

Tuning the VSIDS Decision Heuristic for Bounded Model Checking

Ohad Shacham
IBM Haifa Research Laboratory
University Campus,
Carmel Mountains, Haifa, 31905
ohads@il.ibm.com

Emmanuel Zarpas
IBM Haifa Research Laboratory
University Campus,
Carmel Mountains, Haifa, 31905
zarpas@il.ibm.com

Abstract

Bounded Model Checking (BMC) techniques have been used for formal hardware verification, with the help of tools such as GRASP (Generic search Algorithm for Satisfiability Problem) and more recently zchaff. In order to cope with very large hardware designs, our work exploited the unique characteristics of bounded model checking to enhance the SAT algorithms used to solve our problems. In our work, we tuned the VSIDS (Variable State Independent Decaying Sum) decision heuristics embedded in zchaff [5], in order to improve the efficiency of the DPLL SAT algorithm, which is especially effective for BMC problems. We also checked whether the conclusions reached by Strichman [6] regarding the tuning of GRASP, are also appropriate and hold true for zchaff. Our experimental results on actual hardware designs prove, with a few exceptions, that there is no real improvement when the existing tuned algorithms are applied to zchaff. However, our further modifications to the tuning proved to significantly increase SAT efficiency for BMC problems.

1. Introduction

The use of Propositional Satisfiability SAT methods for hardware verification by symbolic model checking was introduced in the framework of Bounded Model Checking [1]. Since then, it has enjoyed success in the formal verification community [2].

Optimizations have already been implemented on top of CMU BMC [1] and GRASP [7] in order to tune SAT solvers for bounded model checking [6]. We reproduced the main optimization (i.e., static ordering) of [6] on the top of the zchaff SAT solver [5] and benchmarked it on an internal IBM benchmark from real-life bounded model checking problems. The results show tremendous variations; sometimes this heuristic behaves extremely well - more

than one order of magnitude faster than regular zchaff - but sometimes extremely poorly - several order of magnitudes slower. We then focused on tuning the VSIDS [5] decision heuristic on top of zchaff and obtained significantly better results than those found using the original zchaff heuristic.

2. Bounded Model Checking (BMC)

BMC translates a safety formula from ACTL* (the subset of CTL* [4] that contains only universal path quantifiers) into a propositional formula under bounded semantics. The general structure of an **AG** P (safety) formula, as generated in BMC, is as follows:

$$\phi : I_0 \wedge \bigwedge_{i=0}^{k-1} \rho(i, i+1) \wedge \left(\bigvee_{i=0}^k \neg P_i \right)$$

where I_0 is the initial state, $\rho(i, i+1)$ is the transition between cycles i and $i+1$, and P_i is the property in cycle i .

If this propositional formula is proven to be satisfiable, the satisfying assignment provided by the SAT solver is a counter example to the property AGP . To convert the initial propositional formula into Conjunctive Normal Form (used as an input format by most SAT solvers), extra variables are introduced to avoid combinatory explosion. Usually, these extra variables represent more than 80% of the total number of variables in the CNF formula.

3. SAT Basics

SAT is the problem of determining the Satisfiability of a Boolean formula. The problem was first reconized as NP-complete in 1971 by Cook [3]. Today, many implementations are available for solving the problem, such as Grasp and zchaff. Most of them are based on the complete DPLL algorithm, represented as follows:

$$\text{DPLL}(f, \mathbf{x}) \{ \\ f' = \text{UnitRes}(f, \mathbf{x})$$

```

    f'' = UnitSub(f', x)
if (!(x=decide(f'')))
    return SATisfiabLe
    DPLL(f'', x)
    DPLL(f'', !x)
return UNSATisfiabLe}

```

UnitRes(f,x) is a function that perform Unit resolution for a Boolean formula f and a literal x, and returns the new formula.

UnitSub(f, x) is a function that perform Unit subsumption for a Boolean formula f and a literal x, and returns the new formula.

Decide() is a function that chooses the next variable according to which branching will occur. There are many heuristic for choosing this next variable, such as DLIS(Dynamic Largest Individual Sum) and VSIDS (Variable State Independent Decaying Sum). zchaff's decision heuristic is VSIDS. Our work is focus on tuning this heuristic for BMC.

4. CNF Translation

Most SAT solvers require their input in CNF (Conjunctive Normal Form) format. Hence, the propositional formula generated during the BMC procedure must be converted to this format. In order to avoid exponential explosion, extra variables are introduced. In many cases, these variables represent more than 80% of the total number of variables in the CNF formula. Let us take the following formula in DNF format:

$$(a \ \& \ b \ \& \ c) | (c \ \& \ d) | (b \ \& \ d \ \& \ e) | (f \ \& \ g \ \& \ a)$$

where a,b, are literals.

Translation of this formula to CNF format creates a formula which is exponentially bigger than the original one. To avoid this exponential blowup, we use the auxiliary variables x_1, x_2, x_3 and x_4 . The translation will occur in the following manner:

$$\begin{aligned}
& (x_1 | x_2 | x_3 | x_4) \\
& \& (x_1 \rightarrow (a \ \& \ b \ \& \ c)) \\
& \& ((a \ \& \ b \ \& \ c) \rightarrow x_1) \\
& \& (x_2 \rightarrow (c \ \& \ d)) \\
& \& ((c \ \& \ d) \rightarrow x_2) \\
& \& (x_3 \rightarrow (b \ \& \ d \ \& \ e)) \\
& \& ((b \ \& \ d \ \& \ e) \rightarrow x_3) \\
& \& (x_4 \rightarrow (f \ \& \ g \ \& \ a)) \\
& \& ((f \ \& \ g \ \& \ a) \rightarrow x_4)
\end{aligned}$$

Using De Morgan rules, it is straightforward to convert this formula into CNF without experiencing size explosion.

Every environment that satisfies the CNF formula also satisfies the original formula, but the opposite does not hold.

Note that assigning a value to the original variables (dominant variables) can easily lead to assignment of the auxiliary variables that satisfy the CNF formula.

5. Tuning VSIDS

The original zchaff decision VSIDS strategy is as follows:

1. Each variable in each polarity has a counter, which is initialized to 0.
2. When a clause is added (by learning) to the database, the counter associated with each literal in the clause is incremented.
3. The (unassigned) variables and polarity with the highest counter are chosen at each decision.
4. Periodically, all counters are divided by a constant.

One of Shtrichman's [6] main idea is as follows: in the Davis-Putnam decision procedure, the variable of the initial propositional formula is used first and in a specific static order. This static order is determined by a special Breadth First Search of the (k-unfolding of the) Variable Dependency Graph; the search starts from the set $\bigcup_{0 \leq i \leq k} \neg P_i$.

We chose to implement several decision heuristics on top of zchaff. We tuned the VSIDS decision heuristic in zchaff in different ways: first to reflect [6] idea of static order, and second, since it was unclear from [6] if the improvements were due to the static order or to the priority given to the dominant variables (*i. e.*, the variables from the initial propositional formula before the conversion to CNF, or domain variables), to implement a heuristic that gives priority to dominant variables over any other variables but otherwise relies on zchaff decision. Since zchaff uses the VSIDS decision strategy which is less greedy than the decision heuristic used by Shtrichman with GRASP (*i. e.*, DLIS), it was not clear whether the benefits from a static order would still be realized. Therefore we also implemented static order heuristic and dominant variables priority heuristics to act as a *tie breaker* for the zchaff decision. Here are descriptions of the four decisions strategies we implemented as alternatives to VSIDS:

Static Order (SO) If not all of the dominant variables have been assigned, choose the next variable according to the static order. Otherwise, choose the next variable according to the VSIDS algorithm:

1. Each non-dominant variable in each polarity has a counter, which is initialized to 0.
2. When a clause is added (by learning) to the database, the counter associated with each literal in the clause is incremented.
3. The (unassigned) variable and polarity with the highest counter is chosen at each decision.
4. Periodically, all counters are divided by a constant.

Dominant Variables (DV) .

1. Each variable in each polarity has a counter, which is initialized to 0.
2. When a clause is added (by learning) to the database, the counter associated with each literal in the clause is incremented.
3. At each decision, the (unassigned) dominant variable and polarity with the highest counter is chosen. If there are no dominant variables left, the (unassigned) non-dominant variable and polarity with the highest counter is chosen.
4. Periodically, all counters are divided by a constant.

Static Order as a Tie Breaker (SB) .

1. Each variable in each polarity has a counter, which is initialized to 0.
2. When a clause is added (by learning) to the database, the counter associated with each literal in the clause is incremented.
3. The (unassigned) variable and polarity with the highest counter is chosen at each decision.
4. If two dominant variables have the same score, break the tie with the static order. If a dominant variable and a non-dominant variable have the same score, choose the dominant variable.
5. Periodically, all counters are divided by a constant.

Dominant Variables as Tie Breakers (DVB) .

1. Each variable in each polarity has a counter, which is initialized to 0.
2. When a clause is added (by learning) to the database, the counter associated with each literal in the clause is incremented.
3. The (unassigned) variable and polarity with the highest counter is chosen at each decision.
4. If a dominant variable and a non-dominant variable have the same score, choose the dominant variable.
5. Periodically, all counters are divided by a constant.

These four strategies appear similar to the original VSIDS strategy, but each of them leads to very different executions (e.g., number of decisions, max decision level, added conflict clauses) for the zchaff SAT solver.

6. Experimental Results

We did not use the original benchmark from [6] because most of the items were solved too quickly by zchaff. Instead, we benchmarked our algorithms against very difficult verification problems from actual design projects across IBM. We derived the benchmark from our general model checking benchmark. All of the items are real-life hardware verification problems. All items, with the exception of 1-6, come from different designs (Cf. Table 3 for

details on the size of the Conjunctive Normal Formulas derived from the designs after various optimization). Part of this benchmark (IBM_5, IBM_6, IBM_7, IBM_9, IBM_13, IBM_16, IBM_19) was used for the SAT2002 contest (<http://www.satlive.org/SATCompetition/2002/index.jsp>). Tables 1 and 2 present our experimental results. In Table 2, we decided to compute speedup, taking the VSIDS heuristic as a reference and therefore not to take into account IBM_16 and IBM_19, for which VSIDS times out. We could also have decided to take Static Order (SO) heuristic as a reference and not take IBM_10 and IBM_17 into account. However, SO is the only heuristic to time out for IBM_10 and IBM_17; and three out of five heuristics time out for IBM_16 and IBM_19.

The static order (SO) strategy has varying results; sometimes they are extremely good (e.g., IBM 4, IBM 16), but frequently, they are extremely poor (e.g., IBM 8, IBM 10). This is reflected in Table 2 by the strong difference between global speedup and average speedup for Static Order. To understand how static order helps with abstraction of the priority given to the dominating variables, we compare the static order strategies against the dominating variables strategies (DV vs. SO and DVB vs. SB). Using the “tie breaker”, DVB generally performs far better than SB (except for IBM 4, IBM 16, and IBM 17). However, the comparison between DV and SO strategies is less clear and we cannot conclude that the static order provides a general advantage over the dominating variables strategy.

In some cases (e.g., IBM 18, IBM 17), the static order strategy is (far) worse than the dominating variables strategy; this means the static order itself was worse than the VSIDS dynamic order on the dominating variables. We cannot conclude that this heuristic is better than the original zchaff with VSIDS decision strategy. This means that we could not reproduce [6]’s results with zchaff on our benchmark. How do we explain this? There are two possible explanations: [6]’s results may be benchmark-dependent or engine-dependent.

We used a benchmark that was broader than the one we present here and we obtained similar results. Moreover, we used zchaff with the DVB heuristic to create statistics about the proportion of dominant variables in the conflict clauses. We believe that the composition of conflict clauses somehow reflect which variables play an important role in a search. We obtained the following results for the totality of our benchmark (including [6]):

% dominating variables	% conflict clauses
0-5	11
5-10	8.4
10-15	25
15-20	29.7
20-25	12.5
25-30	5.4
30-40	4.4
+40	3.6

For instance, 11% of the conflict clauses from DVB searches are composed of 0% to 5% dominating variables. For [6] benchmark:

% dominating variables	% conflict clauses
0-5	10.1
5-10	8.8
10-15	18.4
15-20	13.2
20-25	11.4
25-30	9.7
30-40	12.7
+45	15.7

We found 87% of the conflict clauses are composed of less than 25% of dominating variables in our whole benchmark; this drops to 62% if we only consider the [6] benchmark. On the other hand, 8% of the conflict clauses are composed of more than 30% of dominating variables in our whole benchmark; this rises to 28.4% for [6] benchmark. This shows that dominating variables played a much larger role in the [6] benchmark than in our whole internal one, and may explain why our experimental results with zchaff are not consistent with the [6] results on GRASP.

The results of [6] may also be engine dependent: the SO heuristic may mixed better with GRASP than with zchaff; however, we think that if this is the case, engine dependency should be marginal.

Which strategy provides us with the best results? First, let's compare the VSIDS strategy with the DVB strategy. Except for IBM 3 and IBM 18, the DVB results are better than the VSIDS results, sometimes by one order of magnitude (e.g., IBM 5). If we compare the total time spent on the benchmark (except IBM 16 and IBM 19), we see that zchaff with DVB performs more than twice as fast as the original zchaff with VSIDS. It is not obvious which strategy (DVB, SB, DV, or SO) is the best. The results vary greatly and they do not share the same timeout. However, DVB performs well in most cases, and only in rare cases is worse than VSIDS, and in which the consequences are not dramatic (Cf. Table 2). On the other hand, the next generation of our formal verification tool will allow several engines to run concurrently.

If we had run the DVB, SB, DV, and SO strategies concurrently, we would have obtained a speedup factor of about six compared to classical zchaff.

7. Conclusion

We have shown that it is possible to improve SAT performance, by a factor two, for Bounded Model Checking by tuning the VSIDS decision heuristic. On the other hand, we have shown that the decision heuristic developed in [6] was, overall, not an improvement for zchaff when applied to our benchmark. Our next step is to implement these SAT algorithms concurrently; this would have given us a speedup of a factor six in our benchmark. An important question to consider for further research is whether our heuristics will still be efficient for SAT solvers other than zchaff. We think it is most probable for DVB, which is a very light and flexible heuristic.

References

- [1] E. C. A. Biere, A. Cimatti, and Y. Zhu. Symbolic Model Checking Without BDDs. In *Proceedings of the workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS99)*, 1999.
- [2] A. Biere, E. Clarke, R. Raimi and Y. Zhu Verifying Safety Properties of a Power PC (tm) Microprocessor Using Symbolic Model Checking Without BDDs. In *N. Halbwachs and D. Peled, editor, Proc. 11st Intl. Conference on Computer Aided Verification (CAV'99)*. Lecture Notes in Computer Science. Springer-Verlag, 1999.
- [3] S. A. Cook. The complexity of theorem proving procedures, In *Proceedings, Third Annual ACM Symp. on the Theory of Computing*, 1971.
- [4] E. A. Emerson and C. L. Lei. Modalities for Model Checking: Branching Time Strikes Back. In *Science of computer programming*, 8:275-306, 1986.
- [5] M. W. Moskewicz, C. F. Maadigan, Y. Zhao, L. Zhang and S. Malik Chaff: Engineering an Efficient SAT Solver. In *38th Design Automation Conference*, page 530-535. ACM/IEEE, 2001.
- [6] O. Shtrichman. Tuning SAT Checkers for Bounded Model Checking. In *Computer-Aided Verification: 12th International Conference*, pages 480-494. Lecture Notes in Computer Science, 1855. Springer-Verlag, 2000.
- [7] J. Silva and K. Sakallah. Grasp - a New Search Algorithm for Satisfiability. In *Technical Report TR-CSE-292996*. University of Michigan, 1996.

	VSIDS	DVB	SB	DV	SO	min(DVB,SB,DV,SO)
IBM 1_2001	18	12	31	14	170	12
IBM 2_2001	1400	620	820	300	1300	300
IBM 3_2001	43	66	190	240	240	66
IBM 4_2001	4000	1000	800	210	120	120
IBM 5_2001	2400	44	240	170	100	44
IBM 6_2001	1000	67	190	3800	250	67
IBM 7_2001	340	120	140	8	19	8
IBM 8_2001	13	10	34	14	4800	10
IBM 9_2001	71	44	110	190	900	44
IBM 10_2001	70	70	110	86	timeout	70
IBM 11_2001	4800	4200	6400	timeout	3100	3100
IBM 12_2001	44	42	73	46	51	42
IBM 13_2001	78	6	58	6	52	6
IBM 14_2001	32	31	42	18	26	18
IBM 15_2001	13	13	13	13	13	13
IBM 16_2001	timeout	timeout	9200	timeout	180	180
IBM 17_2001	7600	4000	3100	2400	timeout	2400
IBM 18_2001	11	85	95	6	3000	6
IBM 19_2001	timeout	timeout	timeout	1600	8700	1600

Table 1. The results are displayed in seconds with two significant digits. timeout was set to 10000 seconds.

	VSIDS	DVB	SB	DV	SO
Total time	21933	10430	12446	> 17521	> 34141
Global speedup (VSIDS total time/total time)		2.10	1.76	< 1.25	< 0.64
Average speedup (average VSIDS time/time)		6.11	2	< 6.26	< 5.15

Table 2. The results are computed without IBM_16 and IBM_19 for which VSIDS times out.

	Clauses	Variables		Clauses	Variables
IBM 1_2001	109 584	1 312 723	IBM 11_2001	56 223	240 946
IBM 2_2001	212 319	2 584 198	IBM 12_2001	89 109	352 457
IBM 3_2001	212 303	2 584 150	IBM 13_2001	88 079	378 174
IBM 4_2001	139 969	1 665 161	IBM 14_2001	58 116	248 548
IBM 5_2001	212 091	2 520 822	IBM 15_2001	133 480	583 044
IBM 6_2001	125 646	1 501 790	IBM 16_2001	88 790	388 707
IBM 7_2001	29 605	150 711	IBM 17_2001	69 569	295 578
IBM 8_2001	34 519	146 887	IBM 18_2001	11 851	295 578
IBM 9_2001	48 109	215 170	IBM 19_2001	22 984	117 949
IBM 10_2001	338 628	4 898 655			

Table 3. CNF sizes