sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP

Marc Thurley

Institut für Informatik, Humboldt-Universität zu Berlin
thurley@informatik.hu-berlin.de

Abstract. We introduce sharpSAT, a new #SAT solver that is based on the well known DPLL algorithm and techniques from SAT and #SAT solvers. Most importantly, we introduce an entirely new approach of coding components, which reduces the cache size by at least one order of magnitude, and a new cache management scheme. Furthermore, we apply a well known look ahead based on BCP in a manner that is well suited for #SAT solving. We show that these techniques are highly beneficial, especially on large structured instances, such that our solver performs significantly better than other #SAT solvers.

Introduction

The appearance of highly optimized SAT solvers [7, 5, 8] encouraged applying these SAT solvers to the closely related problem of counting the solutions of a propositional formula, known as #SAT. Applying the DPLL algorithm [4] to model counting was proposed in [3]. `relsat 2` (cf. [2]) combined clause learning [11, 12] with *component decomposition*. Recently, *Cachet* by Sang et al. [9, 10] provided *component caching* and new branching heuristics.

We introduce sharpSAT - a new #SAT solver that inherits these techniques, improves upon them and contributes new ideas, such that it is able to outperform the best #SAT solvers (its source code is available at [1]).

After some basic definitions we will give a brief overview of our #SAT solver. Then we will discuss a new way of component caching that differs significantly from the scheme known so far (see [9]). It reduces cache sizes by at least by one order of magnitude. In the course of this, we will propose a cache management scheme which bounds the cache size explicitly and deletes old cache entries by means of a simple utility function.

Section 2 provides a discussion of *implicit BCP* - an adaptation of a well known "look ahead" technique based on boolean constraint propagation (BCP) (cf.[6]). Implicit BCP is built to integrate this technique well with other common #SAT solving techniques. This frequently results a smaller search space and reduces the cache size even further.

Eventually, in section 3, we will compare sharpSAT to the sate-of-the-art #SAT solver Cachet. This will reveal that the new techniques perform exceptionally well especially on very large instances, such as those from bounded model checking, which often contain several thousands of variables.

We consider propositional formulas F in *conjunctive normal form* (*CNF*). Let $F|_\sigma$ denote the *residual* formula under an assignment σ, where satisfied literals (and clauses) evaluate to **1** and unsatisfied ones to **0**. If $\mathbf{0} \in F|_\sigma$, i.e $F|_\sigma$ contains the *empty clause* **0** we say that σ *conflicts* with F. $F|_\sigma$ is *satisfied* if all clauses evaluate to **1**.

A short outline of the basic techniques. From SAT solvers sharpSAT inherits clause learning (cf. [11], [12]) and a fast BCP algorithm, based on the "Two Watched Literal" scheme (see [7]). Recall the notion of BCP: Whenever $F|_\sigma$ contains a *unit clause* $C = \lambda$ then λ must be satisfied, i.e. $\sigma(\lambda) = 1$. BCP performs these assignments until either no unit clause is left or a conflict occurs.

From #SAT solvers we adopted *bounded* component analysis and caching - these techniques and their correctness were discussed in [9]. For selecting *branch* variables sharpSAT applies the VSADS heuristic from Cachet (cf. [10]).

1 Component Caching

As is done in Cachet, components can be identified by strings, omitting satisfied clauses and assigned literals. Let, for example $F = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x_4} \vee \bar{x_5}) \wedge (x_6 \vee x_2 \vee x_3) \wedge (x_6 \vee \bar{x_4} \vee \bar{x_5})$, then a string coding this would be $(1, 2, 3, 0, 1, -4, -5, 0, 6, 2, 3, 0, 6, -4, -5, 0)$ (zeros denote ends of clauses). Call this the *Standard scheme* (STD).

sharpSAT codes the components differently. First of all, only *sound components* are cached, i.e. those which contain only clauses with at least two unassigned literals. This is reasonable as length zero clauses (i.e. the empty clause) denote conflicts and unit clauses are handled by BCP.

Let $var(F)$ ($cl(F)$) be the set of variables (clauses, resp.) in F and $var_{id}(F)$ ($cl_{id}(F)$) the corresponding sets of indices. Let $id(\lambda)$ ($id(C)$) denote the index of a literal λ (clause C). With F as in the example above we have $cl(F) = \{\{x_1, x_2, x_3\}, \{x_1, \bar{x_4}, \bar{x_5}\}, \{x_6, x_2, x_3\}, \{x_6, \bar{x_4}, \bar{x_5}\}\}$ but $cl_{id}(F) = \{1, 2, 3, 4\}$. We code components G by writing $var_{id}(G)$ and $cl_{id}(G)$ to strings a and b in increasing order of the indices, which yields a *code* (a, b). For F as above we have $a = (1, 2, 3, 4, 5, 6)$ and $b = (1, 2, 3, 4)$. Call this the *Hybrid coding scheme* (HC). The correctness of HC is displayed by the following lemma.

Lemma 1 *Given $F \in CNF$, (partial) assignments σ, τ and components G of $F|_\sigma$ and B of $F|_\tau$ then*
$(var(B) = var(G)$ and $cl(B) = cl(G))$ iff $(var_{id}(B) = var_{id}(G)$ and $cl_{id}(B) = cl_{id}(G))$.

Proof. The forward direction is trivial. For the reverse let $var_{id}(B) = var_{id}(G)$ and $cl_{id}(B) = cl_{id}(G)$. Obviously, $var(B) = var(G)$ holds, but suppose for contradiction, that $cl(B) \neq cl(G)$. As $cl_{id}(B) = cl_{id}(G)$, there is a clause $\gamma_B \in cl(B)$ for which $\gamma_G \in cl(G)$ exits with $id(\gamma_B) = id(\gamma_G)$ but $\gamma_B \neq \gamma_G$.

As B and G are components of restrictions of F, there is a clause $\gamma \in F$ with $id(\gamma) = id(\gamma_B) = id(\gamma_G)$. Now, as $\gamma_B \neq \gamma_G$, there is a literal λ such that

w.l.o.g. $\lambda \in \gamma_B \setminus \gamma_G$. Since $\lambda \notin \gamma_G$ we have that λ is assigned in G but not in B, contradicting $var(B) = var(G)$.

Note that STD is more general than HC. For example, for F as above, $\sigma = [x_1 \leftarrow 0, x_6 \leftarrow 1]$ and $\tau = [x_1 \leftarrow 1, x_6 \leftarrow 0]$ we have $F|_\sigma = F|_\tau = (x_2 \vee x_3) \wedge (\bar{x_4} \vee \bar{x_5})$ which could be recognized by STD but not by HC as $cl_{id}(F|_\sigma) = \{1, 2\}$ and $cl_{id}(F|_\tau) = \{3, 4\}$. However, our experimental results (see sect. 3) show the effectiveness of HC and hence suggest that this case is not very likely.

We can reduce the sizes of the codes even further. As only sound components G are cached, storing the ids of binary clauses is redundant. Why is this so? Consider a formula F, an assignment σ and a sound component G of $F|_\sigma$ with code (a, b). Assume that G contains $C|_\sigma$ for a binary clause $C = (\lambda \vee \kappa) \in F$. Suppose that at least one of κ and λ is assigned in σ. If exactly one is assigned, $C|_\sigma$ is satisfied (otherwise BCP could be applied). If both are assigned, $C|_\sigma$ is satisfied as well, as G is sound. Thus, $C|_\sigma$ occurs in G iff κ *and* λ are unassigned, and the occurrence of $cl_{id}(C|_\sigma)$ in the code b can be reconstructed by the presence of $id(\lambda)$ and $id(\kappa)$ in a. Hence, we can omit storing the identifiers of binary clauses in the component codes. Call this scheme *Omitting binary clauses*.

We can do even better. Each code (a, b) of a component G is packed before caching and before cache look-up. To obtain the packed form $(\hat{a}, \hat{b})$, we determine $n := \lceil log_2 |var(F)| \rceil$ and $m := \lceil log_2 |cl(F)| \rceil$. Identifiers in a contain information only in the n least significant bits, thus a is packed into $\hat{a}$ by bitshifting. b is treated analogously. Call this the *Packing* scheme.

Table 1 illustrates the coding schemes when applied to formulas from SATLIB. *HCO* is HC omitting binary clauses, and *HCOP* shows this in its packed form.

Problems	vars	clauses	STD	HC	HCO	HCOP
flat200	600	2237	27644	11348	3200	950
uf200	200	860	13760	4240	4240	1075
logistics.a.cnf	828	6718	98532	30184	16964	5301
logistics.b.cnf	843	7301	105300	32576	16576	6006
bmc-ibm-1.cnf	7085*	35419*	822420	170016	49484	20104
bmc-galileo-8.cnf	43962*	183532*	4079 KB	884 KB	302 KB	151 KB

Table 1. Comaring codes sizes in bytes (*= unit clauses removed via BCP)

We compare sizes of the different codes of the input formulas. Experiments show that this gives a good estimate of the relative cache sizes. The starred numbers denote a preprocessing before forming the codes: all unit clauses in the input formula had to be propagated via BCP, as the hybrid scheme does not cache formulas with unit clauses. Observe that the efficiency of HC in comparison to STD increases with clause-to-variable ratio. HCO is futile on formulas without binary clauses (see uf200) but it is highly beneficial for example on the flat200 formulas. Clearly, packing shows the least advantages on large instances (ibm-galileo-8) but still reduces the code size to about 50%.

Cache Management. On hard formulas the cache size quickly exceeds any reasonable bound, which necessitates a good cache management. In our experiments, we observed drawbacks of bounding the cache size by an oldest age bound: a good bound depends highly on the formula size and has to be set manually.

In sharpSAT an absolute bound $maxSize$ in bytes on the cache size is set. Furthermore, we keep scores for each cache entry in a way reminiscent of the VSIDS heuristic (cf. [7]). If an entry is hit its score is increased. All scores are divided periodically. The cache is cleared only if it exceeds a fixed fraction (0.9, say) of $maxSize$, if so all entries with a score lower than $minScore$ are deleted. Directly after cleanup we try to keep the cache size at about $0.5 \cdot maxSize$. To achieve this, we increase or decrease $minScore$ accordingly.

This quickly stabilizes the cache size after cleanup to about the desired value. Furthermore, this scheme is quite fast, as entries are deleted only when necessary and updating scores creates almost no time overhead.

2 Implicit BCP

BCP plays a central role in the performance of SAT and #SAT solvers. Branching heuristics based on BCP, called Unit Propagation (UP) heuristics (cf. [6]) try to maximize the possible effect of BCP by applying a form of "look ahead".

UP heuristics determine branch variables by estimating the effect an assignment has for BCP. To achieve this, for each variable x of a certain set S of free variables the assignments $x \leftarrow 0$ and $x \leftarrow 1$ are made independently and BCP is applied in each case. If any of these cases, say $x \leftarrow 0$, causes a conflict, a *failed literal* ($\bar{x}$) is found and x is chosen directly as branch variable. Otherwise, the variables in S are evaluated according to their effect on BCP and one of these is chosen.

sharpSAT applies an algorithm for finding failed literals. It deviates from the traditional UP heuristics approach, as for example pursued in Cachet, in at least two ways. First, it is applied independently of the branching heuristics and only failed literals are sought. If a failed literal, say $\lambda = \bar{x}$ is found, a conflict clause C^λ is learned directly and the algorithm proceeds as if $x \leftarrow 1$ was found by BCP via C^λ. The process stops either if a conflict occurs, or no failed literals are found anymore. In SAT solvers this might show no big difference to UP heuristics, but in our #SAT solver a large amount of component analysis and cache look up and storing is avoided by this procedure as in the course of implicit BCP these operations are not applied.

Furthermore, the set S of candidates for failed literals is computed differently. We only consider literals from *original* clauses that have become binary in the most recent call of BCP. Thus in instances that allow for few implications only, S is small and thus implicit BCP induces almost no overhead. In cases of many implications S is larger but failed literals are more likely as well.

Problems	vars	clauses	solutions	implicit BCP		w/o implicit BCP		Cachet
				decisions	secs	decisions	secs	secs
flat200	600	2,237						
avg.			2.22e+13	3,378	**1.77**	14,141	2.18	3.98
uf200	200	860						
avg.			1.57e+09	9,597	7.36	70,448	10.9	**6.63**
bmc								
ibm-1	9,685	55,870	7.33e+300	11,991	**16**	37,808	32	47
ibm-2	2,810	11,683	1.33e+19	148	**0.09**	584	0.11	**0.09**
ibm-3	14,930	72,106	2.47e+19	2,657	64.5	14,705	72.2	**58**
ibm-4	28,161	139,716	9.73e+79	59,334	**111**	1.6e+6	1,346	X
ibm-5	9,396	41,207	2.46e+171	191,558	152	198,464	**64.5**	486[1]
ibm-11	32,109	150,027	3.53e+74	429,575	**3,331**	2.3e+6	15,204	26,823[2]
ibm-12	39,598	194,778	2.1e+112	25,456	**833**	–	X	977[3]
galileo-8	58,074	294,821	8.14e+40	12,945	**326**	168,748	1,716	628[3]
galileo-9	63,624	326,999	3.46e+44	15,798	**392**	313,474	3,688	786[4]
logistics								
a	828	6,718	3.78e+14	4,412	**0.8**	19,176	2.14	5.31
b	843	7,301	4.53e+23	15,711	**7.15**	93,885	11.8	17.8
c	1,141	10,719	3.98e+24	2.3e+6	**426**	3.9e+6	480	1,003[5]

Table 2. Comparing sharpSAT with and without implicit BCP and Cachet (X = time out after 10 hours; $^{1-5}$ oldest age bounds: 1 = 500; 2 = 50; 3 = 30; 4 = 15; 5 = 3000)

3 Comparison

We compare sharpSAT with and without implicit BCP and Cachet (version 1.22) on instances from SATLIB, to wit, the flat200, uf200, bmc and logistics suite. Tests were run on a 3GHz Pentium 4 with 1GB of main memory, a time bound of 10 hours and a maximum cache size of 512MB. Table 2 displays the results, entries for bmc-ibm-6,7,10,13 and logistics.d are missing as neither Cachet nor sharpSAT solved them. In contrast to the model counts shown in scientific number form, the solvers used BigNum packages for exact model counting.

Note the effect of implicit BCP. Comparing the savings in terms of the run time to the reduced number of decisions reveals the overhead of implicit BCP, e.g. in the uf200 suite, a reduction by a factor of 7 in the decisions is reflected only in a factor of 1.5 in the actual running time. However, there *is* a benefit in time as a lot of component analysis as well as cache look-up and storing is avoided. Each of these operations is performed once per decision. Hence, less decisions are always beneficial, as irrespective of decreased running times, the cache size is always reduced by about the same factor as the number of decisions.

The footnotes on the running times for Cachet refer to oldest age bounds on the cache entries, which were adjusted manually. Where these are given, Cachet could not solve the instances without them due to out-of-memory errors. For sharpSAT the maximum cache size was set to 512MB for all the instances given.

sharpSAT dominates especially on structured instances. However, a general link between small running time and exactly one of the new techniques is not obvious. On some instances with very low model counts (e.g. galileo-8 and 9) (see table 2) the dominance is clearly due to implicit BCP. On the other hand, for ibm-bmc-5 and logistics.c the dominance is based solely on the new caching scheme. In all other instances we claim that the effect is due to the combination of both techniques.

4 Conclusion

We introduced sharpSAT - a #SAT solver that outperforms the current state-of-the-art solver Cachet on a wide range of structured instances. This is due to new techniques which comprise a highly optimized way of coding the components for caching and the implicit BCP algorithm that performs well in practice.

References

1. www.informatik.hu-berlin.de/∼thurley/sharpSAT.
2. Robert J. Bayardo and Joseph D. Pehoushek. Counting models using connected components. *Proceedings, AAAI-00: 17th International Conference on Artificial Intelligence*, pages 157–162, 2000.
3. Elazar Birnbaum and Eliezer L. Lozinskii. The good old davis-putnam procedure helps counting models. *Journal of Artificial Intelligence Research*, 1999.
4. Martin Davis, George Logeman, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 1962.
5. E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. *Design, Automation and Test in Europe (DATE'02)*, pages 142–149, 2002.
6. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI (1)*, pages 366–371, 1997.
7. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
8. Lawrence Ryan. Efficient algorithms for clause learning sat solvers. Master's thesis, Simon Fraser University, 2004.
9. Tian Sang, Fahiem Bacchus, Paul Beame, Henry Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Seventh International Conference on Theory and Applications of Satisfiability Testing*, 2004.
10. Tian Sang, Paul Beame, and Henry Kautz. Heuristics for fast exact model counting. In *Eighth International Conference on Theory and Applications of Satisfiability Testing, Edinburgh, Scotland*, 2005.
11. Joao P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pages 220–227, November 1996.
12. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *International Conference on Computer-Aided Design (ICCAD'01)*, pages 279–285, 2001.