NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT instances*

Sathiamoorthy Subbarayan[1] and Dhiraj K Pradhan[2]

[1] Department of Innovation, IT-University of Copenhagen, Copenhagen, Denmark
sathi@itu.dk
[2] Department of Computer Science, University of Bristol, Bristol, UK
pradhan@cs.bris.ac.uk

Abstract. The original algorithm for the SAT problem, Variable Elimination Resolution (VER/DP) has exponential space complexity. To tackle that, the backtracking based DPLL procedure [2] is used in SAT solvers. We present a combination of both of the techniques. We use NiVER, a special case of VER, to eliminate some variables in a preprocessing step and then solve the simplified problem using a DPLL SAT solver. NiVER is a strictly formula size not increasing resolution based preprocessor. Best worst-case upper bounds for general SAT solving (arbitrary clause length) in terms of N (Number of variables), K (Number of clauses) and L (Literal count) are 2^N, $2^{0.30897K}$ and $2^{0.10299L}$, respectively [14]. In the experiments, NiVER resulted in upto 74% decrease in N, 58% decrease in K and 46% decrease in L. In many real life instances, we observed that most of the resolvents for several variables are tautologies. There will be no increase in space due to VER on them. Hence, despite its simplicity, NiVER does result in easier instances. In most of the cases, NiVER takes less than one second for preprocessing. In case NiVER removable variables are not present, due to very low overhead, the cost of NiVER is insignificant. We also study the effect of combining NiVER with HyPre [3], a preprocessor based on hyper binary resolution. Based on experimental results, we classify the SAT instances into 4 classes. NiVER consistently performs well in all those classes and hence can be incorporated into all general purpose SAT solvers.

1 Introduction

The VER [1] has serious problems due to exponential space complexity. So, modern SAT solvers are based on DPLL [2]. Preprocessors (simplifiers) can be used to simplify SAT instances. The simplified formula can then be solved by using a SAT Solver. Pure literal elimination and unit propagation are the two best known simplification methods used in most of the DPLL based SAT solvers. Although several preprocessors have been published [3],[4], current state of the art SAT solvers [6],[5] just use these two simplifications. The 2-simplify preprocesor by Brafman [4], applies unit clause resolution, equivalent variable substitution, and a limited form of hyper resolution. It also generates new implications using binary clause resolution. Recent preprocessor, HyPre [3] applies all the rules in 2-simplify and also does hyper binary resolution. In this paper we introduce a new resolution based simplifier NiVER, (Non Increasing VER). Like other simplifiers, NiVER takes a CNF as input and outputs another CNF, with a less or equal number of variables. Preprocessing is worthwhile only if the overall time taken for simplification and solving the simplified formula is less than the time required to solve the unsimplified formula. For several problem classes, NiVER results in reducing the overall runtime. In many cases, NiVER takes less than one second CPU time. For few problems HyPre preprocessor itself solves the problem. But for some instances, it takes a lot of time to preprocess, while the original problem is easily solvable by SAT solvers. Unlike HyPre, NiVER consistently performs well. Hence, like clause learning and decision heuristics, NiVER can also be integrated into the DPLL framework for general purpose SAT solvers. Next section presents NiVER. Section 3 shows some empirical results and we conclude in section 4.

* While preparing final version of the paper , we looked for papers on complexity of SAT algorithms and found that a variant of NiVER method, which does not allow increase in K, was used in [14] to obtain the current best worst-case upper bounds. The method in [14] was used not just as a preprocessor, but, also at each node of a DPLL search. However, we could not find any implementation of it.

2 NiVER: Non Increasing VER

In [7], Franco resolves away variables with two occurences. On a class of random benchmarks, Franco has empirically shown that his procedure, in average case, results in polynomial time solutions. In 2clsVER [8], VER was used, they resolved away a variable rather than splitting on it, if the VER results in less than 200 increase in L (Number of literals). It was done inside a DPLL method, not as a preprocessor. But that method was not successful when compared to state of the art DPLL algorithms. NiVER does not consider the number of occurrences of variables in the formula. In some instances, NiVER removes variables having more than 25 occurrences. For each variable NiVER checks whether it can be removed by VER, without increaseing L. If so it eliminates the variable by VER. The algorithm is shown in Alg. 1. When VER removes a variable, many resolvents have to be added. We discard tautologies. The rest are added to the formula. Then, all clauses containing that variable are deleted from the formula. In real life instances we observed that for many variables, most of the resolvents are tautologies and there is no increase in space due to VER. Except checking for tautology, NiVER does not do any complex steps like subsumption checking. No other simplification is done. Variables are checked in the sequence of their numbering in the original formula. There is not much difference due to different variable orderings. Some variable removals cause other variables to be removable. NiVER iterates until no more variables can be removed. In the present implementation, NiVER does not even check whether any unit clause is present or not. Rarely, when a variable is removed, we observed an increase in K although NiVER does not allow L to increase. Unlike HyPre or 2-simplify, NiVER does not do unit propagation, neither explicitly nor implicitly.

Algorithm 1 NiVER CNF Preprocessor

```
NiVER(F)
repeat
  entry = FALSE
  for all V ∈ Var(F) do
    P_C = {C | C ∈ F, l_V ∈ C }
    N_C = {C | C ∈ F, l̄_V ∈ C }
    R = { }
    for all P ∈ P_C do
      for all N ∈ N_C do
        R = R ∪ Resolve(P,N)
        Old_Num_Lits = Number of Literals in (P_C ∪ N_C)
        New_Num_Lits = Number of Literals in R
        if (Old_Num_Lits ≥ New_Num_Lits) then
          F=F-(P_C ∪ N_C), F=F+R, entry = TRUE
        end if
      end for
    end for
  end for
until ¬entry
return F
```

NiVER preserves the satisfiability of the original problem. If the simplified problem is unsatisfiable, then the original is also unsatisfiable. If the simplified problem is satisfiable, the assignment for the variables in the simplified formula is a subset of at least one of the satisfying assignments of the original problem. For variables removed by NiVER, the satisfying assignment can be obtained by a well known polynomial procedure, in which we just reverse the way NiVER proceeds. We add variables back in the reverse order they were eliminated. While adding each variable, assignment is made to that variable such that the formula is satisfied. For example, let F be the original formula. Let C_x refers to set of clauses containing literals of variable x. Let C_{xr} represent the set of clauses obtained by resolving clauses in C_x on variable x. NiVER first eliminates variable a from F, by removing C_a from F and adding C_{ar} to F, resulting in new formula F_a. Then NiVER eliminates variable b by deleting C_b from F_a and adding C_{br} to F_a, resulting in F_{ab}. Similarly, eliminating c

results in F_{abc}. Now NiVER terminates and let a SAT solver finds satisfying assignment A_{abc} for F_{abc}. A_{abc} will contain satisfying values for all variables in F_{abc}. Now add variables in the reverse order they were deleted. First add C_c to F_{abc}, resulting in F_{ab}. Assign c either value one or value zero, such that F_{ab} is satisfied. One among the assignments will satisfy F_{ab}. Similarly, add C_b and find a value for b and then for a. During preprocessing, just the set of clauses C_a, C_b and C_c should be stored, so that a satisfying assignment can be obtained if the DPLL SAT solver finds a satisfying assignment for the simplified theory. Fig. 1 shows an example of variable elimination by NiVER. In the example, among nine resolvents, five tautologies are discarded. The variable elimination decreases N by one, K by two, and L by two. In Table 1, we show effect of NiVER on a few instances [9]. For the *fifo8_400* instance, NiVER resulted in 74% decrease in N, 58% decrease in K and 46% decrease in L. Best worst-case upper bounds for general SAT solving in terms of N, K and L are 2^N, $2^{0.30897K}$ and $2^{0.10299L}$, respectively [14]. In many of the real life instances, NiVER decreases all the three values, hence resulting in simpler instances.

Clauses with literal l_{44}
$(l_{44} + l_{6315} + \bar{l}_{15605})$
$(l_{44} + l_{6192} + \bar{l}_{6315})$
$(l_{44} + \bar{l}_{3951} + \bar{l}_{11794})$

Clauses with literal $\bar{l}_{44}$
$(\bar{l}_{44} + l_{6315} + l_{15605})$
$(\bar{l}_{44} + \bar{l}_{6192} + \bar{l}_{6315})$
$(\bar{l}_{44} + \bar{l}_{3951} + l_{11794})$

Old_Num_Lits = 18 **Number of Clauses deleted = 6**

Added Resolvents
$(l_{6315} + \bar{l}_{15605} + \bar{l}_{3951} + l_{11794})$
$(l_{6192} + \bar{l}_{6315} + \bar{l}_{3951} + l_{11794})$
$(\bar{l}_{3951} + \bar{l}_{11794} + l_{6315} + l_{15605})$
$(\bar{l}_{3951} + \bar{l}_{11794} + \bar{l}_{6192} + \bar{l}_{6315})$

Discarded Resolvents(Tautologies)
$(l_{6315} + \bar{l}_{15605} + l_{6315} + l_{15605})$
$(l_{6315} + \bar{l}_{15605} + \bar{l}_{6192} + \bar{l}_{6315})$
$(l_{6192} + \bar{l}_{6315} + l_{6315} + l_{15605})$
$(l_{6192} + \bar{l}_{6315} + \bar{l}_{6192} + \bar{l}_{6315})$
$(\bar{l}_{3951} + \bar{l}_{11794} + \bar{l}_{3951} + l_{11794})$

New_Num_Lits = 16 **Number of Clauses added = 4**

Fig. 1. NiVER Example : Elimination of Variable numbered *44* of 6*pipe* **instance from Microprocessor Verification**

Table 1. Effect of NiVER preprocessing. N-org, N-pre: N in original and simplified formulas. %N-Rem : The percentage of variables removed by NiVER. Corresponding information about clauses are listed in consecutive columns. %K-Dec : The percentage decrease in K due to NiVER. %L-Dec : The percentage decrease in L due to NiVER. The last column reports the CPU time taken by NiVER preprocessor in seconds. Some good entries are in bold.

Benchmark	N-org	N-pre	%N-Rem	K-org	K-pre	%K-Dec	L-org	L-pre	%L-Dec	Time
6 pipe	15800	15067	5	394739	393239	0.4	1157225	1154868	0.2	0.5
f2clk_40	27568	10408	**62**	80439	44302	**45**	234655	157761	**32.8**	1.3
ip50	66131	34393	**48**	214786	148477	**31**	512828	398319	**22.3**	5.2
fifo8_400	259762	68790	**74**	707913	300842	**58**	1601865	858776	**46.4**	14.3
comb2	31933	20238	37	112462	89100	21	274030	230537	**15.9**	1
cache_10	227210	129786	**43**	879754	605614	**31**	2191576	1679937	**23.3**	20.1
longmult15	7807	3629	**54**	24351	16057	**34**	58557	45899	**21.6**	0.2
barrel9	8903	4124	**54**	36606	20973	**43**	102370	66244	**35.2**	0.4
ibm-rule20_k45	90542	46231	**49**	373125	281252	25	939748	832479	11.4	4.5
ibm-rule03_k80	88641	55997	37	375087	307728	18	971866	887363	8.7	3.6
w08_14	120367	69151	**43**	425316	323935	24	1038230	859105	**17.3**	5.45
abp1-1-k31	14809	8183	**45**	48483	34118	**30**	123522	97635	**21.0**	0.44
guidance-1-k56	98746	45111	**54**	307346	193087	**37**	757661	553250	**27.0**	2.74

3 Experimental Results

A Linux machine with AthlonXP1900+ processor and 1GB memory was used in all experiments. The SAT benchmarks are from [9], [10] and [11]. Benchmarks used in [3] were mostly used. The NiVER software is available at [13]. Experiments were done with, Berkmin [5], a complete deterministic SAT solver and Siege(v_4) [12] , a complete randomized SAT Solver. Two SAT solvers have different decision strategies and hence the effect of NiVER on them can be studied. In Table 2 runtimes in CPU seconds for experiments using Berkmin are shown. In Table 3 corresponding runtimes using Siege are tabulated. All experiments using Siege were done with 100 as the random seed parameter. For every benchmark, four types of experiments were done with each solver. The first type is just using the solvers to solve the instance. The second one is using the NiVER preprocessor and solving the simplified theory by the SAT solvers. The third type of experiments involves two preprocessors. First the benchmark is simplified by NiVER and then by HyPre. The output of HyPre is then solved using the SAT solvers. Fourth type of experiments use just HyPre simplifier and the SAT solvers. When preprocessor(s) are used, the reported runtimes are the overall time taken to find satisfiability.

Table 2. Results with Berkmin (Ber) SAT solver. CPU Time (seconds) for four types of experiments, along with class type for each benchmark. N+Ber- NiVER+Berkmin. N+H+Ber - NiVER+HyPre+Berkmin. H+Ber - HyPre+Berkmin. An underlined entry in second column indicates that N+Ber results in better runtime than just using the solver. NSpdUp column lists the speedup due to NiVER+Berkmin over Berkmin

BenchMark	Berkmin	N+Ber	N+H+Ber	H+Ber	Class	(UN)SAT	N-SpdUP
6pipe	**210**	222	392	395	I	UNSAT	0.95
6pipe_6_ooo	276	**253**	738	771	I	UNSAT	**1.09**
7pipe	**729**	734	1165	1295	I	UNSAT	0.99
9vliw_bp_mc	**90**	100	1010	1031	I	UNSAT	0.90
comb2	305	**240**	271	302	II	UNSAT	**1.27**
comb3	817	407	**337**	368	II	UNSAT	**2**
fifo8_ 300	16822	13706	**244**	440	II	UNSAT	**1.23**
fifo8_400	42345	1290	**667**	760	II	UNSAT	**32.82**
ip38	256	99	**52**	105	II	UNSAT	**2.59**
ip50	341	313	**87**	224	II	UNSAT	**1.09**
barrel9	106	39	**34**	114	II	UNSAT	**2.71**
barrel8	368	34	**10**	38	II	UNSAT	**10.82**
ibm-rule20_k30	475	554	**116**	305	II	UNSAT	0.86
ibm-rule20_k35	1064	1527	**310**	478	II	UNSAT	0.70
ibm-rule20_k45	5806	8423	**757**	1611	II	SAT	0.69
ibm-rule03_k70	21470	9438	**399**	637	II	SAT	**2.28**
ibm-rule03_k75	30674	29986	**898**	936	II	SAT	**1.02**
ibm-rule03_k80	31206	58893	1833	**1343**	II	SAT	0.53
abp1-1-k31	1546	3282	1066	**766**	IV	UNSAT	0.47
abp4-1-k31	1640	949	1056	**610**	IV	UNSAT	**1.72**
avg-checker-5-34	1361	1099	**595**	919	II	UNSAT	**1.24**
guidance-1-k56	90755	17736	**14970**	22210	III	UNSAT	**5.17**
w08_14	3657	4379	**1381**	1931	III	SAT	0.84
ooo.tag14.ucl	18	**8**	399	1703	III	UNSAT	**2.25**
cache.inv14.ucl	36	**7**	396	2502	III	UNSAT	**5.14**
cache_05	3430	**1390**	2845	3529	III	SAT	**2.47**
cache_10	22504	55290	**12449**	15212	III	SAT	0.41
f2clk_30	100	61	**29**	53	IV	UNSAT	**1.64**
f2clk_40	2014	1848	1506	**737**	IV	UNSAT	**1.09**
longmult15	183	160	128	**54**	IV	UNSAT	**1.14**
longmult12	283	233	180	**39**	IV	UNSAT	**1.21**
cnt10	4170	2799	193	**134**	IV	SAT	**1.49**

Table 3. Results with Siege (Sie) SAT solver. CPU Time (seconds) for four types of experiments, along with class type for each benchmark. N+Sie- NiVER+Siege. N+H+Sie - NiVER+HyPre+Siege. H+Sie - HyPre+Siege. An underlined entry in second column indicates that N+Sie results in better runtime than just using the solver. NSpdUp column lists the speedup due to NiVER+Siege over Siege

Benchmark	Siege	N+Sie	N+H+Sie	H+Sie	Class	(UN)SAT	N-SpdUP
6 pipe	79	**70**	360	361	I	UNSAT	**1.13**
6pipe_6_ooo	187	**156**	743	800	I	UNSAT	**1.20**
7pipe	185	**177**	1095	1183	I	UNSAT	**1.05**
9vliw_bp_mc	52	**46**	975	1014	I	UNSAT	**1.14**
comb2	407	266	**257**	287	II	UNSAT	**1.53**
comb3	550	419	396	**366**	II	UNSAT	**1.31**
fifo8_ 300	519	310	**229**	281	II	UNSAT	**1.68**
fifo8_400	882	657	**404**	920	II	UNSAT	**1.34**
ip38	146	117	**85**	115	II	UNSAT	**1.25**
ip50	405	258	**131**	234	II	UNSAT	**1.57**
barrel9	59	**12**	16	54	II	UNSAT	**4.92**
barrel8	173	25	**6**	16	II	UNSAT	**6.92**
ibm-rule20_k30	216	131	**112**	315	II	UNSAT	**1.65**
ibm-rule20_k35	294	352	**267**	482	II	UNSAT	0.84
ibm-rule20_k45	1537	1422	1308	**827**	II	SAT	**1.08**
ibm-rule03_k70	369	360	**223**	516	II	SAT	**1.03**
ibm-rule03_k75	757	**492**	502	533	II	SAT	**1.54**
ibm-rule03_k80	946	781	**653**	883	II	SAT	**1.21**
abp1-1-k31	559	471	**281**	429	II	UNSAT	**1.19**
abp4-1-k31	455	489	**303**	346	II	UNSAT	0.93
avg-checker-5-34	619	621	**548**	690	II	UNSAT	1
guidance-1-k56	9972	8678	**6887**	20478	II	UNSAT	**1.15**
w08_14	1251	**901**	1365	1931	III	SAT	**1.39**
ooo.tag14.ucl	15	**6**	396	1703	III	UNSAT	**2.5**
cache.inv14.ucl	39	**13**	396	2503	III	UNSAT	**3**
cache_05	238	**124**	2805	3540	III	SAT	**1.92**
cache_10	1373	**669**	10130	13053	III	SAT	**2.05**
f2clk_30	70	48	53	**41**	IV	UNSAT	**1.46**
f2clk_40	891	988	802	**519**	IV	UNSAT	0.90
longmult15	325	198	169	**54**	IV	UNSAT	**1.64**
longmult12	471	256	292	**72**	IV	UNSAT	**1.84**
cnt10	236	139	193	**134**	IV	SAT	**1.70**

Based on the experimental results in two tables, we classify the SAT instances into four classes. Class-I: Instances for which preprocessing results in no significant improvement. Class-II: Instances for which NiVER+HyPre preprocessing results in best runtimes. Class-III: Instances for which NiVER preprocessing results in best runtimes. Class-IV: Instances for which HyPre preprocessing results in best runtimes. The sixth column in the tables lists the class to which each problem belongs. When using SAT solvers to solve problems from a particular domain, samples from the domain can be used to classify them into one of the four classes. After classification, the corresponding type of framework can be used to get better run times. In case of Class-I problems, NiVER results are almost same as the pure SAT solver results. But HyPre takes a lot of time for preprocessing some of the Class-I problems like pipe instances. There are several Class-I problems not listed in tables here, for which neither NiVER nor HyPre results in any simplification, and hence no overhead. In case of Class-II problems, NiVER removes many variables and results in a simplified theory F_N. HyPre further simplifies F_N and results in F_{N+H} which is easier for SAT solvers. When HyPre is alone used for Class-II problems, they simplify well, but the simplification process takes more time than for simplifying corresponding F_N. NiVER removes many variables and results in F_N. But the cost of reducing the same variables by comparatively complex procedures in HyPre is very high. Hence, for Class-II, with few exceptions, H+Solver column values are more than the values in N+H+Solver column. For Class-III problems, HyPre takes a lot of time to preprocess instances, which increases the total time taken to solve by many magnitudes than the normal solving time. In case of *cache.inv14.ucl* [11], N+Sie takes 13 seconds to solve, while H+Sie takes 2503 seconds. The performance of HyPre is similar to that on other benchmarks generated by an infinite state systems verification tool [11]. Those benchmarks are trivial for DPLL SAT Solvers. The Class-IV problems are very special cases in which HyPre outperform others. When NiVER is applied to these problems, it destroys the structure of binary clauses in the formula. HyPre which relies on hyper binary resolution does not perform well on the formula simplified by NiVER. In case of *longmult15* and *cnt10*, the HyPre preprocessor itself solves the problem. When just the first two types of experiments are considered, NiVER performs better in almost all of the instances.

4 Conclusion

We have shown that a special case of VER, NiVER, is an efficient simplifier. Although several simplifiers have been proposed, the state-of-the-art SAT solvers do not use complex simplification steps. We believe that efficient simplifiers will improve SAT solvers. NiVER does the VER space efficiently by not allowing space increasing resolutions. Otherwise, the advantage of VER will be annulled by the associated space explosion. Empirical results have shown that NiVER results in improvement in most of the cases. NiVER+Berkmin outperforms Berkmin in 22 out of 32 cases and gives up to 33x speedup. In the other cases, mostly the difference is negligible. NiVER+Siege outperforms Siege in 29 out of 32 cases and gives up to 7x speedup. In the three other cases, the difference is negligible. Although, NiVER results in easier problems in terms of the three worst case upper bounds, the poor performance of SAT solvers on few NiVER simplified instances is due to the decision heuristics. The NiVER simplifier performs well as most of the best runtimes in the experiments are obtained using it. Due to its consistent performance, like decision heuristics and clause learning, NiVER can also be incorporated into all general purpose DPLL SAT solvers.

Acknowledgements

Special thanks to Tom Morrisette, Lintao Zhang, Allen Van Gelder, Rune M Jensen and the anonymous reviewers for their comments on earlier versions of this paper.

References

1. M. Davis, H. Putnam. : A Computing procedure for quantification theory. J. of the ACM,**7** (1960)
2. M. Davis, et.al.,: A machine program for theorem proving. Comm. of ACM, **5(7)** (1962)
3. F. Bachhus, J. Winter. : Effective preprocessing with Hyper-Resolution and Equality Reduction, SAT 2003 341-355
4. R. I. Brafman : A simplifier for propositional formulas with many binary clauses, IJCAI 2001, 515-522.
5. E.Goldberg, Y.Novikov.: BerkMin: a Fast and Robust SAT-Solver, Proc. of DATE 2002, 142-149
6. M. Moskewicz, et.al.,: Chaff: Engineering an efficient SAT solver, Proc. of DAC 2001
7. J. Franco. : Elimination of infrequent variables improves average case performance of satisfiability algorithms. SIAM Journal on Computing **20** (1991) 1119-1127.
8. A. Van Gelder. : Combining preorder and postorder resolution in a satisfiability solver, In Kautz, H., and Selman, B., eds., Electronic Notes of SAT 2001, Elsevier.
9. H. Hoos, T. Stützle.: SATLIB: An Online Resource for Research on SAT. In: I.P.Gent, H.v.Maaren, T.Walsh, editors, SAT 2000, 283-292, www.satlib.org
10. IBM Formal Verification Benchmarks Library : `http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html`
11. UCLID : `http://www-2.cs.cmu.edu/~uclid/`
12. L. Ryan : Siege SAT Solver : `http://www.cs.sfu.ca/~loryan/personal/`
13. NiVER SAT Preprocessor : `http://www.itu.dk/people/sathi/niver.html`
14. E. D. Hirsch. : New Worst-Case Upper Bounds for SAT, J. of Automated Reasoning **24** (2000) 397-420