Chaff in practice

Samuli Larvala

April 3, 2003

Abstract

We will look into the practical performance of the zChaff SAT solver that won the SAT 2002 competition in both industrial and handmade benchmark categories. The solver will be subjected to various tests like solving random SAT instances, factoring and primality testing that have been transformed into SAT problems and also more practical SAT instances that come from real hardware verification. We only present the results of these tests and do not attempt to compare them with any other SAT solver.[1]

1 Introduction

The Boolean Satisfiability problem (SAT) is one of the most studied problems in computer science because of its importance in theoretical research and its growing practical applications. Given a Boolean formula, the SAT problem asks if it is possible to find an assignment for the variables of the formula so that it evaluates to true, or to determine that such an assignment is not possible (UNSAT).

Chaff [1] is a fairly new SAT solver but it is mainly based on ideas that have already been used in other solvers. For this study we will use zChaff [3] which is an implementation of Chaff maintained by Lintao Zhang. Chaff contains several improvements and new ideas, and the results were impressive enough to award Chaff with the best complete solver in both industrial and handmade benchmark categories, in the SAT 2002 competition.

2 Theoretical background

Chaff is a complete SAT solver and it is based on the well known Davis-Putnam (DP) backtrack search. The basic backtracking starts by assigning one variable from the given formula to either true or false and proceeds by doing the same operation for other variables. The choice of which variable to assign and to which value, is called a *decision* or *decision assignment*.

[1]This paper is part of the mandatory requirements of the T-79.194 seminar course

If the process detects that the previous decision causes a conflict, i.e. that the formula can not be satisfied by the current assignment, we backtrack to the last variable that has not been tried both ways, i.e. both true and false assignments have lead to a conflict, and flip the previously chosen assignment. This process is repeated until an assignment is found so that the given formula is true or that we have tried all possible assignments and none satisfy the given formula.

The formula given to us is mainly in Conjunctive Normal Form (CNF) containing a logical and ($\wedge$) of one or more *clauses*. Each clause is a logical or ($\vee$) of one or more *literals*. If the whole formula is to be true, then all the clauses must also be true. If all but one of the literals in a clause have been assigned to false, then the remaining literal becomes a *unit literal* and the clause becomes a *unit clause*. They are significant since in order for the clause to be true, the unit literal must be true. This is an *implication* in the sense that the decisions done so far and this clause imply that the unit literal must be true or otherwise the whole formula is false.

The process of finding and using these implications is called binary constraint propagation (BCP). The BCP might come across a conflict in the sense that two unit clauses imply the same variable to opposite assignments. From such conflicts we can try to learn and avoid these same conflict in the future. Learning is usually implemented so that we add *conflict clauses* that are based on the decisions and implications that have taken place. We resolve the conflict by undoing previous decisions and implications and have the conflict clause as a reminder for us that we should not go in that direction anymore but look elsewhere in the backtrack tree.

In Chaff, the authors use a Variable State Independent Decaying Sum (VSIDS) based decision algorithm that tries to select the most useful variables that gets assigned. After the assignment, Chaff uses an optimized BCP algorithm based on the watched two literals scheme to find possible unit clauses and propagate any implications. Chaff can also learn from possible BCP conflicts and add conflict clauses. Further details can be found in [1, 2].

3 Test Specifications

To test zChaff, we will use random 3-SAT instances that are approximately at the phase transition point. The phase transition point is a point where the probability that a randomly chosen SAT instance is either SAT or UNSAT is 50%. The point is measured as the ratio between clauses and variables and for our test we will use 4.3 [4]. Since random instances are inherently unpredictable, we will also use factoring integers and primality proving as a more deterministic benchmark. Finally we will do a more practical test run by verifying real hardware specifications provided by IBM [5].

As our test hardware we will use an Intel Pentium 4 1.4GHz machine with 1GB of RAM. The operating system will be Linux using kernel 2.4.18. For all tests a time limit of one hour was set. If a test run took more than that it was terminated. The diagrams will have these terminated cases displayed although the exact runtime is of course larger. The time that was measured was the amount of runtime the solver program got. The timing was done by zChaff itself. Comparisons with the zChaff timing and timing provided by the system (/usr/bin/time) showed them to differ by 0.02 seconds at the most which is acceptable for our purposes. No other means of estimating timing differences or problems was made.

Chaff supports restarting the whole search process but zChaff doesn't report the number of restarts that were made so the affects of restarting will not be taken into account in any way in our measurements.

4 Test Statistics

The diagrams presented in the following sections contain results of the benchmark runs. In all diagrams a diamond represents an instance that was found to be satisfiable and a cross represents an instance that was found unsatisfiable. The solid line will indicate the growth tendency of the satisfiable instances and the dashed line the unsatisfiable instances.

4.1 Total Runtime

4.1.1 IBM Benchmark

The benchmark set based on verifying hardware specification provided by IBM contains 13 instances [5]. For zChaff verifying that they are all satisfiable is a fairly easy task. The runtime for each instance is show in Table 1. According to unofficial tests by the university of Darmstadt [5], all these instances can be solved by Chaff in about 8 minutes using an Intel Pentium 3 700MHz machine. This is a bit surprising since our test hardware should be faster. This difference in run times might be because of compiler parameter tuning or the significant difference in processor architecture on the Pentium 4 that modern compilers can't utilize yet. The zChaff executable was provided in precompiled format so no compiler tuning was done.

The full benchmark test was run only once, but some of the faster cases were run several times as to see if there is much difference between repeated runs. No significant difference was detected and thus only one full test run was made.

4.1.2 Random 3-SAT Instances

For the random 3-SAT benchmark we fixed the number of variables for each run and took the number of clauses to be approximately 4.3 times this

Instance	Variables	Clauses	Time (sec)
bmc-ibm-1	9685	55870	2.66
bmc-ibm-2	2810	11683	0.01
bmc-ibm-3	14930	72106	0.80
bmc-ibm-4	28161	139716	4.51
bmc-ibm-5	9396	41207	0.23
bmc-ibm-6	51639	368352	15.08
bmc-ibm-7	8710	39774	0.08
bmc-galileo-8	58074	294821	111.80
bmc-galileo-9	63624	326999	168.37
bmc-ibm-10	59056	323700	490.14
bmc-ibm-11	32109	150027	39.77
bmc-ibm-12	39598	194778	186.64
bmc-ibm-13	13215	65728	11.01

Table 1: IBM benchmark results

number with a bit of fluctuation both ways so we deviate to both sides of the phase transition point. The range of the deviation is presented in Table 2. For each number of variables v we did 20 tests so that the number of clauses was $4.3v + 10k$ where $k = -10, -9, \ldots, 8, 9$.

It should be noted that such a linear deviation distorts the ratio between clauses and variables as the number of variables grows. For $v = 100$ the ratio is $3.3 - 5.2$, for $v = 200$ it is $3.8 - 4.5$ and for $v = 290$ the ratio is $3.9 - 4.6$. Thus the problems might get harder as the number of variables grows and we focus more on the assumed phase transition point. This depends on how accurate the used 4.3 ratio for the phase transition is. The deviation to the number of clauses was added to allow some movement around the phase transition point in case it is not exactly 4.3.

The runtime for these experiments can be seen in Figure 1. The figure also displays an approximated exponential growth curve for the unsatisfiable instance run times.

The random SAT instances were generated using the Makewff program [6].

4.1.3 Factoring and Prime Numbers

The factoring and primality proving benchmark was used to get a bit more deterministic results since for factoring, there should only be one possible satisfying truth assignment and thus finding it should be challenging. For primality proving, all instances should be unsatisfiable. We use the Genfacbm program [7] to generate our factoring and primality proving instances. Both test cases used the Braun multiplication circuit. The Atree circuit was

Variables	Clauses	Repeat	Variables	Clauses	Repeat
100	330-520	20	200	760-950	20
110	373-563	20	210	803-993	20
120	416-606	20	220	846-1036	20
130	459-649	20	230	889-1079	20
140	502-692	20	240	932-1122	20
150	545-735	20	250	975-1165	20
160	588-778	20	260	1018-1208	20
170	631-821	20	270	1061-1251	20
180	674-864	20	280	1104-1294	20
190	717-907	20	290	1147-1337	20

Table 2: Random 3-SAT test parameters

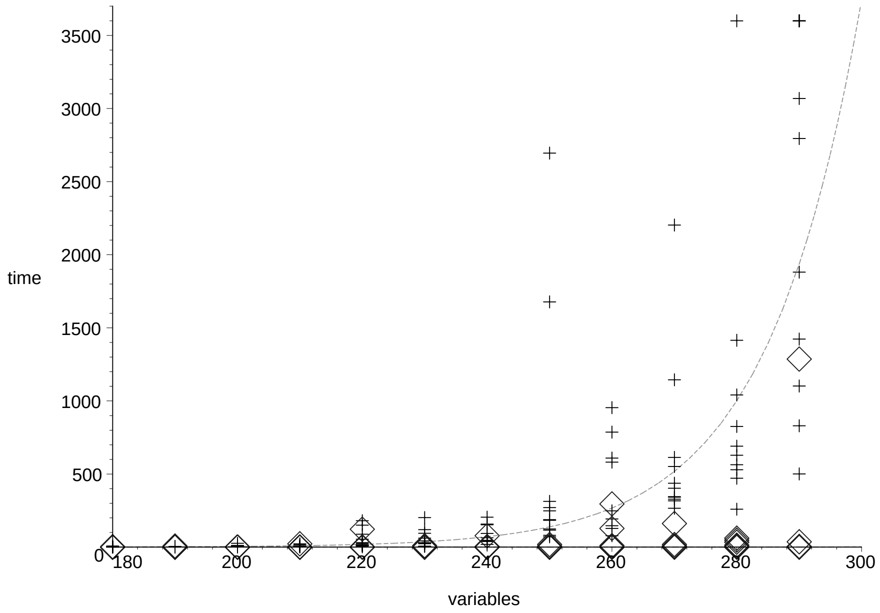

Figure 1: Random 3-SAT test runtime

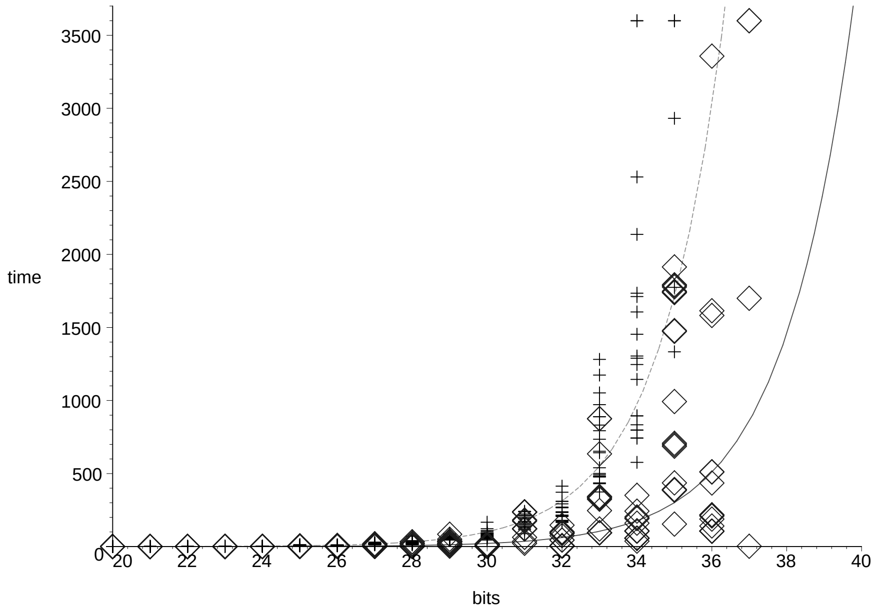

Figure 2: Factoring test runtime

also tested, but it did not perform significantly better nor worse, so the results will be omitted. The runtime of the factoring and primality proving benchmark can be seen in Figure 2.

Since we have a fixed multiplication circuit, the number of clauses and literals can't be controlled arbitrarily. As our measurement unit we use the size in bits of the number being factored or proved prime.

For factoring 20 to 36 bit numbers we created 20 instances for each bit size giving $1|s$ (where $|$ is concatenation) where $s = 1, 2, \ldots, 20$ as the seed value to genfacbm. For bit size 37, only four ($s = 1, 2, 3, 4$) instances were tested since the one hour time limit terminated most of the tests.

For primality proving 20 to 34 bit numbers we created 20 instances and again used $1|s$ as the seed where $s = 1, 2, \ldots, 20$. For bit size 35 we only did five tests ($s = 1, 2, 3, 4, 5$) because of the time limit constraint.

Smaller numbers in factoring or primality proving were not used because genfacbm can't generate many small instances.

The runtime diagram shows a curious 37-bit instance that is found to be satisfiable in only 2.5 seconds. This behaviour is reproducible, but it is unknown if it is a bug in genfacbm or zChaff, or just a very strange instance that happens to be so easy. For further investigation, the instance can be generated by executing 'genfacbm dimacs sat braun 37 13'. For 36-bit numbers the fastest factorization instance was solved in 105 seconds.

4.2 Decision making

The decision assignment of Chaff is based on a VSIDS [1] scheme. It tries to emphasize satisfying the conflict clauses as this, according to the belief of the Chaff authors, is the key that drives the search process.

Various statistics from the decision assignment are provided by zChaff. One statistic is the maximum decision level. In trivial backtracking one would have to go down to the bottom of the search tree before it will find a solution. Since Chaff uses unit clauses to imply the assignment of other variables, the backtracking search doesn't have to go too deep into the tree before the implications start to work for us and conflicts indicate that we should not go deeper. The lower the maximum decision level, the faster the search progresses since we find out at an early stage that we should not go further in the current branch that we are traversing.

Figure 3 displays the maximum decision level where zChaff ventured while working on the random SAT instances. Figure 4 gives the same diagram for the factoring and primality proving test. It is not that surprising that the decision level for unsatisfiable instances in the random SAT test is lower since it is to be expected that we run into conflicts sooner than we would, if working on a really satisfiable instance.

For factoring the depth is more equal which can be an indication of a few things. In random SAT instances we probably have several satisfying truth assignments as opposed to just one in the factoring case. For primality proving the case is the same as with a random UNSAT instance and we have to traverse the whole search tree to conclude that. For factoring and primality proving, the structure of the multiplication circuit might have something to do with the equal depth of traversing the whole tree to prove the case UNSAT or find the single satisfying truth assignment for the factoring case. Further testing and more sophisticated analysis of the results are needed to confirm these assumptions.

4.3 Binary Constraint Propagation

The binary constraint propagation phase of a SAT solver usually takes most of the runtime. Chaff watches two literals in each clause in order to easily and rapidly detect the clauses that will become unit clauses. The amount of implications done by the BCP for the random 3-SAT instances can be seen in Figure 5. This curve seems to follows the total runtime curve as is to be expected if BCP takes the majority of the runtime. The same diagram for the factoring and primality proving tests can be seen in Figure 6.

4.4 Learning

If the BCP runs into a conflict, Chaff can learn from the conflict and generate conflict clauses that help it go into other directions in the backtrack search.

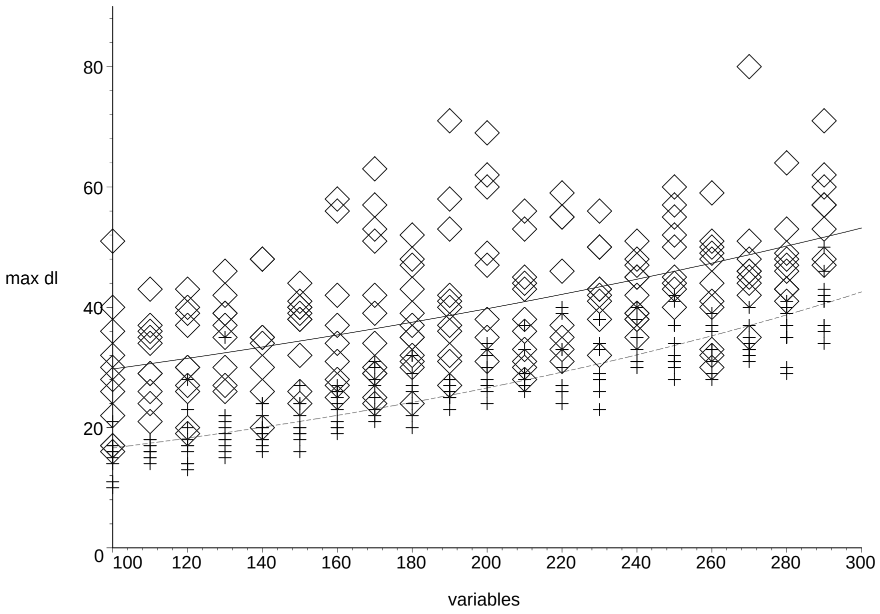

Figure 3: Random 3-SAT instance maximum decision level

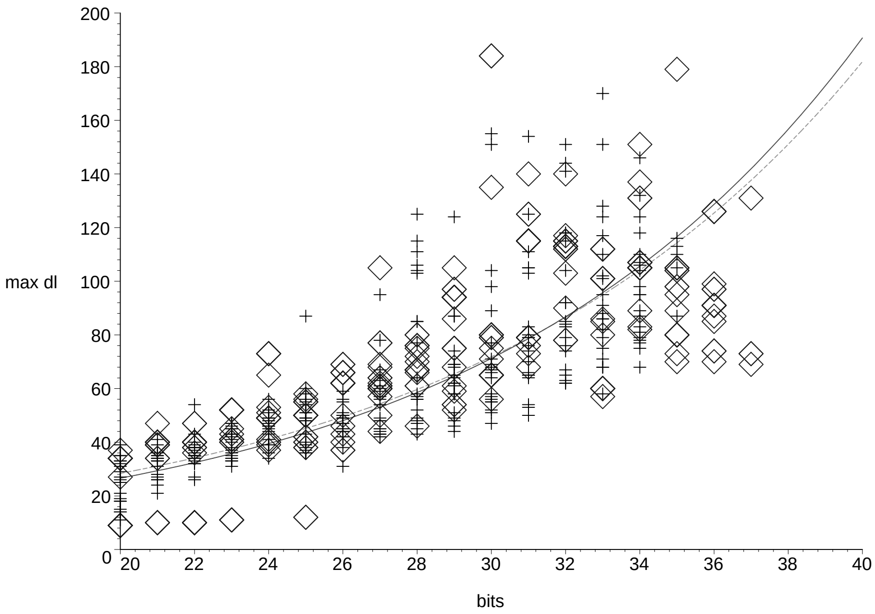

Figure 4: Factoring instance maximum decision level

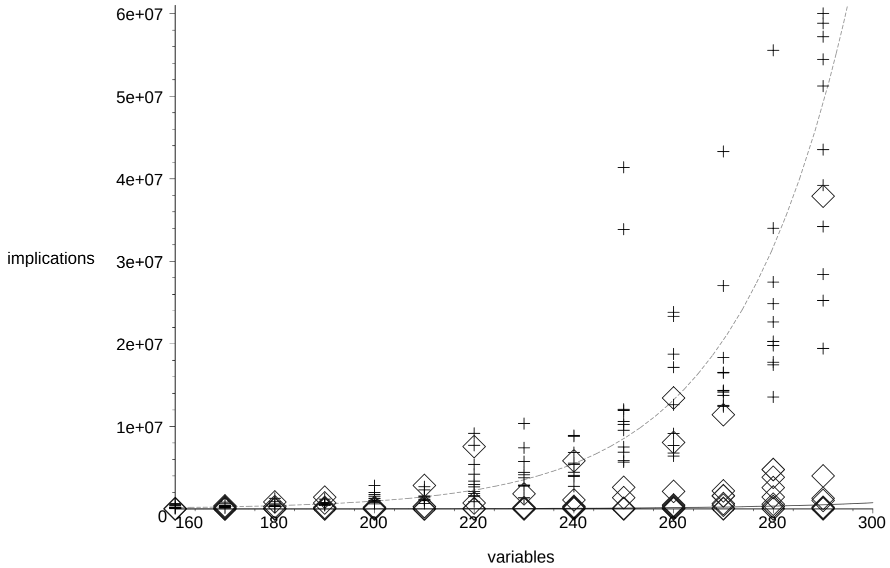

Figure 5: Random 3-SAT instance number of implications

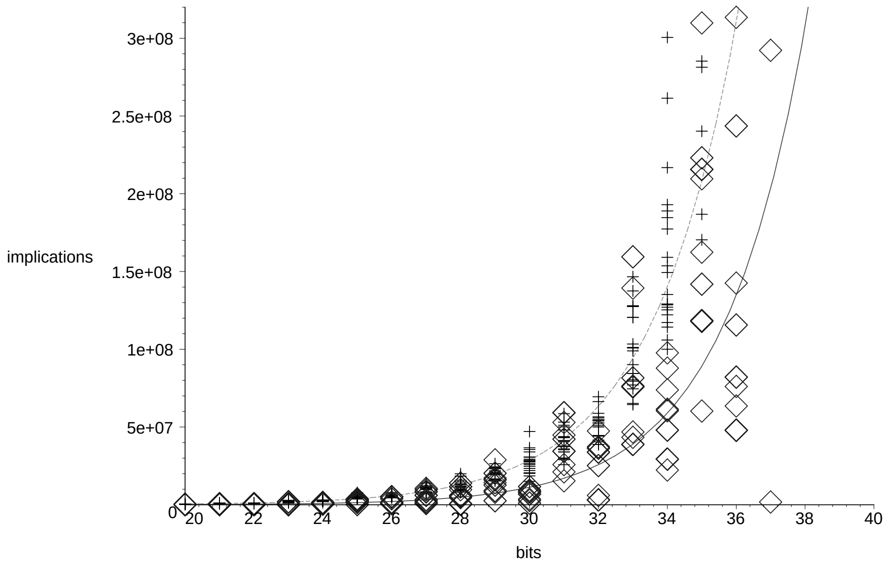

Figure 6: Factoring instance number of implications

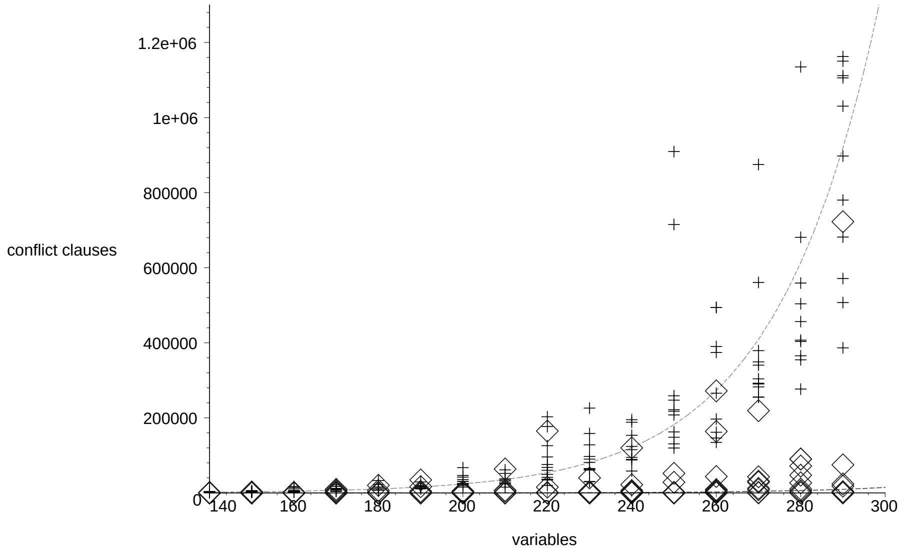

Figure 7: Random 3-SAT instance generated conflict clauses

These clauses can be very effective in pruning the search tree. The number of clauses generated and the average size of the clauses for the random 3-SAT instances can be seen in Figure 7 and Figure 8 respectively. The same results for factoring are provided in Figure 9 and Figure 10.

For the random SAT instances the learning with satisfiable instances seems to get more difficult as the number of variables (and also clauses) increases. The average conflict clause size (number of literals) increases and this means it becomes harder to get the conflict clauses into unit clauses that would help us by implicating other variables. At the same time the number of generated conflict clauses increases and so we get a lot of clauses that slow down the BCP and need more decisions before we gain any knowledge from them. Learning seems to slow down faster with satisfiable instances than with unsatisfiable.

For factoring the learning with unsatisfiable cases gets more difficult faster than with satisfiable cases. A more clear explanation for this would again need more testing and more sophisticated analysis of the results.

As can be seen from Figure 8 there are some SAT instances that don't cause any conflicts clauses at all. These cases happen to be so easy that we can find a satisfying truth assignment by just decisions and implications, without any conflicts. For UNSAT instances this is of course not possible. For factoring such SAT instances should be very rare since there is only one satisfying truth assignment and the chances that the solver finds it without trying any other assignment first, is almost impossible.

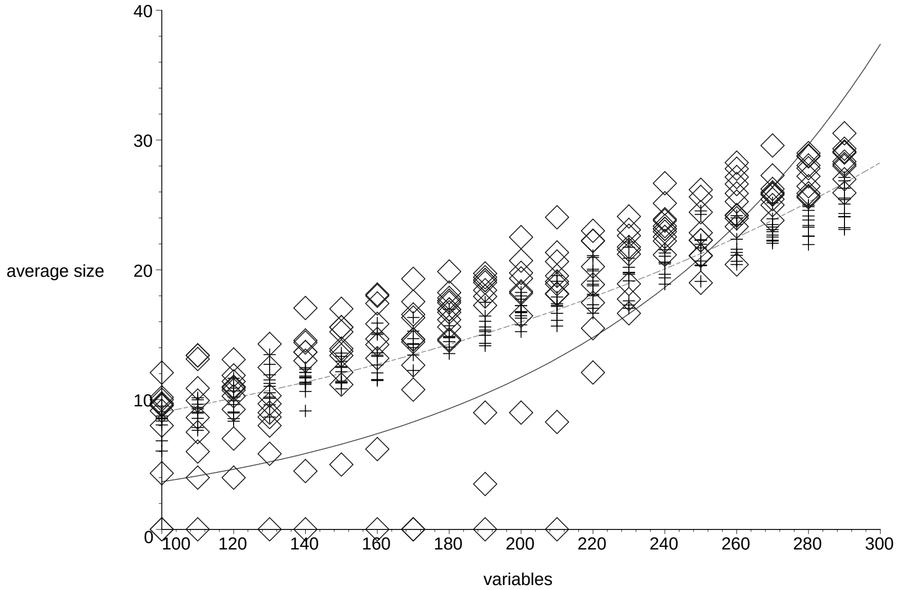

Figure 8: Random 3-SAT instance average conflict clause size

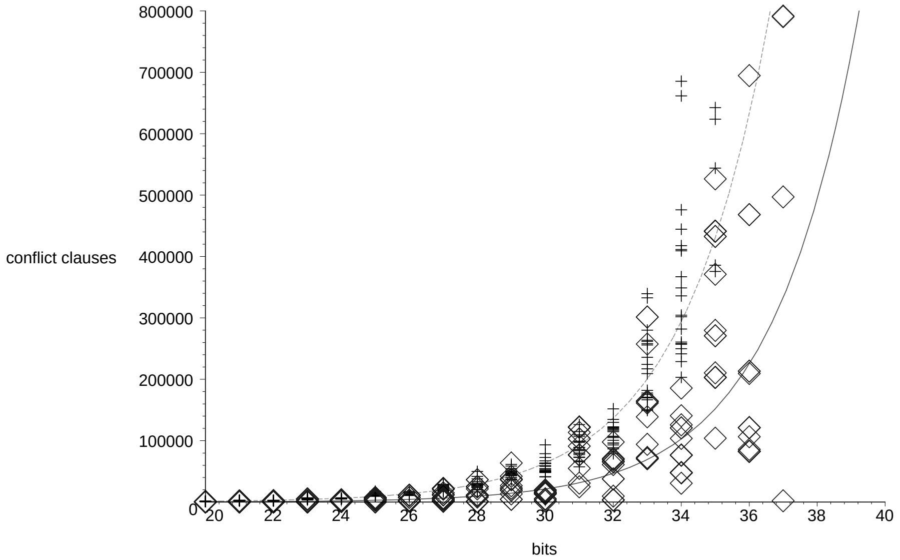

Figure 9: Factoring instance generated conflict clauses

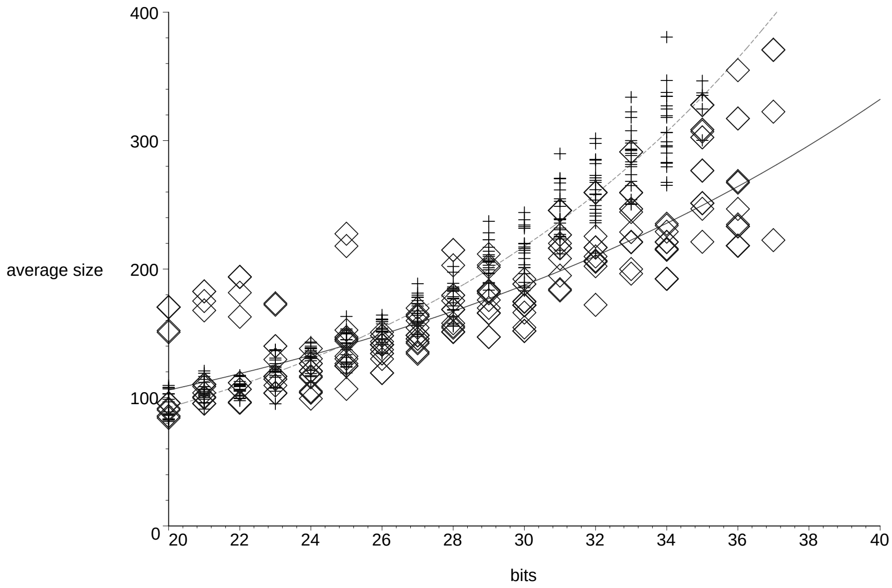

Figure 10: Factoring instance average conflict clause size

5 Further Work

Since the scope of this study was a fairly limited one, further studies on the relation between SAT problems involving pure three literal clauses and n literal clauses might prove to be interesting. All n literal clauses can be broken down into smaller clauses by introducing new variables and splitting the original clause into smaller pieces. It would be interesting to reverse this operation so several small clauses are possibly joined into a larger one and see what kind of effect this has on the total runtime. The test could be conducted by generating n literal clauses and testing if they are solvable. If so, the clauses could be broken into smaller ones and solved again and the runtime compared with previous results to see what size of clause is the optimum.

6 Conclusion

Based on the success in the SAT 2002 competition, zChaff is the current state of the art SAT solver. It is not perfect in all possible benchmarks but has a good overall performance. Comparison with other solvers presented on the seminar course T-79.194 could provide more insight into the performance issue, but results on other solvers are not available at this time and thus a final conclusion can be made at the end of the course. The scope of this study was limited and thus more work should be done to more thoroughly analyze the pros and cons of Chaff.

This study shows that zChaff can within a reasonable time, solve random 3-SAT instances with up to 270 variables, factor numbers of up to 36 bits and prove up to 33-bit numbers as primes.

References

[1] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," presented at 39th Design Automation Conference, 2001.

[2] L. Zhang, C. Madigan, M. Moskewicz, S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," Proceedings of ICCAD 2001, San Jose, CA, Nov. 2001.

[3] L. Zhang, "The zChaff SAT solver," `http://ee.princeton.edu/~chaff/zchaff.php`.

[4] J. Crawford, L. Auton, "Experimental Results on the Crossover Point in Random 3-SAT," Artificial Intelligence, vol. 81, pp. 31-57, 1996.

[5] Bounded Model Checking benchmark provided by IBM available at `http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/Benchmarks/SAT/BMC/description.html`.

[6] H. Kautz, B. Selman, Random SAT formula generator available at `http://www.cs.washington.edu/homes/kautz/walksat/`

[7] T. Pyhälä, Random factoring and primality proving SAT instance generator available at `http://www.tcs.hut.fi/~tpyhala/genfacbm.tgz`