PBS: A Backtrack-Search Pseudo-Boolean Solver and Optimizer

Fadi A. Aloul, Arathi Ramani, Igor L. Markov, Karem A. Sakallah

Department of Electrical Engineering and Computer Science
University of Michigan
{faloul, ramania, imarkov, karem}@eecs.umich.edu

Abstract

*Optimized solvers for the Boolean Satisfiability (SAT) problem [*5, 14, 15, 17, 19, 23, 24*] found many applications in areas such as hardware and software verification, FPGA routing, planning in AI, etc. Further uses are complicated by the need to express "counting constraints" in conjunctive normal form (CNF). Expressing such constraints by pure CNF leads to more complex SAT instances. Alternatively, those constraints can be handled by Integer Linear Programming (ILP), but off-the-shelf ILP solvers tend to ignore the Boolean nature of 0-1 variables.*

This work attempts to generalize recent highly successful SAT techniques to new applications. First, we extend the basic Davis-Putnam framework to handle counting constraints and apply it to solve routing problems. Our implementation outperforms previously reported solvers for the satisfiability with "pseudo-Boolean" constraints and shows significant speed-up over best SAT solvers when such constraints are translated into CNF,.

Additionally, we solve instances of the Max-ONEs optimization problem which seeks to maximize the number of "true" values over all satisfying assignments. This, and the related Min-ONEs problem are important due to reductions from Max-Clique and Min Vertex Cover. Our experimental results for various benchmarks are superior to all approaches reported earlier.

1 Introduction

Boolean Satisfiability has been the topic of intensive research over the past few decades. It has become a promising new technology for a number of applications in the field of electronic design automation (EDA) and has been successfully applied to various problems arising in formal verification [6, 21], timing analysis [18], routing of field-programmable gate arrays [16], automatic test pattern generation (ATPG) [12, 20], etc. As a result, several powerful SAT solvers [5, 14, 15, 19, 23, 24] have been proposed, many of which use one or another variation of the Davis-Logemann-Loveland (DLL) [8] approach. A recently developed solver, Chaff [15], proposed significant enhancements in both algorithm and implementation level to backtrack search algorithms, which has lead to dramatic performance gains on many SAT instances.

SAT solvers typically input Boolean formulas in conjunctive normal form (CNF), and an arbitrary Boolean formula without quantifiers can be quickly converted into a CNF. However, some applications contain counting constraints which impose an upper or a lower bound on the number of certain objects. More generally, given a collection of predicates, one may want to limit the number of predicates which evaluate to true. VLSI routing is a source of such examples - only a certain number of wires can be routed through a given channel (no more than the channel capacity), but the particular choice of wires is often not important. Expressing such a constraint in pure CNF may lead to a serious increase in the number of clauses and variables, or in the addition of a fairly complicated set of clauses derived from adder and comparator circuits. This problem was addressed in a recent work [23] by an algorithm that handles non-CNF constraints such as cube lists and pseudo-Boolean (PB) constraints [3] of the form:

$$\sum c_i x_i \leq n \quad c_i, n \in Z, x_i \in \{0, 1\}$$

The term *pseudo-Boolean constraints* refers to arbitrary linear inequalities 0-1 in terms of variables, however many applications require only integer coefficients. The SATIRE solver [23] was successfully applied to functional delay fault testing. It showed that PB encodings can be solved much more efficiently (by both runtime and memory usage) than pure CNF encodings. Note that decision problems with pseudo-Boolean constraints are a special case of Integer Linear Programming (ILP). However, as has been pointed out in [3], common ILP algorithms fail to take 0-1 variables into account and are outperformed by specialized algorithms.

Our first contribution is a solver architecture that combines data structures used in SATIRE with those used in the currently best pure CNF solver Chaff [15]. Our new SAT solver, PBS, handles CNF constraints and PB inequalities. Unlike previously proposed stochastic local search solvers [22], this solver is complete and is based on a backtrack search algorithm. We believe that our proposed algorithms to handle PB constraints can be added to any backtrack SAT solver.

To demonstrate the effectiveness of PBS, we compare it to (i) the currently best CNF-SAT solver Chaff [15], (ii) the earlier PB-SAT solver SATIRE [23] based on GRASP [14], and (iii) one of the best linear programming solvers OPBDP [4] based on branch-and-bound. Our experimental results show at least an order-of-magnitude speed-up on instances with more than 300 variables and also on many smaller instances.

From the application standpoint, the availability of PB constraints helps to address a broader range of problems, especially those which cannot be efficiently encoded with pure

CNF constraints. In this work, we study two classes of such applications: (i) decision problems in global routing, and (ii) Boolean optimization problems. We propose a new formulation of global routing that leverages PB constraints and show that it leads to dramatic speed-ups and lower memory requirements.

The second class of applications is Boolean optimization problems such as Max-SAT, Max-ONEs and Min-ONEs. These problems have 0-1 variables only, but the goal is to minimize or maximize a particular objective while satisfying a number of constraints. For example, in Max-SAT, one tries to maximize the number of satisfied clauses (unconstrained optimization). The Max-ONEs problem seeks a satisfying assignment that maximizes the number of "true" values. Min-ONEs seeks a satisfying assignment that minimizes that number.

The importance of these SAT-like optimization problems can be explained by analogy with the fundamental role of the CNF-SAT problem in NP-complete constraint satisfaction problems. As shown in [7], a number of NP-hard combinatorial optimization problems, such as Max-Cut, Max-Clique, and Min Vertex Cover have linear-time reductions to Max-SAT, Max-ONEs and Min-ONEs. Many of those problems are vital to Electronic Design Automation, for example, Vertex Cover has applications in two-level logic minimization.

We observe that heuristic solvers based on local search, such as WalkSAT [17], can be used in the context of the Max-SAT problem. However, the Max-ONEs and Min-ONEs problems, while similar to each other, have not been addressed by efficient solvers (to the best of our knowledge).

To fill this void, we propose algorithms to optimally solve Boolean optimization problems with the help of PB constraints. In our experiments for the Max-ONEs problem on standard CNF benchmarks, these algorithms outperform all previously published approaches.

The remaining part of the paper is organized as follows. Section 2 briefly reviews the latest enhancements in backtrack search solvers. PB constraints and PBS are described in Section 3. Section 4 discusses applications of the new PBS. In Section 5, we present our experimental results. Finally, the conclusions are presented Section 6.

2 The State of the Art in CNF-SAT Search

Recent improvements to the DLL procedure [8] have focused on the internal data structures and various heuristics that guide complete search. This section describes the most effective enhancements, and our new solver PBS includes them.

The DLL procedure performs a *depth-first search* in the decision tree over of the problem variables and can be viewed as consisting of three main engines: decision, deduction, and diagnosis. The decision engine makes an *elective* assignment based on a heuristic. The deduction engine makes a *forced* assignment based on the problem constraints and current partial assignments to other variables. The main idea is based on the *unit clause rule* which forces the assignment of the only unassigned variable in a clause whose other literals are all 0. Boolean Constraint Propagation (BCP) is achieved by the repeated application of the unit clause rule over a given clause database, and is known to identify all possible implications of the decisions made thus far. Finally, the diagnosis engine handles the occurrence of conflicts (i.e., assignments that cause the formula to become unsatisfiable) and backtracks appropriately.

2.1 Decision Strategy

Decision heuristics have played an important role in enhancing the performance of SAT solvers. Several studies have proposed various decision heuristics that can be classified as static [1] or dynamic [14, 15, 24]. For example, the GRASP SAT solver [14] is typically used with the dynamic decision heuristic DLIS which selects the literal that appears in the maximum number of unresolved clauses. One heuristic that has been found to be particularly effective in a variety of problems is VSIDS [15]. It was implemented in the Chaff SAT solver. The heuristic maintains two counters for every variable. They are incremented if a positive or negative literal is identified in a new conflict-induced clause, respectively. The variable with highest counter is selected for the next decision. In order to emphasize the variables identified in recent conflicts, the counters are periodically divided by a constant.

2.2 Improved BCP

Enhancements to the implementation of BCP were shown to yield significant performance improvements [15, 26]. Noting that a sizable fraction of a SAT solver's runtime is spent in the BCP procedure, these enhancements can be viewed as a form of "lazy" evaluation that avoids unnecessary traversals of the clause database. In conventional BCP procedures, whenever a variable v is assigned, all clauses containing literals of this variable are traversed to check whether they have become unit or are in conflict. In other words, an implication step requires time bounded by the number of existing literals of the assigned variable. Recently, Moskewicz et al. [15], presented a very efficient implementation of an amortized linear time BCP algorithm. The basic idea is to keep track of *any two unresolved literals* in each clause. A unit clause can, then, be easily detected when the two pointers concur. Furthermore, this approach incurs no penalty when a variable is unassigned.

Empirically, such BCP improvements show great improvements over conventional BCP implementations, especially for problems containing large number of large clauses; for example, a problem consisting of n k-literal clauses needs $2n$ pointers instead of kn pointers.

2.3 Conflict Diagnosis and Clause Deletion

One of the most powerful techniques to expedite the search process is known as conflict diagnosis [14]. Whenever a conflict is detected, the conflict is analyzed and a conflict-induced clause is added to the clause database to ensure

that the undiagnosed variable assignment doesn't occur in future. Using the learned clause, it is possible to backtrack non-chronologically. This technique is implemented in almost all backtrack search SAT solvers and has shown to be very effective in pruning the search space. Since then, several clause learning schemes have been proposed [14, 15], some of which learn multiple clauses at each conflict. Recently, however, Zhang et al. [25] proved empirically that the *1UIP* learning scheme, in which a single clause is learned at each conflict, showed the best performance among a variety of schemes on several hard instances.

Nevertheless, conflict-induced clauses tend to consist of hundreds of literals. Recording all conflict-induced clauses during a search process can be impractical as its possible for the clause database to grow exponentially. Several methods have been proposed to handle this efficiently. One solution is to keep any conflict-induced clause of size smaller than a user-specified threshold k and discard all others as soon as they are generated.

2.4 Random Restarts and Backtracking

Besides learning new clauses, recent studies have shown that using randomization and random restarts can be very effective in solving hard SAT instances [2, 10, 15]. A SAT solver may often get stuck in a local non-useful search space. The restart process periodically unassigns all previous decisions and implications and randomly selects a new sequence of decisions. This process ensures that different sub-trees are searched every time the search process restarts. Clauses learned between various restarts are kept for future use.

Recently, Lynce et al. [13] proposed and empirically evaluated combining randomization with backtracks. Periodically, the diagnosis engine backtracks non-chronologically to a decision level involving any literal in the conflict-induced clause. This process has been shown to be effective on various instances. The completeness of the search is preserved by keeping all conflict-induced clauses during the search process.

3 Processing of Pseudo-Boolean Constraints

In this section we define the PB constraints and describe the details of how PBS can be implemented to process both CNF and PB constraints efficiently. In particular, we explain how to adapt backtrack search SAT solvers to handle both CNF and PB constraints. A PB constraint is a linear inequality in terms of 0-1 variables:

$$c_1 x_1 + \ldots + c_n x_n \leq n$$

In this work we assume that coefficients c_i and n are integers because that is sufficient for most applications and enables minor implementation efficiencies. x_i are literals of Boolean variables. In principle, none of our algorithms rely on the integrality of coefficients and can be implemented for floating-point coefficients.

$$(\overline{v_1} + \overline{v_2} + \overline{v_3})(\overline{v_1} + \overline{v_2} + \overline{v_4})(\overline{v_1} + \overline{v_2} + \overline{v_3})$$
$$(\overline{v_1} + \overline{v_3} + \overline{v_4})(\overline{v_1} + \overline{v_2} + \overline{v_5})(\overline{v_1} + \overline{v_4} + \overline{v_5})$$
$$(\overline{v_2} + \overline{v_3} + \overline{v_4})(\overline{v_2} + \overline{v_3} + \overline{v_5})(\overline{v_2} + \overline{v_4} + \overline{v_5})$$
$$(\overline{v_3} + \overline{v_4} + \overline{v_5})$$

(a)

$$1v_1 + 1v_2 + 1v_3 + 1v_4 + 1v_5 \leq 2$$

(b)

Figure 1. Example representing "at most 2 out of v_1, v_2, v_3, v_4, v_5 can be true" using (a) pure CNF (b) PB form

3.1 Motivating Example

Consider a problem in which the objective is to limit the number of true assignments to a set of k variables out of a total of n variables in the problem. A straightforward CNF encoding requires: $\binom{n}{k+1}$ clauses of size $(k+1)$ each, which does not scale well and requires more than 150 million clauses for *n=30* variables. On the other hand, such a "counting" constraint can be handled literally by a PB solver, as a single PB constraint. Figure 1 illustrates the difference between a problem encoded in pure CNF and PB form. PB constraints are well suited for modeling cost functions and are likely to be useful when applying SAT to optimization problems throughout the EDA domain. Note that even when each counting constraint involves only 5-10 literals, the advantage of PB formulation can be significant if the number of counting constraints is large.

3.2 Algorithms for PB-SAT Search

Besides the conventional "problem-in/solution-out" mode of operation, our solver supports incremental changes to PB constraints after an instance is solved or if it times out. Therefore, it can be used (i) to prove the *consistency* of the formula by identifying a satisfiable assignment, or (ii) to *optimize* the upper/lower bounds for selected PB constraints.

Our algorithms maintain two separate data structures - for CNF constraints, we use the "watched literal" data structure from Chaff [15] and for PB constraints we use data structures somewhat related to those in SATIRE [23]. For further details of CNF data structures, the reader is referred to [15]. Our data structures for PB constraints are described below.

Every PB constraint is represented internally by a record. Each such record contains these fields:

- The objective goal n, the constraint type ($\geq$, $\leq$ or

=), and a list of coefficients with respective literals $c_i x_i$.

- The $initLHS$ field stores the sum of all coefficients in the PB constraint.
- The LHS field stores the value of the left-hand-side of the PB constraint computed based on the currently assigned variables.
- The $maxLHS$ field represents the maximum possible value of the left-hand-side.

For efficiency, the list of $c_i x_i$ is sorted in each record, in the order of increasing c_i values. Another improvement is to convert each PB constraint to the form where all coefficients are positive. This is achieved using negated variables, as illustrated by the following example:

$$c_1 x_1 - c_2 x_2 \le n$$
$$c_1 x_1 - c_2 (1 - \overline{x_2}) \le n$$
$$c_1 x_1 + c_2 \overline{x_2} \le n + c_2$$

For each variable x ,we maintain its value as well as lists *PosPBLits* and *NegPBLits* of PB constraints that include a positive or a negative literal in x.

After assigning *true* to variable v, we traverse the PB pointers in the *PosPBLits* and *NegPBLits* of variable v. The LHS of every PB constraint pointed to by the *PosPBLits* is incremented by the coefficient associated with v in the given PB constraint. On the other hand, the $maxLHS$ of every PB constraint pointed to by the *NegPBLits* is decremented by the coefficient associated with v in the given PB constraint. Undoing an assignment of *true* to a variable v, entails a similar traversal, except that the LHS and $maxLHS$ values for every affected PB constraint are decremented and incremented, respectively. When assigning or un-assigning *false*, we perform similar traversals as well.

In the process of updating the PB constraint fields, we watch for new implications and conflicts. Each PB constraint type is associated with a set of rules for detecting implications and conflicts. Given a PB constraint of type " $\le$ ", any literal x_i whose coefficient is $c_i > n - LHS$ is implied to *false*. The implying literals consist of the set of literals assigned to *true* in the PB constraint. In terms of a PB constraint of type "$\ge$", a literal x_i is implied to *true* if its coefficient is $c_i > maxLHS - n$. The implying literals consist of the set of literals assigned to *false* in the PB constraint. The time needed to identify the implications is ameliorated by maintaining a pointer in each PB constraint to the unassigned literal with the highest coefficient.

When a conflict is detected in a PB constraint of type " $\le$ " (i.e., for $LHS > n$), the conflicting clause consists of literals assigned to *true*. If the LHS is significantly large, it may be possible to simplify the conflicting clause. In order to minimize the number of literals in the conflicting clause, recall that all coefficients are positive. Thus we can select only a subset of literals, whose coefficients add up to $> n$. Similarly, when a conflict is detected in a PB constraint of type "$\ge$" (i.e., $maxLHS < n$), the conflicting clause consists of literals assigned to *false*. The conflicting clause size can be reduced by selecting a subset of literals with false values whose coefficients total is greater than or equal to $initLHS - n$. While finding a smallest possible conflict clause is as difficult as solving the NP-complete problem KNAPSACK, we use a classic heuristic that packs starting from the greatest coefficient towards the smallest.

Each PB constraint of type " = " is handled by following rules for inequalities of types $\ge$ and $\le$ that have the same left-hand and right-hand sides. Moreover, only one PB record is used.

The described algorithm is complete for the same reasons that the DLL algorithm is complete, and can be readily integrated into practically any existing backtrack SAT solver.

4 Applications

Two classes of applications that can be expressed in PB/CNF form are described in this section. First, we show how adding a single PB constraint with a sliding lower/upper bound to a CNF formula can model standard Boolean optimization problems. Second, we discuss the benefits of using multiple PB constraints with fixed bounds in global routing.

4.1 Max-ONEs

The Max-ONEs problem seeks an assignment that satisfies all constraints in the problem and maximizes the number of variables assigned to true. Several problems can be represented as a Max-ONEs problem, such as the "Max-Clique" problem [7]. Similarly, Vertex Cover can be reduced to Min-ONEs, which is essentially the same problem from the computational stand-point. The book [7] also defines weighted versions of those optimization problems, and the same reductions hold as in the un-weighted case.

To model Max-ONEs as an extension of CNF-SAT, we add a single PB constraint of type "$\ge$" that includes each variable with coefficient 1. "Weighted Max-ONEs" [7] problems, in which variables are assigned different coefficients, can be handled similarly. Min-ONEs can be modeled as an extension to CNF-SAT by adding a single PB constraint of type $\le$, that includes each variable with coefficient 1.

In order to find an assignment with the maximal number of *true* values, we iteratively increase the lower bound in the single PB constraint until the instance becomes unsatisfiable. The solver retains learned conflict clauses and therefore runs much faster than it would during independent starts with modified lower bounds.

In the same spirit, one can use PBS to optimally solve the Max-SAT problem [7], which has applications to Max-CUT and other discrete optimization problems. Again, this can also be trivially expressed using a single PB constraint with sliding lower bound. In this work, we perform experiments for the Max-ONEs problem only, since the Max-SAT problem can be addressed, to some extent, by existing local search solvers such as WalkSAT [17].

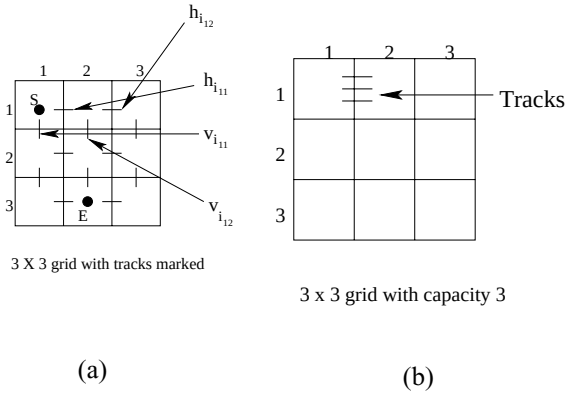

Figure 2. SAT formulations for global routing.

4.2 Global Routing

Earlier works on SAT-based routing focused on detailed routing for FPGAs, primarily because problems in that domain tend to be very constrained. With recent improvements in SAT solvers, it is reasonable to broaden SAT routing approaches to also address global routing formulations. However, such attempts were hindered so far by the presence of counting constraints that are intrinsic to global routing. In this section we demonstrate how PB constraints address this problem and lead to new PB-SAT formulations for global routing.

We use a grid model for our global routing instances. A two-dimensional grid consists of grid cells arranged in rows and columns. We refer to cell boundaries as edges, i.e. if there are two adjacent cells in a route then the route passes through the edge between the two cells. Horizontal edges exist between adjacent cells in the same row, and vertical edges exist between adjacent cells in the same column. The goal is to route a number of two-pin connections in the grid with edge capacity constraints. Edge capacities are introduced to express the constraint that there is an upper bound on the number of routes that can pass through an edge to reduce coupling. A capacity C is associated with each edge, indicating that no more than C routes can pass through that edge. To ensure that the instances are satisfiable but difficult to solve, we propose construction by randomized flooding. Namely, we create a routing configuration by adding shortest possible routes while unused routing resources (edge capacities) remain. Shortest routes are created by breadth-first-search between two randomly chosen grid cells or, if that fails, by finding a maximal shortest route starting at a given grid cell with unused routing resources. After a routing configuration is created, routes are erased and their end points are used to formulate a SAT instance.

4.2.1 CNF-SAT Formulations

Routes are specified in terms of edges across cell boundaries in a grid. A connection is routed across edges through route tracks. In the SAT formulation, each track is treated as a variable. In a grid with H rows and V columns, there are $(V-1)H$ horizontal tracks and $(H-1)V$ vertical tracks per connection. Therefore, if there are n connections, a total of $N = n((H-1)V + (V-1)H)$ variables are required to express connectivity constraints. Figure 2(a) illustrates how variables are labeled in a 3 x 3 grid. Horizontal tracks for net i are labeled $(h, i)_{r,c}$, where r and c are the row and column indices of the cell whose boundary the track crosses. Vertical tracks are labeled $(v, i)_{r,c}$.

Our method of expressing routing instances as SAT problems has two components. One deals with route definition and captures possible ways to route each connection. The other addresses capacity constraints, which restrict the number of nets that can be routed across a grid cell boundary. The route definition component is similar to the connectivity (or "liveness") constraint defined in [16] for SAT-based FPGA routing, and the capacity constraints are similar to the exclusivity constraints. The definitions of CNF constraints are available in the Appendix.

4.2.2 Pseudo-Boolean Constraints in SAT Formulations

The SAT instances outlined above are satisfiable, yet must be at least as hard as the routing instances. Moreover, the conversion to CNF is likely to make them more difficult. Incomplete SAT solvers, e.g., WalkSAT [17], are typically not able to solve such SAT instances even for modest-sized grids. Chaff [15], too, spends considerable time solving them, partly because of their sheer size.

One can see that these SAT formulations owe their size *to the CNF encoding of capacity constraints*. Connectivity constraints are encoded compactly as only one variable per track per connection is needed. For capacity constraints, one needs C variables per track per connection, and a number of clauses relate those variables. In fact, if n is the number of connections, and C is the capacity, we add $(nC + C^2)$ clauses *per track per connection*. The same constraints could be expressed much more efficiently as counting constraints in PB form. Namely, no more than C such connections should be routed through a given edge, in other words the sum of indicator variables for a given edges is upper-bounded by C.

For a PB-SAT solver, *all* capacity constraints could be encoded in just $((H-1)V + (V-1)H)$ clauses (one constraint per edge). In our experiments we compared such smaller PB-SAT instances (solved with PBS) to the original CNF instances (solved with Chaff).

5 Experimental Results

In this section, we empirically validate the algorithms described in Section 2 and Section 3 as well as PB-SAT formulations described in Section 4.

All experiments are conducted on a Pentium-II 333Mhz workstation running Linux and equipped with 512 Mbytes of RAM. Our algorithms are implemented in C++. The runtime limit is set to 5,000 seconds for all experiments. Our solver

TABLE I: Global Routing Runtime Results (in seconds)

Instance	CNF + pseudo-Boolean						pure CNF			PBS Speedup		
	V	C	#PB	PBS	SATIRE	OPBDP	V	C	Chaff	SATIRE	OPBDP	Chaff
grout3.3-1	216	572	12	1.72	**0.41**	4.51	864	7592	40.43	0.24	2.62	24
grout3.3-2	264	700	12	**0.33**	0.96	4.65	1056	10864	11.3	2.9	14.1	34
grout3.3-3	240	636	12	**0.09**	1.1	6.65	960	9156	37.21	12	74	413
grout3.3-4	228	604	12	1.29	**0.2**	4.73	912	8356	103.13	0.16	3.67	80
grout3.3-5	240	634	12	0.84	**0.35**	6.88	960	9154	71.21	0.42	8.19	85
grout4.3-1	672	2004	24	**3.46**	109.7	5000	2688	33924	1361.6	32	>1445	394
grout4.3-2	648	1928	24	**1.92**	32.13	5000	2592	31736	5000	17	>2604	>2604
grout4.3-3	648	1930	24	**5.52**	319.47	5000	2592	31738	5000	58	>906	>906
grout4.3-4	696	2072	24	**16.3**	3772	5000	2784	36176	2523	231	>307	155
grout4.3-5	720	2144	24	**2.06**	567.12	5000	2880	38504	3915	275	>2427	>1900
grout4.3-6	624	1860	24	**134**	5000	5000	2496	29628	5000	>37	>37	37
grout4.3-7	672	2006	24	**55**	5000	5000	2688	33926	772.6	>91	>91	14
grout4.3-8	432	1280	24	**2.9**	177.8	5000	1728	15320	125	61	>1724	43
grout4.3-9	840	2502	24	**376**	5000	5000	3360	51222	3203	>13	>13	8.52
grout4.3-10	840	2504	24	**7.4**	5000	5000	3360	51224	3465	>676	>676	468

PBS is compared against the SAT solvers Chaff [15] and SATIRE [23], as well as to the ILP solver OPBDP [4] whose algorithm is specialized to 0-1 problems.

We used the default settings for Chaff and OPBDP. The setting (+d DLCS) was used with SATIRE. We configured our solver to use the following:

- VSIDS decision heuristic [15],
- Optimized BCP approach with watched literals [15],
- Random restarts [10],
- Single-clause conflict analysis as suggested by the authors of Chaff [15, 25].
- Clause deletion and random backtracking are disabled.

Table I lists the runtimes for the global routing instances described in Section 4.2. Each routing instance was encoded in two ways: (i) using CNF and PB constraints where necessary, (ii) using CNF only. Table I also lists the number of variables (V), clauses (C), and PB constraints in each problem. Chaff was tested on the pure CNF formulation whereas all other solvers were tested using the general formulation which includes CNF and PB constraints. Clearly, the size of the problems, in terms of both the variables and clauses, significantly increases using the pure CNF formulation. Such instances are likely to run out of memory. Moreover, the overall runtime performance of PBS is significantly better than Chaff, SATIRE, or OPBDP. For example, PBS was able to solve the *grout4.3-2* instance in less than 2 seconds while OPBDP and Chaff time out after 5000 seconds. PBS solved all 15 instances, whereas SATIRE, OPBDP, and Chaff were only able to solve 11, 5, and 12 instances, respectively.

The results of the Max-ONEs experiment are listed in Table II. Several satisfiable instances from various benchmarks including the DIMACs [9], Bejing [11], quasi-group [24], and sat-planning [11] were tested. The table of results include the number of variables in each problem (V), the maximum possible number of variables assigned to true in a satisfying assignment (MaxONEs), the runtimes for PBS, SATIRE, and OBP, and the corresponding speedup of PBS against the other two solvers. The table clearly shows the performance improvement obtained by PBS.

6 Conclusions

In this work we have studied discrete optimization and constraint satisfaction problems related to Boolean satisfiability. We have shown that, in applications such as routing, encodings that utilize CNF and pseudo-Boolean constraints can be much more compact than pure CNF encodings. We developed a solver for this extension of the Boolean satisfiability problem, and demonstrated speed-ups (on larger benchmarks) of an order of magnitude or more against three previously published approaches, including an earlier pseudo-Boolean solver and currently-best solver for pure CNF.

Additionally, we applied our new solver to the Max-ONEs problem (a trivial modification for Min-ONEs was not explored), whose significance is due to a reduction from the Max-Clique problem (the Min Vertex Cover problem reduces to Min-ONEs). On most of the Max-ONEs benchmarks, the new solver demonstrated very impressive speed-ups over two alternative approaches constructed using previously published solvers.

By significantly improving the efficiency of SAT-based applications, particularly routing, our work opens a new avenue in this area. Our on-going work in this direction includes embedding SAT-based routers into realistic algorithmic flows and benchmarking them against best known geometric algorithms.

Our progress on the Max-ONEs problem - a fundamental, but unfortunately overlooked formulation, - also opens new applications of Davis-Putnam solvers. Our future work will study applications to Max-Clique, Max Independent Set and Min Vertex Cover - fairly popular problems in logic synthesis and other areas of design automation.

TABLE II: Max-ONEs Experiment Results (in seconds)

Benchmark	Satisfiable Instance	V	MaxONEs	PBS	SATIRE	OPBDP	PBS Speedup	
							SATIRE	OPBDP
DIMACS	**aim-50-1_6-yes1-1**	50	29	**0.01**	**0.01**	0.02	1	2
	aim-100-1_6-yes1-1	100	43	**0.01**	0.02	7.19	2	719
	aim-200-2_0-yes1_1	200	96	**0.01**	0.06	5000	6	>500K
	ii8b1	336	275	**4.69**	3180	56.2	678	12
	jnh1	100	55	0.32	2.2	**0.12**	6.88	0.38
	jnh204	100	58	0.28	1.63	**0.14**	5.82	0.50
	par8-1	350	79	**0.01**	0.06	0.05	6	5
	par8-2-c	68	20	**0.01**	0.02	**0.01**	2	1
Bejing	**3blocks**	283	63	**4.83**	49.53	4494	10.3	930
QG	**qg7-09**	729	81	**0.1**	5.41	9.8	54.1	98
	qg6-09	729	81	**0.21**	5.56	45	26.5	214
Satplan-sat	**bw_a**	459	73	**0.03**	0.43	0.21	14.3	7
	bw_b	1087	136	**0.58**	6.39	17.86	11	31
	bw_c	3016	272	**24.37**	315.5	5000	13	>205

Acknowledgments

This work is funded by the DARPA/MARCO Gigascale Silicon Research Center, an Agere Systems/SRC Research fellowship, and a DAC graduate scholarship.

References

[1] F. Aloul, I. Markov, and K. Sakallah, "Faster SAT and Smaller BDDs via Common Function Structure," *in Proc. of the International Conference on Computer Aided Design*, 2001.

[2] L. Baptista and J. P. Marques-Silva, "Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability," *in Proc. of the Int'l Conf. on Principles and Practice of Constraint Programming*, 2000.

[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] P. Barth, "OPBDP: A Davis-Putnam based Enumeration Algorithm for Linear Pseudo-Boolean Optimization," http://www.mpi-sb.mpg.de/units/ag2/software/opbdp.

[5] R. Bayardo Jr. and R. Schrag, "Using CSP look-back techniques to solve real world SAT instances," *in Proc. of the 14th National Conf. on Artificial Intelligence*, 203-208, 1997.

[6] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," *in Proc. of the Design Automation Conference*, 1999.

[7] N. Creignou, S. Kanna, and M. Sudan, "Complexity Classifications of Boolean Constraint Satisfaction Problems", SIAM, 2001.

[8] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," *in Communications of the ACM*, 5(7), 394-397, 1962.

[9] DIMACS Challenge benchmarks in *ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf*.

[10] C. P. Gomes, B. Selman, and H. Kautz, "Boosting Combinatorial Search Through Randomization," *in Proc. of the Nat'l Conf. on Artificial Intillegence,* 1998.

[11] H. Hoos and T. Stützle, http://www.satlib.org

[12] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *in IEEE Transactions on Computer-Aided Design*, 11(1), 4-15, 1992.

[13] I. Lynce, L. Baptista, and J. P. Marques-Silva, "Stochastic Systematic Search Algorithms for Satisfiability," *in the LICS Workshop on Theory and Applications of Satisfiability Testing*, 2001.

[14] J. P. Marques-Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," *in IEEE Trans. on Computers*, 48(5), 506-521, 1999.

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *in Proc. of the Design Automation Conference*, 2001.

[16] G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," *in the Proc. of the International Symposium on Physical Design*, 2001.

[17] B. Selman, H. Kautz, and B. Cohen. "Noise strategies for local search," *in Proc. of the Eleventh National Conference on Artificial Intelligence*, 337-343, 1994.

[18] L. Silva, J. Silva, L. Silveira and K. Sakallah, "Timing Analysis Using Propositional Satisfiability," *in IEEE Int'l Conf. on Electronics, Circuits and Systems*, 1998.

[19] G. Stalmarck, "System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from Boolean Formula," *United States Patent no. 5,276,897*, 1994.

[20] P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," *in IEEE Trans. on Computer-Aided Design*, 1996.

[21] M. Velev and R. Bryant, "Effective use of Boolean Satisfiability Procedure in the Formal Verification of Superscalar and VLIW Mircroprocessors," *in Proc. of the Design Automation Conference*, 2001.

[22] J. Walsor, "Solving Linear Pseudo-Boolean Constraint Problems with Local Search," *in Proc. of the National Conference on Artificial Intelligence*, 1997.

[23] J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," *in Proc. of the Design Automation Conference*, 2001.

[24] H. Zhang, "SATO: An Efficient Propositional Prover," *in Int'l Conference on Automated Deduction*, 1997.

[25] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," *in Proc. of the Int'l Conference on Computer-Aided Design*, 2001.

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

Appendix

Route Definition (Connectivity constraint)

In Figure 2(a), let the points marked S and E be the terminals of some two-terminal connection i. The SAT formulation proceeds as follows. Consider the terminal marked S. A route for this net must pass through $(h, i)_{1, 1}$ or $(v, i)_{1, 1}$. Therefore, we add the clause $((h, i)_{1, 1} + (v, i)_{1, 1})$. Clearly, both these tracks cannot be selected at the same time so we add the mutual exclusion constraint $(\overline{(h, i)_{1, 1}} + \overline{(v, i)_{1, 1}})$.

We now push the cells reachable from the possible tracks into a queue. The queue contains cells reachable from those already visited. A list of visited cells is also maintained so that a cell is not pushed on the queue twice. While the queue is not empty, cells are popped off it and new clauses are introduced for the route tracks across the cell boundaries. In our example, assume that the cell to the right of S is popped off the queue. Since this cell is not an endpoint of the connection, exactly two of its boundaries must be selected. The cell boundaries in this case are $(h, i)_{1, 2}$, $(h, i)_{1, 1}$ and $(v, i)_{1, 2}$. We therefore introduce the clauses $((h, i)_{1, 1} + (v, i)_{1, 2})$, $((h, i)_{1, 1} + (h, i)_{1, 2})$, and $((h, i)_{1, 2} + (v, i)_{1, 2})$. However, again it is not possible for more than two tracks to be selected. Therefore, we add clauses of the form: $((h, i)_{1, 1} \wedge (h, i)_{1, 2}) \Rightarrow \overline{(v, i)_{1, 2}}$. This procedure is repeated for every cell popped off the queue until the queue is empty.

Capacity Constraints

Each grid cell boundary (edge) has a capacity associated with it, to restrict the number of nets that can be routed through it. The capacity limits are intended to prevent congestion. To encode capacity constraints in the SAT formulation, if C is the capacity limit for a route track, then we include C extra variables per route track for each connection. This essentially says that each connection can be routed through one of C tracks across a cell boundary. This is illustrated in Figure 2(b).

Consider two connections i and j. Consider horizontal route tracks for each connection, $(h, i)_{r, c}$, and $(h, j)_{r, c}$ for some row r and column c. Let

$$(h, i)_{((r, c), 1)}, (h, i)_{((r, c), 2)}, \ldots, (h, i)_{((r, c), C)}$$

and

$$(h, j)_{((r, c), 1)}, (h, j)_{((r, c), 2)}, \ldots, (h, j)_{((r, c), C)}$$

be the C extra variables introduced in the SAT formulation for the horizontal track in question. Then clearly, for any $(h, i)_{((r, c), k)}$, $1 \le k \le C$, the following implications hold:

$$(h, i)_{((r, c), k)} \Rightarrow (h, i)_{r, c}$$

and

$$(h, i)_{r, c} \Rightarrow$$
$$((h, i)_{((r, c), 1)} + (h, i)_{((r, c), 2)} + \ldots + (h, i)_{((r, c), C)})$$

Clauses of this form are added to the SAT instance.

Another restriction is that a route cannot pass through two tracks for the same edge, i.e. if for some k, $1 \le k \le c$, if $(h, i)_{((r, c), k)}$ is true, then for all

$$l, 1 \le l \le C, l \ne k, (h, i)_{((r, c), k)} \Rightarrow \overline{(h, i)_{((r, c), l)}}$$

These clauses are also added. Finally, two connections cannot be routed through the same track, i.e. for all $k, 1 \le k \le C$, $(h, i)_{((r, c), k)} \Rightarrow \overline{(h, j)_{((r, c), k)}}$ for all $j \ne i$, where j represents another connection. By combining the aforementioned techniques, we are able to express routing instances as SAT problems.