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. In this paper, we study an inefficiency caused by far-backtracking, which is the type of non-chronological backtracking commonly practiced by SAT solvers. This inefficiency is caused by an undesirable effect that far-backtracking has on problems with component structure. We then propose a lightweight solution to the problem. The presented solution can be viewed as a partial component caching scheme that helps SAT solvers avoid re-solving sub-problems multiple times. Empirical evaluation of the technique reveals good performance improvements on a broad range of industrial SAT instances.

1 Introduction

The Boolean satisfiability problem (SAT) asks whether a given Boolean formula in conjunctive normal form (CNF) can be made true by some assignment of its variables. For example, the formula $\Delta = (a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c)$ is satisfiable, because assigning $a = true, b = false, c = true$ makes Δ evaluate to true. In this case, we say that $\langle a, \neg b, c \rangle$ is a solution for Δ. On the other hand, $\Gamma = (a \vee b) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg b)$ is unsatisfiable. This class of problems, which was the first member of the complexity class NP-complete [6], has played important roles in Computer Science from both theoretical and practical points of view. Many problems in real-world domains, such as hardware design, formal verification, and planning, can now be solved efficiently with a satisfiability-based approach [19, 15, 4, 18, 27, 16].

The majority of SAT solvers streamlined for dealing with real-world SAT problems are based on the DPLL algorithm introduced by Davis *et al.* in 1962 [8]. The DPLL algorithm in its original form is essentially a depth-first search algorithm that searches for a solution in the space of truth assignments. The algorithm is also augmented with a lookahead mechanism called unit propagation to help reduce the search space size. DPLL advances to the next level of the search tree by assigning *true* to a literal selected according to some heuristics.[1] Such assignment is called a *decision*. A *branching heuristic* is used to select a

* This paper extends the work in [20]

[1] A literal is defined as a variable or its negation.

variable to assign next. A *phase selection heuristic* is used to determine which value of the selected variable to try first.

After assigning the literal ℓ to *true*, any clause that contains ℓ becomes satisfied and can be removed from further consideration.[2] Similarly, the literal $\neg\ell$ now becomes falsified and can be removed from any clause it appears in. The process of removing all satisfied clauses and falsified literals from the formula is called *formula simplification*.[3]

Unit propagation is a process of inferring values of some variables based on the assignments made so far. According to unit propagation, any literal that appears alone in any clause after formula simplification can be set to *true* without changing the satisfiability of the current formula. For example, consider Δ defined above. After setting $a = true$, the second clause contains only c. Thus, we can set $c = true$ and continue working with a simpler formula. Unit propagation refers to the repeated application of formula simplification and the above rule until no more assignment can be inferred. Assignments derived during unit propagation are called *implications*.

During unit propagation, a *conflict* may arise. A conflict happens when a literal and its negation both become implications. A conflict is a sign of an inconsistency, which indicates that the current assignments cannot be extended to a solution. The algorithm must retract some of the assignments before it can continue its quest for a solution. The process of undoing assignments in order to escape from a conflict is called *backtracking*.

Over the years, many techniques have been invented to make DPLL more efficient. One of them is non-chronological backtracking [25, 2, 17, 21]. The original DPLL algorithm backtracks to the most recent decision whose negation has not been tried. This backtracking scheme often ends up running into the same conflict multiple times. Non-chronological backtracking deals with this problem by performing an analysis that helps identify the causes of the conflict. The algorithm can then backtrack past irrelevant assignments and focus on fixing the conflict at the sources.[4]

However, non-chronological backtracking has an undesirable side-effect that may cause an inefficiency when dealing with problems with a certain type of structure. Many SAT problems can be decomposed into components, especially those arising from real-world applications. A component is a sub-formula that can be solved independently. Components may exist originally or form dynamically as decisions are made and formula simplification takes place. Ideally, one would want to solve each component independently and then combine subsolutions to generate a global solution. However, in practice, precise component analysis is prohibitively expensive for SAT solving, because problem structure

[2] A clause is a disjunction of literals.

[3] In practice, literals and clauses are not actually removed from the problem representation. SAT solvers usually have a flag for each of them to keep track of its status.

[4] Unit propagation is an incomplete inference mechanism, hence, an inconsistency generated at an earlier level may not manifest into a conflict until much later.

may change with every decision and every backtrack. Since SAT solvers only look for a single solution, investing time in analyzing component structure can hardly pay off. As a result, virtually no modern solver takes dynamic component analysis into consideration, although some solvers have incorporated static component analysis in the preprocessing phase [11, 1, 13, 9].

The lack of dynamic component analysis is made worse by the use of non-chronological backtracking in modern SAT solvers. In particular, while backtracking past irrelevant assignments allows SAT solvers to deal with conflicts better, it may inadvertently erase solutions of independent components. Erasing these assignments means that the solver needs to solve these components again, potentially leading to great inefficiencies.

This problem has been observed and solutions have been proposed in [12, 5]. Nevertheless, the proposed solutions seem to offer limited practicality for SAT solvers, for which efficiency is a priority. We address this particular problem in this paper and provide two contributions. First, we perform an analytic and empirical analysis that substantiates our observations about the above behavior of DPLL-based solvers. Second, we propose a low-overhead technique that is oriented toward reducing the amount of repeated work performed by solvers that use non-chronological backtracking.

The rest of this paper is structured as follows. Section 2 gives more background on non-chronological backtracking and the way it is practiced by SAT solvers. In Section 3, we describe precisely the problem with this form of backtracking. In Section 4, we conduct an empirical study that exposes concretely this weakness of the backtracking mechanism. Then, in Section 5, we describe a technique that could help mitigate the problem. We present experimental evaluation of the proposed technique in Section 6. Related work is discussed in Section 7 and we close with some concluding remarks in Section 8.

2 Non-Chronological Backtracking in Modern SAT Solvers

Every time a conflict happens, chronological backtracking backtracks to the most recent unnegated decision and flips it. As mentioned earlier, this form of backtracking does not resolve conflicts efficiently. To consider a dramatic example, suppose that setting $v = true$ always result in a conflict (regardless of other assignments). If the solver happened to set (for the first time) $v = true$ at the 100^{th} decision, a conflict would arise and chronological backtracking would simply flip the assignment of v. However, since this conflict takes place rather deep in the search tree, the solver may, in the future, backtrack past v (erase v's assignment) and end up making the same mistake again when it later assigns a value to v. This scenario is depicted in Figure 1. In Figure 1 (a), a conflict due to the assignment $v = true$ causes the assignment of v to be flipped. Then, the search continues and the algorithm eventually backtracks past v to flip some earlier assignments. In Figure 1 (b), the assignment $v = true$ is repeated in a different context, resulting in an immediate conflict. Chronological backtracking

would respond by flipping the assignment of v. Apparently, this conflict can keep occurring under different contexts.

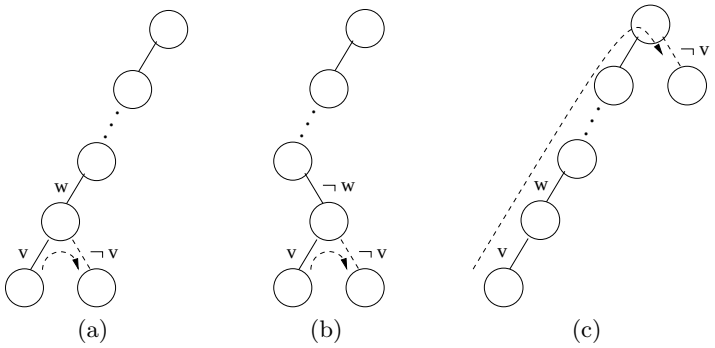

Fig. 1. (a) The assignment of v is flipped by chronological backtracking. (b) The same conflict is repeated in a different context. (c) Response to the same conflict by non-chronological backtracking. The assignment $v = false$ is made at the top of the search tree.

Non-chronological backtracking mitigates this problem by moving the fix for the conflict up the search tree so that it does not get erased easily by later backtracks. According to this scheme, whenever a conflict is found, the solver performs a conflict analysis in order to identify earlier assignments that contributed to the conflict. Then, the solver backtracks past all irrelevant decisions, and flips some assignments to resolve the conflict. In the above example, since the conflict was caused by the assignment $v = true$ alone, non-chronological backtracking would backtrack past all assignments and flip v to $false$ at the very top level of the search tree and continue to search (Figure 1 (c)). As a result, $v = false$ is effectively made permanent as any future backtrack can no longer erase this assignment (unless there is no solution). In general, non-chronological backtracking does not always have to backtrack to the very top level. The amount of backtracking could be as small as one level or as large as that in the above example.

Modern SAT solvers use a specific form of non-chronological backtracking that is tightly-coupled with clause learning [28, 17, 2]. Upon each conflict, SAT solvers would perform a conflict analysis that produces a learned clause, which is added to the formula. Adding learned clauses to the formula does not change its satisfiability. Moreover, learned clauses make some relationships between variables explicit and prevent SAT solvers from repeating the same conflicts in the future. The content of a learned clause is used to determine the destination of the subsequent non-chronological backtrack. The details of conflict analysis and determining backtrack destinations are beyond the scope of this paper.

Modern SAT solvers generate a specific type of learned clauses called *asserting clauses* [28]. The backtrack destination computed from an asserting clause is referred to as the *assertion level*. This specific form of non-chronological backtracking, which backtracks to the assertion levels computed from asserting clauses, is often referred to as *far-backtracking*. Far-backtracking is used by virtually all DPLL-based SAT solvers today.

3 Losing Work with Far-Backtracking

While far-backtracking improves performance of SAT solvers significantly in practice, it comes with an undesirable side-effect. The assignments that get erased during far-backtracking could contain solutions to previously solved parts of the problem. Hence, after backtracking, the solver needs to solve these parts again, potentially leading to great inefficiencies.

3.1 Independent Components

A component can be defined with respect to a CNF formula as follows: Let a set of clauses Δ be a CNF formula. A set of clauses C is a component in Δ if C shares no variable with $\Delta \backslash C$. Throughout this paper, we use the term *independent component* interchangeably with *component* to emphasize the fact that a solution of a component is independent from solutions of the rest of the problem.

Biere and Sinz showed in [5] that many SAT instances from real-world applications do exhibit component structures due to the nature of the problems. Independent components could also be generated dynamically. As the solver makes decisions, the formula gets simplified by removals of falsified literals and satisfied clauses. These simplifications may be substantial enough to disconnect the problem into multiple independent pieces. For example, originally, $\Delta = \{(a \vee b \vee c), (a \vee b \vee \neg c), (a \vee d \vee e), (a \vee d \vee \neg e)\}$ contains a single component. However, after setting $a = false$, the formula gets simplified and disconnected into two components: $\{(b \vee c), (b \vee \neg c)\}$ and $\{(d \vee e), (d \vee \neg e)\}$.

3.2 How Far-backtracking Erases Work

Contrary to chronological backtracking, far-backtracking may erase legitimate assignments which may be solutions to independent components. When far-backtracking is performed, all assignments between the point of conflict and the assertion level are erased.

To illustrate the extent of work that could be erased, let us consider an example in Figure 2. In this figure, the solver has solved a component using the first k decision levels. After a few decisions into another independent component, a conflict occurs. Suppose that the solver derives an asserting clause that contains only one literal ℓ for the conflict. In this case, the assertion level will

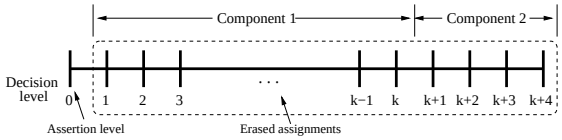

Fig. 2. Erased assignments due to a far-backtrack.

be zero. Consequently, far-backtracking will force the solver to undo all assignments between the current level and level zero. After the implication $\ell = true$ is discovered by unit propagation at level zero, the solver will continue looking for solutions of both components all over again, even though it had previously found a solution for one of the components.

While the above scenario is rather extreme, it serves well as an example to illustrate how much inefficiency could be introduced by far-backtracking. We will provide a more realistic, yet still somewhat synthetic, empirical study in the next section to quantify further this potential inefficiency of this backtracking scheme. We will finally perform an empirical study in Section 6 using real-world problems from the previous SAT competition.

4 An Empirical Study

To better illustrate and quantify the extent of work repetition a SAT solver might go through due to far-backtracking, we constructed instances that have a special structure. The intention was to create instances that contained multiple independent components, leading conventional SAT solvers, which are not aware of the problem structure, to solve each component multiple times. Each instance was simply generated by initially concatenating four identical copies of a small SAT instance. These bigger instances will be referred to as replicated instances throughout the rest of this paper.[5]

Table 1 reports the results of this initial experiment, conducted using a state-of-the-art solver, MiniSat [10], on a computer with Intel Pentium 3.4GHz processor and 2GB of RAM.[6] 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.

The results show that, MiniSat can be more than two orders of magnitude slower on replicated instances, even though each replicated instance contains four identical copies of the original instance. Although, we expected MiniSat to not be able to take advantage of component information, the increases in runtime

[5] Each original instance must be satisfiable. If we created a replicated instance from an unsatisfiable instance, the runtime for the replicated instance would not be interesting, as an inconsistency found in any single component would render the whole instance unsatisfiable.

[6] MiniSat won the last SAT competition in the industrial category [24].

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
IBM_FV_2004_1_02_3_SAT_dat.k70	1.29	923.97	716	1.03	11.78	11
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
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
difp_19	123.14	2939.51	24	37.64	304.42	8.1

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 instances over original instances.

on replicated instances are rather shocking. By just repeating an easy instance four times, we could create an instance that is impossible for MiniSat to solve within an hour.

We performed further investigation on these instances to track down the causes of the disproportionate runtime. In the following experiment, indices of decision variables were recorded during executions of MiniSat. These indices were later plotted in chronological order to reveal what decisions were made during the search.

The plot on the left of Figure 3 is from an execution of MiniSat on the replicated instance of vmpc_21.shuffled-as.sat05-1955. The original instance contains 441 variables, resulting in 1764 variables in the replicated instance. Each component in the replicated instance occupies a contiguous range of variable indices. Variables 1 through 441 belong to the first component. Variables 442 through 882 belong to the second component and so on. Each dark band in this plot indicates a solver's attempt to solve the component that contains the variables in the band's range. From this plot, we can see that the solver does a fine job in staying focused on one component at any given time. The execution trace of the solver indicates that component switching took place when the solver finished a component. Multiple bands that coexist on the same variable range indicate that the solver ended up solving each component more than once.

This is already a bad sign. As discussed earlier, once a solution of a component is discovered, it could be saved and combined with solutions from other components. Clearly, the solver did not take advantage of this fact. Nevertheless, one might argue that re-solving a previously solved component should be relatively straightforward, because modern SAT solvers add learned clauses to the

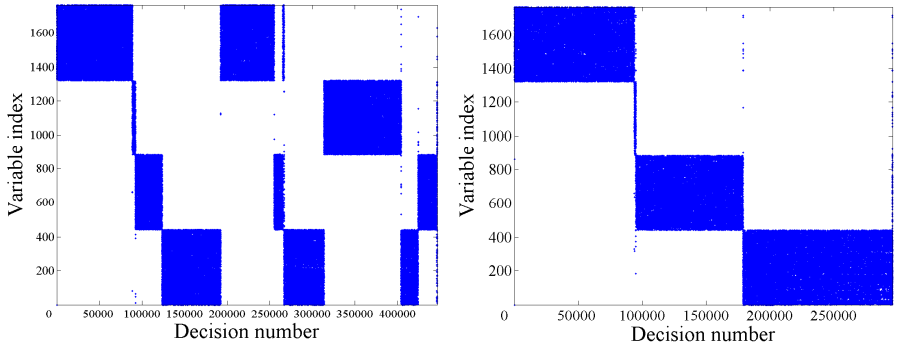

Fig. 3. Decision behavior on a replicated instance of MiniSat (left) and MiniSat with progress saving (right). Both x-axes represent the chronological order of decisions.

formula. These learned clauses should dramatically reduce the amount of work required to rediscover the solution.[7]

According to the left plot of Figure 3, this does not seem to be the case. Recall that bands on the same horizontal level corresponds to the solver's attempts on the same component. The widths of the bands vary greatly within the same component. More importantly, there is no sign that the solver performed significantly less work on later attempts. As a matter of fact, on a few occasions, the solver even spent more time on the later visits to a component.

The results of this experiment 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. More results can be found in Figure 7 in the Appendix.

5 A Lightweight Caching Scheme

The solution we are proposing for this problem is simple and can be thought of as a lightweight component caching technique. In particular, since far-backtracking could erase solutions from memory, 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. The size of this array is equal to the number of variables in the instance and each position of this array is associated with a variable. Every time the solver performs a backtrack and erases assignments, each erased assignment is saved in the saved-literal array. Both decision assignments and implication assignments are saved. Now, any time the solver decides to branch on variable v, it tries the saved value first, if one exists. Otherwise, the solver resorts to the default phase selection heuristic.

[7] Although MiniSat occasionally deletes learned clauses to reduce memory usage, no learned clause was deleted during this execution of MiniSat.

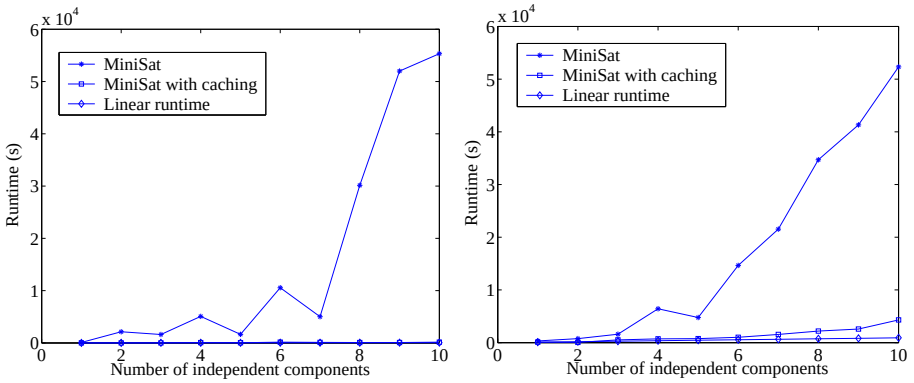

Fig. 4. Runtime of MiniSat on replicated instances with varied number of components. Hypothetical linear (on the number of components) runtime is also depicted in each plot.

Recall that a phase selection heuristic complements a branching heuristic: the latter chooses a variable to branch on, while the former chooses a particular value of the variable to instantiate first. Our technique can therefore be implemented as a phase selection heuristic, which requires very little overhead. This is to be contrasted with explicit component analysis which would be prohibitively expensive.

For evaluation purposes, we integrated progress saving into MiniSat. We performed our first test of the technique on the set of original and replicated instances in Table 1. The results are reported in the last three columns of the same table. The performance of MiniSat with progress saving on replicated instances improves significantly on all cases. The runtime ratio columns provide a strong evidence that progress saving helps reduce work repetition dramatically.

Consider now the plot on the right of Figure 3. This plot, which parallels the one on its left, visualizes the decision behavior of MiniSat with progress saving on the same replicated instance. Reduction in work repetition performed by MiniSat can clearly be seen from the differences of the two plots in the figure. Note that solved components are still revisited, as indicated by very thin strips of dots after each thick band. However, their solutions are almost immediately found, because all decisions made were based on assignments saved in the saved-literal array.

Our next experiment involved the generation of replicated instances of varying sizes to allow a more systematic study of the performance of MiniSat with and without progress saving. This experiment was performed on a machine with Intel Xeon 2.4GHz with 4GB of RAM. Figure 4 depicts results of this experiment on two sets of instances.[8] The plots clearly show that the runtime of MiniSat

[8] More results can be found in the Appendix.

Suite	Instance count	Runtime*		Instances 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
pipe sat 1.0	10	92.108	973.436	6	8
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
liveness sat 1.0	10	114.576	888.857	5	6
narain 05	10	194.998	249.998	5	5
vliw sat 2.0	9	59.312	887.14	5	5
Total	160	16466.916	11899.54	110	131

Table 2. Runtime of different versions of MiniSat. * The solvers' runtime for each suite is calculated from instances solved by all versions.

could grow rapidly as the number of components increases. On the other hand, MiniSat with progress saving scales much better with the increasing number of independent components.

Our experiments up to this point involved synthetic, replicated instances to illustrate the behavior of progress saving as a component caching scheme. In the next section, we will show experiments on unmodified instances from the previous SAT competition to illustrate the performance of this technique under more realistic settings.

6 Experimental Results

We now evaluate progress saving on real-world instances. In the following experiments, we used 1251 industrial instances drawn from the SAT'05 competition [24] and from contemporary benchmark libraries [26, 14]. All experiments were performed on a machine with Intel Pentium 4, 3.8 GHz and 2GB of RAM, with a time-out limit of 1800 seconds.

Table 2 reports runtime on 160 instances selected from the total 1251 instances considered. In this experiment, there are two versions of MiniSat being considered: the normal MiniSat and MiniSat with progress saving. To exclude time-out penalty from the runtime columns, these columns are based only on instances that can be solved by both versions of MiniSat within the time-out limit. According to this table, MiniSat with progress saving solves 21 instances more than MiniSat, improves the overall running time on those instance solved by both solvers, yet leads to worse runtime on some of the instances.

The next set of results provides more comprehensive evidence on the effectiveness of progress saving as it is based on all 1251 instances discussed above. On the left of Figure 5, we compare three versions of MiniSat (different phase selection heuristics) to MiniSat with its default phase selection heuristic augmented with progress saving. By default, MiniSat always branches on the negative literal

first. To demonstrate the effectiveness of progress saving, we compare it against two obvious modifications to the phase selection heuristic; branch on positive literals and branch randomly. 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. This plot shows that simple modifications to the phase selection heuristic do not significantly effect performance on a large set of instances, but progress saving does.

On the right of the same figure, we show a head-to-head runtime comparison between MiniSat and MiniSat with progress saving. Note that both axes here are in log-scale. Each data point in this figure represents one instance. The x-coordinate is the runtime of MiniSat on the instance, while the y-coordinate is the runtime of MiniSat with progress saving on the same instance. The best linear fit depicted in this plot provides further evidence on the effectiveness of progress saving when added to MiniSat.

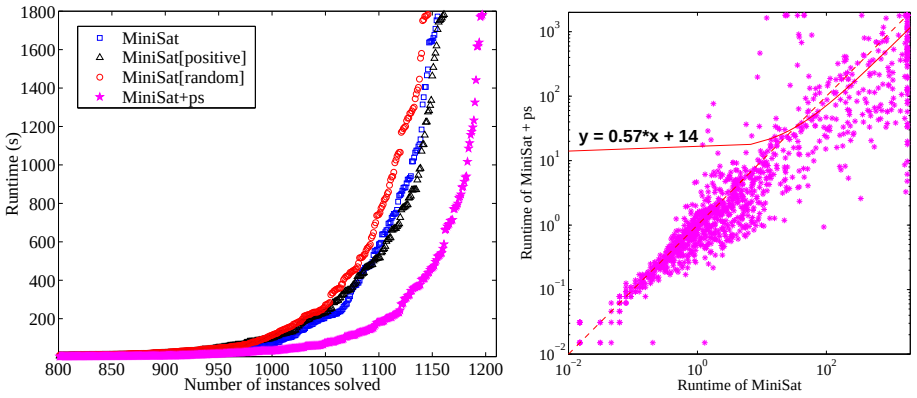

Fig. 5. (Left) Comparing three versions of MiniSat to MiniSat with progress saving. (Right) Head-to-head runtime comparison between MiniSat and MiniSat with progress saving.

For more insights on the effectiveness of progress saving, the above instances are categorized into 4 groups based on the number of unit learned clauses discovered by MiniSat (N). Figure 6 compares the runtime of MiniSat and MiniSat with progress saving on each group. Base on this figure, progress saving seems to provide the most gain on instances in which MiniSat discovered many unit learned clauses (large N). This result is consistent with the analysis in previous sections. Since MiniSat backtracks to the top level every time a unit learned clause is derived, the more unit clauses learned, the more likely partial solutions are erased. For moderate values of N, MiniSat with progress saving still dominates, although not as much as in the previous case. It is interesting to note that, even in the case where no unit clause was learned, progress saving still

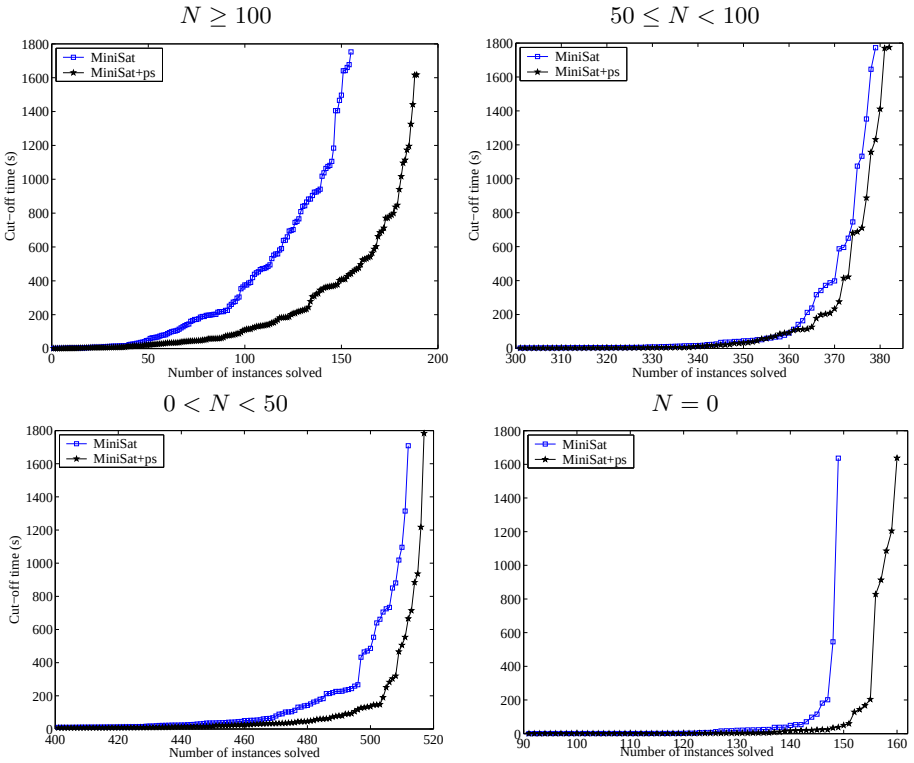

Fig. 6. Runtime comparison with instances broken down by the number of unit learned clauses discovered during the search by MiniSat (N).

improves the performance of the solver significantly. This is possible because progress saving is applied to all backtracks and could potentially cache solutions of components decomposed at any level during the search.

7 Related Work

Model counters and knowledge compilers, which are based on exhaustive search, have proved to benefit greatly from dynamic and semi–dynamic component analysis performed during the search [22, 23, 7, 3]. These techniques are usually too expensive to apply to SAT solving. Ginsberg addressed a very similar problem in the context of CSP [12]. The author proposed a new backtracking scheme called *dynamic backtracking*, which allows the solver to specifically undo bad assignment instead of backtracking to it. This approach is superficially similar to ours. However, it may cause the problem to become overly-constrained after backtracking, as pointed out by the author. Moreover, implementing this approach

in the contemporary SAT framework would require a careful modification to make it work as intended. Neither is the case for our solution.

Biere and Sinz showed in [5] that independent components do exist in real-world instances. They proposed to perform precise component analysis only at certain points during the search. In particular, their system performs component analysis only when the solver backtracks to the top level. Any solutions for independent components are then saved permanently. This approach is semi-dynamic, as it only considers problem decomposition in the absence of any decision. This means that unless components exist originally or the solver backtracks to the top level, this approach will not have any positive effect on performance. In contrast, progress saving has positive impacts even when the solver did not backtrack to the top level, as illustrated by Figure 6 (case $N = 0$).[9] In any case, component analysis certainly introduced a higher overhead, which resulted in slight runtime increase when the proposed technique was tested on real-world instances that contain components.

8 Conclusion

In this paper, we studied a problem with the conventional backtracking scheme used by modern SAT solvers. We then proposed a simple, yet very effective, solution, called progress saving. This technique can be viewed as a component caching scheme that can be implemented with a very low overhead as a phase selection heuristic on any contemporary SAT framework. Experimental results show improvements on a wide range of industrial SAT problems.

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.* (2003).
2. Bayardo, R. J. J., and Schrag, R. C. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)* (Providence, Rhode Island, 1997), pp. 203–208.
3. Beame, P., Impagliazzo, R., Pitassi, T., and Segerlind, N. Memoization and dpll: Formula caching proof systems. In *Proceedings of 18th Annual IEEE Conference on Computational Complexity, Aarhus, Denmark (2003).* (2003).
4. Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. Symbolic model checking without BDDs. *Lecture Notes in Computer Science 1579* (1999), 193–207.
5. Biere, A., and Sinz, C. Decomposing sat problems into connected components. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT) 2* (2006).

[9] The accuracy of this comparison is based on the assumption that this set of instances do not originally contain multiple components. We did not verify this assumption due to limited resource.

6. Cook, S. A. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing* (New York, NY, USA, 1971), ACM Press, pp. 151–158.
7. Darwiche, A. New advances in compiling CNF to decomposable negational normal form. In *Proceedings of European Conference on Artificial Intelligence* (2004).
8. Davis, M., Logemann, G., and Loveland, D. A machine program for theorem-proving. *Commun. ACM 5*, 7 (1962), 394–397.
9. Durairaj, V., and Kalla., P. Variable ordering for efficient sat search by analyzing constraint-variable dependencies. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing* (August 2005).
10. Eén, N., and Sörensson, N. An extensible sat-solver. In *SAT* (2003), pp. 502–518.
11. Fadi A. Aloul, Igor L. Markov, K. A. S. Faster sat and smaller bdds via common function structure. In *Technical Report #CSE-TR-445-01* (November 2001), University of Michigan.
12. Ginsberg, M. L. Dynamic backtracking. *Journal of Artificial Intelligence Research 1* (1993), 25–46.
13. Huang, J., and Darwiche, A. A structure-based variable ordering heuristic for sat. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)* (2003), pp. 1167–1172.
14. IBM. Ibm formal verification benchmark library. http://www.research.ibm.com/haifa/projects/verification/RB_Homepage/fvbenchmarks.html [Online; accessed 30-11-2006].
15. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science* (Washington, D.C., 1990), IEEE Computer Society Press, pp. 1–33.
16. Kautz, H. A., and Selman, B. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)* (1992), pp. 359–363.
17. 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.
18. McMillan, K. Symbolic model checking. PhD thesis, Pittsburgh, PA, USA (1992).
19. Nam, G.-J., Sakallah, K. A., and Rutenbar, R. A. Satisfiability-based layout revisited: detailed routing of complex fpgas via search-based boolean sat. In *FPGA '99: Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays* (New York, NY, USA, 1999), ACM Press, pp. 167–175.
20. Pipatsrisawat, K., and Darwiche, A. A lightweight component caching scheme for satisfiability solvers. To appear in SAT'07.
21. Prosser, P. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3) (August 1993), 268–299.
22. Roberto J. Bayardo, J., and Pehoushek, J. D. Counting models using connected components. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence* (2000), AAAI Press / The MIT Press, pp. 157–162.
23. Sang, T., Bacchus, F., Beame, P., Kautz, H. A., and Pitassi, T. Combining component caching and clause learning for effective model counting. In

Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT) (2004).
24. SAT'05 Competition Homepage, http://www.satcompetition.org/2005/.
25. Stallman, R., and Sussman, G. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence 9* (October 1977).
26. Velev, M. N. Sat benchmarks library. http://www.miroslav-velev.com/sat_benchmarks.html.
27. Velev, M. N., and Bryant, R. E. Effective use of boolean satisfiability procedures in the formal verification of superscalar and vliw. In *DAC '01: Proceedings of the 38th conference on Design automation* (New York, NY, USA, 2001), ACM Press, pp. 226–231.
28. 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.

Appendix

A Additional Experimental Results

In this section, we provide additional results of some experiments discussed in Section 4.

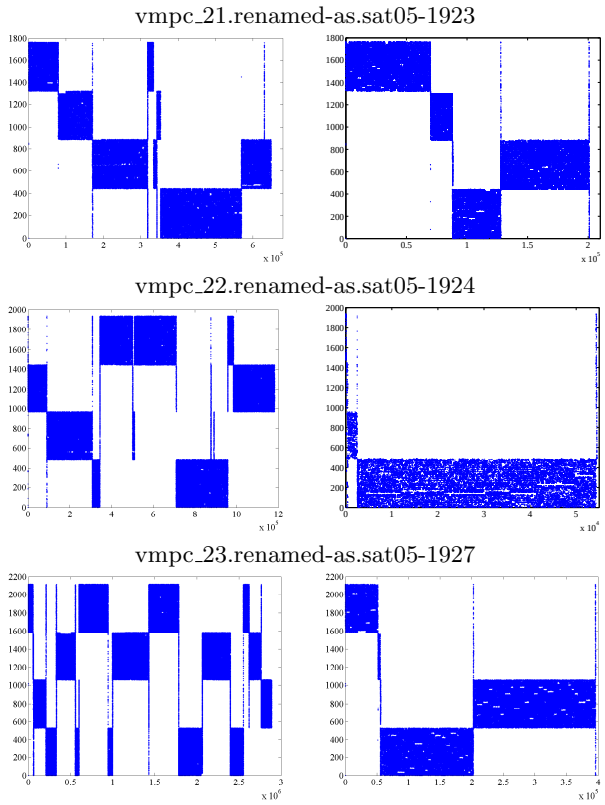

Fig. 7. Additional plots showing decision behavior of MiniSat with and without progress saving. Plots of MiniSat are shown on the left column. Paralleled plots for MiniSat with progress saving are shown on the right column. The x-axis of each plot represents decision number and the y-axis represents variable index.

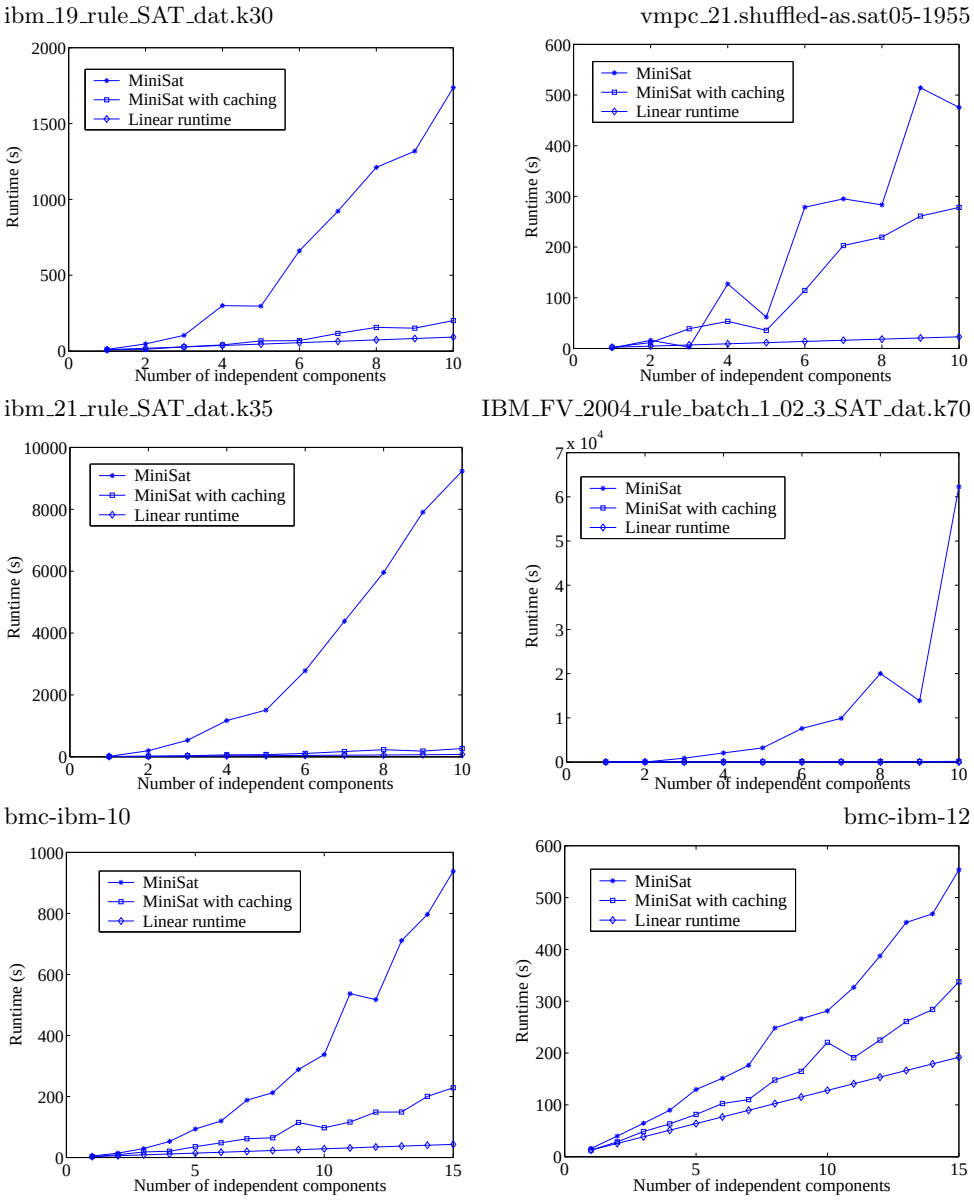

Fig. 8. Runtime of MiniSat on replicated instances with varied number of components.