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 two 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. In the experiments, NiVER resulted in up to 74% decrease in N (Number of variables), 58% decrease in K (Number of clauses) and 46% decrease in L (Literal count). In many real-life instances, we observed that most of the resolvents for several variables are tautologies. Such variables are removed by NiVER. Hence, despite its simplicity, NiVER does result in easier instances. In case NiVER removable variables are not present, due to very low overhead, the cost of NiVER is insignificant. Empirical results using the state-of-the-art SAT solvers show the usefulness of NiVER. Some instances cannot be solved without NiVER preprocessing. NiVER consistently performs well and hence, can be incorporated into all general purpose SAT solvers.

1 Introduction

The Variable Elimination Resolution (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. Preprocessing is worthwhile only if the overall time taken for simplification as well as for solving the simplified formula is less than the time required to solve the unsimplified formula. This paper introduces NiVER (Non-increasing VER), a new preprocessor based on VER. NiVER is a limited version of VER, which resolves away a variable only if

* Research reported supported in part by EPSRC(UK). Most of this work was done when the first author was working at University of Bristol.

H.H. Hoos and D.G. Mitchell (Eds.): SAT 2004, LNCS 3542, pp. 276–291, 2005.
© Springer-Verlag Berlin Heidelberg 2005

there will be no resulting increase in space. For several instances, NiVER results in reducing the overall runtime. In many cases, NiVER takes less than one second CPU time. Because, NiVER consistently performs well, like clause learning and decision heuristics, NiVER can also be integrated into the DPLL framework for general purpose SAT solvers.

The important contribution of this paper is the observation that most of the real-life SAT instances have several NiVER removable variables, which can be resolved away without increase in space. The structure of the real-life instances are such that for several variables, most of the resolvents are tautologies. Hence, there will be no increase in space due to VER on them. Empirical results show that NiVER not only decreases the time taken to solve an instance, but also decreases the amount of memory needed to solve it. Some of the instances cannot be solved without NiVER preprocessing, because of more memory requirement. Importantly, empirical results also show that the benefit of NiVER preprocessing increases with the increase in the size of the problem. As the size of the problem instances increases, it is useful to have a NiVER preprocessor in general SAT solvers. Another advantage is its simplicity: NiVER can be implemented with the conventional data structures used for clause representation in modern SAT solvers. In fact, in our implementation we have used the data structures from the zChaff SAT solver [6]. Hence, there will not be a significant overhead in implementation of NiVER with SAT solvers.

The next section presents a brief overview of previous SAT preprocessing techniques. Section 3 presents the NiVER preprocessor and gives some examples. Empirical results are presented in section 4. We conclude in section 5.

2 SAT Preprocessors

Simplifiers are used to change the formula into a simpler one, which is easier to solve. 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], the current state-of-the-art SAT solvers [6],[5], just use these two simplifications. The 2-simplify preprocessor 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. The recent preprocessor, HyPre [3] applies all the rules in 2-simplify and also does hyper-binary resolution. For some problems HyPre preprocessor itself solves the problem. But for other instances, it takes a lot of time to preprocess, while the original problem is easily solvable by SAT solvers.

VER has already been used as a simplifier, but to a lesser extent. In [7], variables with two occurences are resolved away. For a class of random benchmarks, [7] has empirically shown that the 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 a 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 the state-

of-the-art DPLL algorithms. A variant of the NiVER method, which does not allow an 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, at each node of a DPLL search. However, no implementation was found.

Algorithm 1. NiVER CNF Preprocessor

1: NiVER(F)
2: **repeat**
3: $entry = FALSE$
4: **for all** $V \in \text{Var}(F)$ **do**
5: $P_C = \{\text{C} \mid \text{C} \in F, l_V \in \text{C}\}$
6: $N_C = \{\text{C} \mid \text{C} \in F, \bar{l}_V \in \text{C}\}$
7: $R = \{\ \}$
8: **for all** $P \in P_C$ **do**
9: **for all** $N \in N_C$ **do**
10: $R = R \cup \text{Resolve}(P,N)$
11: **end for**
12: **end for**
13: Old_Num_Lits = Number of Literals in $(P_C \cup N_C)$
14: New_Num_Lits = Number of Literals in R
15: **if** (Old_Num_Lits $\geq$ New_Num_Lits) **then**
16: F=F-$(P_C \cup N_C)$, F=F+R, $entry = TRUE$
17: **end if**
18: **end for**
19: **until** $\neg$entry
20: return F

3 NiVER: Non-increasing VER

Like other simplifiers, NiVER takes a CNF as input and outputs another CNF, with a lesser or equal number of variables. The VER, the original algorithm for SAT solving, has exponential space complexity, while that of DPLL is linear. Both have exponential time complexity. In NiVER, as we do not allow space increasing resolutions, we have linear space complexity. The strategy we use is to simplify the SAT instance as much as possible using NiVER, a linear-space version of VER. Then, the resulting problem is solved using a DPLL-based SAT solver. 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 increasing L. If so, it eliminates the variable by VER. The NiVER procedure is shown in Algorithm 1. When VER removes a variable, all resolvents of the variable have to be added. We discard trivial resolvents (tautologies). The rest of the resolvents are added to the formula. Then, all the clauses containing the variable are deleted from the formula. In many real-life instances (Figure 1 and Figure 2) we observed that for many variables, most of the resolvents are tautologies. So, there will be no increase in space when those

variables are resolved away. Apart from checking for tautologies, 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 become removable. NiVER iterates until no more variable can be removed. In the present implementation, NiVER does not even check whether any unit clause is present. 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.

Clauses with literal l_{24}
$(\bar{l}_{23} + l_{24})$ $(\bar{l}_{22} + l_{24})$ $(l_{24} + \bar{l}_{31})$
$(l_2 + \bar{l}_{15} + l_{24})$ $(\bar{l}_2 + l_{15} + l_{24})$

Clauses with literal $\bar{l}_{24}$
$(l_{22} + l_{23} + \bar{l}_{24} + l_2 + l_{15})$
$(l_{22} + l_{23} + \bar{l}_{24} + \bar{l}_2 + \bar{l}_{15})$

Old_Num_Lits = 22 **Number of Clauses deleted = 7**

Added Resolvents
$(\bar{l}_{31} + l_{22} + l_{23} + l_2 + l_{15})$ $(\bar{l}_{31} + l_{22} + l_{23} + \bar{l}_2 + \bar{l}_{15})$
Eight other resolvents are tautologies

New_Num_Lits = 10 **Number of Clauses added = 2**

Fig. 1. NiVER Example 1: Elimination of Variable numbered *24* of *barrel*8 instance from Bounded Model Checking

Figure 1 shows an example of variable elimination by NiVER, when applied to the *barrel*8 instance from Bounded Model Checking [15]. In this example, variable *24* has 10 resolvents. Among them, eight are tautologies, which can be discarded. Only the two remaining non-trivial resolvents are added. The seven old clauses containing a literal of variable *24* are deleted. The variable elimination decreases N (number of variables) by one, K (number of clauses) by 12, and L (literal count) by five.

Figure 2 shows another example of variable elimination by NiVER, when applied to the 6*pipe* instance from a microprocessor verification benchmark suite[16]. In this example, variable *44* has nine resolvents. Among them, five are tautologies, which can be discarded. Only the four remaining non-trivial resolvents are added. The six old clauses containing a literal of variable *44* are deleted. The variable elimination decreases N by one, K by two, and L by two.

Table 1 shows the effect of NiVER on a few instances from [9]. For the *fifo8_400* instance, NiVER resulted in a 74% decrease in N, a 58% decrease in K and a 46% decrease in L. The benefit of these reductions is shown in the results section. In many of the real-life instances, NiVER decreases N, K and L. This might be a reason for the usefulness of NiVER on those instances.

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

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

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

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

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

Fig. 2. NiVER Example 2: Elimination of Variable numbered *44* of 6*pipe* instance from Microprocessor Verification

Table 1. Effect of NiVER preprocessing. N-org, N-pre: N (Number of variables) in original and simplified formulas. %N↓ : The percentage of variables removed by NiVER. Corresponding information about clauses are listed in consecutive columns. %K↓ : The percentage decreases in K due to NiVER. %L↓ : The percentage decreases in L (Number of Literals) due to NiVER. The last column reports the CPU time taken by NiVER preprocessor in seconds. Some good entries are in bold. A machine with AthlonXP1900+ processor and 1GB memory was used in the experiments

Benchmark	N-org	N-pre	%N↓	K-org	K-pre	%K↓	L-org	L-pre	%L↓	Time
6pipe	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

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 the way NiVER proceeds is simply reversed. The variables are added back in the reverse order in which 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 refer to a 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 the 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 a 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 to c either the value one or the value zero, such that F_{ab} is satisfied. At least one among those 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.

4 Experimental Results

This section contains two subsections. The subsection 4.1 presents the effect of the NiVER Preprocessor on two state-of-the-art SAT solvers: Berkmin and Siege. As the two SAT solvers have different decision strategies, the effect of NiVER on them can be studied. The other subsection presents the effect of NiVER on time and memory requirement for solving large instances from four families. The time listed for NiVER preprocessing in the tables are just the time taken for preprocessing the instance. At present, as NiVER is a separate tool, and, if the instance is *very large*, it might take few additional seconds to read the file and write back into the disk. But, as the data structures used in NiVER implementation are those used in modern SAT solvers to represent clauses, the time taken for reading and writing back can be avoided when integrated with the SAT solver.

4.1 Effect of NiVER on SAT-Solvers: BerkMin and Siege

The SAT benchmarks used in this subsection 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. A fourth type of experiment uses just HyPre simplifier

Table 2. Results with Berkmin (Ber) SAT solver. CPU Time (seconds) for four types of experiments, along with class type (Cls) for each benchmark. An underlined entry in the second column indicates that NiVER+Berkmin results in better runtime than just using the solver. NSpdUp column lists the speedup due to NiVER+Berkmin over Berkmin. A machine with AthlonXP1900+ processor and 1GB memory was used in the experiments

BenchMark	**Berkmin**	**Berkmin with**			**Cls**	**(UN)SAT**	**NSpdUP**
		NiVER	**NiVER +HyPre**	**HyPre**			
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**

and the SAT-solvers. When preprocessor(s) are used, the reported runtimes are the overall time taken to find satisfiability.

Based on the experimental results in these two tables of this subsection, we classify the SAT instances into four classes. Class-I: Instances for which pre-

Table 3. Results with Siege (Sie) SAT solver. A machine with AthlonXP1900+ processor and 1GB memory was used in the experiments

BenchMark	Siege	Siege with			Cls	(UN)SAT	NSpdUP
		NiVER	NiVER +HyPre	HyPre			
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**

processing 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 significant 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, HyPre+Solver column values are more than the values in NiVER+HyPre+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], NiVER+Siege takes 13 seconds to solve, while HyPre+Siege 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.

4.2 Effect of NiVER on Time and Memory Usage by Siege SAT Solver

In this subsection the effect of the NiVER preprocessor on four families of large SAT instances are presented. As the results in the previous subsection show, Siege SAT solver is better than Berkmin. As we are primarily interested in studying the effect of NiVER on memory and time requirements of SAT solvers, all the results in this section are done using the Siege SAT solver, alone. Again, the random seed parameter for the Siege SAT solver was fixed at 100.

In the tables in this subsection: *NiVER Time (sec)* refers to the CPU-time in seconds taken by NiVER to preprocess the benchmark. *Time (sec)-Siege* refers to the CPU-time in seconds taken by Siege to solve the original benchmark. *Time (sec)-NiVER+Siege* refers to the sum of the time taken for NiVER preprocessing and the time taken for solving the NiVER-preprocessed instance by Siege. *Time (sec)-HyPre+Siege* refers to the sum of the time taken for HyPre preprocessing and the time taken for solving the HyPre-preprocessed instance by Siege. The other three columns list the amount of memory, in MegaBytes (MB), used by Siege SAT solver to solve the original and the corresponding preprocessed instances. *MA-S* means memory-abort by Siege SAT solver. *MA-H* means memory-abort by HyPre preprocessor. *SF* mentions segmentation fault by HyPre preprocessor.

The *x-x-barrel* family of instances are generated using the tools, *BMC* and *genbarrel*, available at [15]. The *BMC* tool, bounded model checker, takes as input: a model and a parameter, *k*, the bound for which the model should be verified. It then creates a SAT formula, whose unsatisfiability implies the verification of a property for the specified bound. The *genbarrel* tool takes an integer parameter and generates a model of the corresponding size. The *x-x-barrel* instances are generated using the command: *genbarrel x | bmc -dimacs -k x* > *x-x-barrel.cnf*

The results for the *x-x-barrel* family are listed in Table 4. Three types of experiments are done. One just using the plain Siege SAT solver. Second one using the NiVER preprocessor. Third one using the HyPre preprocessor. Correponding time and memory usage are listed. Figure 3 shows the graph obtained when the time usage for *x-x-barrel* instances are plotted. Figure 4 shows the graph obtained when the memory usage for *x-x-barrel* instances are plotted. The HyPre preprocessor was not able to handle instances larger than the *10-10-barrel*, and aborted with a segmentation fault error message. Although the Siege SAT solver was able to solve all the original *x-x-barrel* instances, it used more time and space, than those for the corresponding NiVER simplified instances. As the Figure 3 shows, with the increase in the size of the instance, the benefit of NiVER preprocessing increases. For example, the speed-up due to NiVER in case of *8-8-barrel* instance is 6.1 (Table3), while it is 13.6 in case of *14-14-barrel* instance. Similar trend is also observed in the graph (Figure 4) for memory usage comparison. More than half of the variables in *x-x-barrel* instances are resolved away by NiVER preprocessor. In several other bounded model checking generated SAT instances, we observed similar amount of decrease in the number of variables due to NiVER preprocessing. For example, the *longmult15* instance and *w08_14* instance in Table 1 are both generated by bounded model checking. In both cases, approximately half of the variables are removed by NiVER. In all the *x-x-barrel* instances, the time taken for NiVER preprocessing is very small, and insignificant, when compared with the original solution time.

The other three families of large SAT instances: *xpipe_k*, *xpipe_q0_k*, and, *xpipe_xooo_q0_T0*, are all obtained from the Microprocessor Verification bench-

Table 4. Results for *x-x-barrel* family. Experiments were done on an Intel Xeon 2.8 GHz machine with 2 GB of memory. *SF*: Segmentation fault by HyPre preprocessor

Benchmark	**N**	**%N↓**	**NiVER Time (sec)**	**Time (sec)**			**Memory (MB)**		
				Siege	**NiVER +Siege**	**HyPre +Siege**	**Siege**	**NiVER +Siege**	**HyPre +Siege**
8-8-barrel	5106	56	0.2	86	14	**9**	59	26	**13**
9-9-barrel	8903	53	0.4	29	**7**	25	20	**8**	41
10-10-barrel	11982	54	1	45	**10**	65	31	**12**	63
11-11-barrel	15699	54	1	77	**12**	SF	52	**14**	SF
12-12-barrel	20114	55	2	580	**73**	SF	147	**50**	SF
13-13-barrel	25287	55	3	429	**42**	SF	120	**34**	SF
14-14-barrel	31278	55	5	1307	**96**	SF	208	**70**	SF

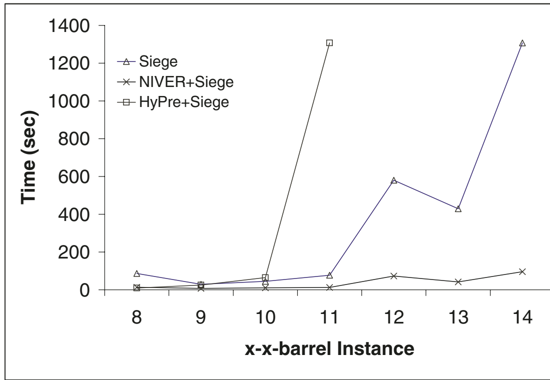

Fig. 3. Time comparison for the *x-x-barrel* family

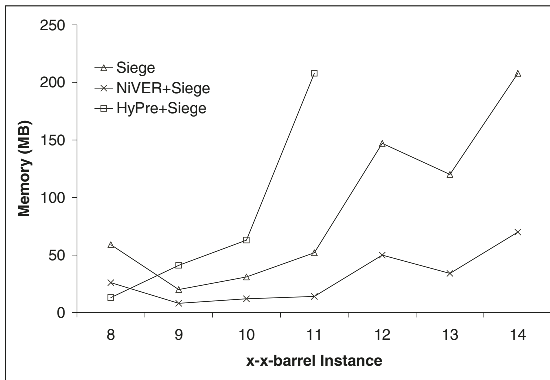

Fig. 4. Memory usage comparison for the *x-x-barrel* family

mark suite at [16]. The results obtained from experiments on *xpipe_k*, *xpipe_q0_k*, and, *xpipe_xooo_q0_T0* families are listed in Tables 5, 6, and 7, respectively. Figures for corresponding time and memory comparison follow them.

In the case of the *xpipe_k* family, without using the NiVER preprocessor, all the instances cannot be solved. Memory values listed in the tables just show the amount of memory used by the Siege SAT solver. Even when preprocessors are used, the memory values listed are just those used by Siege for solving the corresponding preprocessed instance. The amount of memory used by preprocessors is not listed. The aborts (MA-H) in the HyPre column of Table 5 are due to HyPre. In a machine with 2GB of memory, the HyPre preprocessor was not able to handle instances larger than *10pipe_k*. Even for the instances smaller than *10pipe_k*, HyPre took a lot of time. In case of *9pipe_k*, HyPre took 47 times the time taken by Siege. Siege SAT solver was not able to handle the *14pipe_k*

Table 5. Results for the *xpipe_k* family. Experiments were done on an Intel Xeon 2.8 GHz machine with 2 GB of memory. *MA-S*: Memory Abort by Siege SAT solver. *MA-H*: Memory Abort by HyPre preprocessor

Benchmark	N	%N↓	NiVER Time (sec)	Time (sec)			Memory (MB)		
				Siege	NiVER +Siege	HyPre +Siege	Siege	NiVER +Siege	HyPre +Siege
7pipe_k	23909	4	1	**64**	69	361	93	94	**61**
8pipe_k	35065	5	1	144	**127**	947	152	115	**87**
9pipe_k	49112	3	1	109	**106**	4709	**121**	123	**121**
10pipe_k	67300	5	2	565	**544**	5695	308	212	**196**
11pipe_k	89315	5	2	1183	**915**	MA-H	443	**296**	MA-H
12pipe_k	115915	5	3	3325	**2170**	MA-H	670	**418**	MA-H
13pipe_k	147626	5	4	5276	**3639**	MA-H	842	**579**	MA-H
14pipe_k	184980	5	5	MA-S	**8559**	MA-H	MA-S	**730**	MA-H

Table 6. Results for the *xpipe_q0_k* family. Experiments were done on an Intel Xeon 2.8 GHz machine with 2 GB of memory. *MA-H*: Memory Abort by HyPre preprocessor

Benchmark	N	%N↓	NiVER Time (sec)	Time (sec)			Memory (MB)		
				Siege	NiVER +Siege	HyPre +Siege	Siege	NiVER +Siege	HyPre +Siege
8pipe_q0_k	39434	27	1	90	**68**	304	81	62	**56**
9pipe_q0_k	55996	28	2	71	**61**	1337	77	**66**	76
10pipe_q0_k	77639	29	2	295	**280**	1116	170	120	**117**
11pipe_q0_k	104244	31	3	520	**478**	1913	233	164	**154**
12pipe_q0_k	136800	32	4	1060	**873**	3368	351	227	**225**
13pipe_q0_k	176066	33	5	1656	**1472**	5747	481	**295**	315
14pipe_q0_k	222845	34	7	2797	**3751**	MA-H	616	**412**	MA-H
15pipe_q0_k	277976	35	10	5653	**4165**	MA-H	826	**494**	MA-H

Table 7. Results for the *xpipe_x_ooo_q0_T0* family. Experiments were done on an Intel Xeon 2.8 GHz machine with 2 GB of memory. In this table an entry x in the Benchmark column refers to an *xpipe_x_ooo_q0_T0* instance

Benchmark	N	%N↓	NiVER Time (sec)	Time (sec)			Memory (MB)		
				Siege	NiVER +Siege	HyPre +Siege	Siege	NiVER +Siege	HyPre +Siege
7	27846	24	1	97	**87**	229	60	60	**45**
8	41491	25	1	252	**226**	605	100	85	**77**
9	59024	25	1	415	**359**	1524	135	119	**113**
10	81932	27	2	**2123**	2391	3190	215	172	**171**
11	110150	28	2	9917	**8007**	8313	317	240	**234**
12	144721	29	3	56748	30392	**27071**	448	**330**	342

instance. It aborted due to insufficient memory. But after NiVER preprocessing the *14pipe_k* instance was solved by Siege using the same machine. While solving NiVER preprocessed instance, Siege consumed only one third (730MB) of the available 2GB. As the results in Table 5 and the corresponding time comparison graph (Figure 5) shows, the speed-up due to NiVER keeps increasing with the increase in size of the instance. It is also interesting to note that in case of *14pipe_k* instance, only 5% of the variables are removed by NiVER. But still it resulted in solving the instance by using just 730MB of memory. As shown in Figure 6, the memory usage ratio, between the original instance and the corresponding NiVER preprocessed instance, also increases with the increase in the instance size.

In the case of the *xpipe_q0_k* family, again HyPre was not able to handle all the instances. As in the other cases, the benefit of NiVER preprocessing increases with the increase in the size of the instances. This is shown by the graphs in

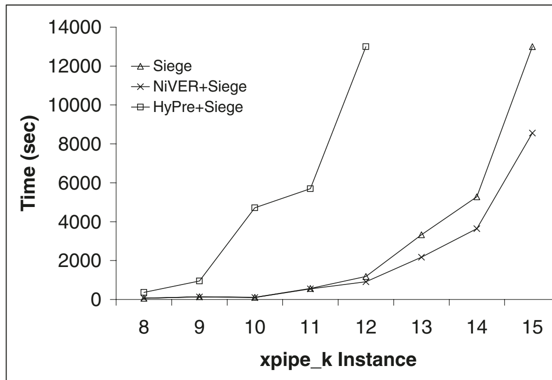

Fig. 5. Time comparison for the *xpipe_k* family

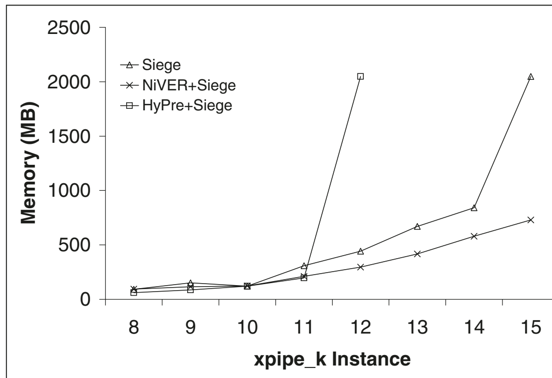

Fig. 6. Memory usage comparison for the *xpipe_k* family

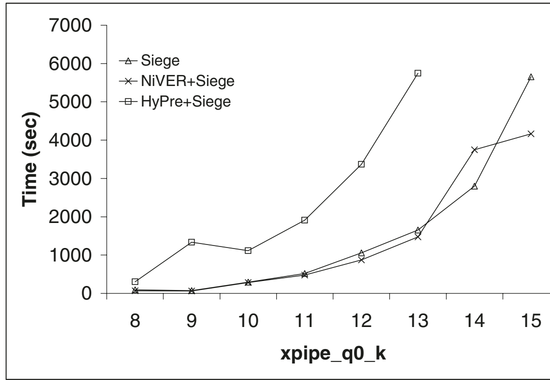

Fig. 7. Time comparison for the *xpipe_q0_k*family

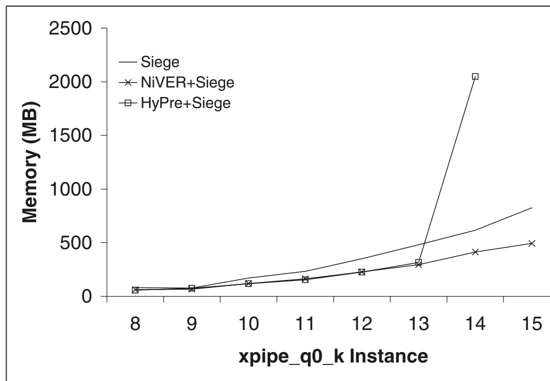

Fig. 8. Memory usage comparison for the *xpipe_q0_k* family

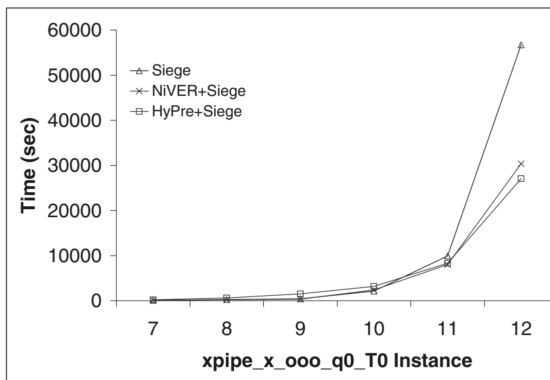

Fig. 9. Time comparison for the *xpipe_x_ooo_q0_T0* family

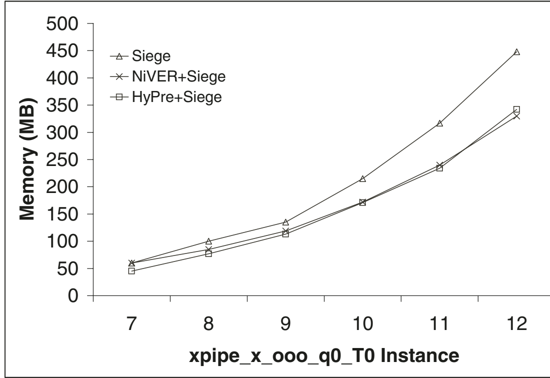

Fig. 10. Memory usage comparison for the *xpipe_x_ooo_q0_T0* family

Figures 7 and 8. For the *15pipe_q0_k* instance, NiVER resulted in decreasing the memory usage by 40%. Even in time usage, there is a significant improvement. In the case of the *xpipe_x_ooo_q0_T0* family (Table 7), both NiVER and HyPre preprocessors give similar improvement over the original instance. But still the advantages of NiVER over HyPre are its simplicity, and usefulness in a wide range of real-life instances.

5 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 simplification steps other than unit propogation and pure literal elimination. 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 would 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 (Table 2) and yields up to 33x speedup. In the other cases, mostly the difference is negligible. NiVER+Siege outperforms Siege in 29 out of 32 cases (Table 3) and gives up to 7x speedup. In the three other cases, the difference is negligible. Although, NiVER results in easier problems when some variables are removed by it, the poor performance of SAT solvers on few NiVER simplified instances is due to the decision heuristics. Experiments on four families of large SAT instances show that, the usefulness of NiVER increases with the increase in size of the problem size. NiVER also decreases the memory usage by SAT solvers. Due to that, more instances can be solved with the same machine configuration. The NiVER simplifier performs well, as most of the best runtimes in the experiments are obtained using it. Due to its usefulness and simplicity, 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
15. A. Biere: BMC, `http://www-2.cs.cmu.edu/~modelcheck/bmc.html`
16. M. N. Velev: Microprocessor Benchmarks, `http://www.ece.cmu.edu/~mvelev/sat_benchmarks.html`