`zChaff`: Modifications and Extensions

Marc Herbstritt

Institute of Computer Science,
Albert–Ludwigs–University,
Goerges-Koehler Allee 051,
79110 Freiburg im Breisgau, Germany
herbstri@informatik.uni-freiburg.de

Technical Report No. 188

July 17, 2003

ABSTRACT

The application of SAT solver in industrial verification tools is mandatory for the successful analysis of problems derived from hardware design, e.g. Combinational Equivalence Checking (CEC) and Bounded Model Checking (BMC). In [8] Chaff*, a new powerful SAT solver, was presented that outperforms most of the existing publicly available SAT solvers when applied to hard real-world problems, e.g. from VLSI CAD. There are two versions available of Chaff:* `mChaff` *and* `zChaff`*. Both are instances of Chaff, but from the viewpoint of implementation they are totally different since they were developped independently by M. Moskewicz (*`mChaff`*) and L. Zhang (*`zChaff`*). In this paper we concentrate on* `zChaff`*. We analyze the core issues and propose modifications and extensions.*

1. INTRODUCTION

The SAT problem is asking if a propositional boolean formula is satisfiable or not. Propositional formulas are built from literals, and clauses. A boolean *variable* x can take values from the set $\{0,1\}$. A *literal* is either the variable x itself or its complemented form $\neg x$. x is then called *positive literal* and $\neg x$ is called *negative literal*. A clause c is a conjunction of pairwise disjoint literals: $c = (l_1 \vee l_2 \vee \ldots \vee l_n)$ and can be defined as a set of literals: $c = \{l_1, l_2, \ldots, l_n)$. A *conjunctive normal form* (CNF) is a disjunction of clauses: $f = (c_1 \wedge c_2 \wedge \ldots \wedge c_m)$ and can also be seen as a set of clauses: $f = \{c_1, c_2, \ldots, c_m\}$. Finally, a CNF for any boolean function f can be written as

$$\{\{l_{1,1}, \ldots, l_{1,i_1}\}, \{l_{2,1}, \ldots, l_{2,i_2}\}, \ldots \{l_{m,1}, \ldots, l_{m,i_m}\}\},$$

where all $l_{i,j}$ are literals of variables $x_1, \ldots, x_n$.

Since we are interested in satisfiable solutions for the SAT problem, we make the following definitions. Let $\varphi : \mathbf{N} \rightarrow \{0,1\}^n$ an assignment to all n boolean variables $x_1, \ldots, x_n$ and let $\varphi_i := \varphi_{|i}$ the projection of φ to the i-th component of φ.

- □ A literal l is *satisfied* iff ($l \equiv x_i$ and $\varphi_i(x_i) = 1$) or ($l \equiv \neg x_i$ and $\varphi_i(x_i) = 0$).
- □ A literal l is *unsatisfied* iff ($l \equiv x_i$ and $\varphi_i(x_i) = 0$) or ($l \equiv \neg x_i$ and $\varphi_i(x_i) = 1$).
- □ A clause $c = \{l_1, \ldots, l_k\}$ is *satisfied* iff at least one literal $l \in c$ is satisfied under φ.
- □ The empty clause contains no literals and is denoted by □ and is always unsatisfied.
- □ A clause $c = \{l_1, \ldots, l_k\}$ is *unsatisfied* iff all literals in c are unsatisfied under φ.
- □ A CNF $f = \{c_1, \ldots, c_m\}$ is *satisfied* iff all clauses are satisfied under φ.

A clause $c \in f$ is called *unresolved* if it is neither satisfied nor unsatisfied under φ. A clause $c = \{l\}$, i.e. containing only one literal, is called *unit clause*. It is clear that for any satisfying assignment φ it must hold: $\varphi_i(x_i) = 1$ if $l \equiv x_i$ and $\varphi_i(x_i) = 0$ if $l \equiv \neg x_i$.
If there exists no satisfying assignment φ such that f is satisfied under φ, f is called *unsatisfiable*.

2. DAVIS-PUTNAM PROCEDURE AND EXTENSIONS

The most popular approaches for solving SAT are based on the Davis-Putnam procedure [2, 3] sketched in Figure 1.

There have been some successful extensions to the basic DP-procedure:

- □ A lot of *branching rules* were proposed [5, 7, 4], each aiming at accelerating the search under different

This work was done while the author was with Siemens AG during August/September 2001.

```
Davis-Putnam( CNF f ) {
  if f is empty return SATISFIABLE
  if □ ∈ f return UNSATISFIABLE
  /* unit propagation */
  if ∃c ∈ f with c = {l})
    then satisfy l and simplify f to f'
         return Davis-Putnam( f' )
    else
         /* branching rule */
         select unassigned variable x_i and an
         assignment φ_i(x_i) = b;
         simplify f to f';
         if Davis-Putnam( f' ) == SATISFIABLE
           return SATISFIABLE
         else
           change assignment of x_i to φ_i(x) = ¬b;
           simplify f to f'';
           return Davis-Putnam( f'' )
}
```

Figure 1: Davis-Putnam procedure.

preferences (e.g. eliminating small clauses, weighting variables that occur often as positive as well as negative literal, ...). A good overview and a comparison gives [6]. Most branching rules are *variable state dependent*, i.e. statistics on the occurence of unassigned variables are used whereby the underlying clause database is assumed to be simplified using the already assigned variables.

- *Conflict analysis* is used to analyze contradictions occuring during the search. E.g. an assignment is chosen that makes a clause unresolved; this directly transfers to the case where an assignment is implied that is complementary to an assignment already made. The concept of conflict analysis was first applied by `relsat` [1] and is used in a more detailed fashion in GRASP [7]. As a result of the conflict analysis one or more additional clauses are learned that *explain* the conflict. This issue is called *Conflict Learning*. Learning clauses from a conflict prevents the algorithm in the future to search parts of the search space that will lead to the same conflict.
- *Non-chronological backtracking* is an issue closely related to conflict analysis. In the original DP-procedure, when testing variable assignment $x = c$ fails, this assignment is automatically *flipped*, i.e. $x = \neg c$ is assigned. But when a conflict occured and a clause c is learned it will often still be unresolved when flipping the last decision assignment. Instead we can identify the last assignment contained in the learned clause and backtrack to this assignment. Thereby, several decision levels can be skipped, leading to a *non-chronological*[1] backtrack behaviour of the SAT algorithm.
- *Restarts* conceptually abort the search after exceeding some parameter treshold, and starts the search from scratch. There are several possibilities for the parameter treshold: the number of backtracks, the number of decisions, ... Restarts prevent the SAT algorithm to lose in non-relevant parts of the search space. Conflict learning affects the concept of restarts in that new clauses are added during two restarts and thus modifiying the clause database. In general, branching rules will then behave different, leading to a varying exploration of the search space.

3. ZCHAFF

`zChaff` incorporates the extensions mentioned in Section 2 and is sketched in Figure 2.

```
zChaff() {
  while ( 1 ) {
    if( decide_next_branch() ) {
      while ( deduce() == CONFLICT ) {
        blevel = analyze_conflicts()
        if( blevel == 0 ) {
          return UNSATISFIABLE
        }
        else {
          back_track( blevel )
        }
      }
    }
    else {
      return SATISFIABLE
    }
  }
```

Figure 2: `zChaff` (without Restarts).

We briefly describe the functions used in `zChaff`.

- *decide_next_branch*: This function implements the branching rule. Each variable x maintains *literal count* counters for counting the occurences of x and $\neg x$ in clauses contained in the clause database. Additionally, *scores* are maintained for x and $\neg x$. These scores are based on the values of its corresponding literal count counters. All variables are kept in decreasing order where the function

$$maxscore(x) = \max(score(x), score(\neg x))$$

is used as ordering relation. Due to conflict learning the literal count counters can be increased. `zChaff`

[1] Sometimes this issue is called *Conflict-Directed Backjumping* (CBJ) [9]

performs updates of the scores from time to time. Let x be a variable, $score_{old}(x)$, $score_{old}(\neg x)$ its actual score values, and let $increase(x)$, $increase(\neg x)$ be the number of literal count increments of x and $\neg x$ since the last update. Then the new score is computed as:

$$\begin{aligned} score_{new}(x) &= score_{old}(x)/2 + increase(x) \\ score_{new}(\neg x) &= score_{old}(\neg x)/2 + increase(\neg x) \end{aligned}$$

For getting a new branching variable and an initial assignment to this variable, the variable x with maximum value $maxscore(x)$ is selected and it is assigned $x = 1$ if $score(x) \geq score(\neg x)$ and $x = 0$ otherwise (it is clear that only unassigned variables are considered).

Since the literal counts are only increased when adding a conflict clause to the clause database, this branching rule directly makes preferences to variables occuring in learned conflict clauses. In addition, the scores of variables that never or very seldom occur in conflict clauses are decreased due to the division by 2.

Note that in contrast to traditional branching rules this one is not dependent on the variable state, i.e. the literal counts are always based on the non-simplified clause database (including learned conflict clauses). Therefore the branching rule is called *Variable State Independent Decaying Sum* (VSIDS).

Each decision assignment a *decision level* is assigned which is also used to store the decision assignment in an *assignment stack*.

- *deduce*: This function incorporates the *unit propagation rule* in an iterated manner, and this process is commonly called *Boolean Constraint Propagation* (BCP). The underlying data structure is a queue storing all implications (i.e. unit propagations). For the ease of implementation, decision assignments made by *decide_next_branch* are also put into the implication queue. The main function of *deduce* is to iteratively pop an implication from the implication queue and check all unsatisfied clauses if they are now satisfied or if they are becoming a unit clause. In the latter case, the unit literal of the unit clause is pushed into the implication queue as a new implication together with its origin clause, which is needed in conflict analysis as the reason for this implication. In addition, this implication is assigned a decision level, namely that from the last decision assignment. All the implication assignments for a decision level are stored in the assignment stack (in a list attached to the decision assignment of this decision level). If a conflict occurs, i.e. a clause is identfied to be unresolved w.r.t. to the actual handled implication, all unresolved clauses are collected to allow conflict analysis based on the unresolved clauses. Unresolved clauses occuring during the search are also called *Conflicting Clauses*. The function returns `NO_CONFLICT` if no conflict occured, otherwise it returns `CONFLICT`.
- *analyze_conflicts*: This function analyzes the conflict occured in the *deduce* function. A conflict occurs when a contradictory variable assignment is deduced. E.g. variable x was already assigned $x = 1$, and within *deduce* a unit clause $c = \{\neg x\}$ is identified, thus concluding the implication $x = 0$ which is a contradiction to $x = 1$. Therefore c is a conflicting clause. In combination with the assignment stack and the stored reasons for all implications on the assignment stack, it is possible to make a traversal of the implication graph leading to this conflict. In [10] several schemes for conflict analysis are analyzed. Most schemes are based on the concept of *Unique Implication Points* (UIPs), which are dominators in the implication graph. The best results are obtained by the *1UIP-Scheme* (1UIP), i.e. the traversal is stopped when the first UIP on the last decision level is hit. 1UIP is also applied in `zChaff` and learns exactly one clause. Finally, from the learned conflict clause a backtracking level is computed.
- *back_track*: After conflict analysis, we need to backtrack to the backtracking level b computed from *analyze_conflicts*. Thus, all implications and decision assignments with decision level $r \geq b$ are reset. If in conflict clause c only one literal has as its decision level the backtracking level, this literal is directly implied. Note that this implication can trigger new implications by the *deduce* function.

  A backtracking level 0 indicates that the CNF formula is unsatisfiable.

In the top level search procedure as depicted in Figure 2 before each call to *decide_next_branch* it is periodically checked wether the variables scores must be updated and wether a restart should be started. The variable score update is done every 255 decisions (i.e. calls of *decide_next_branch*). To decide wether a restart should take place, some parameter tresholds are tested that are based on the number of backtracks. To guarantee completeness of `zChaff`, these tresholds are incremented after each restart.

4. MODIFICATIONS AND EXTENSIONS

We now propose some modifications and extensions to `zChaff`.

4.1. Variable Ordering Extension (VOX)

The branching rule VSIDS needs to access all variables sorted in decreasing order by their maximum score values $maxscore(x)$. Problem instances from VLSI CAD often have a large number of variables up to several hundred thousands and millions (e.g. problem instances derived from BMC where the number of variables is dependent on the number of timeframes for which a circuit is unrolled). Keeping all variables in a sorted sequence can therefore become a time-consuming task.

`zChaff` uses a STL[2] Vector with pointers to the variables. After each update of the variable scores the vector is sorted using the STL method `stable_sort`. This sorting algorithm preserves the relative ordering of equivalent elements, i.e. if element e_1 is located before e_2 and $e_1 = e_2$ then after sorting using `stable_sort`, e_1 is still located before e_2. This results in a deterministic behaviour with respect to elements that have the same value. If we have n variables a lower bound on the time complexity for `stable_sort` is $\Omega(n \log n)$ (as for all sorting algorithms based on element comparison!) and an upper bound is $O(n(\log n)^2)$.

But only the scores of variables that occur in conflict clauses are changed, thus only these variables must be rearranged in the ordering. In practice, the number of variables that have a changed score are relatively small compared to the whole number of variables. Thus, sorting a vector is perhaps overkill.

To overcome the limitations of using a STL Vector in combination with `stable_sort`, we introduced a STL Multiset that is also ordered (due to the implementation of the STL Multiset as a Red-Black-Tree that needs an ordering relation on the elements). Unfortunately, there's no *Update* function available for STL Multiset. Instead an element to be updated has to be deleted from the set and re-inserted with its updated score. Candidates for an update are variables that occur in conflict clauses, therefore we remember those variables (using a vector) whenever a conflict clause is added. Before an update of all modified score variables we sort this vector (with `stable_sort` in decreasing order of the variables' maxscores) to achieve some kind of approximation to the original `zChaff` implementation. Since each *Deletion* and *Insertion* operation for the STL Multiset are bound by $O(\log n)$, in case of $k < n$ update operations we have achieved an upper bound for the runtime of $O(k(\log k)^2 + k \log n)$ compared to a lower bound of $\Omega(n \log n)$ using `stable_sort`. But we have forgotten that the variable scores are divided by 2 each update. This introduces only linear overhead of $\Theta(n)$, resulting in an overall runtime of $O(n + k(\log k)^2 + k \log n)$ when using a STL Multiset. Using `stable_sort` we have a lower bound of $\Omega(n + n \log n)$.

[2]STL = Standard Template Library (for C++)

4.2. Analysis of Multiple Causes (AMC)

In `zChaff`, when a conflict occurs, only the "'first"' *conflicting clause* for a contradiction is analyzed. But in general it is possible that there are more than one conflicting clauses for a contradicition. It is known that BCP requires approximately 80% of the whole process, thus 20% is needed by conflict analysis, branching rules and backtracking. On the other side, conflict learning is mandatory to solve hard SAT instances. Therefore we can spent some more time in conflict analysis, hoping to identify more powerful conflict clauses that will lead to a reduction of overall runtime.

To implement our concept of the analysis of multiple causes, we proceed as follows. If a conflict occurs within *deduce*, we collect all conflicting clauses, since each of them can be seen as the cause of the conflict. For each conflicting clause, we then make conflict analysis as in original `zChaff` (learning one clause from a conflict using 1UIP), but we do not immediately insert the learned clause into the database. Instead we collect all learned conflict clauses seperately. After we have processed all conflicting clauses, we select one clause from the set of learned conflict clauses, that (1) yields the minimum backtracking level among all other learned conflict clauses, and (2) is the smallest clause among all clauses that yields the same minimum backtracking level.
More formal: Given the set $L = \{c_1, \ldots, c_l\}$ of learned conflict clauses. We add the clause c_a to the clause database that fulfills the following property ($bt\_level(c)$ gets the backtracking level yielded by clause c, and $size(c)$ returns the number of literals of c):

$$\begin{aligned}&(\forall c \in L : bt\_level(c) \geq bt\_level(c_a)) \wedge \\ &(\forall c \in L : bt\_level(c) = bt\_level(c_a) \rightarrow size(c) \geq size(c_a))\end{aligned} \tag{1}$$

Unfortunatly this property is still ambigious when there are several clauses that yield the same backtracking level and that have the same size. In this case the first one that is processed is selected.

4.3. Multiple Individual Conflicts (MIC)

If a conflict occurs within *deduce* then there are possibly remaining implications in the implication queue that are not propagated in original `zChaff`, because BCP is aborted to activate conflict analysis. But starting BCP with some implications in the implication queue the order in which these implications should be processed can be chosen arbitrarily. Therefore it may be possible that using a different order on the implications a "'more interesting"' conflict is triggered that could lead to a better learned conflict clause.

Therefore we propose some kind of multiple conflict scheme that we call *Multiple Individual Conflicts* (MIC). The main idea is to reset the implication assignment that led to a conflict during BCP and proceed BCP with the remaining implications in the implication queue until another conflict occurs or all implications are processed. In the latter case we must remember that at least one conflict occured. Otherwise multiple conflicts occured and again we have a pool of conflicts that can be analyzed.

MIC is embedded into the *deduce* function and is simultaneously combined with conflict analysis. The resulting *deduce_MIC* function is sketched in Figure 3.

```
deduce_MIC( impl_queue ) {
  while ( not empty( impl_queue ) ) {
      impl = pop( impl_queue )
      conflicting_clauses = simplify_db( db, impl )
      if( not empty( conflicting_clauses ) ) {
        conflict_clause =
          compute_conflict_clause( conflicting_clauses )
        add( conflict_clause, conflict_clauses )
        reset_db( db, impl )
        remove_triggered_impl( impl_queue, impl )
      }
  }
  blevel = NO_BACKTRACK
  if ( not empty( conflict_clauses ) {
      learned_clause = select_best_clause( conflict_clauses )
      add( learned_clause, db )
      blevel = bt_level( learned_clause )
  }
  return ( blevel )
```

Figure 3: *deduce_MIC*: MIC embedded in *deduce* combined with conflict analysis.

The toplevel search function as used in `zChaff` must be reorganised to cope with the integration of BCP and conflict analysis into one function. The modified algorithm `zChaff_MIC` is given in Figure 4.

We briefly describe the important functions used in *deduce_MIC*.

- *simplify_db(impl)*: This function just assigns a satisfying assignment to *impl* which is a literal. It simplifies the clause database according to this assignment and returns the set of conflicting clauses, i.e. the set of clauses that are unresolved due to the implication assignment.
- *compute_conflict_clause(conflicting_clauses)*: This function incorporates the conflict analysis. We are free to decide if we want to apply the classical conflict analysis from `zChaff` or to apply our AMC scheme (as described in Section 4.2). The only restriction is that only one conflict clause must be returned.
- *reset_db(db, impl)*: This is the reverse function of *simlify_db* and resets the implication assignment *impl*. In fact, only the assignment is removed from the assignment stack. Due to the clever *Two Literal Watching Scheme* (TLW) in `zChaff` that allows fast detection of unit clauses, nothing has to be changed in the clause database since the invariants for TLW are still valid after removing the assignment from the assignment stack.
- *remove_triggered_impl(impl_queue, impl)*: During *simplify_db(db, impl)* some new implications may be deduced which are then (as a side effect) pushed into the implication queue. Since these implications are only valid under the assumption that *impl* is satisfied we must delete these implications after resetting the assignment of *impl*.
- *select_best_clause(conflict_clauses)*: As in the case of AMC, after MIC enhanced BCP we have a set of "'good"' conflict clauses. Since it is known from previous results that learning many clauses is not reasonable due to memory explosion, we proceed as in AMC. I.e. we select the conflict clause yielding the minimum backtracking level and that is the smallest among all conflict clauses that yield that minimum backtracking level (see Equation 1).

```
zChaff_MIC() {
  while ( 1 ) {
    if( decide_next_branch() ) {
      blevel = 0
      while ( blevel ≠ NO_BACKTRACK ) {
        blevel = deduce_MIC()
        if( blevel == 0 ) {
          return UNSATISFIABLE
        }
        elsif ( blevel ≠ NO_BACKTRACK ) {
          back_track( blevel )
        }
      }
    }
    else {
      return SATISFIABLE
    }
  }
```

Figure 4: `zChaff_MIC`.

5. EXPERIMENTAL RESULTS

We performed experiments to compare the proposed extensions to `zChaff`. We used SAT benchmarks from the Siemens Combinational Equivalence Checking tool `gateComp`, and from the Siemens Model

Checking tool gateProp. Table 1 and 2 gives a summary on these benchmarks. Column zChaff denotes the results from original zChaff. Column zChVox (zChAMC,zChMIC3) gives results for zChaff where VOX (AMC, MIC3 with conflict treshold 3) is enabled. The row aborted gives the number of aborted instances, and row best denotes the number of instances that a solver had the best runtime for an instance among all solvers. For each class a specific time limit was, e.g. for "'easy"' benchmarks from gateComp (e.g. cnf_birl) a limit of 300 seconds was set, and for "'hard"' benchmarks from gateProp (e.g. arb a limit up to 2000 seconds was set.

The experimental results lead to a definitive conclusion: None of the proposed extensions is able to overcome limitations w.r.t. to the number of aborted instances. Only zChVOX gains some small improvement against zChaff on 6 benchmarks from fox, t_8_152, turpin_9_152. These 6 benchmarks are unsatisfiable and have about 130.000 — 200.000 variables. zChaff, zChAMC and zChMIC3 result in significantly larger runtimes or abort these instance within a time limit of 1000 seconds. This is the only case where an improvement due to VOX can be observed (see Table 3).

On three benchmark classes (comp1,comp1.1, and dlx_mod) AMC achieves runtime improvements compared to zChaff. Most of these benchmarks are unsatisfiable, thus indicating that the concept of AMC gains some advantages compared to the conflict analysis in zChaff.

Compared to zChaff the extension MIC3 does not achieve any improvement. Although on the benchmarks from gateComp significantly less instances are aborted compared to zChVox and zChAMC, the runtimes and number of aborted instances regarding benchmarks from gateProp are the worst. One explanation is perhaps, that MIC increases the amount of BCP so much that the gain from a possible better learned conflict clause is not counterbalanced.

Interesting are the results in Table 4 from the benchmark class dmcap113 from gateProp. The instances with suffix -p01 are index permuted versions of the corresponding instances without the suffix. We experienced that these instances gain from the STL Vector implementation of zChaff for the variable ordering. Using the sorted vector, and especially stable_sort, results in giving preference to variables with low indices. These variables correspond to signals "'near"' the inputs of the underlying circuit, thus imitating some kind of simulation. Nevertheless, for these instances it seems helpful to give preferences to these variables. The results on the index permuted versions of these instances corroberate this assertion. Interestingly, VOX aborts most of both instances. This lies in the fact, that due to implementation of VOX the preference given to variables with low indices no longer holds. On the instance pairs (loadp_i_di4a, loadp_i_di4a-p01) and (loadp_i_di4p, loadp_i_di4p-p01) VOX does not abort on the index permuted instances whereas zChaff does. Since these instances have more than 600.000 variables, these improvements can be traced back to the use of the Multiset in VOX.

Finally, we performed experiments enabling VOX together with AMC. Results are given in Table 5 in column zChVOXAMC (we left out the benchmark class dmacp113). Here also the same improvements as for VOX only can be observed: benchmarks from fox, t_8_152, turpin_9_152 are having better runtimes or are not aborted compared to zChaff (see also Table 3).

6. CONCLUSIONS

We proposed 3 extensions to the original zChaff implementation. VOX used a Multiset instead of the a vector for handling the variable ordering. AMC not only uses one conflicting clause when a conflict occurs, but all conflicting clauses, resulting in a set of conflict clauses from which the best is added to the clause database. MIC proceeds in a special way BCP when a conflict already occured during BCP. This also leads to a set of conflict clauses from which the best clause is added to the clause database.

In summary, none of the proposed extensions can overcome limitations of the original zChaff implementation. This indicates that zChaff is highly optimized w.r.t. to data structures and the complex interaction between the internal processes of zChaff. But overall, no improvement is made by combining VOX and AMC.

7. ACKNOWLEDGEMENTS

The author wants to thank Reinhard Enders for many valuable discussions.

8. REFERENCES

[1] R. Bayardo Jr. and P. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *National Conference on Artificial Intelligence (AAAI)*, 1997.

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

[3] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:506–521, 1960.

[4] V. Hooker, J.N. snd Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[5] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 315–326, 1997.

[6] J.P. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *9th Portuguese Conference on Artificial Intelligence (EPIA)*, 1999.

[7] J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. 48(5):506–521, 1999.

[8] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39. Design Automation Conference (DAC)*, Las Vegas, 7 2001.

[9] H. Zhang. SATO: An efficient propositional prover. In *Proceedings of the International Conference on Automated Deduction*, pages 155–160, July 1997.

[10] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, San Jose, California, 11 2001.

Problem	#Instances	zChVOX	zChAMC	zChMIC3	zChaff
gc/cnf_big	8	549.40	378.61	1334.73	152.04
aborted		0	0	1	0
best		0	1	0	7
gc/cnf_birl/	94	26.69	25.38	106.70	20.10
best		19	23	49	29
gc/cnf_bpf/	8	2400.00	2032.44	836.43	929.05
aborted		8	5	0	0
best		0	0	6	2
gc/cnf_c3645/	39	6330.35	10338.58	5624.53	3811.09
aborted		12	30	12	4
best		1	0	7	27
gc/cnf_fft/	223	24174.97	24802.46	23765.43	22988.65
aborted		79	81	74	72
best		17	0	75	79
gc/cnf_peak/	3	3000.00	3000.00	3000.00	3000.00
aborted		3	3	3	3
best		0	0	0	0
gc/cnf_sru/	11	6568.30	8800.00	2643.90	999.54
aborted		7	11	0	0
best		0	0	0	11
All	386	43049.71	49377.47	37311.72	31900.47
aborted		109	130	90	79
best		37	24	137	155

Table 1: Results on `gateComp` benchmarks.

Problem	#Instances	zChVOX	zChAMC	zChMIC3	zChaff
gp/arb/	4	4000.19	4000.22	4000.17	4000.22
aborted		2	2	2	2
best		1	0	1	0
gp/cnf_pilock/	8	776.90	2732.77	2843.08	371.56
best		1	1	1	6
gp/comp1	8	1698.39	267.43	3758.08	458.65
aborted		0	0	2	0
best		0	3	2	3
gp/comp1.1	10	596.41	124.99	4184.93	298.62
aborted		0	0	1	0
best		2	7	1	2
gp/dlx_mod/	2	3.53	58.93	49.52	70.83
best		1	0	1	0
gp/dmacp113/	28	18601.68	27008.41	27584.64	18385.99
aborted		6	9	8	4
best		17	1	0	8
gp/fifo_mod/	4	187.06	243.57	550.64	242.41
best		2	0	0	2
gp/fox/	4	86.70	2011.15	2172.10	1926.64
aborted		0	0	2	0
best		4	0	0	0
gp/ga/	3	251.53	104.16	590.11	54.00
best		0	0	0	3
gp/gpcomp1	2	0.48	0.48	0.91	0.40
best		0	0	1	2
gp/gpcomp3/	3	2.69	4.29	24.42	4.14
best		3	0	0	0
gp/gpcomp8/	1	29.43	22.05	65.08	17.38
best		0	0	0	1
gp/hb	15	2.33	4.57	3.30	1.67
best		0	1	8	9
gp/itc/	28	924.14	2999.52	2392.95	1507.44
best		14	3	12	5
gp/other/	17	2428.65	2793.43	2483.15	2119.46
aborted		2	2	2	2
best		0	1	10	4
gp/qunit_w.8_20/	2	4.48	54.24	1000.04	37.71
aborted		0	0	1	0
best		1	0	1	0
gp/t_3_6/	14	1.78	2.00	12.39	1.63
best		0	0	10	5
gp/t_8_152/	22	265.13	2015.50	2040.09	2015.37
aborted		0	2	2	2
best		10	6	6	2
gp/t_8_20/	74	15.23	34.67	1329.02	21.80
best		12	19	52	36
gp/turpin_9_152/	22	300.24	2015.26	2040.08	2014.71
aborted		0	2	2	2
best		8	5	4	6
All	271	30176.97	46497.64	57124.70	33550.63
aborted		10	17	22	12
best		76	47	110	94

Table 2: Results on `gateProp` benchmarks.

Problem	Vars	Clauses	Result	zChVOX	zChAMC	zChMIC3	zChaff
gp/fox							
fox1-p01	136828	396973	SAT	15.53	538.37	>1000.10	981.72
fox1	136828	396973	SAT	15.01	29.73	86.71	54.80
fox981124p-p01	134999	391486	SAT	20.19	920.19	>1000.10	790.67
fox981124p	134999	391486	SAT	35.97	522.86	85.39	99.45
gp/t_8_152							
bench1	36605	95710	SAT	1.43	2.04	12.42	2.10
bench10	14409	38590	SAT	0.56	0.54	0.44	0.60
bench11	14421	38623	SAT	0.54	0.55	0.44	0.57
bench12	13604	34435	UNSAT	0.38	0.37	0.40	0.36
bench13	14433	38656	SAT	0.53	0.54	0.40	0.61
bench14	14443	38683	SAT	0.48	0.63	1.16	0.56
bench15	13604	34435	UNSAT	0.40	0.40	0.38	0.38
bench16	13604	34435	UNSAT	0.32	0.37	0.36	0.33
bench17	13604	34435	UNSAT	0.36	0.33	0.36	0.37
bench18	13604	34435	UNSAT	0.38	0.34	0.39	0.38
bench19	13604	34435	UNSAT	0.40	0.42	0.36	0.39
bench2	15558	41272	SAT	0.55	0.67	1.55	0.63
bench20	13604	34435	UNSAT	0.34	0.34	0.38	0.38
bench21	13604	34435	UNSAT	0.42	0.35	0.40	0.37
bench22	13604	34435	UNSAT	0.39	0.31	0.37	0.37
bench3	36571	95608	SAT	1.38	2.00	3.12	2.00
bench4	221965	572692	UNSAT	170.34	>1000.10	>1000.10	>1000.10
bench5	36580	95635	SAT	1.57	2.12	3.28	1.97
bench6	36586	95653	SAT	1.50	2.03	12.29	1.79
bench7	203132	516193	UNSAT	81.61	>1000.10	>1000.10	>1000.10
bench8	26017	58705	UNSAT	0.73	0.61	1.17	0.64
bench9	14396	38554	SAT	0.52	0.54	0.42	0.57
gp/turpin_9_152/							
bench1	36605	95710	SAT	1.51	1.92	12.26	1.90
bench10	14409	38590	SAT	0.52	0.58	0.45	0.58
bench11	14421	38623	SAT	0.54	0.58	0.43	0.55
bench12	13604	34435	UNSAT	0.41	0.33	0.37	0.35
bench13	14433	38656	SAT	0.56	0.58	0.43	0.57
bench14	14443	38683	SAT	0.51	0.56	1.10	0.52
bench15	13604	34435	UNSAT	0.38	0.35	0.37	0.35
bench16	13604	34435	UNSAT	0.36	0.36	0.36	0.35
bench17	13604	34435	UNSAT	0.35	0.33	0.39	0.36
bench18	13604	34435	UNSAT	0.44	0.37	0.38	0.35
bench19	13604	34435	UNSAT	0.35	0.39	0.38	0.34
bench2	15558	41272	SAT	0.58	0.60	1.50	0.64
bench20	13604	34435	UNSAT	0.42	0.36	0.38	0.37
bench21	13604	34435	UNSAT	0.42	0.33	0.36	0.35
bench22	13604	34435	UNSAT	0.39	0.37	0.37	0.36
bench3	36571	95608	SAT	1.49	2.08	3.20	1.96
bench4	221965	572692	UNSAT	199.92	>1000.10	>1000.10	>1000.10
bench5	36580	95635	SAT	1.62	1.99	3.25	1.81
bench6	36586	95653	SAT	1.59	1.94	12.61	1.84
bench7	203132	516193	UNSAT	86.68	>1000.10	>1000.10	>1000.10
bench8	26017	58705	UNSAT	0.65	0.68	1.07	0.63
bench9	14396	38554	SAT	0.55	0.56	0.42	0.53

Table 3: Results on `gateProp` benchmarks with some improvement using VOX.

Problem	Vars	Clauses	Result	zChVOX	zChAMC	zChMIC3	zChaff
loadp_i_di4a-p01	609356	1800745	SAT	1379.91	>2000.10	>2000.10	>2000.10
loadp_i_di4a	609356	1800745	SAT	>2000.10	88.96	94.66	95.36
loadp_i_di4p-p01	631132	1864834	SAT	1779.08	>2000.10	>2000.10	>2000.10
loadp_i_di4p	631132	1864834	SAT	>2000.10	296.81	372.72	128.76
loadp_i_di5a-p01	566217	1673233	ABORT	>2000.10	>2000.10	>2000.10	>2000.10
loadp_i_di5a	566217	1673233	SAT	>2000.10	212.83	201.42	112.24
loadp_i_di5p-p01	598108	1766941	ABORT	>2000.10	>2000.10	>2000.10	>2000.10
loadp_i_di5p	598108	1766941	SAT	>2000.10	497.42	>2000.10	298.66
loadp_i_di6a-p01	385797	1142533	SAT	487.47	>2000.10	>2000.10	1509.60
loadp_i_di6a	385797	1142533	SAT	269.16	263.02	54.15	42.71
loadp_i_di6p-p01	417047	1234525	UNSAT	702.24	>2000.10	>2000.10	1151.61
loadp_i_di6p	417047	1234525	UNSAT	783.74	>2000.10	1430.30	358.24
loadp_i_di_sa-p01	310887	921655	SAT	68.18	490.41	344.39	335.70
loadp_i_di_sa	310887	921655	SAT	87.45	332.12	46.13	34.09
loadp_i_di_sp-p01	333401	988123	UNSAT	119.64	883.34	926.19	380.33
loadp_i_di_sp	333401	988123	UNSAT	78.55	1110.02	417.56	177.05
loadp_i_dia-p01	239666	710590	SAT	12.93	351.01	508.34	303.84
loadp_i_dia	239666	710590	SAT	13.24	17.95	13.60	17.92
loadp_i_dip-p01	266049	788569	SAT	13.99	423.99	746.96	379.76
loadp_i_dip	266049	788569	SAT	15.97	21.02	20.91	22.44
storep_di_i_s8a-p01	245301	723442	SAT	61.15	1491.53	972.04	709.87
storep_di_i_s8a	245301	723442	SAT	74.82	35.20	128.38	32.77
storep_di_i_s8p-p01	257027	757864	SAT	66.43	1045.40	1205.30	718.23
storep_di_i_s8p	257027	757864	SAT	57.28	212.11	623.65	556.88
storep_di_i_s9a-p01	285797	843250	SAT	33.92	1120.72	1545.79	1121.08
storep_di_i_s9a	285797	843250	SAT	101.51	114.55	68.79	42.51
storep_di_i_s9p-p01	297757	878371	UNSAT	227.01	>2000.10	>2000.10	1425.43
storep_di_i_s9p	297757	878371	UNSAT	168.01	>2000.10	1863.36	430.91

Table 4: Results on `dmacp113` from `gateProp` benchmarks.

Problem	#Instances	zChVOXAMC	zChaff
gc/cnf_big	8	1075.60	152.04
aborted		2	0
best		0	8
gc/cnf_birl/	94	35.72	20.10
best		87	12
gc/cnf_bpf/	8	2400.00	929.05
aborted		8	0
best		0	8
gc/cnf_c3645/	39	10846.81	3811.09
aborted		30	4
best		0	35
gc/cnf_fft/	223	24818.39	22988.65
aborted		82	72
best		77	80
gc/cnf_peak/	3	3000.00	3000.00
aborted		3	3
best		0	0
gc/cnf_sru/	11	8800.00	999.54
aborted		11	0
best		0	11
gp/arb/	4	4000.18	4000.22
aborted		2	2
best		2	1
gp/cnf_pilock/	8	3327.35	371.56
aborted		1	0
best		3	5
gp/comp1	8	3230.64	458.65
aborted		2	0
best		2	6
gp/comp1.1	10	3242.44	298.62
aborted		2	0
best		3	7
gp/dlx_mod/	2	2.97	70.83
best		2	0
gp/fifo_mod/	4	274.93	242.41
best		2	2
gp/fox/	4	85.95	1926.64
best		4	0
gp/ga/	3	825.92	54.00
best		0	3
gp/gpcomp1	2	0.47	0.40
best		0	2
gp/gpcomp3/	3	3.22	4.14
best		3	0
gp/gpcomp8/	1	19.83	17.38
best		0	1
gp/hb	15	11.99	1.67
best		1	14
gp/itc/	28	2356.28	1507.44
aborted		2	0
best		17	14
gp/other/	17	3068.13	2119.46
aborted		3	2
best		6	10
gp/qunit_w.8_20/	2	6.73	37.71
best		2	0
gp/t_3_6/	14	1.77	1.63
best		10	6
gp/t_8_152/	22	608.55	2015.37
aborted		0	2
best		20	5
gp/t_8_20/	74	17.55	21.80
best		39	51
gp/turpin_9_152/	22	629.47	2014.71
aborted		0	2
best		14	10
All	629	72690.89	47065.11
aborted		148	87
best		294	291

Table 5: Results of VOX combined with AMC compared to `zChaff`.