulae (see Figure 1). An initial CNF formula (that may originate from circuits or other applications) is converted into a hypergraph (see Figure 2). An ordering of hypergraph vertices is

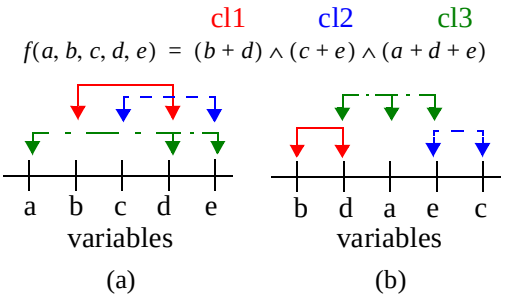

Fig. 2: Example of (a) default vertex-ordering (b) improved vertex-ordering

then found via min-cut linear placement and translated back into an ordering of CNF variables. The original CNF formula is reordered and used (i) as input to an arbitrary SAT solver, or (ii) to construct a BDD representation of the boolean function it represents. The results produced by SAT solvers and BDD manipulations are then translated back into the original variable order.

Note that this approach does not require modifications in SAT solvers, BDD manipulation software or the min-cut placer.[1] We call this heuristic MINCE (MIN-Cut, Etc.) and implemented it by chaining publicly available software with PERL scripts.

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

Wood and Rutenbar have already used linear hypergraph placement as a variable-ordering technique for BDD minimization in 1998 [29]. However, they used spectral methods which entail converting hyperedges to edges and then minimizing quadratic edge length, rather than the half-perimeter (linear) edge length. Spectral placement methods used in [29] do not appear to have direct connection to cut minimization. As of 2001, spectral methods for partitioning and placement are practically abandoned due to their unacceptable runtime on large instances and poor solution quality as measured by half-perimeter edge length. This can be contrasted with min-cut placement that is among the fastest known approaches, provides good solutions and is obviously related to cut minimization.

On the empirical side, our results with BDD minimization presented below show that MINCE, by itself, outperforms variable-sifting (used without static ordering) in both runtime and memory. According to [11], as of 2000, variable sifting is the best published dynamic variable reordering heuristic for BDDs with near-linear performance.[2] From this, we conclude that our proposed technique outperforms all other published scalable approaches to BDD minimization. Of course, dynamic variable reordering techniques can be applied on top of MINCE or can use MINCE order as a tie-breaker.

1. Commercial EDA software can be used, e.g. Cadence QPlace.
2. Some generic or simulated annealing reordering algorithms can generate smaller BDDs but may incur longer runtimes.

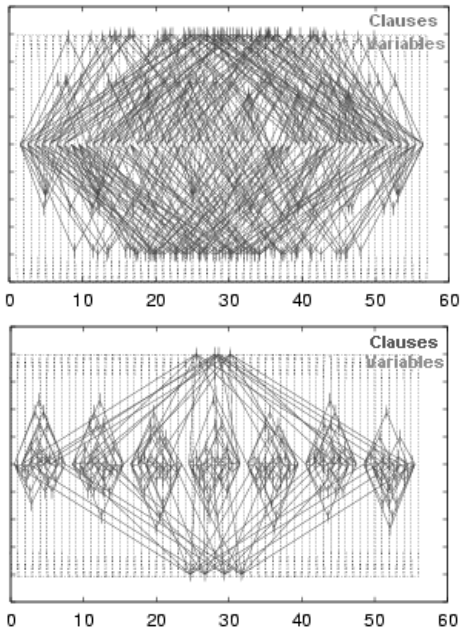

Fig. 3: Sample hypergraph representing the structure of the *hole-7* instance using (a) default vertex-ordering (b) improved vertex-ordering

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 [10].

2.2 Illustration

Figure 2 illustrates the difference between a good and a bad variable order for a CNF formula. We use the Capo placer to find an ordering of vertices, i.e. variables, that produces a small total (equivalent average) clause span. Figure 2(b) shows a sample order returned by MINCE for the example described. The total span of all clauses in this CNF formula is reduced from 8 to 4 by this better variable order. In addition, the number of edges crossing each variable (cut) is reduced. The original problem has a maximum *variable cut* (at variable c) of 3 which is reduced to 1 in the MINCE order.

In general, *structured* problems such as the *hole-n* series of benchmarks (e.g., hole-10, hole-11, etc.) are divided by MINCE into several partitions. Figure 3 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 Figure 3(b), this reduction exposes the problem's structure. Our experiments show that such MINCE variable-ordering generally speeds up SAT solvers and improves runtime/memory of BDD manipulations.

Similar techniques and intuitions apply in related contexts. For example, one can apply MINCE to DNF formulae rather than CNF formulae. In this and related cases, one starts with a description of a Boolean function that is sparse, i.e., connects very few groups of variables (by clauses, minterms, in terms of circuit connectivity, etc.). Recursive partitioning orders the "connected"

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

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

Selected Instances	MSTS Time	MSOS Time	DLCS Time	DLIS Time	Fixed Time	MINCE Order +	Solve =	Total
aim100-2n2	0.04	0.02	**0.01**	**0.01**	**0.01**	0.72	0.01	0.73
bf0432-007	1.72	3.85	1.74	**1.48**	10K	7.34	1.6	8.94
hanoi4	4.54	2267	10K	10K	**1.75**	8.8	1.66	10.5
hole8	6879	10K	140	70.3	61	**0.44**	**6.44**	**6.88**
hole9	10K	10K	1556	752	623	**0.53**	**52.8**	**53.3**
hole10	10K	10K	10K	8637	5597	**1.94**	**599**	**601**
ii16b1	174	217	10K	10K	4840	**90.7**	**0.47**	**91.2**
ii16b2	133	153	71.3	238	5507	**53.2**	**1.38**	**54.6**
ii32c4	650	696	24.9	**1.24**	10K	84.3	6.11	90.4
par16-2-c	1321	1325	2469	3570	184	**3.16**	**110**	**113**
par16-5	7329	7348	315	10K	111	**11.3**	**14.4**	**25.7**
pret150_25	0.15	0.13	0.14	**0.12**	**0.12**	0.62	0.14	0.76
ssa0432-003	**0.04**	0.04	0.06	0.05	0.06	1.91	0.03	1.94
ssa2670-141	95.2	9.78	2.15	**1.1**	10K	6.9	1.41	8.3
Nqueens-20	482	1485	**23**	24.9	3160	40	0.31	40.3
Nqueens-25	10K	10K	178	183	94.9	**93**	**0.79**	**93.4**
Nqueens-30	10K	10K	5233	5402	10K	**217**	**2.27**	**219**
Nqueens-35	10K	10K	10K	10K	10K	**317**	**1.06**	**318**

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

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.

3 Application of MinCut to SAT & BDDs

In this section, we present experimental evidence of the improvements obtained by MINCE. We used GRASP as our SAT solver [23] and CUDD as our BDD engine [25]. Experiments were conducted on a Pentium-II 333 MHz, running Linux with 512 MB RAM. We used the DIMACS [8] and the *n-queens* CNF benchmarks, as well as flat versions of the ISCAS89 circuit benchmarks [3] expressed in CNF. For all experiments, the CPU time and memory limits were set to 10K seconds and 500 MB, respectively.

SAT Experiment: Table I and Table II show runtime in seconds for MINCE versus the dynamic MSTS, MSOS, DLCS and DLIS orderings, as well as the static *fixed* variable-ordering [23]. "#I" denotes the number of instances solved by each decision heuristic. The average *variable cut* is included for the original and the MINCE variable orders. As the data clearly illustrate, deciding on closely-connected variables leads to a reduction in search time. Since "connected" variables are ordered next to each other, this approach allows the solver to quickly identify and avoid unpromising partial solutions. In other words, instead of deciding on variables from separate partitions, one partition is considered at a time. This approach is more effective on *structured* problems, such as the *hole-n* or the *n-queens* problem, which consist of multiple partitions. On these problems, MINCE finds variable orders compatible with the problem's structure, and this speeds up SAT solvers and BDD engines. For example, a speedup of 16, 16, 16, 14, and 9, was obtained for the *hole-10* benchmark over the MSTS, MSOS, DLCS, DLIS, and *fixed* decision heuristics, respectively. MINCE also achieved significant speed-ups over other decision heuristics for large instances from the *n-queens* set. Particularly, none of the dynamic or *fixed* decision heuristic were able to solve the *nqueens-35* instance in 10K seconds, but it was solved in less than 320 seconds using MINCE.

In general, GRASP run time is almost always reduced when the recursive bisection ordering is used. However, for particularly easy[1] SAT instances recursive bisection itself requires more time than GRASP[2] with either *fixed*, MSTS, MSOS, DLCS, or DLIS ordering. Observe that MINCE has a worst- and best-case performance of $\Theta(N\log^2 N)$. On the other hand, SAT problems have an exponential worst-case and a best-case of $\Theta(N)$, where N is the number of variables. Hence, *MINCE should be useful in solving instances that otherwise require exponential runtime*, but unhelpful in solving easy instances.

To explore the variability of orders returned by independent random starts of MINCE, we applied GRASP to three different orders of benchmark ii32d3.cnf, generated by MINCE. Two cases time out in 10K seconds and the third was solved in 14.8 seconds. This empirically confirms the heavy-tail distribution theory for SAT instances [10] and suggests that a solution can be produced in 60 seconds if multiple starts of the SAT solver are launched with a 20-sec time-out (MINCE took 55 seconds per order on that instance, which is negligible compared to a 10K time-out).

Although not presented in the tables of results, we tested the given benchmarks using the SATO SAT solver [30]. SATO implements an intelligent dynamic decision heuristic and was able

1. We define *easy* instances as those that can be solved in near-linear time.
2. Same is expected with comparable and faster solvers, e.g. Chaff [18].

Instance	Fixed		Random		Fixed-Sifting		Random-Sifting		MINCE				Avg Cut	
	Build Time	Max Node	Build Time	Max Node	Build Time	Max Node	Build Time	Max Node	Time Order + Build = Total			Max Node	Fix	New
s208.1	timeout		timeout		14.8	6.4	21.57	13.6	**0.88**	**0.66**	**1.54**	**3.4**	104	16
s27	**0.06**	**0.2**	0.07	0.2	0.08	0.2	0.08	0.2	0.18	0.07	0.25	0.07	11	5
s298	timeout		timeout		47.83	28.0	56.55	26.8	**0.97**	**3.03**	**4**	**14.5**	157	28
s344	timeout		timeout		525.1	130.5	703.45	192.1	**1.28**	**7.48**	**8.76**	**14.2**	137	17
s349	timeout		timeout		267.78	83.3	707.24	248.3	**1.36**	**10.77**	**12.13**	**19.3**	149	18
s382	timeout		timeout		159.08	88.2	113.25	32.6	**1.23**	**5.48**	**6.71**	**13.6**	176	26
s386	timeout		timeout		258.12	96.7	168.84	48.0	**1.8**	**91.74**	**93.54**	**310.4**	172	55
s400	timeout		timeout		564.91	193.9	292.7	114.9	**1.12**	**5.8**	**6.92**	**20.0**	182	26
s420	timeout		timeout		361.84	93.9	590.79	122.5	**1.47**	**4.69**	**6.16**	**17.7**	183	19
s444	timeout		timeout		252.83	85.0	605.12	241.9	**1.71**	**5.02**	**6.73**	**7.7**	192	25
s526	timeout		timeout		timeout		timeout		**2.92**	**17.74**	**20.66**	**37.7**	271	42
s526n	timeout		timeout		timeout		timeout		**1.99**	**10.35**	**12.34**	**18.45**	262	40
s641	timeout		timeout		timeout		timeout		**2.35**	**42.12**	**44.47**	**158.9**	190	23
s713	timeout		timeout		timeout		timeout		**2.86**	**62.86**	**65.72**	**174.3**	216	25
s838.1	timeout		timeout		timeout		timeout		**3.74**	**105.46**	**109.2**	**147.8**	419	29
s838	timeout		timeout		timeout		timeout		**3.82**	**322.13**	**325.95**	**885.6**	366	29
aim-100-1_6-no-1	0.55	33.0	0.45	18.9	**0.33**	**3.1**	0.45	3.5	0.72	0.08	0.8	0.2	84	32
dubois50	timeout		timeout		12.36	3.2	14.66	4.3	**0.69**	**0.25**	**0.94**	**0.4**	201	11
hole10	26	131.1	116.43	3390.7	12.75	19.0	12.37	19.0	**1.46**	**0.38**	**1.84**	**19.5**	201	30
hole8	2.08	20.2	4.43	145.4	3.01	5.1	3.03	4.1	**0.44**	**0.14**	**0.58**	**3.8**	108	21
hole9	7.74	52.2	23.3	810.5	5.37	8.5	5.5	10.2	**0.62**	**0.2**	**0.82**	**8.7**	149	25
ii8a1	25.71	372.0	timeout		4.43	7.2	7.22	6.8	**0.89**	**1.18**	**2.07**	**17.6**	75	20
par16-1-c	535.65	893.8	timeout		1826	500.5	2140	434.2	**3.84**	**115.71**	**119.55**	**171.5**	271	101
par8-1	133.14	90.7	113.37	160.0	88.38	34.5	97.35	28.6	**2.43**	**32.07**	**34.5**	**36.2**	253	39
pret150_25	timeout		timeout		649	271.6	302.82	156.1	**0.7**	**1.01**	**1.71**	**3.4**	152	18
ssa0432-003	timeout		timeout		timeout		timeout		**2.83**	**29.87**	**32.7**	**168.7**	287	46

TABLE III: Statistics for constructing the BDDs of the ISCAS89 Circuits and Selected DIMACS Benchmarks

to solve the given DIMACS benchmarks in approximately 45,000 seconds (4 instances timed-out after 10,000 seconds) as opposed to 16,200 seconds using GRASP with recursive bisection ordering. However, for some instances, SATO was faster. MINCE failed to generate effective variable-orderings for these instances, since most of them were not structured.

Our preliminary experiments with the recently published Chaff SAT solver [18] indicate that MINCE is not helpful on most standard benchmarks. This, in part, is due to the highly optimized implementation of Chaff, but is also explained by the relative simplicity of the instances. Indeed, if an instance of an NP-complete problem is solved in near-linear time by a generic algorithm, this instance must be easy. Note, however, that while the worst-case complexity of both GRASP and Chaff is exponential, MINCE always runs in near-linear time, perhaps with a greater constant. Finally, even when MINCE's runtime makes it prohibitively expensive for a particular SAT instance where it reduces a solver's runtime, capturing the instance structure may lead to a better understanding and be useful for practical purposes.

BDD Experiment: Table III shows the BDD construction runtimes for circuit consistency functions of the ISCAS89 circuit benchmarks and selected Boolean functions from the DIMACS set. Note that this is not representative of symbolic state traversal, but is a standard experimental procedure for evaluating BDD packages [13]. The table shows runtimes (sec) and numbers of nodes (K) in BDDs constructed using the *fixed*, *random*, *fixed* with *sifting*, *random* with *sifting*, and MINCE orderings, respectively. Clearly, the MINCE 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". MINCE enabled the BDD construction for all 16 ISCAS89 circuits as opposed to only 10 with sifting and 1 with a *fixed* variable-ordering. MINCE's ordering time is negligible in most cases. MINCE reduced the average *variable cut* for the ISCAS89 circuits from 200 to 26, and for selected DIMACS benchmarks from 178 to 34. The technique is simple and easy to use in practice. Its static nature allows for a variety of applications where dynamic approaches fail.

4 Conclusions & Future Work

Our work proposes a static variable-ordering heuristic MINCE for CNF formulae with applications to SAT and BDDs. The main advantage of this heuristic is its very good performance on standard benchmarks in terms of implied runtime of SAT solvers as well as memory/runtime of BDD primitives. We believe that this is due to the fact that the proposed variable-ordering is *global* and relies on high-performance hypergraph partitioning and placement (MLPart [5] and CAPO [6]). Unlike problem-specific dynamic variable-ordering heuristics, such as DLCS, DLIS, and variable-sifting, MINCE can be implemented once and used for different applications without modifying the application code. Given that MINCE shows strong improvements in seemingly unrelated applications (SAT and BDD) and for a wide variety of standard benchmarks, we believe that it is able to capture structural properties of CNF instances. 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. In general, this technique should have better impact on BDDs, since they are more sensitive to variable-ordering than SAT. SAT solvers can

reduce the damage incurred by a bad variable-ordering using the addition of conflict-induced clauses (a conflict clause connects literals of related variables even if they are very far from each other in the ordering).

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

Our on-going work addresses additional types of benchmarks, better justifications of the MINCE heuristic and also analyses of the cases when it fails to produce near-best variable-orderings. An important research question is to account for polarities of literals. We are aware of work conducted in [28] which is similar to ours. Our colleagues use hMetis, modify the source-code of GRASP and attempt to account for polarities of literals by post-processing. Comparisons of preliminary results show that MINCE is surprisingly successful without using polarities of literals. We are also looking into further improving the runtimes by detecting symmetries in the problem's structure. A public-domain implementation of MINCE is available at *http://andante.eecs.umich.edu/mince*.

5 Acknowledgments

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

6 References

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

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

[3] F. Brglez, D. Bryan, and K. Kozminski, "Combinational problems of sequential benchmark circuits," *in Proc. of the International Symposium on Circuits and Systems*, 1989.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[18] 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)*, 2001.

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

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

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

[22] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *in Proc. of the International Conference on Computer Aided Design (ICCAD)*, 1993.

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

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

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

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

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

[28] D. Wang and E. Clarke, "Efficient Formal Verification through Cutwidth," *unpublished manuscript*, 2001.

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

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