A Lightweight Component Caching Scheme for Satisfiability Solvers

Knot Pipatsrisawat and Adnan Darwiche
{thammakn,darwiche}@cs.ucla.edu

Computer Science Department
University of California, Los Angeles

Abstract. We introduce in this paper a lightweight technique for reducing work repetition caused by non–chronological backtracking commonly practiced by DPLL–based SAT solvers. The presented technique can be viewed as a partial component caching scheme. Empirical evaluation of the technique reveals significant improvements on a broad range of industrial instances.

1 Introduction

As a DPLL–based SAT solver makes decisions, the knowledge base gets simplified due to Boolean constraint propagation. This simplification may be substantial enough to disconnect the knowledge base into independent components[1]. Knowledge about independent components could reduce the amount of work done by a solver. However, precise component analysis is prohibitively expensive for SAT solving in general, although some solvers have incorporated static component analysis in the preprocessing phase [1, 9, 6].

The lack of dynamic component analysis is made worse by the use of non–chronological backtracking, because it may cause solvers to erase assignments that are not related to the conflict. In the worst case, erased assignments may contain solutions to independent components. As a result, the solver may need to solve some components multiple times. This problem has already been observed and solutions have been proposed in [8, 4]. Nevertheless, the proposed solutions seem to offer limited improvements on real–world instances.

We address this particular problem in this paper and provide two contributions. First, an analytic and empirical analysis that substantiates the observations about work repetition in modern SAT solvers that use non–chronological backtracking. Second, a low–overhead technique that helps reduce work repetition in such solvers.

The rest of this paper is structured as follows. In Section 2, we describes precisely the above problem. An empirical study that further exposes and quantifies the problem is presented in Section 3. A solution is proposed in Section 4 and is evaluated in Section 5. Section 6 discusses related work and we conclude in Section 7.

[1] A component is defined as a set of clauses. Two components are independent if they share no variable.

2 Losing Work with Non–Chronological Backtracking

The use of non–chronological backtracking in SAT and CSP allows solvers to better focus on fixing the cause of the conflict [18, 2, 11]. The most common non–chronological backtracking scheme used by SAT solvers today, called far–backtracking [16], is based on generating *asserting clauses* [20]. This approach involves undoing the assignments from the point of conflict up to (not including) the assertion level. Although the results and analysis we present in this paper can be adapted to work for non–chronological backtracking in general, we stay focused on far–backtracking as it is the most common in modern SAT solvers.

One caveat on this backtracking scheme is that all decisions and implied variable assignments made between the level of the conflict and the assertion level are effectively erased by backtracking. As we shall see next, these assignments may contain solutions of sub–problems (components) and would be lost in the backtracking process, requiring their rediscovery at a later stage in the search.

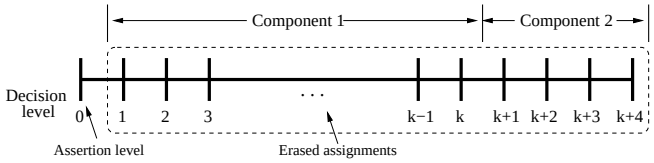

Fig. 1. Erased assignments due to a backtrack.

To consider a dramatic example of this phenomena, examine Figure 1 in which the solver has solved a component using the first k decision levels. After several decisions on a second component, the solver runs into a conflict and derives a unit learned clause. The assertion level in this case will be level 0, leading the solver to erase all assignments, assert the unit clause, and restart the search process all over. After the learned clause is asserted, the solver will continue looking for solutions for both components, as it did not save the solution it previously found. This can lead to great inefficiency as the solver may end up solving some components multiple times.

We will provide a more realistic, yet still somewhat synthetic, empirical study in the next section to quantify further this potential inefficiency.

3 An Empirical Study

To illustrate the extent of work repetition, we artificially generated instances that would cause work repetition in conventional SAT solvers. Each instance was generated by merging four identical copies of a satisfiable instance. These bigger instances will be referred to as replicated instances throughout this paper.

Instance Name	Runtime					
	MiniSat			MiniSat with ps		
	Original	Replicated	Ratio	Original	Replicated	Ratio
vmpc_21.renamed-as.sat05-1923	6.01	731.98	122	1.5	21.58	14
vmpc_21.shuffled-as.sat05-1955	0.48	59.37	124	1.25	26.01	21
vmpc_23.renamed-as.sat05-1927	39.19	3202.67	82	2.4	28.61	12
vange-color-54	28.26	4624.42	163	4.24	96.34	23
velev-fvp-sat-3.0-12	6.70	>200*	>29	4.07	41.87	10
ibm_19_rule_SAT_dat.k30	7.91	209.73	26	6.47	28.6	4.4
ibm_21_rule_SAT_dat.k35	8.87	819.00	92	5.15	44.12	8.6

Table 1. Runtime (in seconds) of MiniSat with and without progress saving. (*) indicates insufficient memory. The ratio columns show approximate ratios of runtime on replicated over original instances.

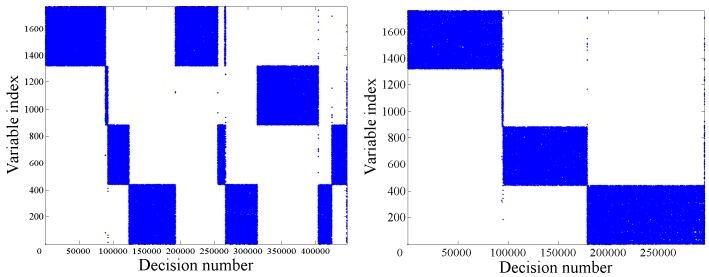

Fig. 2. Decision behavior on a replicated instance of MiniSat (left) and MiniSat with progress saving (right). Both x-axes represent the chronological order of decisions.

Table 1 reports the results of this initial experiment, conducted using MiniSat [7], on a computer with Intel Pentium 3.4GHz processor and 2GB of RAM. The table reports runtime of MiniSat on each original and replicated instance. Let us first consider the first three runtime columns. The remaining columns will be discussed in the next section. According to this table, MiniSat can be more than two orders of magnitude slower on replicated instances, even though a replicated instance contains four identical copies of the original instance.

Further investigation on these instances reveals the source of inefficiency. In the next experiment, we plot indices of decision variables in chronological order. The left plot in Figure 2 shows such plot based on running MiniSat on the replicated instance of vmpc_21.shuffled-as.sat05-1955. Variable indices in the replicated instance range from 1 to 1764 (4×441 original variables). Each independent component in the instance occupies a contiguous range of variable indices. Each dark band in this plot indicates the solver's attempt to solve a component. We can see in this plot that MiniSat ended up solving all components multiple times. Most of the attempts to re-solve a component take non-trivial amount of work, as illustrated by the width of each band. This clearly illustrates that work repetition is responsible for a fair amount of the disproportionate increase in runtime of the solver on the replicated instances. Further experiments revealed that similar behavior persisted on other instance pairs as well[2].

[2] More experimental results are available in an extended version of the paper at http://reasoning.cs.ucla.edu/publications.html

Suite	Instance Count	Runtime*		# Solved	
		MiniSat	P. Saving	MiniSat	P. Saving
fvp sat 3.0	20	1134.027	45.59	10	20
grieu 05	32	5069.342	1789.555	16	19
IBM 2004 1_11	19	4422.805	1623.339	14	18
IBM 2004 1_14	19	329.039	194.078	19	19
manol pipe	31	5050.709	5247.547	30	31
pipe sat 1.0	10	92.108	973.436	6	8
liveness sat 1.0	10	114.576	888.857	5	6
vliw sat 2.0	9	59.312	887.14	5	5
narain 05	10	194.998	249.998	5	5
Total	160	16466.916	11899.54	110	131

Table 2. Runtime of MiniSat with and without progress saving. * The solvers' runtime for each suite is calculated from instances solved by both versions.

4 A Lightweight Caching Scheme

The solution we are proposing for this problem is simple and can be thought of as a lightweight partial component caching technique. Since far–backtracking could erase partial solutions, we simply save them. This technique, which we refer to as *progress saving*, requires keeping an additional array of literals, called the saved–literal array. Every time the solver performs a backtrack and erases assignments, each erased assignment is saved in the this array. Now, any time the solver decides to branch on variable v, it uses the saved literal, if one exists. Otherwise, the solver uses the default phase selection heuristic. Note that this technique would fit nicely on any Chaff-like solver implementation.

We integrated progress saving into MiniSat for the purpose of evaluation[3]. In this integration, the variable ordering heuristic needs not be changed. Now, consider the last three columns of Table 1, in which the runtimes of MiniSat with progress saving on original and replicated instances are compared. In all cases, there are significant improvements in runtime on replicated instances.

Furthermore, the decision behavior of MiniSat with progress saving on the replicated instance of vmpc_21.shuffled-as.sat05-1955 is shown on the right of Figure 2. This plot indicates a decease in work repetition. Though previously solved components are still revisited (thin strips of dots after dark bands), their solutions are almost immediately found, because of the saved literals.

5 Experimental Results

We now evaluate progress saving on a set of 1251 industrial benchmarks drawn from the SAT'05 competition [17] and [19, 10]. All experiments were performed on a Pentium 4, 3.8 GHz and 2GB RAM, with time limit of 1800 seconds.

Table 2 reports runtime on 160 instances selected from the total 1251 instances considered. According to this table, progress saving solves 21 more instances than MiniSat, improves the overall running time on those instance solved by both solvers, yet leads to worse running time on some of the instances.

[3] Progress saving was originally introduced in RSat [12, 14].

Figure 3 provides more comprehensive evidence on the effectiveness of progress saving as it considers all 1251 instances discussed above. On the left, we compare three versions of MiniSat (different phase selection heuristics) to MiniSat with its default heuristic augmented with progress saving. The x-axis lists the number of solved instances for a given cutoff time (y-axis), showing that progress saving dominates all three versions of MiniSat. On the right, we show a head-to-head runtime comparison between MiniSat and MiniSat with progress saving. Note that both axes here are in log-scale. This figure, which also depicts the best linear fit, provides further evidence on the effectiveness of progress saving.

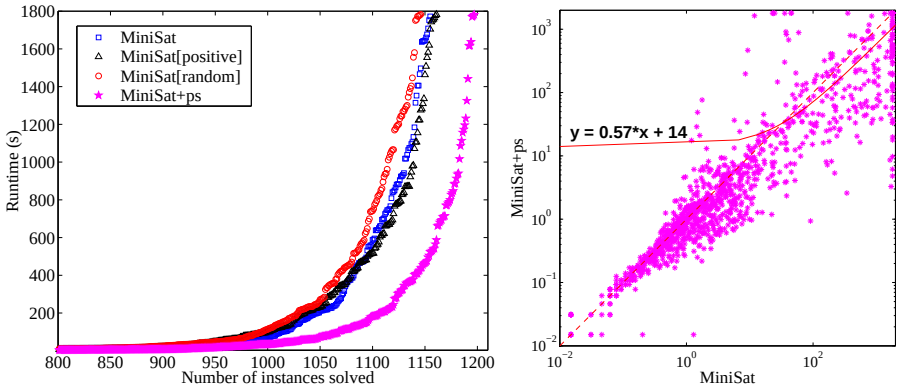

Fig. 3. Comparing three versions of MiniSat (different phase selection heuristics) to MiniSat with progress saving. The default phase selection heuristic of MiniSat splits on negative literals. We also consider splitting on positive and randomly chosen literals.

6 Related Work

Model counters and knowledge compilers have proven to benefit greatly from dynamic and semi–dynamic component analysis performed during the search [13, 15, 5, 3]. These techniques are usually too expensive to apply to SAT solving. Ginsberg addressed a very similar problem in the context of CSP [8]. The author proposed a new backtracking scheme called *dynamic backtracking*. This approach is superficially similar to ours. However, it may cause the search space after backtracking to become overly constrained, as pointed out by the author. Moreover, it would require a careful modification of the contemporary SAT framework to make it work as intended. Neither is the case for our solution.

Biere and Sinz showed that independent components do exist in some real-world SAT instances and proposed an efficient method to take advantage of the structure [4]. However, their approach is semi–dynamic, as it only considers permanent decompositions that occur in the absence of any decision. While improvements on artificially–generated instances were reported, similar gains did not materialized in their experiment on real–world instances.

7 Conclusion

We studied an inefficiency introduced by the conventional backtracking scheme of modern SAT solvers. We then proposed a low–overhead solution, called progress saving, that can be viewed as a component caching technique. The practicality of our solution is illustrated by experimental results, which show improvements on a wide range of problems when the technique is integrated into MiniSat.

References

1. Aloul, F., Markov, I., and Sakallah, K. Force: a fast and easy-to-implement variable-ordering heuristic. In *Proc. of the 13th ACM Great Lakes Symposium on VLSI 2003. pp. 116-119.* (2003).
2. Bayardo, R. J. J., and Schrag, R. C. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI'97* (Providence, Rhode Island), pp. 203–208.
3. Beame, P., Impagliazzo, R., Pitassi, T., and Segerlind, N. Memoization and dpll: Formula caching proof systems. In *Proc. of 18th Annual IEEE Conf. on Computational Complexity, Aarhus, Denmark.* (2003).
4. Biere, A., and Sinz, C. Decomposing sat problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 2* (2006).
5. Darwiche, A. New advances in compiling CNF to decomposable negational normal form. In *Proc. of European Conference on AI.* (2004).
6. Durairaj, V., and Kalla., P. Variable ordering for efficient sat search by analyzing constraint-variable dependencies. In *SAT'05* (August 2005).
7. Eén, N., and Sörensson, N. An extensible sat-solver. In *SAT'03* (2003).
8. Ginsberg, M. L. Dynamic backtracking. *Jrnl of Artf. Intel. Resrh. 1* (1993).
9. Huang, J., and Darwiche, A. A structure-based variable ordering heuristic for sat. In *(IJCAI'03)* (2003), pp. 1167–1172.
10. IBM. Ibm formal verification benchmark library. http://www.research.ibm.com /haifa/projects/verification/RB_Homepage/fvbenchmarks.html.
11. Marques-Silva, J. P., and Sakallah, K. A. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design* (1996), pp. 220–227.
12. Pipatsrisawat, K., and Darwiche, A. *SAT Solver Description: RSat.*
13. Roberto J. Bayardo, J., and Pehoushek, J. D. Counting models using connected components. In *Proc. of the 17th Natl. Conf. on AI.* (2000), AAAI Press / The MIT Press, pp. 157–162.
14. Rsat sat solver homepage. http://reasoning.cs.ucla.edu/rsat.
15. Sang, T., Bacchus, F., Beame, P., Kautz, H. A., and Pitassi, T. Combining component caching and clause learning for effective model counting. In *SAT'04.*
16. Sang, T., Beame, P., and Kautz, H. A. Heuristics for fast exact model counting. In *SAT* (2005), pp. 226–240.
17. SAT'05 Competition Homepage, http://www.satcompetition.org/2005/.
18. Stallman, R., and Sussman, G. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artf. Intel. 9* (1977).
19. Velev, M. N. Sat bnchmrk lib. www.miroslav-velev.com/sat_benchmarks.html.
20. Zhang, L., Madigan, C. F., Moskewicz, M. W., and Malik, S. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD* (2001), pp. 279–285.