ZBDD-Based Backtrack Search SAT Solver

Fadi A. Aloul, Maher N. Mneimneh, Karem A. Sakallah
Department of Electrical Engineering and Computer Science
University of Michigan
{faloul, maherm, karem}@eecs.umich.edu

Abstract

We introduce a new approach to Boolean satisfiability that combines backtrack search techniques and zero-suppressed binary decision diagrams (ZBDDs). This approach implicitly represents satisfiability instances using ZBDDs, and performs search using an efficient implementation of unit propagation on the ZBDD structure. We describe how to perform backtrack search using ZBDDs as the underlying structure for clause representation. This methodology, which adapts backtrack search algorithms to such implicit representations, allows for a potential exponential increase in the size of the problems that can be handled. Our experimental results show consistent speedups over conventional approaches.

1 Introduction

In the past few years, interest in the direct application of Boolean Satisfiability (SAT) to various Electronic Design Automation (EDA) computational tasks [9, 14, 15, 17] has been on the rise. Several powerful SAT solvers [11, 13, 20] have been proposed, many of which using one or another variation of the Davis-Logemann-Loveland (DLL) [5] approach. Several of these variations employ some form of learning new clauses that were not part of the original set [11], optimize the performance of unit propagation [13, 19], and propose intelligent decision heuristics to enhance the search performance [11, 13, 20].

These improvements extended the application of SAT solvers to large problem instances. Nevertheless, despite these advances, the continuing growth in today's designs is outpacing the capabilities of existing SAT solvers as these tend to *explicitly* represent the clause database. In general, *explicit* representation and manipulation of large problems typically leads to time and memory explosion.

Rather than *explicitly* representing the clause database using arrays or linked lists, an alternative is to *implicitly* represent the clause database. Zhang et al. [18] introduced an efficient implementation of the DLL procedure using a trie to represent the clauses. Tries allow sharing of nodes among clauses beginning with identical sequences of literals.

Recently, Chatalic et al. [3] proposed a new implementation of the *resolution*-based Davis-Putnam (DP) [4] procedure using zero-suppressed binary decision diagrams (ZBDDs) [10, 12] as the underlying data structure for clause encoding. This approach succeeded in solving several SAT instances that defied search-based methods. Unlike tries, ZBDDs allow the sharing of nodes among clauses with common literals regardless of the location of literals in the clauses. The high compression power of this data structure resulted in exponential reductions in space and time complexity for certain instance classes.

In this paper, we present a new *search*-based technique, where the focus is on the data structure used for encoding sets of clauses. ZBDDs are used to implicitly represent the clause database, and unit propagation is implemented by an efficient procedure that processes *sets of, instead of single, clauses*. Our initial implementation does not include many of the recent enhancements to backtrack search, such as conflict diagnosis and recursive learning; our objective was simply to determine if ZBDDs, as data structures, outperform conventional representations for backtrack search. Experimental results do indeed confirm the memory reduction advantages of ZBDDs over lists and tries. Furthermore, reduced memory consumption consistently translates into faster runtimes despite the initial overhead of building the ZBDD structure. We consider these results to be quite positive and conjecture that incorporation of various forms of clause learning will further accentuate the performance advantage of ZBDDs.

Aloul et al. [1] proposed an approach where search is performed by explicitly manipulating the ZBDD data structure. However, multiple ZBDDs must be constructed during the search process, which incurs significant runtime overhead. We propose a more efficient method where a single ZBDD is constructed to represent the CNF problem and search is performed by manipulating pointers to the ZBDD nodes.

The organization of the paper is as follows. In Section 2, we present a variety of data structures used to represent clause databases and describe recent efficient implementations of unit propagation [13, 19]. We describe how to use ZBDDs as an underlying data structure in Section 3. The proposed *backtrack*-based algorithm using ZBDDs is presented in Section 4. Finally, experimental results and conclusions are presented in Section 5 and Section 6, respectively

2 Preliminaries

Most modern complete SAT algorithms can be classified as enhancements to the basic DLL *backtrack* search approach [5]. The DLL procedure performs a depth-first search in the *n*-dimensional space of the problem variables and can be viewed as consisting 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.

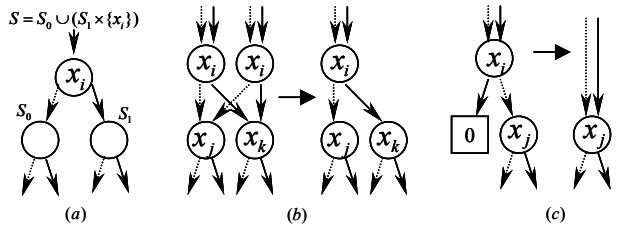

Figure 1: (a) ZBDD node semantics
(b) node merging rule and (c) node elimination rule

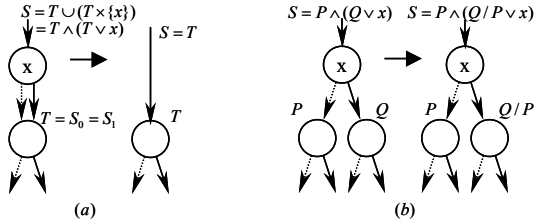

Figure 2: (a) clause subsumption rule and
(b) subsumption elimination rule

- *A diagnosis* engine that handles the occurrence of conflicts (i.e., assignments that cause the formula to become unsatisfiable) and backtracks appropriately.

The deduction engine in the DLL procedure is based on the application of two rules: 1) the *unit clause rule* which forces the assignment of the only unassigned variable in a clause whose other literals are all 0; and 2) the *pure literal rule* which forces the assignment of monoform variables to the values that satisfy all the clauses containing them. *Boolean Constraint Propagation* (BCP) is achieved by the repeated application of the unit clause rule over a given clause database, and is known to identify all possible implications of the decisions made thus far. Recent enhancements to the DLL procedure have focused on improving the DLL engines or the data structure used to represent the problem. In what follows, we describe two effective enhancements. The first uses a trie data structure to implicitly represent the clause database. The second enhances the deduction engine by implementing an optimized BCP procedure.

In general, most SAT solvers store the SAT problem, represented in *conjunctive normal form* (CNF), explicitly using arrays or linked lists [11, 13]. Unfortunately, this method does not scale well for large instances, especially when a large number of "learned" clauses are added to the problem. As described earlier, tries have been used successfully to implicitly represent the CNF problem [18, 20]. The reason for this success is their ability to share nodes among clauses beginning with identical sequences of literals. Tries used for encoding CNF problems are 3-ary trees whose nodes correspond to literals in the CNF problems. Each path from a root node to a leaf represents a clause in the problem. Each node has a label (var) representing a problem variable and three edges $(pos, neg, other)$ each denoting a clause set in which the variable appears positively, negatively, or not at all. Such tries are built assuming a fixed variable order.

More recently, enhancements to the implementation of BCP were shown to yield significant performance improvements [13, 19]. Noting that a sizable fraction of a SAT solver's runtime is spent in the BCP procedure, these enhancements can be viewed as a form of "lazy" evaluation that avoids unnecessary traversals of the clause database. In conventional BCP procedures, whenever a variable v is assigned, all clauses containing literals of this variable are traversed to check whether they have become unit or are in conflict. In other words, an implication step requires time bounded by the number of existing literals of the assigned variable. Zhang et al. [19] presented a very efficient implementation of an amortized linear time BCP algorithm. The basic idea is to keep track of the first (head) and last (tail) unresolved literals in each clause using two pointers. A unit clause can, then, be easily detected when the two pointers concur. Furthermore, such an approach limits the search to literals appearing in the clause between the first and last unresolved literals. This method, however, has the drawback that the first/last pointers must be updated whenever a variable is unassigned. A variation of this algorithm, proposed recently by Moskewicz et al. [13], just keeps track of *any two unresolved literals* in each clause. Unlike Zhang's algorithm, this variation incurs no penalty when a variable is unassigned, but is more likely to traverse more literals in each clause when a variable is assigned.

Empirically, such BCP optimizations show great improvements over conventional BCP implementations, especially for problems containing large number of large clauses; for example, a problem consisting of n k-literal clauses needs $2n$ pointers instead of kn pointers.

3 Zero-Suppressed Binary Decision Diagrams

ZBDDs [10, 12] were inspired by the need to efficiently represent and manipulate sets of combinations. A ZBDD is a directed acyclic graph (DAG) consisting of two terminal nodes, the 0-terminal (the empty set) and the 1-terminal (the set of a single empty combination), and non-terminal nodes each of which has two children, the 1-successor and the 0-successor. In addition, each non-terminal node is labeled with a Boolean variable. Given a universe $U = \{c_1, c_2, \ldots, c_n\}$ of n objects, a combination $C = \{c_1, c_2, \ldots, c_m\}$ of m objects from U can be represented by an n-bit *binary* vector $X = (x_1, x_2, \ldots, x_n)$ where $x_i = 1$ if object c_i is in C, and 0 otherwise, $1 \le i \le n$. A set S of combinations can be represented by a characteristic function $\chi_S: \{0,1\}^n \rightarrow \{0, 1\}$ where $\chi_S(X) = 1$ if $X \in S$ and 0 otherwise, $X \in \{0, 1\}^n$. In what follows, we use a set S and its characteristic function χ_S interchangeably.

ZBDD node semantics are illustrated in Figure 1(a). If a node v with label x_i represents a set S, and v's 0-successor and 1-successor represent S_0 and S_1 respectively, then

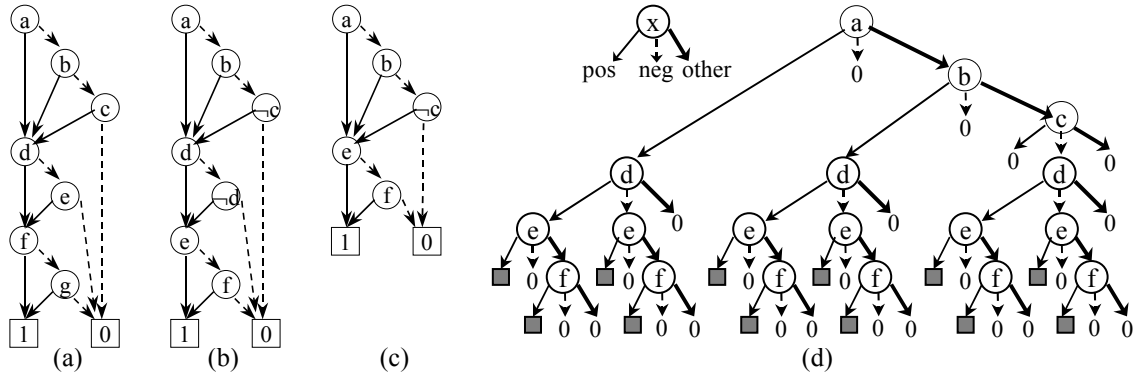

Figure 3: (a) ZBDD representing set S in Eq. (2), (b) ZBDD representing clause set φ in Eq. (3), (c) ZBDD representing clause set φ in Eq. (3) after applying reduction rules, and (d) *Trie* representing clause set φ in Eq. (3)

$S = S_0 \cup (S_1 \times \{c_i\})$, where:

$$A \times B = \bigcup_{a \in A,\, b \in B} (a \cup b). \quad (1)$$

For example, given the combination sets $A = \{\{g,h\},\{h,f\}\}$ and $B = \{\{g,r\},\{e\}\}$, their product is[1]

$$A \times B = \{\{g,h,r\},\{h,f,g,r\},\{g,h,e\},\{h,f,e\}\}. \quad (2)$$

ZBDD construction is based on two reduction rules illustrated in Figure 1(b) and (c). The *node merging rule* merges two nodes if they have the same label and identical 0- and 1-successors, whereas the *node elimination rule* eliminates a node if its 1-successor is the 0-terminal. Each path from the root node to the 1-terminal corresponds to one combination C of S where $x_i = 0$ if no node labeled x_i exists along the path, or the successor leaving the x_i node on the path is the 0-successor. It is this property that renders ZBDDs a compact representation for sparse combinations. As an example consider the universe $U = \{a, b, c, d, e, f, g\}$ and the set:

$$S = \begin{matrix} \{\{a,d,f\},\{a,d,g\},\{a,e,f\},\{a,e,g\} \\ \{b,d,f\},\{b,d,g\},\{b,e,f\},\{b,e,g\} \\ \{c,d,f\},\{c,d,g\},\{c,e,f\},\{c,e,g\}\} \end{matrix} \quad (3)$$

S can be represented by the ZBDD shown in Figure 3(a). Minato [12] presented efficient algorithms that implement set theoretic operations on ZBDDs. These operations include *union*, *intersection, difference*, and *product*, among others.

It was demonstrated in [3, 12] that the above approach can be extended to efficiently encode sets of clauses. In this case, each variable and its complement are objects of U, and each path from the root to the 1-terminal corresponds to a single clause. The number of paths to the 1-terminal equals the number of clauses in the clause database. As an example, the clause database:

$$\varphi = \begin{matrix} (a \vee d \vee e) \wedge (a \vee d \vee f) \wedge (a \vee \neg d \vee e) \wedge \\ (a \vee \neg d \vee f) \wedge (b \vee d \vee e) \wedge (b \vee d \vee f) \wedge \\ (b \vee \neg d \vee e) \wedge (b \vee \neg d \vee f) \wedge (\neg c \vee d \vee e) \wedge \\ (\neg c \vee d \vee f) \wedge (\neg c \vee \neg d \vee e) \wedge (\neg c \vee \neg d \vee f) \end{matrix} \quad (4)$$

corresponds to a set of combinations from $U_\varphi = \{a, \neg a, b, \neg b, c, \neg c, d, \neg d, e, \neg e, f, \neg f\}$ and can be represented by the ZBDD shown in Figure 3(b). Using this approach, the semantics of Boolean Algebra, such as subsumption, can be superimposed on ZBDD reduction rules to achieve further compression. As an example consider the ZBDD illustrated in Figure 2(a) where the 1-successor and the 0-successor of the root are identical. Using ZBDD node semantics, $S = T \wedge (T \vee x)$ and by the subsumption rule of Boolean Algebra, $S = T$. Another CNF-specific reduction rule is the *subsumed difference* [3]: given two sets S and T, the subsumed difference of S by T, denoted as S/T, is the set of clauses of S that are not subsumed by any clause of T. The clause set S represented by a ZBDD node, whose 0- and 1-successors represent sets P and Q respectively, can be expressed as $S = P \wedge (Q \vee x)$. Since P is independent of x, clauses in P can subsume clauses in $Q \vee x$, while clauses in $Q \vee x$ can't subsume any clause in P. Consequently, $S = P \wedge [(Q \vee x)/P] = P \wedge (Q/P \vee x)$. This reduction rule is illustrated in Figure 2(b). It was shown that recursive application of this rule results in a ZBDD that is free of subsumed clauses. In addition, the subsumed difference can be used as the building block for *subsumption-free union* and *subsumption-free product* operations [3].

The small example in Figure 3, illustrates the compression power of ZBDDs: clause set φ requires 36 nodes in an explicit list representation, 18 nodes using a trie, and just 7 nodes using a ZBDD. Further *reduction techniques*, proposed in [1], that perform a combination of Boolean algebraic manipulations (i.e. absorption, subsumption, and resolution) can be locally applied to further reduce the formula as well as the ZBDD size. Figure 3(c) shows the result of such reductions on clause set φ: whereas the original problem had 12 clauses, 36 literals, and required 7 ZBDD nodes, the reduced formula consist of 6 clauses, 12 literals,

1. Note that $A \times B \neq A \cup B$. For this example, $A \cup B = \{\{g,h\},\{h,f\},\{g,r\},\{e\}\}$.

and requires only 5 ZBDD nodes.

4 Implementing Backtrack Search Using ZBDDs

In the following we describe an efficient DLL implementation using ZBDDs as the underlying data structure. The approach we present borrows from the previous work described in Section 2. Specifically, we describe how to incorporate the efficient BCP algorithms described in [13, 19] with ZBDDs.

Our algorithm involves manipulating pointers to ZBDD nodes representing literals that are adjacent in the clause database. For each node p representing the literal l, we define *PrevLit(p)* and *NextLit(p)* as the set of nodes corresponding to l's previous and next adjacent literals, respectively. For example, the ZBDD in Figure 4(a) encodes the clause set:

$$(a + b + d + e) \wedge (a + \neg b + e) \wedge (a + c + \neg f) \qquad (5)$$

Thus, $NextLit(a) = \{b, \neg b, c\}$ denoting that b, $\neg b$, and c follow a in these three clauses, and $PrevLit(c) = \{a\}$ indicating that a precedes c in the last clause. Computing the sets *PrevLit(n)* and *NextLit(n)* for all nodes n can be done by a single traversal of the ZBDD.

The ZBDD-based DLL algorithm makes use of node pointers that serve the same function as the literal pointers used in [13]. Each variable v has a *value* field in addition to two lists, *PosNode* and *NegNode*, pointing to nodes labeled with literals v and $\neg v$, respectively. Initially, the *value* field is set to unassigned and the *PosNode* and *NegNode* lists are empty for all variables. Two nodes, representing literals, from every clause are then inserted into their corresponding *PosNode* or *NegNode* lists. We will refer to nodes in the *NegNode* or *PosNode* list as *marked* nodes.

In order to assign a variable v to *true*, the *value* field in the variable list is updated to 1. In the process of updating the marked nodes, we ignore the *PosNode* list and traverse the nodes in the *NegNode* list of variable v. For each node p in the *NegNode* list, we search *NextLit(p)* for an unassigned literal. The state of each node z in *NextLit(p)* can be classified into one of four cases:

a) Unmarked, representing a true or unassigned literal: The algorithm marks z and continues traversing the next node in *NextLit(p)*.

b) Marked, representing a true literal: The algorithm continues traversing the next node in *NextLit(p)*.

c) Marked, representing an unassigned literal: z could possibly represent a unit literal. The algorithm recursively traverses the nodes in *NextLit(z)* looking for an unassigned literal. If at least a single node from *NextLit(z)* identifies a path with only false literals, z is added to a *PossibleUnitVar* list. It is later implied if at least one node from *PrevLit(p)* identifies a path with no unassigned or satisfied literals.

d) Representing a false literal: The algorithm recursively traverses the nodes in *NextLit(z)* looking for an unassigned literal. If a path is detected, in which all nodes represent false literals, the procedure immediately returns without traversing the rest of the nodes in *NextLit(p)*. Since the given path could lead to a conflict, the nodes in *PrevLit(p)* are traversed. A conflict is detected, if a single node from *PrevLit(p)* identifies a path with only false literals. On the other hand, an implication is detected if a node from *PrevLit(p)* identifies a path with a single unassigned literal.

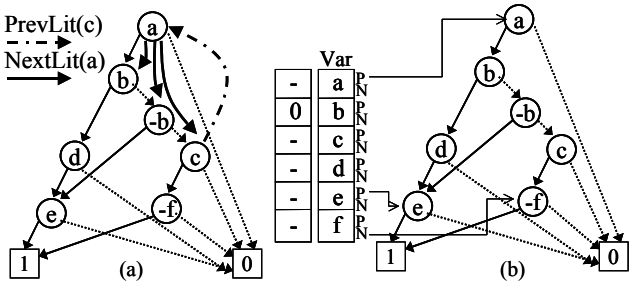

Figure 4: (a) Example showing *NextLit(a)* and *PrevLit(c)* (b) Example showing pointer updates

Inorder to preserve the two pointer invariant, p is unmarked only if the algorithm marks a node (using case-a) in each path traversed by the nodes in *PrevLit(p)* or *NextLit(p)*.

Figure 4(b) shows an example of a problem consisting of 3 clauses. Initially, the first and last literal are marked in each clause. When decision $b = 0$ is made, its *value* field is updated but no further pointer maintenance is necessary as this assignment involves unmarked variables only. Assuming that the second decision is $a = 0$, on the other hand, triggers traversals to update the node pointers. We start by traversing the nodes in *NextLit(a)*. The first node b in *NextLit(a)* is identified as a false literal. The algorithm recursively traverses the nodes in *NextLit(b)* to check if a conflict exists. Node d, which represents an unmarked, unassigned literal is detected. The algorithm marks node d and returns. The algorithm continues to check $\neg b$, the second node in *NextLit(a)*. Node $\neg b$ represents an unmarked, true literal which implies a satisfiable clause. Node $\neg b$ is marked and the algorithm returns and checks the final node in *NextLit(a),* node c. Again, c represents an unmarked, unassigned literal. The algorithm marks node c and returns. Since no possible unit literals or conflicts were detected, nodes in *Prevlit(a)* are not traversed. In addition, a is unmarked, since a node was marked in each path identified by its *NextLit(a)* nodes.

As shown in the above example, the variable decision order is independent of the ZBDD variable order. Furthermore, unassigning a variable is simply done by unassigning its corresponding *value* field and does not require any pointer updates. We should note that unassigning variables should be analogous to the order in which they were assigned.

5 Experimental Results

In this section, we present experimental evidence of the improvements obtained using ZBDDs, instead of *explicit* lists or tries, as the underlying data structure for the DLL pro-

TABLE I: Experimental Results for classic lists, lists, tries, and ZBDDs using VSIDS Decision Heuristic

Instance	S/U	ZBDD Comp.		Classic List	List	Trie			ZBDD			ZBDD Speedup		
		List	Trie	Solve	Solve	Build	Solve	Total	Build	Solve	Total	C-List	List	Trie
3blocks	SAT	3.07	1.69	429	191	0.28	101.8	102	0.28	93.4	93.7	4.58	2.04	1.09
4blocks	SAT	2.64	1.41	8319	2873	1.7	1239	1240	1.55	1186	1188	7.00	2.42	1.04
4blocksb	SAT	3.29	1.74	5258	1690	0.8	769.7	770	0.81	709.1	709	7.41	2.38	1.09
par16-2	SAT	1.37	1.32	16.72	12.51	0.39	10.43	10.8	0.41	6.45	6.86	2.44	1.82	1.58
pret60_25	UNS	1.22	1.14	49.52	49.66	0.03	46.01	46	0.03	45	45	1.10	1.10	1.02
hole8	UNS	2.48	2.66	7.88	7.91	0.04	7.49	7.53	0.04	6.42	6.46	1.22	1.22	1.17
hole9	UNS	2.73	2.87	111.2	111.2	0.04	106.2	106	0.04	89	89	1.25	1.25	1.19
hole10	UNS	3.08	3.19	1758	1763	0.04	1664	1664	0.04	1385	1385	1.27	1.27	1.20
NQueens15	SAT	1.52	1.35	168.7	151.4	0.27	108.2	108.5	0.29	103	103	1.63	1.46	1.05
barrel5	UNS	1.60	1.47	27.34	24.57	0.52	18.11	18.63	0.56	15.7	16.2	1.68	1.51	1.14
barrel6	UNS	1.62	1.48	197.1	180.6	1.17	126.4	127	1.21	98.4	99.6	1.98	1.81	1.28
barrel7	UNS	1.63	1.49	885	718	2.22	592.8	595	2.22	485	487	1.81	1.47	1.22
barrel8	UNS	1.58	1.45	2508	2177	4.15	1935	1939	4.29	1600	1604	1.56	1.36	1.21
longmult8	UNS	1.26	1.30	104	95.46	2.27	82.38	84.6	2.44	76	78	1.33	1.21	1.08
longmult9	UNS	1.34	1.38	225.9	203.7	2.63	181.8	184.4	2.66	162	164	1.37	1.24	1.12
longmult10	UNS	1.33	1.38	402	348.6	3.01	304.4	307	3.09	274	277	1.45	1.26	1.11
queueinvar12	UNS	1.52	1.32	30.66	25.99	1.67	23.94	25.61	1.78	21.2	23	1.33	1.13	1.11
queueinvar14	UNS	1.53	1.33	154.7	124.2	2.61	116.9	119.5	2.65	104	107	1.45	1.16	1.12
queueinvar16	UNS	1.51	1.36	622	539.9	1.71	493.7	495	1.83	428	430	1.44	1.25	1.15
queueinvar18	UNS	1.61	1.34	4535	3548	8.13	3291	3299	8.5	2755	2763	1.64	1.28	1.19
Average		**1.90**	**1.63**	**1290**	**742**	**1.68**	**561**	**563**	**1.74**	**482**	**484**	**2.25**	**1.48**	**1.16**

cedure. All experiments were conducted on a Pentium-II 333Mhz workstation, running Linux and equipped with 512 Mbytes of memory. Our algorithm is implemented in C++ and uses the CUDD package [16] to build the ZBDDs. The runtime limit was set to 10,000 seconds for all experiments.

We used a basic DLL implementation with the Dynamic VSIDS[1] (as used in Chaff [13]) variable decision (i.e. splitting) heuristic[2]. We present four sets of results corresponding to ZBDDs, tries, lists, and classic lists. The first three approaches initially use two literal pointers per clause; the tries and lists implementations replicate SATO [20] and Chaff [13], respectively. It should be noted, however, that the number of pointers per clause could increase beyond two during the search process for ZBDDs and tries. The last approach replicates the method implemented in GRASP [11] in which a pointer exists for every literal in the instance; unmarking nodes is disabled in this approach. In order to make the experiments fair, the initial two pointers were set to the first and last literals of each clause for the first three methods. In terms of ZBDDs, pointing to the first and last literal in every clause ensures that each clause has only two pointers. Specifically, the pointers are placed at the nodes identified by recursively traversing 0-successors starting at the root. Furthermore, we point to nodes whose 1-successor is the 1-terminal.

The search space size was identical for all four approaches. Specifically, the number of decisions, implications, and conflicts were equivalent for each instance among classic lists, lists, tries, and ZBDDs experiments.

We used a fixed ZBDD and trie variable order $(1, 2, \ldots, k)$. It is instructive to point out that different ZBDD or *trie* variable orderings can lead to further compression. In terms of benchmark suites, we focused on the bounded model checking (BMC) [3] benchmarks, in addition to various benchmarks from the DIMACs [6], Bejing [7], and NQueens [20] set.

Table I shows the compression capability obtained when using ZBDDs as opposed to explicit lists or tries. The first two columns list the compression power measured as the ratio of the *actual memory* used by the nodes in lists and ZBDDs and tries and ZBDDs, respectively. This implicit representation provides additional memory reduction. On average, ZBDDs were able to reduce the representation size by a factor of 1.9 over lists and 1.6 over tries. The *hole10*, representing a pigeon hole problem, is a structured instance; it achieves significant compression on the order of 3 over tries and lists. In general, structured problems are expected to exhibit higher compression ratios.

Table I also summarizes the search results using classic lists, lists, tries, and ZBDDs. The use of ZBDDs provides a reduction in search time for all problems, despite their construction time overhead. ZBDDs achieved an average speedup of 2.25, 1.5, and 1.16 over classic lists, lists, and tries, respectively, with a maximum speedup of 2.4 over lists in some cases. Our results show some correlation between compression rates and search runtimes. Specifically, ZBDDs are expected to provide higher speedups when the problem is highly compressed, since fewer number of nodes are traversed during the search process. For example, a speedup of

1. The order of the decision variables was set initially based on the frequency of occurrence of variables in the problem. Since conflict clauses are not generated, the order of decision variables didn't change during the search process.
2. Experiments with fixed decision heuristic yield similar speedups.

7.4 over classic lists for the *4blocksb* instance whose compression ratio is 3.3.

The table also shows the runtimes using the optimized BCP approach, expressed under *lists*, and the normal BCP approach, expressed under *classic lists*. Interestingly, the optimized BCP approach in lists leads to a reduction in search time when compared to classic lists in most cases but not all. This is justified by the fact that the "Watched Literal" method [13] used in the optimized BCP approach is more efficient with problems consisting of large clauses. These clauses are usually added by the conflict diagnosis process and tend to consist of many literals.

We believe that ZBDDs will be able to further compress clauses generated from conflict diagnosis, since these are typically subsumed by clauses learned later in the search process. The reuse of nodes in ZBDDs is also more likely for larger clauses which tend to appear among conflict-induced clauses. We also conjecture that reordering the ZBDD variables either by using sifting [8] or preprocessing the problem by analyzing its structure can lead to even higher compression ratios, and hence, faster runtimes.

6 Conclusions

We described a new implementation of the classic DLL search procedure that uses ZBDDs as the underlying data structure. Compared to explicit lists and tries, ZBDDs have the potential of significantly compressing a clause database leading to faster search times and the possibility of tackling much larger instances. This potential is indeed borne out by the preliminary experimental data presented in this paper. We believe that the incorporation of clause recording, through conflict analysis and recursive learning, will increase the performance edge of ZBDDs over conventional data structures. We are currently exploring appropriate ways of augmenting a ZBDD clause database with conflict-induced clauses during the search. Issues that must be addressed include weighing the tradeoff between adding clauses individually or in sets, or building separate ZBDDs for the conflict-induced clauses. We also propose further improving the unit propagation algorithm by caching ZBDD nodes during search.

7 Acknowledgments

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

8 References

[1] F. Aloul, M. Mneimneh, K. Sakallah, "Backtrack Search Using ZBDDs," *in International Workshop on Logic Synthesis*, pp. 293-297, 2001.

[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*, pp. 317-320, 1999.

[3] P. Chatalic and L. Simon, "Multi-Resolution on Compressed Sets of Clauses," *in International Conference on Tools with Artificial Intelligence*, pp. 2-10, 2000.

[4] M. Davis and H. Putnam, "A Computing Procedure For Quantification Theory," *in Journal of the ACM*, 7(3), pp. 201-215, 1960.

[5] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," *in Communications of the ACM*, 5(7), pp. 394-397, 1962.

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

[7] H. Hoos and T. Stützle, *http://www.satlib.org*.

[8] G. Hachtel and F. Somenzi, "Logic Synthesis and Verification Algorithms," *Kluwer Academic Publishers*, 1996.

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

[10] M. Lobbing, O. Schroer, and I. Wegner, "The Theory of Zero-Suppressed BDDs and the Number of Knight's Tours," *Workshop on Apps. of the RM Expansion in Circuit Design, 1995*.

[11] J. P. Marques-Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *in IEEE Transactions on Computers*, 48(5), pp. 506-521, 1999.

[12] S. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," *in Proc. of the Design Automation Conference*, pp. 272-277, 1993.

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

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

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

[16] F. Somenzi, CUDD: CU Decision Diagram Package, University of Colorado at Boulder, *ftp://vlsi.colorado.edu/pub/*.

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

[18] H. Zhang, and M. Stickel, "Implementing the Davis-Putnam Algorithm by Tries," *Technical Report, Univ. of Iowa*, 1994.

[19] H. Zhang, and M. Stickel, "An efficient algorithm for unit-propagation," *in International Symposium on Artificial Intelligence and Mathematics*, pp. 166-169, 1996.

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