Pueblo: A Modern Pseudo-Boolean SAT Solver

Hossein M. Sheini, Karem A. Sakallah

CSE-TR-492-04

July 21st, 2004

THE UNIVERSITY OF MICHIGAN

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan 48109-2122
USA

Pueblo: A Modern Pseudo-Boolean SAT Solver

Hossein M. Sheini, Karem A. Sakallah

Advanced Computer Architecture Laboratory
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Michigan 48109-2122

July 21st, 2004

Abstract

In this report we introduce a new SAT solver that integrates logic-based reasoning and integer programming methods to systems of CNF and PB constraints. Its novel features include an efficient PB literal watching strategy that takes advantage of the preponderance of unit-coefficient literals in most PB constraints. Additionally, the solver incorporates several PB learning methods that take advantage of the pruning power of PB constraints while minimizing their overhead. Empirical evidence suggests that such judicious injection of IP techniques can be quite effective in practice.

1 INTRODUCTION

Modern backtrack search SAT solvers augment the basic DPLL procedure of Davis, Putnam, Logemann, and Loveland [6, 7] with powerful conflict-based learning [13] and efficient watched-literal schemes [14] for Boolean constraint propagation (BCP). These advances have increased the scope and applicability of SAT solvers to problem instances with millions of CNF clauses and tens of thousands of variables. Many large design and analysis problems from the field of Electronic Design Automation (EDA) are now routinely cast as SAT instances and solved using these powerful solvers.

The closely-related 0-1 integer programming (IP) problem has also been studied extensively. In particular, the extension of SAT techniques to systems of CNF and so-called pseudo Boolean (PB) constraints was addressed in [3, 1]. Algorithms that combine the logic-based reasoning techniques of CNF SAT and the constraint relaxation and polyhedral analysis ("cutting planes") methods of IP were also explored with some success [11, 4].

In this report we introduce Pueblo, a new CNF/PB SAT solver that integrates logic-based reasoning and integer programming methods to handle systems of CNF and PB constraints. Pueblo provides a platform for exploring the trade-offs between these complementary approaches to constraint solving. In particular, Pueblo incorporates a novel watched literal strategy that takes advantage of the fact that many literals in PB constraints have unit coefficients in order to reduce the overhead of Boolean constraint propagation. In addition, Pueblo features several learning strategies including two that combine conflict-based CNF learning and cutting plane PB learning.

The report is structured as follows. In Section 2, we cover some preliminaries. Section 3 describes our PB propagation method. Section 4 details Pueblo's various PB learning strategies. Experimental results are reported in Section 5. Conclusions and suggestions for further work are presented in Section 6.

2 PRELIMINARIES

A linear pseudo-Boolean (PB) constraint is said to be in normal form when expressed as[1]:

$$\sum_{i=1}^{n} a_i \dot{x}_i \geq b \qquad a_i, b \in \mathbb{Z}^+, \dot{x}_i \in \{0,1\} \tag{1}$$

where $\dot{x}_i$ denotes x_i or x_i'. A PB constraint in which some coefficients are negative can be transformed to normal form by noting that $\dot{x}_i' = 1 - \dot{x}_i$. An example PB constraint in normal form is:

$$3x_1 + 2x_2' + x_3 + x_4 \geq 3 \tag{2}$$

In general, a PB constraint is equivalent to a large, potentially exponential, number of CNF clauses [1]. When the right-hand side and all left-hand side coefficients are equal to 1, however, a PB constraint is equivalent to a single CNF clause:

[1] Less-than-or-equal and equality constraints can be easily transformed to equivalent greater-than-or-equal forms.

$$\sum_{i=1}^{n} \dot{x}_i \geq 1 \Leftrightarrow \bigvee_{i=1}^{n} \dot{x}_i \tag{3}$$

3 PSEUDO-BOOLEAN PROPAGATION

It was noted in [14] that modern conflict-based backtracking SAT solvers spend most of their run time in Boolean constraint propagation (BCP). Conventionally, BCP entails "watching" all the literals in a clause until the clause becomes unit, whereupon the remaining free literal is implied to true. In [14, 16] it was shown that watching just two literals per clause, regardless of clause size, is sufficient to detect when the clause becomes unit. This optimization drastically reduces the overhead of BCP during search and is one of the most significant recent enhancements to SAT algorithms. In this so-called two-watched-literal scheme, two non-false (initially unassigned) literals are chosen to be watched in each clause. A clause is processed only when either of its two watched literals is set to false; such an assignment triggers the search for another unassigned literal to replace the one that just became false. The clause becomes unit if the only unassigned literal found is the other watched literal which must now be implied to true to satisfy the clause.

The watched literal concept was extended in [3] to handle PB constraints. The basic idea is to watch the fewest number of non-false literals such that when the unassigned watched literal with the largest coefficient is set to false a) the constraint is still guaranteed to be satisfied and b) the constraint can identify the literals that must now be implied to true. Specifically, let T and U denote the sets of true and unassigned literals in the constraint, and let $W \subseteq T \cup U$ denote the set of watched literals. We will refer to W as the *watch list* and to the sum of coefficients of the watched literals as the *watched sum*, i.e.,

$$S_W = \sum_{\dot{x}_i \in W} a_i \tag{4}$$

We also introduce $a_{\max}$ defined as:

$$a_{\max} = \max\{a_i | \dot{x}_i \in U\} \tag{5}$$

The invariant that must be maintained to detect when the PB constraint becomes unit can now be succinctly expressed as:

$$S_W \geq b + a_{\max} \tag{6}$$

When a watched literal is set to false, it must be removed from the watch list and replaced by one or more non-false literals to maintain the above invariant. When that is no longer possible, the constraint becomes unit and the unassigned watched literals with the largest coefficients must be set to true to insure that the constraint is not violated. In other words, any unassigned watched literal whose coefficient a satisfies the *unit constraint* condition

$a > S_W - b$

Algorithm 1 Watch Fewest Literals [3]

```
// Execute when watched literal ẋ_i is set to false
1. UPDATE a_max using (5)
2. UPDATE watch list
   a. Remove ẋ_i from W and update S_W using (4)
   b. Fill W and update S_W until invariant (6) is
      restored
3. IMPLY watched literals whose coefficients satisfy
   the unit constraint condition
```

Assignment	$a_{\max}$	W	S_W	$b + a_{\max}$	Unit?	$S_W - b$
None	3	$\{x_1, x_2', x_3\}$	6	6	No	—
Decide $x_1 = 0$ Imply $x_2 = 0$	2	$\{x_2', x_3, x_4\}$	4	5	Yes	1
Decide $x_3 = 0$ Imply $x_4 = 1$	1	$\{x_2', x_4\}$	3	4	Yes	0

Figure 1: Execution trace of Algorithm 1 on (2)

must now be implied to true[2]. The major computational steps of this ***Watch Fewest Literals*** procedure are highlighted in Algorithm 1 and a trace of its execution on the example PB constraint in (2) is shown in Figure 1.

Empirical evaluation of this procedure suggests that about two thirds of its run time is spent in updating $a_{\max}$ (see Table 1) and corroborates the conclusion in [3] that the "watching scheme is beneficial for clauses and cardinality constraints, but not for LPB constraints; therefore we use counters to implement Boolean constraint propagation on LPB constraints." In other words, PB constraints are processed using a ***Watch All Literals*** strategy similar to that of PBS [1].

Further analysis of the data, however, suggests a potentially more efficient *hybrid* watching strategy that differentiates between the literals with unit- and non-unit coefficients in the same PB constraint. Specifically, let L denote the set of literals whose coefficients are greater than 1 (the *large* literals) and let C denote those literals whose coefficients are equal to 1 (the *cheap* literals). As indicated in columns B and C of Table 1, the majority of the literals belong to the C set and their processing accounts for a correspondingly large fraction of the time spent in Algorithm 1. Much of this time can be eliminated by applying the procedure of Algorithm 1 only to the literals in the (relatively small) L set. To achieve this, the computation of $a_{\max}$ in (5) is modified to become:

$$a_{\max} = \begin{cases} 1 & \text{if } L \cap U = \varnothing \\ \max\{a_i | \dot{x}_i \in L \cap U\} & \text{if } L \cap U \neq \varnothing \end{cases} \tag{7}$$

[2] Note that conditions (6) and reduce to the 2-lit strategy when the PB constraint is just a CNF clause ($a_i = b = 1$)

Algorithm 2 Watch Cheap Literals

```
// Execute when watched literal ẋ_i is set to false
1. UPDATE a_max using (7)
2. UPDATE watch list
   a. Remove ẋ_i from W and update S_W
   b. Fill W and update S_W until invariant (6) is
      restored
3. IMPLY watched literals
   a. If a_max > 1, imply large watched literals in
      decreasing order of their coefficients until
      (6) is restored or the constraint is satisfied
   b. If a_max = 1, imply all remaining unassigned
      cheap literals
```

Furthermore, the watch list W is modified to include all of the literals in C:

$$\begin{aligned} W' &\subseteq L \cap (T \cup U) \\ W &= W' \cup C \end{aligned} \tag{8}$$

This modification, in turn, requires a slight change to the manner in which the watched sum is calculated since the watched literals are no longer guaranteed to be true or unassigned. Specifically, the watched sum must now be decremented by 1 when a C literal is set to false, and incremented by 1 when a C literal is unassigned from false.

In addition to the efficiencies that accrue from processing the shorter list of non-unit coefficients in the UPDATE steps of Algorithm 1, a further gain is possible in the IMPLY step by noting that all unassigned C literals can be simultaneously implied to true when there are no unassigned L literals. We will refer to this procedure as the ***Watch Cheap Literals*** algorithm (see Algorithm 2.) As the data in columns D and E of Table 1 show, this algorithm watches many more literals than Algorithm 1 but manages to achieve a net gain in performance because watching the C literals is "cheap."

Table 1: Analysis of PB Propagation

Benchmark	|PB|[a]	A[b]	B[c]	C[d]	D[e]	E[f]
fpga10-8-sat-pb	842	62%	92%	94%	464%	60%
fpga10-9-sat-pb	1907	61%	91%	92%	552%	53%
fpga11-10-sat-pb	278	66%	99%	100%	710%	50%
fpga15-14-sat-pb	596	69%	100%	100%	903%	40%

[a] Total number of original and learned PB constraint
[b] Fraction of PB propagation time spent in step 1 of Algorithm 1
[c] Fraction of cheap (unit-coefficient) literals
[d] Fraction of time spent in steps 1 and 2 of Algorithm 1 to process the cheap literals
[e] Number of propagations processed by Algorithm 2 relative to those processed by Algorithm 1
[f] Gain in PB propagation run time of Algorithm 2 over Algorithm 1

4 PSEUDO-BOOLEAN LEARNING

As described earlier, one of the major improvements in SAT solvers was the deployment of conflict learning into the DPLL algorithm. Generating and recording a so-called conflict-induced clause [13], enables the solver to prune away a portion of the search space thus avoiding a recurrence of the same conflict. Additionally, conflict-induced clauses enable the solver to backtrack non-chronologically in the search tree without compromising completeness [13]. We will refer to this style of learning as *CNF learning*. An alternative approach, based on cutting plane methods [5], can be used to create a conflict-induced PB constraint instead. Since PB constraints are generally more expressive than CNF clauses, a learned PB constraint has the potential of pruning more of the search space than a learned CNF clause. This style of learning is employed in the PB solver galena [4] and will be referred as *PB learning.* As we observed earlier in connection with PB propagation, however, the steep overhead of manipulating PB constraints can more than offset their pruning benefits. Thus, an adaptive approach that combines CNF and PB learning, and introduces PB constraints selectively, might be superior to either approach alone. We describe next two variations on this theme. In both variations, CNF and PB learning is done in parallel by backward traversal of the implication graph; the two variations, however, differ in the way the learned PB constraint is processed.

Scheme 1: Learn Strong PB Constraints. In this scheme a PB constraint is learned and recorded if and only if

1. it is unit, i.e., it rejects the current conflicting assignment, and
2. it corresponds to more than just a single CNF clause, and
3. the number of its large literals (those in the L set) is less than a given threshold;

Otherwise, the learned PB constraint is discarded and the CNF clause learned in parallel is retained.

Scheme 2: Convert Learned PB Constraint to CNF. In this scheme, the learned PB constraint is recorded but not used in BCP. Rather, it is viewed as a compact representation of a set of CNF clauses, subsets of which can be extracted as needed during the search. Extraction of suitable CNF clauses from the PB constraint is carried out using a simple knapsack algorithm. The rationale for this scheme is to capitalize on the pruning power of the learned PB constraint without incurring its high propagation overhead.

The data in Figure 2 drive home the high cost of unlimited PB learning and propagation. The pruning ability of learned PB constraints is evident from the decrease in the number of decisions as more PB constraints are added. However, the run time increases initially and does not drop significantly until enough PB constraints have been collblected to effectively prune out most of the non-solution part of the search space.

5 EXPERIMENTAL RESULTS

We conducted several experiments to evaluate the learning and propagation strategies described above using our new CNF/PB SAT Pueblo. Pueblo is built on top of MiniSAT [8] and inherits its strategy for random restarts. It

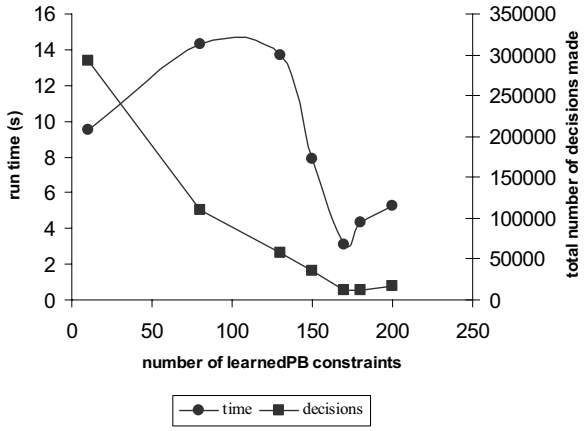

Figure 2: Effect of PB learning and propagation on number of decisions and overall run time for the global routing instance s4-3-2pb [1]. Learning was done according to scheme 1, with an added restriction on the maximum allowable number of learned PB constraints

additionally extends MiniSAT's VSIDS decision heuristic and its clause removal mechanism to PB constraints. All experiments were conducted on a Pentium-IV 2800MHz machine with 1 GB of RAM running Linux 2.4.20.

Table 2 shows the effect of the four learning strategies described in Section 4 on a set of large CNF/PB benchmarks arising from microprocessor verification [15]. The data provide clear evidence that PB learning is too costly and is, in many cases, even inferior to CNF learning. Of the integrated approaches, scheme 1 has a clear edge over scheme 2. The last two columns in the table give the number of learned CNF and PB constraints in scheme 1. A further breakdown of these data is shown in Table 3 which indicates the reason for learning a CNF clause instead of a PB constraint.

Table 4 depicts a comparison of Pueblo and galena PB learning strategies on a set of representative benchmarks including the FPGA Routing and Global Routing instances of [1]. Overall, scheme 1 performs robustly and compares favorably to the cardinality strategy of galena.

6 CONCLUSIONS

The integration of logic-based reasoning and integer programming methods promises to be a vibrant area of research for the next several years. As we learn more about the trade-offs involved, we will be able to develop effective integration strategies that outperform individual techniques. Our contribution in this paper should be viewed as one additional milestone along this road.

The concepts described in this paper do not exhaust all the possibilities for taking advantage of the pruning power of PB constraints while minimizing their computational overhead. Other ways of generating cutting planes, for example, that are provably superior to current approaches should be investigated. One promising direction of future research involves extending the techniques described above to small-domain integer programs. Such problems arise

naturally in many application areas. Specifically, the use of uninterpreted functions to abstract away datapath components in the verifications of digital systems leads to a small-domain decision problem that can benefit form the application of these techniques.

REFERENCES

[1] F.A. Aloul, A. Ramani, I.L. Markov, and K.A. Sakallah, "Generic ILP versus specialized 0-1 ILP: An Update", IEEE/ACM Intl. Conference on Computer-Aided Design, pp. 450-457, 2002.

[2] E. Balas, S. Ceria and G. Cornnéjols, "A lift-and-project cutting plane algorithm for mixed 0-1 programs", Math. Programs, vol. 58, pp. 295-324 1993.

[3] P. Barth, "A Davis-Putnam based Enumeration Algorithm for Linear Pseudo-Boolean Optimization," *Technical Report MPI-I-95-2-003, Max-Planck-Institut Für Informatik*, 1995.

[4] D. Chai, A. Kuehlmann, "A Fast Pseudo-Boolean Constraint Solver", Proc. Design Autom. Conf., pp. 830-835, 2003.

[5] V. Chvátal, "Edmonds Polytopes and a Hierarchy of Combinatorial Problems", Discr. Math., vol. 4, pp. 305-307, 1973.

[6] M. Davis and H. Putnam, "A computing procedure for qualification theory", Journal of the ACM, vol. 7, pp. 102-215, 1960.

[7] M. Davis, G. Logemann, and D. Loveland. "A machine program for theorem proving", Communications of the ACM, vol. 7, pp. 394-397, 1962.

[8] N. Eén, and N. Sörensson, "An Extensible SAT-solver," in *Sixth International Conference on Theory and Applications of Satisfiability Testing* (SAT), 2003.

[9] R.E. Gomory, "Outline of an algorithm for integer solutions to linear programs", Bulletin of the AMS, vol. 64, pp. 275-278, 1958.

[10] J.N. Hooker, "Generalized Resolution for 0-1 Linear Inequalities", Annals of Mathematics and Artificial Intelligence, vol. 6, pp. 271-286, 1992.

[11] J.N. Hooker, G. Ottosson, E.S. Thorsteinsson, and H.K. Kim, "A Scheme for Unifying Optimization and Constraint Satisfaction Methods" Knowledge Engineering Review vol. 15 pp.11-30, 2000.

[12] L. Lovász and A.J. Schrijver, "Cones of matrices and set-functions and 0-1 optimization", SIAM J. Opn., vol. 1, pp. 166-190, 1991.

[13] J.P. Marques-Silva and K.A. Sakallah, "GRASP: a search algorithm for propositional satisfiability", IEEE Transaction on Computers, vol. 48, no. 5, pp. 506-521, 1999.

[14] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver", Proc. of the Design Automation Conference, pp. 530-535, 2001.

[15] UCLID: A Verification Tool for Infinite-State Systems, http://www-2.cs.cmu.edu/~uclid/.

[16] H. Zhang, and M. Stickel, "An efficient algorithm for unit-propagation", Proc. of the Int'l Symposium on Artificial Intelligence and Mathematics, pp. 166-169,1996.

Table 2: Run time comparison of different learning strategies

Benchmark				Time, sec.				Learned Constraints	
Name	|Vars|	|CNF|	|PB|	CNF Learning	PB Learning	Scheme 1	Scheme 2	|CNF|	|PB|
elf.rf8.ucl (UNS)	6059	15940	744	0.11	0.12	0.1	0.11	81	1
elf.rf9.ucl (UNS)	20445	56420	2024	0.98	1.18	1.14	0.98	368	3
elf.rf10.ucl (UNS)	55066	156587	4032	13.26	23.16	13.27	13.11	2753	2
ooo.rf6.ucl (UNS)	1804	4707	224	0.04	0.05	0.04	0.04	134	6
ooo.rf7.ucl (UNS)	3736	10089	368	0.52	0.43	0.29	0.28	409	36
ooo.rf8.ucl (UNS)	6832	18953	524	1.73	7.06	1.42	2.21	1180	183

Table 2: Run time comparison of different learning strategies

Benchmark				Time, sec.				Learned Constraints	
Name	|Vars|	|CNF|	|PB|	CNF Learning	PB Learning	Scheme 1	Scheme 2	|CNF|	|PB|
ooo.rf10.ucl (UNS)	18069	51555	920	88.6	*	96.87	63.71	18835	12298
ooo.tag8.ucl (UNS)	3249	8599	384	2.1	5.3	2.07	*	3686	622
ooo.tag10.ucl (UNS)	9071	25190	724	32.34	111.82	11.52	*	7323	1341
ooo.tag12.ucl (UNS)	20605	58675	1176	170.39	672.5	95.09	*	26957	8754
ooo.tag14.ucl (UNS)	40605	117190	1740	334.54	*	99.83	*	18665	2095

Table 3: Reason for learning a CNF clause rather than a PB constraint in scheme 1

Benchmark	Over satisfaction	No PB involved in conflict level	CNF equiv	Large PB
elf.rf8.ucl (UNS)	1	74	6	0
elf.rf9.ucl (UNS)	0	368	11	0
elf.rf10.ucl (UNS)	1	2724	28	0
ooo.rf6.ucl (UNS)	0	100	34	0
ooo.rf7.ucl (UNS)	20	227	112	0
ooo.rf8.ucl (UNS)	51	644	485	0
ooo.rf10.ucl (UNS)	1136	6783	10855	61
ooo.tag8.ucl (UNS)	1881	940	778	87
ooo.tag10.ucl (UNS)	2625	2857	1771	70
ooo.tag12.ucl (UNS)	7801	6059	12514	583
ooo.tag14.ucl (UNS)	3312	10910	3936	507

Table 4: Run time comparison between different PB learning strategies

Benchmark	Time (sec)			
Name |Var|/|CNF|/|PB|	CNF Pueblo	Card. Galena	LPB Galena	Scheme1 Pueblo
s4-3-1pb (SAT) 672/2004/24	0.2	0.90	0.78	0.09
s4-3-2pb (SAT) 648/1928/24	4.92	0.40	0.7	0.58
s4-3-3pb (SAT) 648/1930/24	3.37	0.40	0.6	0.84
s4-3-4pb (SAT) 696/2072/24	2.61	0.60	1.12	0.05
s4-3-5pb (SAT) 720/2144/24	3.25	0.80	0.15	1.74
fpga10_12 (UNS) 240/24/20	*	0	0	0.01
fpga10_8 (SAT) 120/88/18	0.17	1.31	29.60	0.03

Table 4: Run time comparison between different PB learning strategies (Continued)

Benchmark	Time (sec)			
Name |Var|/|CNF|/|PB|	**CNF Pueblo**	**Card. Galena**	**LPB Galena**	**Scheme1 Pueblo**
fpga10_9 (SAT) 135/99/19	4.42	0.20	0.70	0.08
fpga11_10 (SAT) 165/120/21	*	0.50	0.12	0.01
fpga11_9 (SAT) 149/108/20	41.19	43.39	*	0.09
fpga12_14 (UNS) 336/28/24	*	0.10	0	0
fpga15_10 (SAT) 225/160/25	*	*	85.40	7.45
fpga15_14 (SAT) 315/224/29	*	0.37	1.14	0.02
cache-ibm-q-full.all.ucl 81558/235865/4604	*	356.53	*	567.98
cache.inv12.ucl.cnf 25800/76319/380	36.01	18.91	254.25	57.48
dlx1c.rwmem1.ucl 7578/20501/900	1.14	1.43	2.24	1.05
ooo.burch_dill.3.accl.ucl 4622/11753/816	10.03	10.29	641.84	8.72