Efficient SAT Solving: Beyond Supercubes [*]

Domagoj Babić Jesse Bingham Alan J. Hu

Department of Computer Science, University of British Columbia
{babic, jbingham, ajh}@cs.ubc.ca

ABSTRACT

SAT (Boolean satisfiability) has become the primary Boolean reasoning engine for many EDA applications, so the efficiency of SAT solving is of great practical importance. Recently, Goldberg *et al.* introduced *supercubing*, a different approach to search-space pruning, based on a theory that unifies many existing methods. Their implementation reduced the number of decisions, but no speedup was obtained. In this paper, we generalize beyond supercubes, creating a theory we call *B-cubing*, and show how to implement B-cubing in a practical solver. On extensive benchmark runs, using both real problems and synthetic benchmarks, the new technique is competitive on average with the newest version of ZChaff, is much faster in some cases, and is more robust.

Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: Computer-Aided Design;
I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*backtracking, graph and tree search strategies*

General Terms

Algorithms, Verification

Keywords

SAT, formal verification, learning, search space pruning

1. INTRODUCTION

The problem of satisfiability of boolean formulas (SAT) is a well-known NP-complete problem. In short, given a boolean function f, one needs either to find a satisfying assignment or to prove that such doesn't exist. SAT has been intensively used in many domains. Our focus is the application of SAT to structured problems, especially those resulting from EDA domain, like model checking (bounded [5] and unbounded [13]), FPGA routing [16], and ATPG [19]. Such industrial applications require complete SAT solvers, meaning that the solver must be capable of proving that a problem is either satisfiable or definitely unsatisfiable.

Since the early days of SAT solving [7], it was clear that the efficiency of SAT solvers depends heavily on search space pruning rules and decision heuristics. Decision heuristics have received a fair amount of attention in the literature [10, 3, 22]. Although some limited success has been achieved, many proposed heuristics are extremely sensitive to chosen parameters, and strongly dependent on the details of implementation of the rest of the solver[1]. In general, the heuristics tend to perform well only on a restricted set of problems.

On the other hand, in spite of the obvious relation between pruning rules and performance of SAT solvers, there exists only a handful of SAT pruning techniques. Even fewer are actually used in modern SAT solvers. Learning and conflict directed backtracking are probably the most well known, and work quite well. Inventing new pruning techniques for DPLL solvers is encumbered by the requirements for compatibility with the DPLL framework and existing pruning techniques. In addition, the savings achieved must justify the additional cost.

1.1 Existing Pruning Techniques

Boolean Constraint Propagation (BCP) and the Pure Literal Rule (PLR)[2] were proposed in [7]. Modern SAT solvers have efficient BCP engines for detection of unit clauses, propagation of unit literals, and conflict detection. A major step forward in SAT solving was the invention of lazy algorithms for BCP based on head/tail lists [22] and watched literals scheme [14].

The literals which appear in boolean formula with only one phase are called pure literals. According to PLR, all the clauses that contain pure literals may be eliminated. PLR is considered to be too expensive to be performed on every step in SAT solving, as exact counters of appearances of each literal need to be maintained. As it will be explained later, B-cubing subsumes PLR. So, our experimental solver HyperSAT uses PLR only during the preprocessing step and implicitly through B-cubing.

When a conflict is detected, the solver finds the reason for the conflict and tries to resolve it. The simplest method is to backtrack to the last decision variable which hasn't been explored with both phases, flip its current assignment, and proceed with the search. A more elaborate method is to analyze the conflict and backtrack to the decision variable that is actually responsible for the conflict. This scheme is called Conflict-Directed Backtracking (CDB) and GRASP [12] was the first SAT solver to implement it.

[*]This work was supported in part by a research grant from the Natural Sciences and Engineering Research Council of Canada and graduate fellowships from the University of British Columbia.

[1]Mainly clause database organization, learning mechanism, and preprocessing.

[2]PLR was called Affirmative-Negative Rule in the original paper.

Learning goes hand-in-hand with conflict-directed backtracking. Different solvers feature different learning strategies and clause deletion schemes. One thing in common is that they all traverse the implication graph in the reverse direction and add clauses that correspond to different cuts in the graph [23]. From experimental results [23] it seems that adding a clause that corresponds to the cut made before the first Unique Implication Point (1-UIP) is a better choice than other cuts proposed so far. Those results have recently been challenged by a suggestion that adding intermediate clauses that correspond to the cuts made closer to the conflict might perform better [18], but no experimental results were given. For a more extensive introduction into CDB and learning, and an exhaustive list of references, the reader is referred to [24].

Although much work remains to be done, it is clear that learning schemes proposed so far can use only a fraction of the information inferable from conflicts. The limiting factors are memory requirements and computation time required for reasoning. For example, due to memory constraints, it is impossible to add all the clauses that can be learned to the clause database. Similarly, some reasoning techniques, like hyper-resolution [2], have been shown to be computationally too expensive to be performed at every step during SAT solving.

Recently, a theory of essential points [17, 9] has been proposed. The theory unifies many existing search space pruning schemes (like PLR, CDB, and learning) under a single theoretical framework and serves as a tool for developing new pruning techniques. A new pruning technique called *supercubing* was proposed as an example of application of the theory of essential points. Their solver was a proof of concept, and although supercubing reduced the number of decisions, no actual speedup has been reported. Subsequent work [1] pointed out that supercubing is not readily compatible with learning and proposed an alternative backtracking and learning scheme to integrate supercubing and learning. The reported performance results of the solver were comparable to an earlier version of ZChaff (v2003.11.04)[14].

1.2 Contributions

In this paper, we generalize the theory of supercubing to introduce a new search-space pruning technique, performing a far more elaborate conflict analysis and moving beyond cubes as a way to store knowledge of learned conflicts. The theoretical ideal, which we dub B-cube, will blow up in space on practical problems, so we introduce a data structure *Boolean Constraint Trees* for compactly representing a safe approximation of the ideal B-cubes. The new technique can be made compatible with learning, but it requires significant modifications of the backtracking and decision making mechanisms, as in [1]. We have implemented our new technique in an experimental solver HyperSAT, which features both learning and B-cubing. Although HyperSAT is in its infancy, we report encouraging results and show that it can compete with leading-edge solvers like the newest version of ZChaff [14] on a wide range of problems.

2. NEW PRUNING TECHNIQUE

We start with some basic definitions, and continue with explanations of supercubing and B-cubing. The proofs are omitted, as the space constraints do not permit the presentation of the entire theoretical framework on which the proofs are based. We assume some basic familiarity with modern DPLL-based SAT solvers.

Let $\mathbb{B} = \{0,1\}$, and let $\mathscr{V}$ be a finite set of boolean variables. A *literal* is denoted by x^b, where $x \in \mathscr{V}$ and $b \in \mathbb{B}$. Define $\bar{0} = 1$ and $\bar{1} = 0$, then we say that the literal $x^{\bar{b}}$ is obtained by *flipping* x^b.

A *cube* (*clause*) is a conjunction (disjunction) of literals in which each variable from $\mathscr{V}$ appears at most once. A *minterm* (also called an *assignment*) is a cube in which each variable appears exactly once. The set of all minterms is denoted by M. A *CNF formula* is a conjunction of clauses. For $x \in \mathscr{V}$ and a cube c, we write $\mathit{flip}(c,x)$ to denote the cube formed by flipping the x-literal of c (if it exists), and for a set of cubes S, we define $\mathit{flip}(S,x) = \{\mathit{flip}(c,x) \mid c \in S\}$.

We assume a simple SAT solver which systematically explores a search tree without restarts or CDB, and the solver's input is a CNF formula φ. We use T to denote the binary search tree traversed by the solver[3]. The nodes of T are labeled with variables of $\mathscr{V}$. A *decision* is a node in T that has two children, the *0-child* and *1-child*, that correspond respectively to assigning 0 and 1 to the decision's variable. For a decision d and $b \in \mathbb{B}$, we let d^b denote the subtree of T rooted at the b-child of d.

Assuming φ is not satisfiable, the leaves of T are called *conflicts*. Both supercubing and B-cubing require the solver to construct a *decision conflict clause* (DCC) whenever a conflict is encountered. A DCC contains all the decision variables involved in the conflict, and is typically computed by traversing the implication graph backwards until the resolvent contains only decision literals [22]. The negation of a DCC is a cube (via an application of DeMorgan's Law) that we will call a *certificate* (of unsatisfiability) and denote by $\mathit{cert}(u)$, where u is a conflict node in T. The certificate $\mathit{cert}(u)$ has the property that no minterm m such that $m \to \mathit{cert}(u)$ will satisfy φ.

Consider a decision node d for variable x, and the certificates encountered when exploring d^b for some $b \in \mathbb{B}$. Note that for any such certificate c, c may or may not contain x^b, but c certainly doesn't contain $x^{\bar{b}}$. We are interested in those certificates c that involve x^b.

Definition 1. The set of all conflict nodes found in d^b that include the literal x^b will be denoted $A_b(d)$, where x is the variable of d.

Definition 2. Suppose that there are no satisfying assignments in d^b. The *B-cube* is then defined as a set of certificates $B_b(d) = \{\mathit{cert}(u) \mid u \in A_b(d)\}$ and we also define $B^*_b(d) = \mathit{flip}(B_b(d),x)$, where x is the variable of d.

Definition 3. Let $S_b(d)$ be the set of minterms defined by $S_b(d) = \{m \in M \mid m \to c \text{ for some } c \in B^*_b(d)\}$

THEOREM 1. *Suppose d^b has no satisfying assignments. Then for any minterm m found in $d^{\bar{b}}$ that satisfies φ, we have $m \in S_b(d)$.*

Supercubing and B-cubing are pruning techniques that both exploit Theorem 1 in the following manner. While exploring d^b, some over-approximation S' of $S_b(d)$ is computed[4]. Then, while exploring $d^{\bar{b}}$, attention is restricted to the assignments of S'; i.e. assignments in $d^{\bar{b}}$ that are not in S' are pruned. The difference between supercubing and B-cubing is that the latter's over-approximation is a tighter fit than the former's, hence B-cubing allows for more pruning.

2.1 Supercubing

Supercubing over-approximates $S_b(d)$ using a single cube, defined as follows. The supercube $sc_b(d)$ is the least cube that subsumes $S_b(d)$, i.e. $sc_b(d)$ is the conjunction of all literals ℓ such that $S_b(d) \to \ell$.

[3]For brevity, we leave T formally undefined in this paper.

[4]To be more precise, S' need only over-approximate the intersection of $S_b(d)$ with the subspace corresponding to $d^{\bar{b}}$.

Example 1. Suppose decisions in the search tree are (in order) x_1^0, x_2^1, x_3^0, and let d be the decision node for x_3. The solver explores the search subtree d^0 (i.e. x_3^0) and finds no solution. In the process of exploring d^0, the following five certificates are constructed:

$$\begin{aligned} c_1 &= x_2^1 \wedge x_4^1 \\ c_2 &= x_3^0 \wedge x_4^0 \wedge x_5^0 \wedge x_6^1 \\ c_3 &= x_1^0 \wedge x_5^0 \wedge x_6^0 \\ c_4 &= x_2^1 \wedge x_5^1 \wedge x_6^1 \\ c_5 &= x_1^0 \wedge x_3^0 \wedge x_4^0 \wedge x_5^1 \wedge x_6^0 \end{aligned}$$

Here we have that $B_0(d) = \{c_2, c_5\}$, and the least cube that covers $B_0(d)$ is $sc_0(d) = x_3^0 \wedge x_4^0$. Hence, since $\mathit{flip}(sc_0(d), x_3)$ over-approximates $S_0(d)$, in the subtree d^1 (i.e. after flipping x_3 to 1), the solver can immediately assign x_4^0.

The implementation of supercubing stores an array representing a supercube for each decision variable. Storing supercubes is not memory demanding, as the average size of the supercube per decision node is small (density of supercubes, [1]). Also, since decisions above d are the same in both d^0 and d^1, such variables need not be stored in the supercube, which reduces space requirements further.

Supercubing can prune the search space that can't be pruned by learning, as explained in [9]. An algorithm for computing supercubes and a thorough discussion of the integration of supercubing and learning are given in [1].

2.2 B-cubing

Going back to Example 1, the solver can immediately assign x_4^0 after x_3^1, but then there are no more literals that are common to all certificates of $B_0(d)$. However, there is a *variable* that appears in all certificates in $B_0(d)$, and that is x_5. So the solver can choose x_5 as a new decision variable. If x_5^1 is chosen, we can immediately assign x_6^0, since assignments in the space $x_3^1 \wedge x_5^1 \wedge x_6^1$ have already been certified to be unsatisfying by c_4. Similarly, after picking x_5^0, the variable assignment x_6^1 can be immediately asserted. In this manner, only assignment of $S_0(d)$ are considered after x_3 has been flipped, which is a legal pruning thanks to Theorem 1.

The fact that more information can be learned from certificates was first observed by Nadel [15] and implemented in Jerusat SAT solver. It seems that Jerusat keeps all the certificates and does the analysis when a new decision is needed. Needless to say, such an approach requires a huge amount of memory and it is infeasible even for moderately large problems. For that reason, Jerusat seems to keep certificates only for certain number of decision levels. When it backtracks out of the window, it discards certificates. This approach has several serious drawbacks.

First, certificates contain a significant amount of redundant information. In Example 1, certificates c_1 and c_2 both contain information that only x_4^0 needs to be explored after flipping x_3^0. Clearly, if we had a suitable data structure to represent the corresponding B-cube, less memory would be required.

Second, discarding certificates means that the search space will be less constrained and therefore more search will be needed. This is especially serious when the certificates are discarded for decision nodes close to the root of the search tree. For example, if the root decision node contains three literals in its supercube (or in the stem of its BCT data structure, as will be explained later), after flipping the root node, the supercube would ideally reduce the search space eightfold.

An advantage of the Jerusat approach is its simplicity. If all the conflicts are kept (within the predefined window), the reasoning procedure can be entirely implemented inside the decision engine. The technique we are proposing requires substantial changes in the backtracking mechanism, conflict analysis, and decision engine.

When it comes to the integration of B-cubing and learning, one runs into the same compatibility problems as with supercubing. This problem has been extensively discussed in [1].

3. APPROXIMATION OF B-CUBES

As mentioned before, keeping all the conflicts (i.e. entire B-cube) is not an option. Hence, we need to find a more compact, approximate representation that keeps as much relevant search space pruning information as possible. BDDs [6] or ZBDDs [11], perhaps with heuristic approximation techniques, certainly come to mind. Standard decision diagrams, however, are not suited for the task. In particular, a key advantage of SAT is the ability to have different decision orders along different parts of the search, meaning that the data structure must efficiently handle different variable orders for different certificates, ruling out standard ordered decision diagrams. We have chosen instead to create a more appropriate data structure loosely based on decision trees [8] that is specifically designed to efficiently support the operations we need.

Let's consider some of the key properties of the DPLL algorithm and try to picture an ideal B-cube that would be of the greatest use for search space pruning. The SAT search tree is a binary tree, in which decision nodes have two outgoing edges[5], and implied nodes have one. Ideally, our new pruning technique would provide the solver with a large number of literals that can be immediately assigned after flipping some decision variable. Obviously, such literals would need to be present in all the certificates, so we will call them *supercubed literals*. The more supercubed literals we have, the higher the probability that more unit clauses will be generated, increasing the chances for quick conflict detection. Thus the first desired property is to have as many supercubed literals as possible.

After supercubed literals are removed from certificates, there are no more common literals, but there might be common variables. Common variable can be used to sort the certificates in two classes according to the phase of the corresponding literal. By recursively applying the partitioning and searching for common literals and variables we obtain a binary tree. This tree represents an approximation of the set of all certificates in a compact manner. The case when there are no common variables is more complicated.

Example 2. Suppose x_1 is the variable of the root r of the search tree, and the B-cube $B_0(r)$ is the set $\{x_1^0 \wedge x_2^0 \wedge x_3^1, x_1^0 \wedge x_2^0 \wedge x_4^0, x_1^0 \wedge x_2^0 \wedge x_5^1\}$. After flipping x_1^0, the solver can assign the supercubed variable x_2^0. At this point we know that either x_3^1 or x_4^0 or x_5^1 need to be explored. Whichever choice the solver makes, it might need to backtrack later to that choice and try the remaining ones. Faced with a multiway choice, the solver would need some heuristic to determine the order of choice exploration. Choosing the next decision variable from the priority queue might be a better option.

As there is no clear intuition about whether multiway nodes would actually improve performance, and because multiway nodes are not easily added on top of DPLL, an approximation of the B-cube could simply discard such literals.

If the B-cube is approximated by a binary tree, the stem of the tree clearly contains supercubed variables and corresponds to a supercube. As it has been proven in [9], supercubing subsumes PLR. From the fact that such an approximation of B-cubing contains all supercubed literals as a stem, it follows that the approximation also subsumes PLR.

[5]Except for the nodes skipped over during CDB.

3.1 Boolean Constraint Trees

Boolean Constraint Trees (BCTs) are presented in this section as an approximation of B-cubes.

Definition 4. A **Boolean Constraint Tree** is a rooted binary tree such that *branch nodes* are labeled with a variable and have two outgoing edges. *Literal nodes* are labeled with a literal and have one outgoing edge. Any variable can appear at most once on a path from the root to a leaf. Given a BCT C, the prefix of node x is defined to be the cube of literals on the path from the root of the BCT to the node x and denoted by $pref_C(x)$. A leaf node can be either a literal node or a *termination node*. A termination node t is always a child of a branch node and marks that there were at least two certificates containing cube $pref_C(t)$, but no other common literals or variables.

There are two simplification rules for BCTs. A branch node with both children being termination nodes contains no useful information and can be discarded. The second rule says that two adjacent branches cannot contain equal literals. Such literals must be inserted above the branch as they are common to both paths.

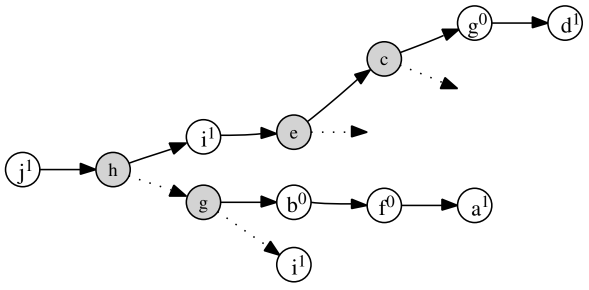


Figure 1: Boolean Constraint Tree

Example 3. A BCT C is given in Fig. 1. Shaded nodes are branch nodes. Dotted edges denote `FALSE` branches. Termination nodes are depicted as **X**. Prefix of node e is $pref_C(e) = [j^1, h^1, i^1]$. Literal j^1 was common to all certificates. Variable h was also common to all certificates. Variable g was common to all the certificates that included h^0, and so on.

The construction of BCTs goes as follows. The algorithm finds the longest path in the currently constructed BCT on which all the literals correspond to the new certificate. The literals that have no match in the certificate are removed from the BCT and pushed on a stack. When a leaf node is reached, the algorithm checks whether there was at least one matching variable between those eliminated literals, and creates a new branch if there was. Otherwise, all the literals from the stack get discarded. Hence, new nodes are added to a BCT only if a new branch is created. In all other cases, adding a new certificate prunes the BCT.

BCTs can grow quite large. To reduce the memory requirements and speed up the BCT construction process, we set the limit on the maximum number of nodes that a BCT can contain. Setting the limit is achieved by disallowing the creation of new branches, while BCT pruning is still allowed as it always reduces the size. The limit for our experiments was set to 2000 nodes.

B-cubing interacts with decision heuristic and learning. Supercubed literals are always welcome, as they increase the probability of creation of new unit literals and do not create new branches in the search tree. Suppose that the search procedure has assigned all the supercubed and implied literals, and has reached a branch node x in the BCT. According to our heuristic, the search will choose the more constrained branch of x by checking the next couple of nodes. In the case when BCT is very branchy, none of the nodes that follow will actually prune the search space. Even worse, just picking the next variable with the highest priority might perform better. For that reason, we also set the limit on the maximum percentage of branch nodes in the BCT. The limit was set to 40% for our experiments. Growth limits are empirically established values. If the number of nodes in the BCT is larger than the given limit, a special restrictive construction mode is entered in which new certificates do not increase the BCT size.

An important property of our algorithm is that it delays the creation of branches as long as possible. The resulting BCTs tend to have longer chains of literal nodes closer to the root while branch nodes are pushed closer to leaves. Such BCTs prune the search space more efficiently.

3.2 Search Space Pruning

A BCT can be seen as a blueprint of the search space that needs to be explored. Suppose the search has just flipped a decision and that the assignment generates a certain number of unit literals. First unit literals will be propagated. Next, if no conflict is found, our procedure will traverse the corresponding BCT, propagating newly generated unit literals aggressively after each new decision. When traversing the BCT, the search procedure might run into nodes already assigned as unit literals. If such a literal node matches the current assignment, it is skipped over, otherwise it is deemed to be a conflict. When a branch node is assigned, the edge to be followed is chosen depending on the current assignment.

Our experiments show that the search procedure rarely traverses the entire BCT. Therefore, large BCTs just slow down the search, while the percentage of used nodes is low. This motivates our decision to discard multiway nodes and set BCT growth limits.

Our B-cube pruning technique is *local* in the sense that it applies knowledge gained from conflicts in the first branch of a node to pruning in the second branch. The gained knowledge cannot be applied to arbitrary parts of the search tree. Learning doesn't have this limitation, but it is less effective in pruning the search space locally.

4. HYPERSAT

Our experimental HyperSAT solver is based on a simple watched literal scheme as implemented in LIMMAT [4], with some minor optimizations and extended to support equivalence clauses. Preprocessing eliminates unit and pure literals, detects tautologies and binary equivalences. Equivalence clauses are detected and reduced as described in [20, 21]. The clause cache is initially set to store 2^{13} clauses, and enlarged as needed. The 1-UIP learning scheme is used and the deletion strategy is very aggressive - half the clauses get deleted every time the cache is enlarged. The clauses to be deleted are chosen according to their size and number of occurrences in conflicts. Larger clauses that appear less often are deleted first. The solver is not randomized and it doesn't use restarts. The weakest point of our solver is a very simple and fragile implementation of VSIDS [14]. Also, only the preprocessor and BCP are optimized for performance so far. Our priority is to optimize other parts of HyperSAT, find a new heuristic which suites the specific search dynamics of the solver, and do memory optimization.

5. EXPERIMENTAL RESULTS

We have chosen eight benchmark sets for empirical evaluation of our new pruning technique. The sets represent typical practical applications of SAT with an emphasis on EDA problems. The PicoJava instances result from Bounded Model Checking (BMC) of the Sun PicoJava II^{TM} microprocessor.[6] The second set (IBM BMC) is an encoding of BMC of real industrial hardware designs.[7] The third set contains the well-known barrel, longmult, and queueinvar BMC benchmarks from CMU.[8] The next three sets are all from Fadi Aloul[9] and represent SAT encodings of FPGA routing and integer factorization problems. The seventh set is a SAT encoding of Constraint Satisfaction Problems (CSP).[10] Only three subsets (`frb30,35,40`) were used from this set, as no solver could solve the remaining ones. The last set is the `rule_1` subset from IBM Formal Verification Benchmarks Library without the `k100` instances.[11] The number of instances in each set is given in parentheses after the name of the set.

All experiments were done on a 2.6 GHz Pentium 4 machine with 3 Gb of memory. ZChaff II version 2004.5.13 running times are given for comparison. The timeout was set to 3600 seconds. The results are shown in Table 1. The number of timeouts is in parentheses following the total run time. Our HyperSAT tool with B-cubing is denoted as "BCT" and the version that implements only supercubing as "SC".

5.1 Discussion

Evaluation of any module (eg. preprocessing, decision heuristic, learning scheme,...) of a SAT solver is a difficult task as it is hard to extract exact information about the influence of a particular module on the overall performance from the background noise created by other modules and their interactions. In most cases, the new technique seems to be effective. On the given set of benchmarks HyperSAT had fewer timeouts. B-cubing doesn't seem to be particularly effective on IBM BMC Benchmarks. We believe that the reason is that our greedy heuristic often makes better decisions, resulting in faster convergence to conflicts than what can be achieved by choosing a BCT branch node as a new decision. HyperSAT performs significantly better on the CMU BMC, integer factorization, and CSP problem sets.

Benchmark	Decisions	Avg. Imp. Chain Len.
30.cnf + BCT	8708	354
30.cnf + SC	18401	427
57.cnf + BCT	945	104
57.cnf + SC	1039	102

Table 2: Number of decisions

The number of decisions is typically smaller for HyperSAT with BCTs while the average length of the implication chains is approximately the same. Examples of typical values for two benchmarks from the PicoJava^{TM} set are given in Table 2. The gain achieved by reducing the number of decisions is dampened by additional computation time required for constructing the BCTs. Currently, this construction is implemented through a series of complex recursive functions. We expect better results after a thorough optimization of our BCT algorithms.

On the scatter plots in Fig. 2 timeouts are placed on the border line. The results are particularly interesting for IBM FVS set, where it is obvious that HyperSAT is faster on most smaller instances, but performs worse on some larger ones. From the extensive experiments we did, it seems that the reason is our aggressive clause deletion strategy. Adapting the clause deletion heuristic decreased the overall performance of the solver, but improved the behaviour on larger IBM FVS instances.

[6]http://www-cad.eecs.berkeley.edu/~kenmcmil/satbench.html
[7]http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/BMC/description.html
[8]http://www-2.cs.cmu.edu/~modelcheck/bmc/bmc-benchmarks.html
[9]http://www.eecs.umich.edu/~faloul/benchmarks.html
[10]http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/benchmarks.htm
[11]http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html

6. CONCLUSIONS

We have introduced B-cubing, a powerful new search-space pruning technique, and have shown how to implement a practical SAT solver based on B-cubing, using Binary Constraint Trees. Our prototype implementation HyperSAT, despite being a preliminary, not-fully-optimized program and despite using completely different search-space pruning, is competitive with the latest version of ZChaff, one of the best state-of-the-art solvers. Furthermore, our new solver is slightly more robust, suffering fewer timeouts over the benchmark runs. Having a new approach that is competitive with, but with different strengths than, the best existing approaches allows solving problems that would otherwise be unsolvable.

Future work includes continued engineering and optimization of the solver itself, as well as exploring ways to approximate B-cubing more accurately and/or more efficiently.

7. REFERENCES

[1] D. Babić and A. J. Hu. Integration of Supercubing and Learning in a SAT Solver. In *Asia South Pacific Design Automation Conference*, pages 438–444. ACM/IEEE, 2005.

[2] F. Bacchus and J. Winter. Effective Preprocessing with Hyper-Resolution and Equality Reduction. In *SAT*, pages 341–355, 2003.

[3] A. Bhalla, I. Lynce, J. de Sousa, and J. Marques-Silva. Heuristic backtracking algorithms for SAT. In *4th International Workshop on Microprocessor Test and Verification*, pages 69– 74, 2003.

[4] A. Biere. The Evolution from LIMMAT to NANOSAT. Technical Report 444, Dept. of Computer Science, ETH Zürich, 2004.

[5] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *36th ACM/IEEE Design Automation Conference*, pages 317–320. ACM Press, 1999.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[7] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[8] D.-Z. Du and K. Ko. *Theory of Computational Complexity*. John Wiley and Sons, 2000.

[9] E. Goldberg, M. R. Prasad, and R. K. Brayton. Using Problem Symmetry in Search Based Satisfiability Algorithms. In *Proceedings of the conference on Design, Automation, and Test in Europe*, pages 134–142, 2002.

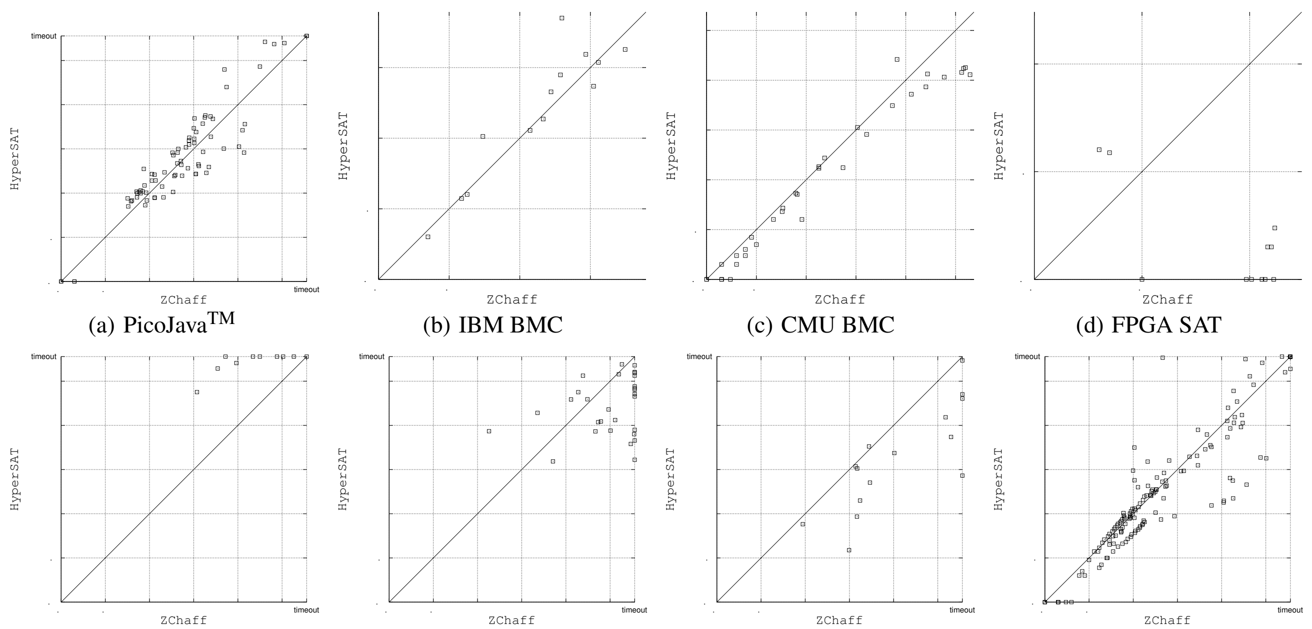
[10] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.

[11] S. i. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *30th Design Automation Conference*, pages 272–277. ACM Press, 1993.

[12] J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search

Benchmark Set	Instances	ZChaff II	HyperSAT (BCT)	HyperSAT (SC)
1. PicoJava BMC (76)	all	10756 (2)	16963 (2)	19952 (5)
2. IBM BMC (13)	all	78	118	117
3. CMU BMC (34)	all	7711	1310	1360
4. FPGA UNS (10)	all	7993 (1)	30271 (7)	32771 (8)
5. FPGA SAT (11)	all	11	0.33	0.23
6. Int Fact (29)	all	58887 (12)	17634	21789 (2)
7. CSP (15)	frb30,frb35,frb40	18130 (4)	4154	4246
8. IBM FVS (209)	rule_1, except k100	268440 (71)	273036 (71)	274414 (74)

Table 1: Experimental Results



(a) PicoJava^{TM} (b) IBM BMC (c) CMU BMC (d) FPGA SAT

(e) FPGA UNS (f) INT FACT (g) CSP (h) IBM FVS

Figure 2: Scatter plots

Algorithm for Propositional Satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, 1999.

[13] K. L. McMillan. Interpolation and SAT-based model checking. In *CAV 03: Computer-Aided Verification, LNCS 2725*, pages 1–13. Springer, 2003.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535. ACM Press, 2001.

[15] A. Nadel. Backtrack Search Algorithms for Propositional Logic Satisfiability: Review and Innovations. Master's thesis, Tel-Aviv University, 2002.

[16] G. Nam, K. Sakallah, and R. Rutenbar. A boolean satisfiability-based incremental rerouting approach with application to FPGAs. In *Proceedings of the conference on Design, Automation and Test in Europe*, pages 560–565. IEEE Press, 2001.

[17] M. R. Prasad. *Propositional Satisfiability Algorithms in EDA Applications*. PhD thesis, University of California at Berkeley, 2001.

[18] L. Ryan. Efficient algortihtms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.

[19] P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(9):1167–1176, Sept 1996.

[20] J. P. Warners and H. van Maaren. A two phase algorithm for solving a class of hard satisfiability problems. *Operations Research letters*, 23:81–88, 1998.

[21] J. P. Warners and H. van Maaren. Recognition of tractable satisfiability problems through balanced polynomial representations. In *5th Twente Workshop on Graphs and Combinatorial Optimization*, pages 229–244. Elsevier Science Publishers B. V., 2000.

[22] H. Zhang and M. E. Stickel. An efficient Algorithm for Unit Propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.

[23] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285. IEEE Press, 2001.

[24] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *Proceedings of the 18th International Conference on Automated Deduction*, pages 295–313. Springer-Verlag, 2002.