Zchaff2004: An Efficient SAT Solver

Yogesh S. Mahajan, Zhaohui Fu, and Sharad Malik

Princeton University, Princeton, NJ 08544, USA
{yogism, zfu, malik}@Princeton.EDU

Abstract. The Boolean Satisfiability Problem (SAT) is a well known NP-Complete problem. While its complexity remains a source of many interesting questions for theoretical computer scientists, the problem has found many practical applications in recent years. The emergence of efficient SAT solvers which can handle large structured SAT instances has enabled the use of SAT solvers in diverse domains such as electronic design automation and artificial intelligence. These applications continue to motivate the development of faster and more robust SAT solvers. In this paper, we describe the popular SAT solver zchaff with a focus on recent developments.

1 Introduction

Given a propositional logic formula, determining whether there exists a variable assignment that makes the formula evaluate to true is called the Boolean Satisfiability Problem (SAT). SAT was the first problem proven to be NP-Complete[1] and has seen much theoretical interest on this account. Most people believe that it is unlikely that a polynomial time algorithm exists for SAT. However, many large instances of SAT generated from real life problems can be successfully solved by heuristic SAT solvers. For example, SAT solvers find application in AI planning[2], circuit testing[3], software verification[4], microprocessor verification[5], model checking[6], etc. This has motivated research in efficient heuristic SAT solvers.

Consequently, there are many practical algorithms based on various principles such as Resolution[7], Systematic Search[8], Stochastic Local Search[9], Binary Decision Diagrams[10], Stålmarck's[11] algorithm, and others. Gu *et al.*[12] provide a review of many of the algorithms.

Given a SAT instance, SAT algorithms which are complete either find a satisfying variable assignment, or prove that no such solution exists. Stochastic methods, on the other hand, are geared toward finding a satisfiable solution quickly but do not prove unsatisfiability. Stochastic methods are likely to be adopted in AI planning[2] and FPGA routing[13], where instances are likely to be satisfiable and proving unsatisfiability is not required. However, for many other domains, particularly some verification problems[4, 6], the primary task is to prove unsatisfiability of the instances. Hence, complete SAT solvers are required in these cases. The zchaff SAT solver is a complete solver.

H.H. Hoos and D.G. Mitchell (Eds.): SAT 2004, LNCS 3542, pp. 360–375, 2005.

The well known Davis-Logemann-Loveland (DLL)[8] algorithm forms the framework for many successful complete SAT solvers. DLL is sometimes referred to as DPLL for historical reasons. Researchers have been working on DPLL-based SAT solvers since the 1960s. The last ten years have seen tremendous growth and success in DPLL-based SAT solver research. In the mid 1990's, techniques like conflict driven clause learning and non-chronological backtracking were integrated into the DPLL framework[14, 15]. These techniques have greatly improved the efficiency of the DPLL algorithm for structured (as opposed to random) SAT instances. Improvements to the memory efficiency of the Boolean constraint propagation procedure[16, 17] [17] have helped modern SAT solvers cope with large problem sizes. A lot of research has gone into developing new decision strategies. Chaff[17] introduced an innovative conflict clause driven decision strategy and BerkMin[18] introduced yet another decision strategy making use of recent conflict clauses. Today, the latest generation of SAT solvers like zchaff, BerkMin, siege[19], and others[20, 21] are able to handle structured instances with tens of thousands of variables and hundreds of thousands of clauses.

The performance of SAT solvers varies significantly according to the domain from which the problem instance is drawn. The SAT 2004 competition[21] broadly categorizes the instances into instances derived from industrial problems, handmade instances, and randomly generated instances. Solvers that perform well in one category rarely perform well in another category as the techniques that are successful differ from category to category.

Zchaff is a solver that targets the industrial category and hopes to be reasonably successful in the handmade category. It implements the well known Chaff algorithm[17] which includes the innovative VSIDS decision strategy and the very efficient two literal watching scheme for Boolean constraint propagation. Zchaff is a popular solver whose source code is available to the public. It is possible to compile zchaff into a linkable library for easy integration with other applications. Successful integration examples include the BlackBox AI planner[22], NuSMV model checker[23], GrAnDe theorem prover[24], and others. Zchaff compares well with other SAT solvers based on solving runtime performance – versions of zchaff have emerged as the Best Complete Solver in the 'industrial' and 'handmade' instances categories in the SAT 2002 Competition[25] and as the Best Complete Solver in the 'industrial' category in the 2004 SAT Competition[21].

This paper provides an overview of the zchaff solver with a focus on recent developments. Section 2 gives an overview of the DPLL framework on which zchaff is based. Section 3 gives an overview of the main features of the 2003 version of the zchaff solver. Section 4 presents the new features in the SAT 2004 versions of zchaff. Section 5 lists some additional features recently integrated with zchaff after SAT 2004. Section 6 gives some experimental results. Section 7 concludes the paper.

2 The DPLL Algorithm with Learning

In 1960, Davis and Putnam[7] proposed an algorithm for solving SAT which was based on resolution. Their method used resolution for the existential abstrac-

tion of variables from the original instance and produced a series of equivalent SAT decision problems with fewer variables. However, their proposed algorithm had impractically large memory requirements. Davis, Logemann and Loveland[8] proposed an algorithm that used search instead of resolution. This algorithm is often referred to as the DPLL algorithm. It can be argued that these two algorithms are closely related because the DPLL search tree can be used to derive a corresponding resolution proof, but we note that the types of proofs of unsatisfiability that the two methods discover can be different.

In most implementations, the propositional formula is usually presented in a Product of Sums form, which is usually called Conjunctive Normal Form (CNF). There exist polynomial algorithms[26] to transform any propositional formula into a CNF formula that has the same satisfiability as the original one. Henceforth, we will assume that the problem is presented in CNF. A formula in CNF is a conjunction of one or more clauses, where each clause is a constraint formed as the disjunction of one or more literals. A literal, in turn, is a Boolean variable or its negation. A propositional formula in CNF has some nice properties that can help prune the search space and speed up the search process. To satisfy a CNF formula, each clause must be satisfied individually. If a variable assignment causes any clause in the formula to have all its literals evaluate to 0 (false), then that current variable assignment or any extension of it will never satisfy the formula. A clause that has all its literals assigned to value 0 is called a *conflicting clause* and directly indicates to the solver that some of the currently assigned variables must be unassigned first before continuing the search for a satisfying assignment.

DPLL is a depth-first backtracking framework. At each step, the algorithm picks a variable v and assigns a value to v. The formula is simplified by removing the satisfied clauses and eliminating the false literals. An inference rule may then be applied to assign values to some more variables which are implied by the current assignments. If an empty clause results after simplification, the procedure *backtracks* and tries the other value for v. Modern DPLL algorithms have an additional feature – they can learn and remember new clause constraints via a procedure called conflict analysis. The worst case time complexity remains exponential in terms of the total number of variables. However, in the case of some classes of real-life applications, a good implementation shows a manageable time complexity when combined with appropriate heuristics.

An outline of 'DPLL with learning' as it is used in zchaff is given in Fig. 1. Initially, none of the variables of the CNF are assigned a value. The unassigned variable are called free variables. The function `decideNextBranch()` uses some heuristics to choose a free variable v to branch upon and assigns it a value. The assignment operation is said to be a decision made on variable v. The heuristics used here constitute the Decision Strategy of the solver. Each assigned v also has a decision level associated with it which equals the solver decision level at the time the decision was made. The decision level starts at 1 for the first decision and is incremented by 1 for subsequent decisions until a backtrack occurs. After each decision, the function `deduce()` determines some variable assignments which are implied by the current set of decisions. This inference is referred to as

```
while(there exists a free variable)
  decideNextBranch();                // pick & assign free variable
  status = deduce();                 // propagate assigned values
  if(status == CONFLICT)
    blevel = analyzeConflict();      // & learn conflict clause
    if(blevel > 0)
      backTrack(blevel);             // resolve the conflict
    else if(blevel == 0)
      return UNSATISFIABLE;          // conflict cannot be resolved
  runPeriodicFunctions();
}
return SATISFIABLE
```

Fig. 1. Algorithm DPLL with Learning

Boolean Constraint Propagation (BCP). Variables that are assigned during BCP will assume the same decision level as the current decision variable. If `deduce()` detects a conflicting clause during BCP, then the current partial variable assignment cannot be extended to a satisfying assignment, and the solver will have to backtrack. The solver calls the conflict analysis procedure `analyzeConflict()` which finds a reason for the discovered conflict and returns the decision level to backtrack to. The reason for the conflict is obtained as a set of variable assignments which imply the current conflict and gets recorded by the solver as a clause.[1] The solver decision level is updated appropriately after the backtrack. The reader is referred to [27, 28] for details of conflict analysis. The solver enters decision level 0 only when making an assignment that is implied by the CNF formula and a backtrack to level 0 indicates that some variable is implied by the CNF formula to be both true and false i.e. the instance is unsatisfiable. (The function `runPeriodicFunctions()` in the main loop is used to schedule some periodic jobs like clause deletion, restarts, etc.)

The outline in Fig. 1 can be extended to include some simplification procedures - like applying the Pure Literal Rule or identifying equivalence classes[29]. Since these can be expensive to implement dynamically during the search, they may be used in a separate preprocessing phase.

3 Overview of the Zchaff Solver Till 2003

In this section, we will present a quick overview of the main features of the zchaff solver. The overall structure of zchaff is as in Fig. 1.

3.1 Decision Strategy - VSIDS

During the search, making a good choice for which free variable is to be assigned and to what value is very important because even for the same basic algorithm

[1] It is well known that the DPLL algorithm without clause recording can discover tree-like resolution proofs. With the ability to record clauses resulting from conflict analysis, the solver can discover more general proofs of unsatisfiability.

framework, different choices may produce search trees with drastically different sizes. Early branching heuristics like Maximum Occurrences in Minimum Sized clauses (MOMS)[30] used some statistics of the clause database to estimate the effect of branching on a particular variable. In [31], the author proposed literal count heuristics which count the number of *unsatisfied* clauses in which a given variable appears (in either phase). These counts are state-dependent because different variable assignments will give different counts and need to be updated every time a variable is assigned or unassigned.

The Chaff[17] solver proposed the use of a heuristic called Variable State Independent Decaying Sum (VSIDS). VSIDS keeps a score for each literal of a variable. Initially, the literal scores equal the number of occurrences of the literal in the input CNF. The literal counts are updated every time the conflict analysis procedure learns a conflict clause by incrementing the scores of each literal in the learned clause by 1. Periodically, after a fixed large number of decisions, all literal scores are divided by 2. The VSIDS literal scores are effectively weighted occurrence counts with higher weights given to occurrences in recently learned clauses. The score of a variable is considered to be the larger of the two associated literal scores. An ordering of the variables is induced by these scores and when a decision is to be made, VSIDS chooses the free variable highest in the variable order and assigns the variable to true if the score of the positive literal exceeds the score of the negative literal and false otherwise. VSIDS provides a quasi-static variable ordering which focuses the search on the recently derived conflict clause. The statistics required for VSIDS are relatively inexpensive to maintain and this makes it a low overhead decision strategy.

3.2 Boolean Constraint Propagation - Two Literal Watching

During the search for a satisfying assignment, the application of an inference rule can detect some variables whose values are implied by the current set of assignments and simplify the problem remaining to be solved. The Unit Clause Rule is a commonly used inference rule. A Unit Clause is a clause which has exactly one unassigned literal and all other literals assigned to false. The unit clause rule says that the unassigned literal in a unit clause must be assigned to true. This implied assignment is called an implication and the unit clause causing the implication is referred to as the antecedent for that variable assignment.

BCP needs to operate on very large clause databases and the pattern of accesses to the clause database often lacks locality. This leads to a large number of cache misses. BCP often contributes as much as 50-90% to the total runtime of modern solvers[32] and it is imperative to optimize the cache/memory usage of the BCP procedure. Early implementations for BCP like [33] maintained counts for the number of assigned literals in each clause in order to identify unit/conflicting clauses. This was costly to implement. The authors of SATO[16] proposed a mechanism for BCP using head/tail lists[34] which significantly improved the efficiency of BCP. In both the counting-based schemes and the head/tail lists methods, unassigning a variable is a costly operation and its complexity may be comparable to that of assigning a variable.

Zchaff uses the Two Literal Watching scheme[17] for BCP. Initially, two of the non-false literals in each clause are marked as watched literals. Each literal maintains a list of the clauses in which it is watched. Whenever a clause becomes a unit/conflicting clause, at least one of the watched literals in that clause must be assigned to false. Hence, when a literal gets assigned to false, it is sufficient to check for unit/conflicting clauses only among those clauses in which that literal is watched . The details of the mechanism for identifying unit/conflicting clauses can be found in [17]. A key benefit of the two literal watching scheme is that at the time of backtracking, there is no need to modify the watched literals in the clause database. Unassigning a variable can be done very simply by doing nothing more than just setting the variable value to "unknown".

3.3 Conflict Driven Clause Learning and Non-chronological Backtracking - Learning the FirstUIP Conflict Clause

Conflict driven clause learning along with non-chronological backtracking were first incorporated into a SAT solver in GRASP[27] and relsat[15]. These techniques are essential for efficient solving of structured problems.

Conflict Driven Clause Learning: When the BCP procedure detects a conflicting clause that results from the current variable assignments, the solver needs to backtrack. The function `analyzeConflict()` finds a subset of the current variable assignments which is also sufficient to make the analyzed clause a conflicting clause. The solver records this information as a clause which evaluates to true exactly when this subset of variable assignments occurs. This prevents the same conflict from occurring again. New clauses are learned using an operation called resolution.[2] The clauses derived using resolution are implied by the resolvents and thus such clauses are logically redundant and adding these clauses does not affect any of the satisfying assignments. However, these added clauses directly help the BCP procedure to prune some of the search space.

The question of which clauses should be selected for resolution can have many answers. In conflict driven clause learning, the solver's search process discovers sequences of clauses which are suitable to be resolved. As mentioned earlier, each assigned non-decision variable, i.e. implied variable, has an antecedent clause associated with it. The antecedent clause for setting the variable v to 1 will contain the positive literal v and all other literals will be assigned false. The conflicting clause C_f comprises of only false literals. Thus, C_f can be resolved with the antecedent of any of its variables to derive a clause C_l which will also have all false literals. The process can be continued to derive other clauses treating C_l as the conflicting clause. A lot of flexibility remains, e.g. in choosing which variable's antecedent is to be used for resolution, which of the learned clauses are to be actually added to the clause database, and when to stop learning. Zchaff

[2] Conflict driven conflict clause learning can be looked at in two equivalent ways as resulting from successive resolutions and as a cut in the implication graph. We refer the reader to the description in [28].

answers these questions with the FirstUIP[28] clause learning scheme. A single variable assignment at the conflict decision level, which along with all the variable assignments at previous decision levels is sufficient to cause the conflict is a Unique Implication Point (UIP) at the conflict decision level. This provides a *single* reason at the conflict decision level for the current conflict. The most recent UIP variable assignment at the conflict decision level is called the FirstUIP and can always be found since the decision at the conflict decision level is itself an UIP. In the FirstUIP scheme, all the antecedent clauses that appear in the sequence of resolved clauses are antecedents of variables at the conflict decision level and the FirstUIP clause is found when the only literal remaining at the conflict decision level corresponds to the FirstUIP assignment. Details of the procedure may be found in [27] and [28]. Such a conflict clause is called an asserting clause and it will become a unit clause after backtracking.

Non-chronological Backtracking: In order to resolve a conflict, the solver must backtrack to a prior state which does not directly entail the identified conflict i.e. none of the clauses in the database must be conflicting clauses after the backtrack. To do this, the solver finds the second highest decision level involved in the derived conflict clause (decision level 0 if a single literal clause) and unassigns all the variables assigned at decision levels greater than this level. The solver decision level is reset to be the backtrack level. The newly added conflict clause becomes a unit clause because it was a FirstUIP conflict clause and causes an implication via the BCP procedure. The original conflicting clause that was identified (and analyzed) before the backtrack will certainly be non-conflicting after the backtrack as this clause became a conflicting clause only at the very last decision level prior to backtracking.

4 The SAT 2004 Versions of Zchaff

Two new versions of the zchaff solver participated in the SAT 2004 Competition. These two versions are `zchaff.2004.5.13` (submitted as `zchaff`) and `zchaff_rand`. Both `zchaff.2004.5.13` and `zchaff_rand` can be downloaded from `http://www.princeton.edu/~chaff/SAT2004_versions.html`. The two versions are closely related and we will use the term zchaff2004 to refer to both of them. During its development, many features from `zchaff_rand` were integrated into `zchaff.2004.5.13`. Some features like the "shrinking" decision heuristic are implemented differently in `zchaff.2004.5.13` and `zchaff_rand`. While there are many differences between them, the solvers are comparable in performance. In the SAT 2004 competition, `zchaff.2004.5.13` was more successful on satisfiable instances while `zchaff_rand` appeared to be more successful on unsatisfiable instances. We have found that the performance of zchaff2004 compared to the 2003 version is slightly worse for bounded model checking, but better for microprocessor verification problems.

Many of the new features have a common theme of increased search locality and the derivation of short conflict clauses. Other researchers, e.g. the author of

Siege[19], have noted the interaction between the search heuristic and the length of the learned clauses. Like BerkMin, zchaff2004 also uses frequent restarts and an aggressive clause deletion policy. Zchaff2004 also features some heuristics whose parameters are dynamically adjusted during the search. Techniques like the VSIDS decision strategy, Two Literal Watching based BCP, FirstUIP based conflict clause learning and non-chronological backtracking which have proved to be useful in earlier versions of zchaff are retained in zchaff2004.

4.1 Increased Search Locality

When VSIDS was first proposed, it turned out to be very successful in increasing the locality of the search by focusing on the recent conflicts. This was observed to lead to faster solving times. Though VSIDS scores are biased toward recent regions of the search by the decaying of the scores, the decisions made are still *global* in nature, due to the slow decay of variable scores[18]. However, recent experiments show that branching within greater locality helps dramatically to prune the search space. SAT solvers BerkMin and siege have both exhibited great speedups from such decision heuristics. Zchaff2004 has three decision heuristics. The first one to be tried is a "shrinking" heuristic. If this is not currently active and does not make a decision, then a modified BerkMin like decision heuristic is tried. The more global VSIDS decision strategy is used last.

Variable Ordering Scheme for VSIDS: This is the default decision heuristic for zchaff2004. One way of trying to make VSIDS more local is to increase the frequency of score decay. The variable ordering scheme also differs from the previous version by incrementing the scores of the literals which get resolved out during conflict analysis. Zchaff2004 increments the scores of involved literals by 10000 instead of by 1. As a result, the decaying scores remain non-zero for longer. Due to the details of the implementation of variable ordering in zchaff, incrementing scores by 10000 also has the side effect that the variable order is no longer the same as given by the variable scores, and the active variables move closer to the top of the variable order. In `zchaff_rand`, the VSIDS scores are reset to new initial values determined by the literal occurrence statistics in the current clause database after every clause deletion phase.

BerkMin Type Decision Heuristic: The use of the most recent unsatisfied conflict clauses as is done by BerkMin also turns out to be a good cost-effective approach to estimate the locality. The main ideas of this approach are described by the authors of BerkMin in [18]. In zchaff2004, we maintain a chronological list of derived conflict clauses. An unassigned variable with the highest VSIDS score in a recent unsatisfied conflict clause is chosen to be branched upon. As in VSIDS, the variable is assigned to true if the score of the positive literal exceeds the score of the negative literal and false otherwise. In `zchaff.2004.5.13`, the most recent unsatisfied clause is identified exactly. In `zchaff_rand`, after searching through a certain threshold number (set to 10000 or randomly) of as yet unexamined conflict clauses, the solver defaults to the VSIDS decision heuristic in case it fails to find an unsatisfied clause. Also, `zchaff_rand` skips conflict clauses which have all unassigned literals during the search for a recent unsatisfied conflict clause.

Conflict Clause Based Assignment Stack Shrinking: This is related to one of the techniques used by the Jerusat solver[35]. We use our modification of the general idea as presented in [35]. When the newly learned FirstUIP clause exceeds a certain length L, we use it to drive the decision strategy as follows. We sort the decision levels of the literals of the FirstUIP clause and then examine the sorted sequence of decision levels to find the lowest decision level that is less than the next higher decision level by at least 2. (If no such decision level is found, then shrinking is not performed.) We then backtrack to this decision level, and the decision strategy starts *re-assigning* to false the unassigned literals of the conflict clause till a conflict is encountered again. We found that reassigning the variables in the reverse order, i.e. in descending order of decision levels (used in `zchaff_rand`), performed slightly better than reassigning the variables in the same order as they were assigned in previously (used in `zchaff.2004.5.13`). Since some of the variables that were unassigned during the backtrack may not get reassigned, the size of the assignment stack is likely to reduce after this operation. As the variables on the assignment stack are precisely those that can appear in derived conflict clauses, new conflict clauses are expected to be shorter and more likely to share common literals. In our experiments, no fixed value for L performed well for the range of benchmarks we tested. Instead, we set L dynamically using some measured statistics. Zchaff2004 has two such metrics. The first metric, used in `zchaff.2004.5.13`, is the averaged difference between lengths of the clause being used for shrinking and the immediate new clause we get after the shrinking. If this average is less than some threshold, L is increased to reduce the amount of shrinking and if L exceeds some threshold, L is decreased to encourage more shrinking. `zchaff_rand` measures the mean and the standard deviation of the lengths of the recent learned conflict clauses and tries to adjust L to keep it at a value greater than the mean. This dynamic decision heuristic of conflict clause based assignment stack shrinking is observed to often reduce the average length of learned conflict clauses and leads to faster solving times, especially for the microprocessor verification benchmarks.

4.2 Learning Shorter Clauses

Short clauses potentially prune large spaces from the search. They lead to faster BCP and quicker conflict detection. Conflict driven learning derives new (conflict) clauses by successively resolving the clauses involved in the current conflict. The newly derived clause is small in size when the number of resolvents is small, when the resolvents are short clauses themselves, or when the resolvents share many literals in common. Zchaff2004 has the following strategies to try to derive short conflict clauses.

Short Antecedent Clauses Are Preferred: When the clauses do not share many common literals, the sum of the lengths of all the involved clauses will determine the length of the learned conflict clause. We can directly influence the choice of clauses for the resolution by preferring shorter antecedent clauses. One way to do this is to update a variable's antecedent clause with a shorter

one whenever possible. As implemented in zchaff2004, BCP queues the implied variable values along with their antecedents but does not perform the assignment immediately. The assignment occurs only when the implied value is dequeued and propagated. Thus, it sometimes happens that the same variable is enqueued multiple times with the same value but different antecedent clauses. When BCP encounters a new antecedent clause for an already assigned variable, the previous antecedent can be replaced with the new one if the new antecedent is shorter. `zchaff_rand` maintains a separate database for binary clauses [36] and processes the binary clauses before the non-binary clauses during BCP.

Multiple Conflict Analysis: This is a more costly technique than replacing antecedents. It is observed that BCP often discovers more than one conflicting clauses (most of which are derived from some common resolvents). For each conflicting clause, zchaff2004 finds the length of the FirstUIP clause to be learned, and only records the one with the shortest length. Variables that are assigned at decision level zero are excluded from all the learned conflict clauses.

Interaction with Decision Strategy: When the clauses being resolved during conflict analysis share many common literals, the resulting conflict clause is likely to be short. There is a strong interaction between the learned clauses and a "locality centric" decision strategy. For example, the shrinking strategy reduces the size of the set of literals that can appear in new conflict clauses. This in turn increases the likelihood that the new clauses that are learned during the search are shorter and share more literals. Decision strategies like VSIDS and BerkMin which focus on recent conflict clauses can then discover which of these new clauses are suitable for resolution, and the resulting clause is again likely to be short. The observation that the decision strategy influences the length of the derived conflict clauses has been made by the author of siege [19] who considers conflict driven clause learning to be primarily a resolution strategy.

Learning Intermediate Resolvents: While performing conflict analysis, the solver remembers the result of the first 5 resolutions. If this intermediate resolution result is shorter than the recorded FirstUIP clause, then the intermediate resolvent is also recorded after the FirstUIP clause is recorded. This is implemented in `zchaff_rand`.

4.3 Aggressive Clause Deletion

Learned conflict clauses slow down the BCP procedure and use up memory. Clauses which are not useful must be deleted periodically in the interest of keeping the clause database small. Clauses satisfied at decision level 0 can be deleted as they no longer prune any search space. As in BerkMin, some learned conflict clauses can be deleted periodically without affecting the correctness of the solver. Zchaff2004 periodically deletes learned clauses using usage statistics and clause lengths to estimate the usefulness of a clause. Each clause has an activity counter which is incremented every time the clause is involved in the derivation of a new learned clause. This counter is used by Zchaff2004 to calculate an approximation

to the clause's activity to age ratio. Any clause with this ratio less than a certain threshold is considered for deletion. The final decision to delete the clause is then made based on the irrelevance of the clause which is estimated by the number of unassigned literals in the clause. The clause is deleted only if its irrelevance exceeds a certain irrelevance threshold. The irrelevance threshold may be a constant or may be set dynamically based on the measurements of observed clause length statistics. `zchaff_rand` uses max{L,45} for the irrelevance parameter where L is the length parameter used by the dynamic shrinking decision strategy. In `zchaff_rand`, the clause activities are also decremented periodically by a very small amount.

4.4 Frequent Restarts

Luck plays an important role in determining the solving time of a SAT solver even for the case of unsatisfiable instances. The order in which the BCP procedure queues implications and the order in which variables get watched are determined more or less arbitrarily via the order in the CNF input file. Consequently, the same CNF formula can take widely different run times after shuffling the clauses and variables. When a VSIDS decision is made with all unassigned variables having score 0, zchaff arbitrarily picks the first variable in the list. The wide distribution of run times for slightly different algorithms running on the same instance has been noted in [37] and the authors point out that a rapid restart policy of a randomized solver can help reduce the variance of run times and thereby contribute to increasing the robustness of the solver. Zchaff2004 also uses a rapid fixed interval restart policy. The frequent restarts are observed to make the solver more robust. With restarts disabled, `zchaff_rand` with a timeout of 300 seconds and random seed 0 takes 688 seconds on the beijing benchmark suite (16 instances) and leaves two instances unsolved. With restarts enabled, all the 16 instances get solved within 65 seconds.

5 Recent Developments

In this section, we briefly mention some of the new features have been added to zchaff2004 after it was submitted to the SAT 2004 competition. One of the motivations was to make BCP more efficient.

5.1 Early Conflict Detection

Early conflict detection is a technique used by solvers like Limmat[38]. During BCP, the variable assignment is completed at the same time that the implied value is queued. This has the advantage that conflicting values in the implications queue can be identified early - as soon as they occur. Another advantage of this is that the implied values still in the queue are already known to the Boolean constraint propagation procedure and this could help BCP by not watching literals which are set false according to the implication queue. This technique has mixed effects on the solver run times. It may be noted that replacing the

antecedent clauses becomes more complicated when early conflict detection is enabled, since extra checks have to be performed to ensure that no cycle is introduced into the current implication graph. In particular, we check that all the false literals were assigned before the single true literal got assigned.

5.2 Reorganized Variable Data Structure

During the addition of the new features, the variable object had grown in size to about a hundred bytes. All the variable objects are stored in a `STL::vector<>` as a result of which the actual variable values were widely separated in memory. Since BCP mainly needs just the values, all the variable values were put into a `vector<char>` by themselves. Other fields like the watched literal lists, variable scores, implication related data, etc. were put into `vector<>`'s of their own. This reorganized variable data structure brings small but consistent speedups.

5.3 Miscellaneous Features

The features listed here are considered to be experimental in status. The first one is a modification to the BerkMin heuristic which uses short satisfied clauses on the conflict clause stack which have less than 4 true literals and length less than 10 to make decisions. An unassigned literal from such a clause is selected and set to false. The motivation is to recreate the assignments at the time the short clause was derived. With this strategy, the performance on satisfiable benchmarks improved for the tested benchmarks and no serious disadvantages were noticed for unsatisfiable instances. The second modification is to increment the literal scores by the number of conflicting clauses analyzed for the current conflict. When multiple clauses are analyzed, they share many common resolvents and have similar literals. Hence, incrementing the score by the number of discovered conflicting clauses gives more importance to literals which are frequently involved in deriving conflict clauses. Secondly, incrementing by more than 1 will also move such literals to the top of the variable ordering.

6 Impact of New Features

In this section, we try to evaluate the impact of the various modifications made to the zchaff solver. To do this, we have created a series of versions of zchaff starting with a version similar to the 2003 version and then adding features eventually ending with a recent development version of zchaff2004. While we do not explore all the possible combinations of the the features, we hope these comparisons will yield some insight into the usefulness of the features. All experiments were run on identical machines having Pentium 4 2.80 GHz processors with 1 MB L2 cache and 1 GB RAM using a random seed equal to 0.

6.1 Experimental Results

First, we present some details about the various versions that appear in the comparisons in Tables 1 and 2. The version `'base'` is similar to the 2003 version of

Table 1. Comparisons of various configurations of zchaff

Benchmark	`base`	`s`	`sMR`	`sMRB`	`SMRBK`	`SMrBKEV`
01_rule(20)	11477(11)	8817(7)	6970(6)	8810(9)	9641(9)	9467(7)
07_rule(20)	11755(12)	7656(8)	14481(11)	200(0)	103(0)	99(0)
barrel(8)	99(0)	89(0)	84(0)	227(0)	235(0)	256(0)
beijing(16)	2042(2)	2813(2)	1934(2)	79(0)	56(0)	57(0)
ferry(10)	17(0)	10(0)	8(0)	4(0)	3(0)	7(0)
fifo(4)	1978(1)	1269(1)	1229(1)	1586(1)	2367(2)	2378(2)
fpga_routing(32)	1091(1)	993(1)	967(1)	254(0)	1228(1)	1315(1)
fvp-sat.3.0(20)	9216(7)	7109(4)	8500(6)	9988(9)	10149(7)	8770(1)
fvp-unsat.2.0(24)	3253(2)	3017(2)	3859(3)	1179(0)	456(0)	402(0)
hanoi(5)	1678(1)	1037(1)	1088(1)	75(0)	215(0)	143(0)
hard_eq_check(16)	12958(14)	12855(14)	12833(13)	11307(10)	9813(9)	9798(9)
ip(4)	2087(2)	1818(0)	601(0)	236(0)	279(0)	730(0)
total(179)	57653(53)	47485(40)	52554(44)	33945(29)	35066(28)	33422(20)

zchaff. The version `'s'` is obtained by modifying `'base'` to also score the literals which get resolved out during conflict analysis. This scoring is similar to what is done in the BerkMin solver. Literals which are involved in the recorded conflict clause and which get resolved out during conflict analysis have their score incremented by 1. The version `'sMR'` adds multiple conflict analysis (Sect. 4.2) to `'s'` and replaces antecedent clauses (Sect. 4.2). The version `'sMRB'` adds BerkMin like heuristics like the decision strategy (Sect. 4.1), aggressive clause deletion (Sect. 4.3) and frequent restarts (Sect. 4.4). In this version, scores are decayed every 20 backtracks. The solver maintains activities for the clauses (Sect. 4.3) and clauses which survive deletion for many iterations have their activities increased by a small amount. This version also includes an experimental decision strategy modification (Sect. 5.3). The version `'SMRBK'` adds the dynamic shrinking decision strategy as in `zchaff_rand` (Sect. 4.1) and clause deletion using the shrinking length parameter L to estimate irrelevance (Sect. 4.3). `SMRBK` also increments the scores by the number of conflicting clauses analyzed (Sect. 5.3) and resets the VSIDS scores after each restart based on the current literal occurrence counts. The version `'SMrBKEV'` adds early conflict detection (Sect. 5.1) to `SMRBK` and has a cleaned up variable data structure (Sect. 5.2). Due to the overhead of the extra checks required before replacing antecedents when using early conflict detection, only the antecedents of binary clauses are replaced when early conflict detection is used. The version `SBKEV` is derived from `SMrBKEV` by disabling multiple conflict analysis and antecedent-replacement. The version `SMrBEV` is derived by disabling dynamic shrinking from `SMrBKEV`.

Table 1 reports the total solving time in seconds for the various versions on benchmarks spanning microprocessor verification, bounded model checking, fpga routing, etc. The random seed was 0 for all runs and the timeout was 900 seconds per instance. The numbers in parentheses give the number of instances in the benchmark suite and also the number that remained unsolved. We see that

Table 2. More comparisons on the same benchmarks

Benchmark	SMrBKEV	SBKEV	SMrBEV	zchaff.2004.5.13	zchaff_rand
01_rule(20)	9467(7)	8776(8)	9448(8)	8759(9)	9915(9)
07_rule(20)	99(0)	99(0)	174(0)	111(0)	121(0)
barrel(8)	256(0)	303(0)	349(0)	162(0)	68(0)
beijing(16)	57(0)	63(0)	61(0)	52(0)	65(0)
ferry(10)	7(0)	3(0)	3(0)	7(0)	2(0)
fifo(4)	2378(2)	2184(2)	2069(2)	1669(1)	1765(1)
fpga_routing(32)	1315(1)	1356(1)	765(0)	1102(1)	516(0)
fvp-sat.3.0(20)	8770(1)	9361(5)	9737(10)	6432(4)	7171(1)
fvp-unsat.2.0(24)	402(0)	365(0)	1753(0)	853(0)	702(0)
hanoi(5)	143(0)	516(0)	174(0)	1180(1)	764(0)
hard_eq_check(16)	9798(9)	10302(10)	12218(12)	12095(12)	9877(9)
ip(4)	730(0)	606(0)	588(0)	1146(0)	214(0)
total(179)	33422(20)	33935(26)	37340(32)	33568(28)	31180(20)

Table 3. Effect of dynamic shrinking on microprocessor verification benchmarks

Benchmark	SMrBEV	SMrBKEV
fvp-unsat.1.0(4)	65(0)	29(0)
fvp-unsat.2.0(24)	1753(0)	402(0)
engine-unsat.1.0(10)	8182(4)	8087(4)
pipe-unsat.1.1(14)	17854(12)	10168(4)
fvp-sat.3.0(20)	17550(7)	9010(0)
total(68)	45404(23)	27696(8)

multiple conflict analysis and replacing the antecedents do not have a significant effect on the solver run times by themselves. The BerkMin like decision heuristics produce a definite improvement. Adding dynamic shrinking has a mixed effect for these benchmarks. Adding early conflict detection and reorganizing the variable data structure gives a small improvement.

Table 2 shows the effect of disabling multiple conflict analysis and the effect of disabling shrinking from the final version. Performance is degraded in both cases. Table 2 also compares `zchaff.2004.5.13` with `zchaff_rand`.

As remarked earlier, the dynamic shrinking strategy has the most significant effect for the microprocessor verification benchmarks. We present the results without and with dynamic shrinking for some microprocessor verification benchmarks in Table 3. The timeout for these experiments was 1800 seconds.

7 Summary

We have presented some details of the 2004 versions of zchaff including the versions that participated in the SAT 2004 Competition. The new features include

decision strategies that increase the "locality" of the search and focus more on recent conflicts, strategies that directly focus on deriving short conflict clauses, an aggressive clause deletion heuristic that keeps only the clauses most likely to be useful, a rapid restart policy that adds robustness and some techniques that are intended to improve the efficiency of the Boolean constraint propagation procedure. We have also presented some data that might help in evaluating the usefulness of the various features.

References

1. Cook, S.A.: The complexity of theorem-proving procedures. In: Third Annual ACM Symposium on Theory of Computing. (1971)
2. Kautz, H., Selman, B.: Planning as Satisfiability. In: European Conference on Artificial Intelligence. (1992)
3. Stephan, P., Brayton, R., Sangiovanni-Vencentelli, A.: Combinational test generation using satisfiability. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **15** (1996) 1167–1176
4. Jackson, D., Vaziri, M.: Finding bugs with a constraint solver. In: International Symposium on Software Testing and Analysis, Portland, OR. (2000)
5. Velev, M.N., Bryant, R.E.: Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW. In: 38th DAC, New York, NY, USA, ACM Press (2001) 226–231
6. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Proc. of TACAS. (1999)
7. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of ACM **7** (1960) 201–215
8. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM **5** (1962) 394–397
9. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability. (1993)
10. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35** (1962) 394–397
11. Gunnar Stålmarck: System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from Boolean Formula (1994) United States Patent. Patent Number 5,276,897.
12. Gu, J., Purdom, P.W., Franco, J., Wah, B.W.: Algorithms for the Satisfiability (SAT) Problem: A Survey. DIMACS Series in Discrete Mathematics and Theoretical Computer Science (1997)
13. Nam, G.J., Sakallah, K.A., Rutenbar, R.A.: Satisfiability-Based Layout Revisited: Detailed Routing of Complex FPGAs Via Search-Based Boolean SAT. In: ACM/SIGDA International Symposium on FPGAs. (1999)
14. Marques-Silva, J.P., Sakallah, K.A.: Conflict Analysis in Search Algorithms for Propositional Satisfiability. In: IEEE International Conference on Tools with Artificial Intelligence. (1996)
15. Bayardo, R., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: National Conference on Artificial Intelligence (AAAI). (1997)
16. Zhang, H.: SATO: An efficient propositional prover. In: International Conference on Automated Deduction. (1997)

17. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: 38th DAC. (2001)
18. Goldberg, E., Novikov, Y.: BerkMin: A Fast and Robust SAT Solver. In: DATE. (2002)
19. Siege Satisfiability Solver, http://www.cs.sfu.ca/~loryan/personal/ (2004).
20. SAT Competition 2003, http://www.satlive.org/SATCompetition/2003/ (2004).
21. SAT Competition 2004, http://www.satlive.org/SATCompetition/2004/ (2004).
22. http://www.cs.washington.edu/homes/kautz/satplan/blackbox/ (2004).
23. NuSMW Home Page, http://nusmv.irst.itc.it/ (2004).
24. GrAnDe, http://www.cs.miami.edu/~tptp/ATPSystems/GrAnDe/ (2004).
25. SAT Competition 2002, http://www.satlive.org/SATCompetition/2002/ (2004).
26. Plaisted, D.A., Greenbaum, S.: A stucture-preserving clause form translation. Journal of Symbolic Computation **2** (1986) 293–304
27. Marques-Silva, J.P., Sakallah, K.A.: GRASP - A New Search Algorithm for Satisfiability. In: IEEE International Conf. on Tools with Artificial Intelligence. (1996)
28. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD. (2001) 279–285
29. Li, C.M.: Integrating Equivalency reasoning into Davis-Putnam procedure. In: AAAI. (2000)
30. Freeman, J.W.: Improvements to propositional satisfiability search algorithms. PhD thesis, University of Pennsylvania (1995)
31. Marques-Silva, J.P.: The impact of branching heuristics in propositional satisfiability algorithms. In: 9th Portuguese Conf. on Artificial Intelligence. (1999)
32. Zhang, L.: Searching for Truth: Techniques for Satisfiability of Boolean Formulas. PhD thesis, Princeton University (2003)
33. Crawford, J., Auton, L.: Experimental results on the cross-over point in satisfiability problems. In: National Conf. on Artificial Intelligence (AAAI). (1993)
34. Zhang, H., Stickel, M.: An efficient algorithm for unit-propagation. In: Fourth International Symposium on Artificial Intelligence and Mathematics, Florida. (1996)
35. Nadel, A.: The Jerusat SAT Solver. Master's thesis, Hebrew University of Jerusalem (2002)
36. Pilarski, S., Hu, G.: Speeding up SAT for EDA. In: DATE. (2002)
37. Kautz, H., Horvitz, E., Ruan, Y., Gomes, C., Selman, B.: Dynamic restart policies. In: The 18th National Conf. on Artificial Intelligence. (2002)
38. http://www2.inf.ethz.ch/personal/biere/projects/limmat/ (2004).