# Strong Conflict Analysis for Propositional Satisfiability*

HoonSang Jin
CAE center, System LSI division
Semicon. Business, Samsung Elec. CO., LTD

Fabio Somenzi
University of Colorado at Boulder
Dept. of Electrical and Computer Engineering

## Abstract

We present a new approach to conflict analysis for propositional satisfiability solvers based on the DPLL procedure and clause recording. When conditions warrant it, we generate a supplemental clause from a conflict. This clause does not contain a unique implication point, and therefore cannot replace the standard conflict clause. However, it is very effective at reducing excessive depth in the implication graphs and at preventing repeated conflicts on the same clause. Experimental results show consistent improvements over state-of-the-art solvers and confirm our analysis of why the new technique works.

## 1. Introduction

The satisfiability of a propositional formula is one of the most studied problems in computer science, from both the theoretical and the practical standpoints. Algorithm based on techniques like stochastic local search [5, 15], backtracking search [18, 21, 10, 6], and Stålmarck's proof procedure [16] have been implemented; their increasing efficiency has led to more problems being tackled by reduction to SAT. In EDA, in particular, most successful satisfiability (SAT) solvers employ variants of the David-Putnam-Logemann-Loveland (DPLL) procedure [3], which is based on backtracking search. Recent solvers improve over the classical DPLL procedure in several ways. Clause recording based on conflict analysis and non-chronological backtracking [18] have been introduced to prune the search space. Efficient implementations like those based on two-watched literal schemes [10] enhance the speed of implication. The choice of the decision variables also has a large impact the search time. Hence, considerable attention has been devoted to the problem. (See, for instance, [17, 10, 6, 8].)

Among these techniques, of interest to us are conflict analysis and clause recording: When a conflicting assignment is found, it is analyzed to identify a subset that is still conflicting. The disjunction of the negation of the literals in the subset is a *conflict-learned clause* (or, more concisely, a *conflict clause*) that can be added to the given SAT instance to prevent the examination of regions of the search space that are guaranteed to contain no solutions. Not all conflict clauses are worth keeping; many SAT solvers periodically discard those that have proved ineffective. Experimental evidence [18, 21, 10, 22] shows that conflict analysis can be efficiently implemented to solve problems of industrial scale. One may argue that effective conflict analysis is the most significant mechanism in scaling DPLL to realistic problems because it directly affects the fraction of the search space that must be explicitly explored. Despite its importance, however, there has been little recent work on conflict analysis.

The SAT solver Grasp [18] used conflict clauses (called *no-goods* to guide the exploration of the search space. A clause containing a *Unique Implication Point* (UIP) becomes *asserting* after backtracking, and therefore guarantees progress in the examination of the possible solutions. The authors of [22] studied several ways of extracting a clause containing a UIP from the chains of implications that led to a conflict (the so-called *implication graph* of the conflict) and found that choosing the *first* UIP (i.e., the one closest to the conflict) worked best.

A compact conflict clause, which contains few literals, is beneficial because it leads to fast Boolean Constraint Propagation (BCP) and detects conflicts earlier. Moreover, it prunes more of the search space. A conflict clause is the result of a series of resolutions steps applied to the current conflicting clause. We are more likely to find a compact conflict clause if we start from compact clauses involved in the current conflict. Therefore, improving the quality of conflict clauses tends to have a rippling effect and may lead to dramatic speedups. In difficult SAT instances, solvers often spend inordinate amounts of time on comparatively small fractions of the search space. We call such small sets of configurations *hot spots*. The existence and location of such hot spots obviously depends on the SAT algorithm and one objective of an effective conflict analysis procedure is to reduce their occurrence as much as possible by adding appropriate conflict clauses. Though clauses based on first UIPs are better than other UIP-based clauses, they do not directly address the issue of hot spots.

Just as different clauses may be derived from the same implication graph, different implication graphs may be obtained from the same sequence of decisions, depending on the order in which implications are propagated. Two recent papers propose improvements in the quality of the conflict clauses obtained by modifying the implication graph produced by the SAT solver. The authors of [4] point out that even though conflict analysis may start from the same conflicting clause, it may produce different conflict-learned clauses on different implication graphs. They propose a method that updates the antecedents of a variable if a smaller clause is found while propagating implications so that the conflict analysis may find a smaller conflict clause.

*Shrinking* is a technique, used in the Jerusat solver [11], that removes some literals from a conflict clause generated by conflict analysis. To do so, it backtracks enough to undo all assignments to the literals in the clause, and then re-applies only the assignments in the clause. This creates a new, often smaller, implication graph for the same conflict. Unfortunately, shrinking is quite an expensive operation because of the amount of backtracking required, and it does not guarantee a reduction in the number of literals. Multiple conflict analysis is another way to enhance conflict analysis. However, it is a costly technique, which should be guided by a good criterion to limit as much as possible its application to unprofitable cases.

*This work was supported in part by SRC contract 2004-TJ-920.

3-9810801-0-6/DATE06 © 2006 EDAA

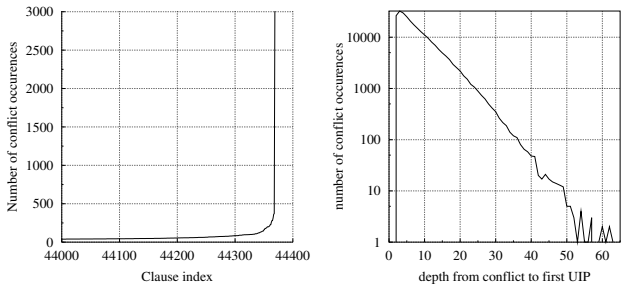


Figure 1: Distributions of conflict occurrences (left) and implication graph depth (right) for SAT instance "c880"

In our experience, techniques that restructure the implication graph must be used sparingly because they tend to be expensive and they are often ineffective. In this paper we focus instead on a stronger conflict analysis, one that often learns better clauses than those including the first UIP. We observe that there are many cases when multiple conflicts occur on one clause. A clause generated by conflict analysis prevents the same conflict from occurring again. However, the current conflict analysis is not strong enough to prevent further conflicts on the same clause. When this happens, the clauses learned through conflict analysis are often ineffective at pruning the search. The left part of Fig. 1 shows an example of distribution of conflicts. The $x$ axis gives the clause index and the $y$ axis shows how many times that clause is the site of a conflict during the SAT solver run. There is a reasonable number of cases in which many conflicts happen on the same clause. Our approach learns two clauses for such conflicts. One clause contains the first UIP: Its main responsibility is to drive the search forward. The second clause is devoted to a better pruning of yet-to-be-visited parts of the search space. It is generated when the first clause is likely to be ineffective and the chance of repeated conflicts on the same clause is deemed high.

The right part of Fig. 1 shows that there is a significant variation in the *depth* of the implication graphs between the conflict and the first UIP. The depth is here measured in terms of the longest path in the (acyclic) graph. This depth is one of our metrics to decide whether to expect repeated conflicts and whether supplemental clauses would be beneficial.

The rest of this paper is organized as follows. Background material is covered in Section 2. The motivation of the proposed method is examined in Section 3. Section 4 discusses our algorithm, while experimental results are presented in Section 5, and conclusions in Section 6.

## 2. Preliminaries

In this paper we assume that the input to the SAT solver is a formula in Conjunctive Normal Form (CNF). This assumption simplifies the description of the algorithms, which, however, can be extended to non-clausal formulae.

A CNF formula is a set of *clauses*; each clause is a set of *literals*; each literal is either a variable or its complement. The function of a clause is the disjunction of its literals, and the function of a CNF formula is the conjunction of its clauses.

Conflict analysis relies on a Directed Acyclic Graph (DAG) called the *implication graph*. Each vertex of this graph corresponds to a variable that is assigned either by decision or by implication. An edge connects a pair of vertices if they are antecedent and consequent of an implication. Whenever an implication takes place, the implication graph is extended by adding edges from the antecedents of the implication to its consequent. The roots of the implication graph correspond to decision assignments, while the conflicting assignments, when they are identified, are among the leaves. Every vertex cutset of the implication graph corresponds to a partial assignment sufficient to imply the conflict. Hence, from any such cut of the graph one can derive a conflict clause by disjunction of the negation of the literals in the cutset. If a cut contains only one literal assigned at a certain decision level, such literal is a Unique Implication Point (UIP).

Figure 2 shows an example of UIP. In the figure, the label of a node represents the value assigned to the variable. For example, $x_{18}$ and $\neg x_{18}$ denote $x_{18} = \mathsf{true}$ and $x_{18} = \mathsf{false}$, respectively. A gray circle represents a decision made at an earlier decision level, while a dark circle represents the conflicting condition. The other nodes are variables that are decided or implied at the current decision level. The dashed line labeled "cut1" shows the result of conflict analysis based on the first UIP and the conflict clause. As shown in the figure, there may be more than one UIP in the implication graph. The clauses corresponding to "cut1" and "cut2" contains UIPs $x_{10}$ and $x_{14}$, respectively; $x_{14}$ is the first UIP, since it is the UIP closest to the conflict. The conflict clause analysis based on the first UIP detects the conflict clause $(\neg x_4 \vee \neg x_5 \vee \neg x_{14})$ and adds it to the clause database. Then, the solver backtracks to the highest decision level in the conflict clause, except the current decision level, which is the higher level between those of $x_4$ and $x_5$. *Deduce* will immediately find the implication $x_{14} = \mathsf{false}$, since there is only one variable unassigned and the other literals are assigned to $\mathsf{false}$. We say that the conflict clause is *asserting*. Conflict clauses based on the first UIP have been empirically found to work well [22].

The termination of SAT solvers that use conflict clauses based on UIPs was studied in [23]. The termination proof relies on the conflict clause becoming asserting after backtracking. The proof defines a progress measure on the states of the solver. In particular it looks at the contents of the decision stack every time Boolean Constraint Propagation (BCP) is completed, either due to a new decision or to a backtrack.

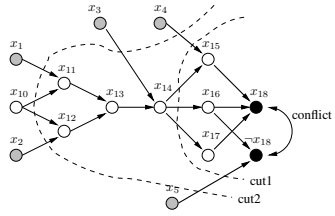


Figure 2: Example of unique implication point

## 3. Motivation

Any cut of the implication graph can be used to build a conflict clause, even though a clause with multiple literals from the current decision level cannot generate further implications after backtracking. In Fig. 3 the dashed line labeled "cut1" shows the result of AnalyzeConflict(). Based on it, the conflict clause $(\neg x_{10} \vee \neg x_3 \vee \neg x_4 \vee \neg x_5 \vee \neg x_6)$, which contains the first UIP, is added to the clause database. We can also add the clauses based on "cut2" and "cut3", since they are also generated by series of resolutions. For example, the conflict clause $(\neg x_{13} \vee \neg x_{14} \vee \neg x_4 \vee \neg x_6)$ corresponds to "cut3."

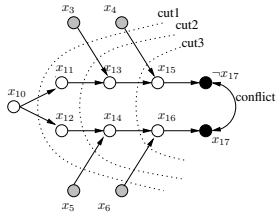


Figure 3: Example of conflict clauses based on arbitrary cut

Suppose the conflict clause $(\neg x_{10} \vee \neg x_3 \vee \neg x_4 \vee \neg x_5 \vee \neg x_6)$ is added to the clause database and causes backtracking to a certain earlier decision level. Suppose next that $x_1 = \mathsf{true}$ is assigned by decision making and that BCP creates the implication graph shown in Figure 4. This graph is similar to the one of Figure 3 and the conflicting clause is the same. This is indeed a case of multiple conflicts on the same clause. In our example, the assignment $x_2 = \mathsf{true}$ also results in a similar implication graph. Each time, the learned conflict clause is different.

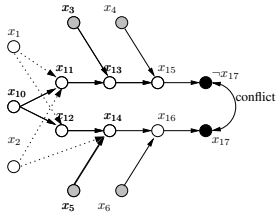


Figure 4: Example of creating similar implication graph

Suppose the conflict clause $(\neg x_{13} \vee \neg x_{14} \vee \neg x_4 \vee \neg x_6)$ based on "cut3" is added to the database besides the UIP-based conflict clause. When the assignment $x_1 = \mathsf{true}$ is later made, this clause will cause a conflict after fewer implications. This will lead to a simpler implication graph. It turn, this is likely to lead to a more concise conflict clause.

Since, in many cases, we can prune more of the search space with a more compact conflict clause, the addition of the second conflict clause may have a substantial rippling effect. An attempt was made in [12] to use intermediate conflict clauses generated by arbitrary cuts of the implication graph, but effective criteria for when those clauses are useful and for how to make them useful were not identified. In this paper we propose a detailed criterion and rationale for using supplemental conflict clauses.

The two conflict clauses—one containing the first UIP, and the other corresponding to a cut close to the conflict—play different roles. The former becomes asserting after backtracking and, hence, is responsible for termination. The new conflict clause is used to prune the search space.

Adding clauses to the database indiscriminately may substantially slow down BCP. To prevent this, supplementary clauses should be generated only when there is reasonable expectation that they will be useful.

Conflict clauses based on first UIPs are preferred to other UIP-based clauses because they usually have fewer literals and are close to the conflict in the implication graph so that they may block conflicting conditions more efficiently. However, there are cases when the first UIP is the decision variable. Such a first UIP may be far away from the conflict. If that is the case, we may have conflicts on the same conflicting clause in the future and we should add other clauses. Notice since most SAT solvers periodically delete

```
1   AnalyzeConflictStrong(c, cl) {
2       bl = 0;
3       size = MAXINT;
4       nBacktrace = 0;
5       cut = ∅;
6       learned = ∅;
7       H = QueueInitialize();
8       bl = AnalyzeConflictAux(H, learned, bl, clause, cl);
9       while (lit = ExtractMaxIndex(H)) {
10          if (Size(H) == 0) {
11              learned = learned ∪ lit;
12              break ;
13          }
14          if (nBacktrace > BOUND && size > Size(H)) {
15              cut = learned ∪ lit ∪ All elements of H;
16              size = Size(H);
17          }
18          ante = Antecedent(lit);
19          bl = AnalyzeConflictAux(H, learned, bl, ante, cl);
20          nBacktrace++;
21      }
22      AddConflictClause(learned);
23      AddConflictClause(cut);
24      return (blevel);
25  }
```

Figure 5: Proposed conflict analysis algorithm

ineffective conflict clauses, the overhead due to these supplemental clauses can be kept low.

# 4. Algorithm

In [4] multiple conflict clauses are derived from analyzing different conflicting clauses that are produced by continuing BCP after the first conflict is detected. By contrast, in our approach, even though we add multiple conflict clauses, we concentrate on one conflict analysis. By adding multiple conflict clauses from a single conflict, we try to prevent further conflicts from reaching the same clause and simplify the future implication graphs as a consequence. In this work we add one more conflict clause when it is needed. It is natural to consider adding more clauses based on different cuts; we are currently investigating a good criterion for the addition of multiple conflict clauses.

Figure 5 shows the pseudocode of our proposed strong conflict analysis. A conflict clause containing more than one variable assigned at the current decision level is detected in addition to the first UIP-based conflict clause. This additional conflict clause tends to have fewer variables, since it is closer to the conflict.

During conflict analysis, every resolution step produces a cut in the graph. The initial cut is at the conflict site; successive resolutions move it backward toward the roots. The procedure keeps track of the number of variables assigned at the current decision level in the cut. The variables assigned at earlier decision levels are saved in "learned" of Figure 5 while backtracing the implication graph. By collecting the literals in "learned" and those in the priority queue "H", we can generate the conflict clause based on the current cut.

The pseudocode of Figure 5 ignores some details for the sake of clarity. The real implementation applies a filter to select better conflict clauses. Cuts that are too close to the conflict are avoided, since their ability to block inconsistent assignments is too similar

to that of the conflicting clause. On the other hand, if a cut is too close to the decision variable then it is seldom effective at blocking conflicts from other sources of implication. (See Fig. 4.) Therefore the procedure tries to locate a cut half way between decision and conflict. The distance is measured by the depth in the implication graph. The size of the cut should also be taken into account: If it is too close or larger than the size of the first UIP cut, then the effectiveness of the supplemental clause will be reduced. If we cannot find such a clause then we choose a cut that constains less than 2/3 of the literals of the UIP-based conflict clause. Finally, we only generate a supplemental clause if there has been a previous conflict on the same clause or if the UIP is the decision variable at the current level.

The generation of a family of similar implication graphs is one reason why the search spends excessive time on a hot spot. This problem is addressed by the addition of clauses based on alternate cuts of the implication graph. Redundancy in conflict clauses is another reason for the search to dwell in a hot spot. The shrinking method of [11] can be effective in removing such redundancy. However, we invoke it only when the number of conflicts on a given clause exceeds a threshold because it is very expensive.

The additional clauses produced by ANALYZECONFLICTSTRONG help simplify future implication graphs, since conflicts may occur on these additional clauses. It is likely that a simpler implication graph generates more concise conflict clauses. This is helpful in pruning the search space and in resolving the empty clause earlier.

## 5. Experimental Results

We have implemented the proposed conflict analysis algorithm in CirCUs [8, 7, 9]. To show its efficiency we conduct three sets of experiments. To emphasize robustness, the hard equivalence checking benchmarks and microprocessor formal verification benchmarks used in the annual SAT competitions are selected. A set of Bounded Model Checking (BMC [2, 7]) benchmarks are also included since BMC is a prime example of a problem that is tackled by reduction to SAT. The industrial benchmarks of the SAT 2004 competition [13] are also used for our experiments. The value of the BOUND parameter in ANALYZECONFLICTSTRONG() is 10 and the shrinking method is invoked only when the number of conflicts on the conflicting clause exceeds 10.

The first set of experiments have been performed on 2.4 GHz Pentium IV with 1GB of RAM running Linux. We have set the time out limit to 10,000 s. Table 1 compares the CPU time, the numbers of decisions and conflicts of three SAT solvers, namely CirCUs (with strong conflict analysis), BerkMin561 [1], Zchaff (2004.11.15) [20] and SatELiteGTI [14]. In the table, 'TO' denotes time out cases and 'MO' denotes memory out cases. BerkMin561 is a close relative of forklift, which was winner of the 2003 SAT competition for the industrial benchmarks. We use BerkMin561 since it is the only version that is publicly available. Zchaff is the winner of the 2004 SAT competition in the industrial category. SatELiteGTI is the winner of the 2005 SAT competition in 5 categories including the industrial category. The hard equivalence checking and CPU verification instances (namely 12pipe_bug) are used for this set of experiments. If no solver can solve an instance within the time limit, then the instance is excluded from the table. CirCUs shows rather consistent improvement over BerkMin561, Zchaff, and SatELiteGTI. Table 2 compares the CPU times, and the numbers of decisions and conflicts of CirCUs with and without the proposed strong conflict analysis. The performance of CirCUs without proposed conflict analysis is comparable to that of Zchaff. As one can see in the table, the improvements are clearly from the proposed method. Even though it adds more conflict clauses from one conflict, CirCUs ends up with fewer conflict clauses. This shows the ability of the proposed conflict analysis to prune the search space. We also found that the shrinking method is still expensive in spite of its limited application in CirCUs. The criterion for shrinking should be further investigated.

Table 2: Comparison Table of CirCUs with and without Strong Conflict Aanalysis

| name | new CirCUs | | | old CirCUs | | |
|---|---|---|---|---|---|---|
| | CPU | # Dec | # Conf | CPU | # Dec | # Conf |
| c880 | 20 | 126k | 55k | 312 | 530k | 341k |
| c3540 | 983 | 747k | 449k | TO | TO | TO |
| c7552 | 41 | 347k | 38k | 51 | 370k | 46k |
| dalu | 3,469 | 1,597k | 829k | TO | TO | TO |
| des | 252 | 1,412k | 70k | 264 | 1,493k | 70k |
| frg1 | 4,987 | 1,632k | 1,201k | TO | TO | TO |
| frg2 | 331 | 842k | 175k | 814 | 1,189k | 373k |
| i10 | 3,098 | 1,515k | 737k | TO | TO | TO |
| i8 | 3,802 | 1,955k | 620k | TO | TO | TO |
| rot | 155 | 466k | 187k | 4323 | 2,324k | 1,591k |
| term1 | 329 | 328k | 228k | TO | TO | TO |
| vda | 586 | 391k | 254k | 578 | 538k | 307k |
| bug1 | 130 | 156k | 12k | 4390 | 2,857k | 626k |
| bug2 | 239 | 305k | 26k | 213 | 191k | 210k |
| bug3 | 63 | 102k | 4k | 1125 | 662k | 151k |
| bug4 | 18 | 44k | 1k | TO | TO | TO |
| bug5 | 281 | 316k | 31k | 90 | 115k | 10k |
| bug6 | 1 | 10k | 0.06k | 14 | 10k | 0.06k |
| bug7 | 546 | 319k | 62k | 3015 | 2,034k | 418k |
| bug8 | 19 | 60k | 0.6k | 3041 | 2,035k | 447k |
| bug9 | 90 | 136k | 11k | 86 | 100k | 12k |
| bug10 | 313 | 345k | 33k | 4681 | 3,084k | 756k |

Table 3: Number of average literals in conflict clauses

| CNF name | new CirCUs | old CirCUs | speed-up |
|---|---|---|---|
| frg2 | 26.9 | 105.1 | 2.5X |
| rot | 26.5 | 86.6 | 27.7X |
| c880 | 17.4 | 48.5 | 15.5X |
| c7552 | 15.4 | 28.6 | 1.3X |
| des | 42.6 | 86.1 | 1.1X |
| vda | 108.1 | 242.3 | 1.0X |

Table 3 shows the average number of literals in conflict clauses. When it obtains a big improvement in CPU time, typically the new CirCUs generates much more concise conflict clauses than the old CirCUs. This observation backs up our claim that concise conflict clauses can be generated thanks to the additional conflict clauses.

The second experimental setup is as follows. We build BMC instances with given Linear Time Logic (LTL) properties from the VIS benchmark suite [19]. We check for paths of length up to 20. These experiments have been performed on 1.7 GHz Pentium IV with 1 GB of RAM running Linux with a 10,000 s timeout. Figure 6 (left) shows the log-log scatterplot comparing CirCUs to Zchaff. The upper line is the diagonal. The lower line is a regression curve of the form $y = \kappa \cdot x^{\eta}$, where $\kappa$ and $\eta$ are obtained by least-square fitting. The separation of the two lines indicates that the new conflict analysis provides a speedup over Zchaff.

We conducted the third set of experiments on the SAT2004 competition benchmark set. One can find more information on these

Table 1: Comparison Table of CirCUs, BerkMin561, Zchaff and SatELiteGTI

| | CirCUs | | | BerkMin561 | | | Zchaff | | | SatELiteGTI | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| name | CPU | #Dec | #Conf | CPU | # Dec | # Conf | CPU | # Dec | # Conf | CPU | # Dec | # Conf |
| c880 | 20 | 126k | 55k | 68 | 301k | 152k | 5,378 | 2,009k | 1,200k | 67 | 928k | 435k |
| c3540 | 983 | 747k | 449k | 2,935 | 3,362k | 1,911k | TO | TO | TO | 4705 | 9,706k | 5,220k |
| c7552 | 41 | 347k | 38k | 70 | 645k | 49k | 126 | 643k | 35k | 28 | 406k | 40k |
| dalu | 3,469 | 1,597k | 829k | TO | TO | TO | TO | TO | TO | TO | TO | TO |
| des | 252 | 1,412k | 70k | 349 | 1,935k | 72k | 551 | 1,795k | 61k | 169 | 1,795k | 68k |
| frg1 | 4,987 | 1,632k | 1,201k | TO | TO | TO | TO | TO | TO | TO | TO | TO |
| frg2 | 331 | 842k | 175k | 1,260 | 2,518k | 528k | 2,475 | 2,974k | 359k | 454 | 2,441k | 465k |
| i10 | 3,098 | 1,515k | 737k | TO | TO | TO | TO | TO | TO | TO | TO | TO |
| i8 | 3,802 | 1,955k | 620k | 3,276 | 6,635k | 1,107k | TO | TO | TO | 1208 | 2,999k | 519k |
| rot | 155 | 466k | 187k | 809 | 1,806k | 588k | 664 | 1,457k | 213k | 265 | 1,362k | 654k |
| term1 | 329 | 328k | 228k | 910 | 1,087k | 661k | TO | TO | TO | 3849 | 7,572k | 5,487k |
| vda | 586 | 391k | 254k | 572 | 484k | 336k | 3,384 | 1,623k | 413k | 396 | 559k | 309k |
| bug1 | 130 | 156k | 12k | 211 | 304k | 23k | 3,408 | 1,828k | 47k | MO | MO | MO |
| bug2 | 239 | 305k | 26k | 238 | 342k | 26k | 4,365 | 2,323k | 60k | 40 | 77k | 1k |
| bug3 | 63 | 102k | 4k | 209 | 268k | 24k | 3,902 | 1,956k | 50k | 46 | 158k | 13k |
| bug4 | 18 | 44k | 1k | 141 | 237k | 14k | 2,685 | 1,491k | 35k | 34 | 27k | 0.2k |
| bug5 | 281 | 316k | 31k | 2521 | 2,892k | 329k | TO | TO | TO | 35 | 149k | 10k |
| bug6 | 1 | 10k | 0.07k | 69 | 108k | 7k | 2 | 8k | 0.02k | MO | MO | MO |
| bug7 | 546 | 319k | 62k | 2159 | 2,549k | 283k | 3,826 | 1,995k | 52k | 40 | 122k | 5k |
| bug8 | 19 | 60k | 0.6k | 7 | 10k | 0.6k | 457 | 387k | 7044 | MO | MO | MO |
| bug9 | 90 | 136k | 11k | 2490 | 2,843k | 323k | 3,069 | 1,713k | 42k | 35 | 71k | 1k |
| bug10 | 313 | 345k | 33k | 39 | 63k | 4k | 4,289 | 2,321k | 57k | 40 | 105k | 5k |

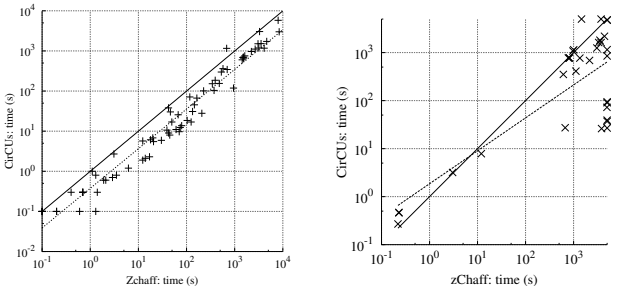


Figure 6: Comparison on BMC benchmark (left) and comparison on SAT2004 competition benchmark (right)

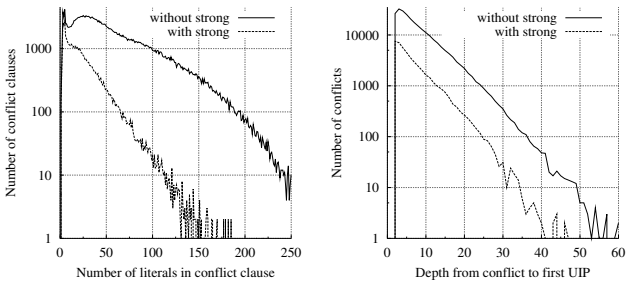


Figure 7: Conflict occurrences versus number of literals (left) and number of conflicts versus implication graph depth (right) for instance “c880”

benchmarks in the SAT’04 competition web page [13]. These experiments also have been performed on 1.7 GHz Pentium IV with 1 GB of RAM running Linux with a 5,000 s timeout. The results are summarized in Figure 6 (right).

We now consider two examples for further analysis: One that shows a large improvement as a result of applying strong conflict analysis (Figure 7), and one that shows only a modest improvement (Figure 8). Figures 7 and 8 show the correlation between the conciseness of the conflict clause and the depth of the implication graphs. In Figure 7 (right), strong conflict analysis reduces the depth of the implication graphs much more than in Figure 8 (right). There is good correlation between the ability to make the implication graphs shallow and the speedup. The reduction in the numbers of literals in conflict clauses are shown in the left parts of Figure 7 and 8. These data support our claim that additional conflict clause and shallower graphs result in fewer literals per clause. We cannot get a significant improvement with the proposed algorithm on the

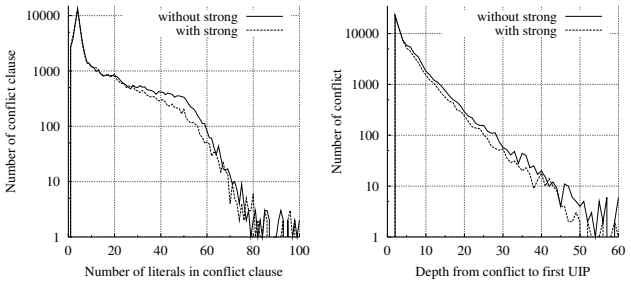


Figure 8: Conflict occurrences versus number of literals (left) and number of conflicts versus implication graph depth (right) for instance “des”

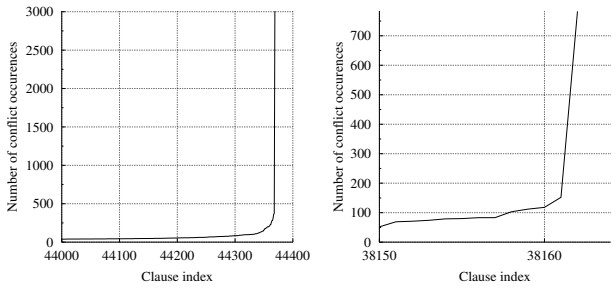


Figure 9: Distributions of conflict occurrences for "c880" (left) and for "des" (right)

"des" benchmark because there is not much room for improving the depth of the implication graphs.

Figure 9 shows the distribution of conflict occurrences for instances "c880"and "des". Only the parts of the graphs where the conflict occurrences exceed 50 are shown, since the full graphs are too large. One can easily identify that only 12 clauses of "des" have more than 50 conflicts as compared to 369 clauses of "c880". As expected, our algorithm is more useful when there is a hot spot.

## 6. Conclusions

We have presented a strong conflict analysis procedure for propositional satisfiability that adds more than one conflict clause from one conflict. We have shown that the additional conflict clauses help reduce the number of literals in conflict clauses as well as the depth of the implication graphs. The experimental results show large improvements compared to the state-of-the-art SAT solvers. Experimental evidence supports the claim that the performance gains are indeed due to the improved conflict analysis.

Several details of the procedure warrant further study. For instance, even though we have devised a criterion to apply the shrinking method based on the occurrence of conflicts on the same clause, the technique remains expensive in term of CPU time, and we need to investigate a more efficient way of applying it.

Random restart techniques have been used to enhance several SAT solvers. We conjecture that deep implication graphs and multiple conflicts on same clause make the search dwell in hot spots. Therefore in some case, restarts help the search out of a hot spot. Our strong conflict analysis should reduce the need for such random restarts.

## References

[1] URL: http://eigold.tripod.com/BerkMin.html.

[2] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Fifth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, pages 193–207, Amsterdam, The Netherlands, Mar. 1999. LNCS 1579.

[3] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.

[4] Z. Fu, Y. Mahajan, and S. Malik. New features of the SAT'04 versions of zChaff. SAT Competition 2004 - Solver Description, May 2004.

[5] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of 11th National Conference on Artificial Intelligence*, 1993. ISBN 0262510715.

[6] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 142–149, Paris, France, Mar. 2002.

[7] H. Jin, M. Awedh, and F. Somenzi. CirCUs: A satisfiability solver geared towards bounded model checking. In R. Alur and D. Peled, editors, *Sixteenth Conference on Computer Aided Verification (CAV'04)*, pages 519–522. Springer-Verlag, Berlin, July 2004. LNCS 3114.

[8] H. Jin and F. Somenzi. CirCUs: A hybrid satisfiability solver. In *International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, Vancouver, Canada, May 2004.

[9] H. Jin and F. Somenzi. An incremental algorithm to check satisfiability for bounded model checking. *Electronic Notes in Theoretical Computer Science*, 2004. Second International Workshop on Bounded Model Checking. http://www.elsevier.nl/locate/entcs/.

[10] M. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535, Las Vegas, NV, June 2001.

[11] A. Nadel. The Jerusat SAT solver. Master's thesis, Hebrew University of Jerusalem, 2002.

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

[13] URL: http://www.lri.fr/∼simon/contest/results.

[14] URL: http://www.cs.chalmers.se/Cs/Research/ FormalMethods/MiniSat/MiniSat.html.

[15] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, oct 1993.

[16] M. Sheeran and G. Stålmark. A tutorial on Stålmark's proof procedure for propositional logic. In G. Gopalakrishnan and P. Windley, editors, *Formal Methods in Computer Aided Design*, pages 82–99. Springer-Verlag, Palo Alto, CA, Nov. 1998. LNCS 1522.

[17] J. P. M. Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portuguese Conference on Artificial Intelligence (EPIA)*, Sept. 1999.

[18] J. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, San Jose, CA, Nov. 1996.

[19] URL: http://vlsi.colorado.edu/∼vis.

[20] URL: http://www.princeton.edu/∼chaff/zchaff/index1.html.

[21] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 272–275, July 1997. LNAI 1249.

[22] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285, San Jose, CA, Nov. 2001.

[23] L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design, Automation and Test in Europe (DATE'03)*, pages 880–885, Munich, Germany, Mar. 2003.