Efficient SAT Solving: Beyond Supercubes

Anonymous version for DAC review.

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 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.

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 [3] and unbounded [12]), FPGA routing [15], and ATPG [18]. Such industrial applications require complete SAT solvers, meaning that the solver must be capable of proving that problem is either satisfiable or definitely unsatisfiable.

Since the early days of SAT solving [5], it was clear that the efficiency of SAT solvers depends heavily on search space pruning rules and decision heuristics. Decision heuristics have got a fair amount of attention in the literature [8, 2, 22, 10]. Although some limited success has been achieved, many proposed heuristics are extremely sensitive to chosen parameters, strongly dependent on the details of implementation of the rest of the solver[1], and in general 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 DPLL framework and existing pruning techniques. In addition, the savings achieved must justify the additional cost.

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

Design and Automation Conference 2005

1.1 Existing Pruning Techniques

The goal of this section is to provide a quick introduction to some pruning techniques that will be mentioned later in the text.

Boolean Constraint Propagation (BCP) and Pure Literal Rule (PLR)[2] were proposed in [5]. 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 [13].

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 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 search. More elaborate method is to analyze the conflict and backtrack to the decision variable that is actually responsible for the conflict. That scheme is called Conflict-Directed Backtracking (CDB) and GRASP [11] was the first SAT solver to implement it.

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 reverse direction and add clauses that correspond to different cuts in the implication 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 [17], 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 on different learning schemes, it is clear that learning schemes proposed so far can use

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

only a fraction of information inferable from the conflicts. Due to memory constraints, it is impossible to add all the clauses that can be learned to the clause database.

Recently, a theory of essential points [16, 7] 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)[13].

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

A *cube* (*clause*) is a conjunction (disjunction) of literals in which each variable from $\mathcal{V}$ 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 $x \in \mathcal{V}$ 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 $\mathit{flip}(S, x) = \{\mathit{flip}(m, x) \mid m \in S\}$.

We assume a simple SAT solver which systematically explores a search tree without restarts or CDB, and the solver's input is the 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 $\mathcal{V}$. 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 $m \rightarrow \mathit{cert}(u)$ 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 $x^{\overline{b}}$. We are interested in those certificates c that involve x^b.

Definition 1. The set of all certificates 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 $S_b(d) = \{m \in M \mid m \rightarrow c \text{ for some } c \in B^*_b(d)\}$

THEOREM 1. *Suppose d^b has no satisfying assignments. Then for any minterm m found in $d^{\overline{b}}$ 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 $d^{\overline{b}}$, attention is restricted to the assignments of S'; i.e. assignments in $d^{\overline{b}}$ 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 $S_b(d) \rightarrow \ell$.

Example 1. Decisions in the search tree, sorted ascending by decision level, are x^0_1, x^1_2, x^0_3, and let d be the decision node for x_3. The solver explores the search subtree d^0 (i.e. x^0_3) and finds no solution. Assume there were three conflicts and a certificate is found for each: $c_1 = x^0_1 \wedge x^1_4 \wedge x^1_5 \wedge x^1_6$, $c_2 = x^1_2 \wedge x^0_3 \wedge x^0_4 \wedge x^1_5 \wedge x^0_6 \wedge x^1_7$, and $c_3 = x^0_1 \wedge x^1_2 \wedge x^0_3 \wedge x^0_4 \wedge x^0_5 \wedge x^1_8$. The least cube that contains the certificates that include x^0_3, namely c_2 and c_3, is $sc_0(d) = x^1_2 \wedge x^0_3 \wedge x^0_4$. As x_2 has a higher decision level than d, the corresponding literal can be eliminated from the supercube, as it is already assigned. Hence, since $sc_d(0)$ over-approximates $S_0(d)$, in the subtree d^1 (i.e. after flipping x_3 to 1), the solver can immediately assign x^0_4.

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.

[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 $d^{\overline{b}}$.

Supercubing can prune the search space that can't be pruned by learning, as explained in [7]. An algorithm for computing supercubes and a thorough discussion of the integration of supercubing and learning are given in [1].

2.2 B-cubing

Even more information can be learned from certificates in Example 1. After assigning $x_3^1 x_4^0$ the solver propagates implied variables, if there are any, or makes a new decision. Let's assume that it made a new decision x_5^1. At that point, the solver can immediately assign $x_6^0 x_7^1$ as implied variables (i.e. it doesn't need to explore x_6^1 or x_7^0).

Going back to Example 1, the solver can immediately assign x_4^0 after x_3^1, but than there no more literals that are common to all certificates, but there is a variable which appears in all certificates in $A_0(c)$, and that is x_5. So, the solver can choose x_5 as a new decision variable. If x_5^1 is chosen, it can immediately assign $x_6^0 x_7^1$, according to the previous discussion. Equivalently, after picking x_5^0, x_8^1 can be immediately assigned.

B-cubing is a generalization of supercubing. The fact that more information can be learned from certificates was first observed by Nadel [14] 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 approach requires huge amounts 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 root decision node contains 3 literals in its supercube (or in the stem of BCT data structure, as it will be explained later), after flipping the root node, the supercube would ideally reduce the search space eight fold.

An advantage of Jerusat approach is its simplicity. If all the conflicts are kept (within the predefined window), reasoning procedure can be entirely implemented inside of the decision engine. The solution we are proposing requires substantial changes in 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 [4] or ZBDDs [9], 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[6] 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 is the probability that more unit clauses will be generated, increasing the chances for quick conflict detection. So, the first desired property is certainly 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. The obtained data structure is a binary tree, similar to a search tree. The case when there are no common variables is more complicated.

Example 2. Suppose x_1^0 is a decision labeled with the lowest decision level. B-cube of x_1^0 is represented by a set of certificates $c_1 = x_1^0 \wedge x_2^0 \wedge x_3^1$, $c_2 = x_1^0 \wedge x_2^0 \wedge x_4^0$, and $c_3 = x_1^0 \wedge x_2^0 \wedge x_5^1$. After flipping x_1^0, solver can assign the supercubed variable x_2^0. At that 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. Hence, it would be a multiway branching point. Having multiple choices, the solver would need some heuristic to determine the order of evaluation. In addition, choosing the next decision variable right of the priority queue might be a better option.

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

If 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 [7], supercubing subsumes PLR. From the fact that approximation of B-cube contains all supercubed literals as a stem it follows that the approximation also subsumes PLR.

3.1 Boolean Constraint Trees

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

Definition 4. **Boolean Constraint Tree** is a 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 BCT C, the prefix of node x is defined as a cube of literals on the path from the root of the BCT to the node x and denoted as $pref_C(x)$. Leaf node can be either a literal node or a termination node. Termination node t is always a child of a branch node and marks that there were at least two certificates c containing cube $pref_C(t)$, but no other common literals or variables.

There are two simplification rules for BCTs. Leaf branch node doesn't contain any useful information and can be discarded. The second rule says that two adjacent branches cannot contain equal

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

literals. Such literals must be inserted above the branch as they are common to both paths.

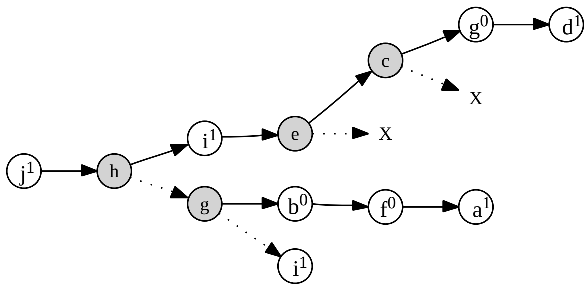


Figure 1: Boolean Constraint Tree

Example 3. 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 of BCT. 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, say x, in BCT. According to our heuristic, 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 BCT. The limit was set to 40% for our experiments.

3.2 Search Space Pruning

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 that assignment generated certain number of unit literals. Unit literals will be propagated first and then, if no conflict is found, the search procedure will traverse the corresponding BCT, propagating newly generated unit literals after assigning each node. When traversing BCT, the search procedure might run into nodes assigned as unit literals. If it is a literal node matching current assignment, it's skipped over, otherwise it's 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.

Growth limit is empirically established value. If the number of nodes in BCT is larger than the given limit, a special restrictive construction mode is entered and new certificates do not increase the size of the BCT.

B-cubing technique applies the knowledge gained from the conflicts in the first branch to pruning the second adjacent branch, effectively partially solving both branches at once. The gained knowledge cannot be applied to different parts of the search tree. Learning doesn't have those limitations, but it is less effective in pruning the search space locally.

4. HYPERSAT

Our experimental HyperSAT solver is based on modified Van Gelder's watched literals scheme [19], 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]. Clause cache is initially set to store 2^{13} clauses, and enlarged as needed. 1-UIP learning scheme is used and 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 the conflicts. Larger clauses that appear less often are deleted first. The solver is not randomized and it doesn't feature restarts. The weakest point of our solver is a very simple and fragile implementation of VSIDS [13]. Also, only the preprocessor and BCP are optimized for performance so far. Our high priority is to optimize other parts of HyperSAT, find a new heuristic which suites the specific search dynamics of the solver, and do the memory optimization.

5. EXPERIMENTAL RESULTS

We have chosen eight benchmark sets for empirical evaluation of our new pruning technique. The number of instances in each set is given in parentheses after the name of the set. PicoJava instances result from Bounded Model Checking (BMC) of Sun PicoJava II™ microprocessor. Instances are generated by scripts written by Ken McMillan. Second set (IBM BMC) is encoding of BMC of real industrial hardware designs. The third set contains well known barrel, longmult, and queueinvar BMC benchmarks from CMU. The following three sets are all from Fadi Aloul and represent SAT encodings of FPGA routing and integer factorization problems. The seventh set is SAT encoding of Constraint Satisfaction Problems (CSP). Only three subsets (`frb30,35,40`) were used from the entire set, as no solver could solve the remaining ones. The last set is `rule_1` subset from IBM Formal Verification Benchmarks Library without `k100` instances.

All experiments were done on 2.6 GHz Pentium 4 with 3 Gb of memory. ZChaff II version 2004.5.13 running times are given for comparison.

The timeout was set to 3600 sec. Results are shown in Table 1. The number of timeouts is in parentheses following the total run time. HyperSAT with B-cubing is denoted as BCT and the version that implements only supercubing as SC.

5.1 Discussion

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

Evaluation of any module (eg. decision heuristic, learning scheme,...) of a SAT solver is a difficult task as it is hard to extract exact information about the influence of that 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. Performance is comparable to the latest version of ZChaff, with each tool significantly outperforming the other on some instances; HyperSAT has 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 a conflict than what can be achieved by choosing a BCT branch node as a new decision.

HyperSAT performs significantly better on 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 implication chains is approximately the same. Two examples of typical values for two benchmarks from 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 BCTs. Currently, construction is implemented through a series of complex recursive functions. We expect better results after a thorough optimization of BCT algorithms.

Timeouts on scatter plots in Fig. 2 are placed on the border line. 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 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 instances.

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. Babic and A. J. Hu. Integration of Supercubing and Learning in a SAT Solver. In *Asia South Pacific Design Automation Conference*, 2005. To appear.

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

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

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

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

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

[7] 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.

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

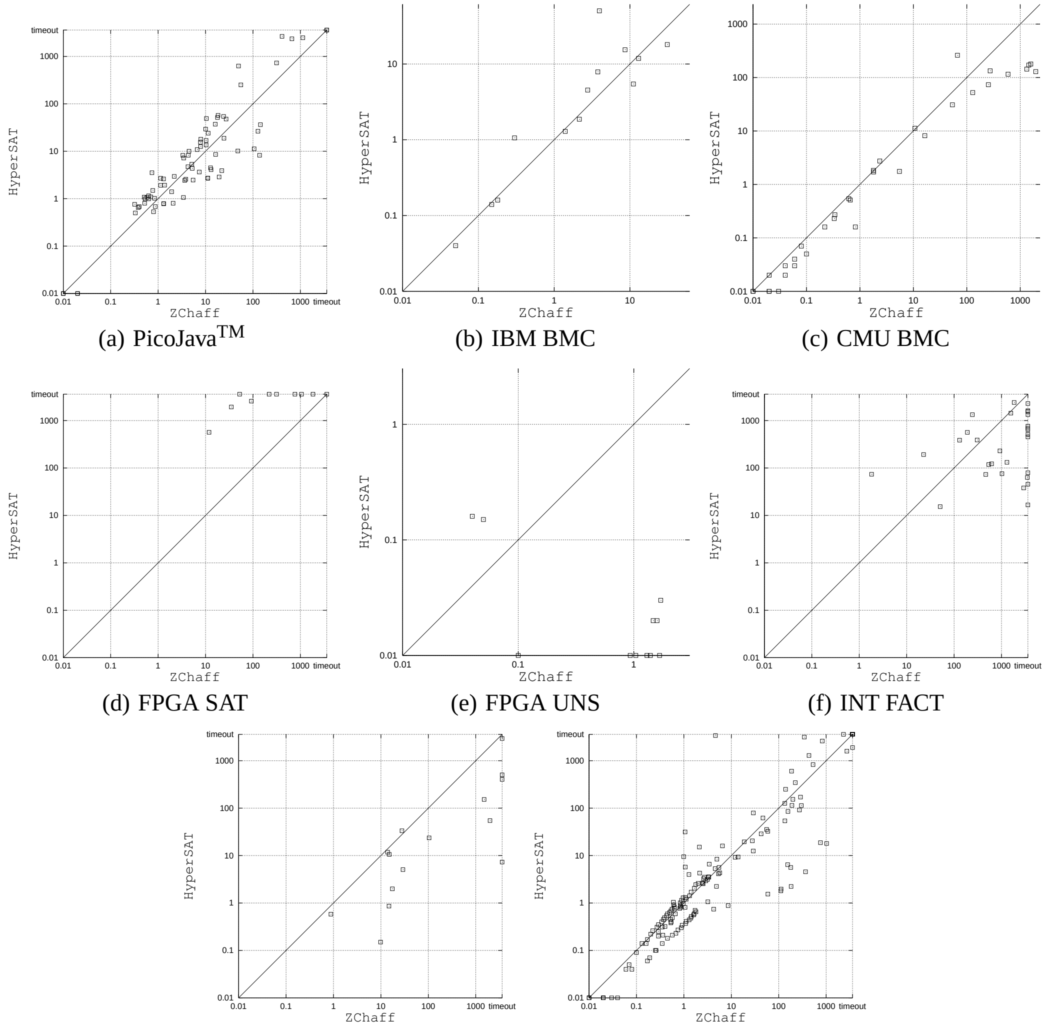
[9] S. ichi Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th international conference on Design automation*, pages 272–277. ACM Press, 1993.

[10] C. M. Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *IJCAI (1)*, pages 366–371, 1997.

[11] J. P. Marques-Silva and K. A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Comput.*, 48(5):506–521, 1999.

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

[13] 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



(a) PicoJava™ (b) IBM BMC (c) CMU BMC

(d) FPGA SAT (e) FPGA UNS (f) INT FACT

(g) CSP (h) IBM FVS

Figure 2: Scatter plots

530–535. ACM Press, 2001.

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

[15] 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.

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

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

[18] 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.

[19] A. Van Gelder. Generalizations of Watched Literals for Backtracking Search. In *Seventh Int'l Symposium on AI and Mathematics*, Ft. Lauderdale, FL, 2002.

[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 *Proceedings of the 5th Twente workshop on 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.