Bit-Vector Optimization

Alexander Nadel(✉) and Vadim Ryvchin

Intel Corporation, P.O. Box 1659, 31015 Haifa, Israel
{alexander.nadel,vadim.ryvchin}@intel.com

Abstract. A variety of applications of Satisfiability Modulo Theories (SMT) require finding a satisfying assignment which optimizes some user-given function. Optimization in the context of SMT is referred to as Optimization Modulo Theories (OMT). Current OMT research is mostly dedicated to optimization in arithmetic domains. This paper is about Optimization modulo Bit-Vectors (OBV). We introduce two OBV algorithms which can easily be implemented in an eager bit-vector solver. We show that an industrial problem of fixing cell placement during the physical design stage of the CAD process can be reduced to optimization modulo either Bit-Vectors (BV) or Linear Integer Arithmetic (LIA). We demonstrate that our resulting OBV tool can solve industrial instances which are out of reach of existing BV and LIA OMT solvers.

1 Introduction

Nowadays, Satisfiability Modulo Theories (SMT) solving is widely applied. Traditionally, SMT solvers are expected to return *any* model (satisfying assignment), given a satisfiable formula, but many applications require a model which *optimizes* some user-given function [12,13,23,24,32,38]. The problem of finding the optimal model in SMT is called *Optimization Modulo Theories (OMT)* [35].

OMT was first addressed in [32], which presented a general OMT framework, in which the minimization/maximization cost function is restricted to *Boolean* variables. The restriction of the cost function to Boolean variables was lifted in [35]. In that work, a solution for optimization modulo linear arithmetic over the rationals was proposed, where the cost function can be an arbitrary arithmetic term. The two basic approaches to optimization, given a satisfiability solver, applied in [35], are binary and linear search, respectively, for the optimal assignment. In [35], both approaches are customized and tuned to arithmetic reasoning in the context of the DPLL(T) approach to SMT [18].

Bit-vector (BV) SMT theory [5] is a highly expressive theory, where the variables are fixed-size bit-vectors and the set of operators includes arithmetic, comparison, bit-wise, and bit-propagating (e.g., extraction, concatenation, shifts) operators. BV solvers are widely applied [15,20,25,26,34,42]. Given a BV formula F, we define the problem of *Optimization modulo Bit-Vectors (OBV)* to be the problem of finding a satisfying assignment to F which *maximizes* some user-given *target* bit-vector term t in the formula, where the term is interpreted as an unsigned number. (Minimization can be modeled as maximization of the target's negation.)

M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 851–867, 2016.
DOI: 10.1007/978-3-662-49674-9_53

Our definition lets the cost function be as generic as possible (similarly to the approach of [35] to arithmetic optimization) as the target term can be an arbitrary function over the formula's input variables. Let our maximization target be $t = [v_{n-1}, v_{n-2}, \ldots, v_0]$, where v_i's are bits and v_0 is the Least Significant Bit (LSB). Note that our semantics induces a strict priority for satisfying the bits of t in the following sense. The solver will prefer satisfying bit i while leaving the lower bits $i-1, \ldots, 0$ unsatisfied, to satisfying bits $i-1, \ldots, 0$ while leaving bit i unsatisfied (since, e.g., the value $[1000] = 8$ is higher than the value $[0111] = 7$).

Surprisingly, OBV research is scarce. We are not aware of any paper dedicated to OBV. The only existing solver supporting OBV is an extension to the Z3 SMT solver, called νZ [8,9]. νZ solves OBV by applying the following reduction to weighted MAX-SAT, proposed in [7] (where, given a set of hard Boolean clauses and a set of soft weighted Boolean clauses, weighted MAX-SAT finds a satisfying assignment to the hard clauses maximizing the weight of the satisfied soft clauses). First, the input BV formula is translated to hard Boolean clauses. Second, for each $i \in \{0, 1, \ldots, n\}$, a soft weighted unit clause (v_i) of the weight 2^i is added to the formula. The reduction guarantees that the solver will give a strictly higher priority to satisfying bit number i than to bits $i-1, \ldots, 0$, thus ensuring that t's value is maximized. Note that applying a similar reduction with *equal* weights given to the bits of t would result in maximizing the number of *satisfied bits* in t, rather than t's *value*.

This paper proposes two new algorithms for OBV solving by leveraging binary and linear search to eager BV solving [17,21]. Both algorithms are easy to implement. Both are incremental. Both take advantage of the SAT solver's conflict analysis capabilities to prune the search space on-the-fly.

The application which triggered our OBV research emerged during the placement sub-stage of the physical design stage of the Computer-Aided Design (CAD) [39] flow at Intel. Assume that after a placement of standard cells has already been generated, a new set of design constraints of different priority, introduced late in the process, has to be taken into account by the placement flow. Re-running the placer from scratch with the new set of constraints would not satisfy backward compatibility, stability, and run-time requirements, hence a new post-processing *fixer* tool is required. The goal of the fixer is to fix as many as possible of the *violations* resulting from applying the additional design constraints, with preference being given to fixing high-priority violations. We will demonstrate that this problem can be reduced to optimization modulo either bit-vectors or linear integer arithmetic (LIA). Section 6 of this work shows that our algorithms have substantially better capacity on real-world and crafted placement fixer benchmarks than νZ in both LIA and BV mode and OptiMathSAT [36,37] in LIA mode (the crafted benchmarks are publicly available at [29]).

In what follows, Sect. 2 contains preliminaries. Section 3 introduces our reduction of the placement fixer problem to optimization modulo BV and LIA. Sections 4 and 5 present our OBV algorithms. Section 6 presents the experimental results, and Sect. 7 concludes our work.

2 Preliminaries

We start off with some basic notions. A *bit* is a Boolean variable which can be interpreted as 0 or 1. A *bit-vector* of width n, $v^{[n]} = [v_{n-1}, v_{n-2}, \ldots, v_0]$, is a sequence of n bits, where bit v_0 is the Least Significant Bit (LSB) and v_{n-1} is the Most Significant Bit (MSB). We consider Boolean variables and bit-vector variables of width 1 to be interchangeable. A *constant* is a bit-vector each one of whose bits is substituted by 0 or 1. A *bit-vector operation* receives one or more bit-vectors and returns a bit-vector. A *Term DAG* is a Directed Acyclic Graph (DAG), each of whose *input nodes* (that is, nodes with in-degree 0) comprises a bit-vector or a constant and each of whose *internal nodes* (that is, nodes with in-degree > 0) is an application of a bit-vector operation over previous nodes. A *BV formula* F is a term DAG, where some of its Boolean terms are asserted to 1 (that is, they must be assigned 1 in every assignment which satisfies F).

The only assumption this paper makes about the input BV formula is that it can be translated to Conjunctive Normal Form (CNF) in propositional logic (a CNF formula is a conjunction of clauses, where each clause is a disjunction of Boolean literals, and a Boolean literal is a Boolean variable or its negation). This assumption holds for the BV language as defined in the SMT-LIB standard [5]. See [19] for a further overview of BV syntax and semantics.

Let μ be a full assignment to the variables of a BV formula F and v be a term in F. We denote by $\mu(v)$ the value assigned to v in μ, interpreted as an unsigned number.

A BV formula F is *satisfiable* iff it has a model (where a *model* is a satisfying assignment). A model μ to F is *t-maximal* iff $\mu(t) \geq \nu(t)$ for every model ν to F.

Given a BV formula F and a term t in F, where t is called the *optimization target*, let the problem of *Bit-Vector Optimization (OBV)* be the problem of finding a t-maximal model to F.

A SAT solver [6,27,40] receives a CNF formula F and returns a model, if one exists. In *incremental SAT solving under assumptions* [14,30,31], the user may invoke the SAT solver multiple times, each time with a different set of *assumption literals* and, possibly, additional clauses. The solver then checks the satisfiability of all the clauses provided so far, while enforcing the values of the current assumptions only. In the widely used Minisat's approach [14] to incremental SAT solving under assumptions, the same SAT solver instance solves the entire sequence internally. The assumptions are modeled as first decision literals in the user-given order. Each assignment to an assumption is followed by Boolean Constraint Propagation (BCP). If the solver discovers that the negation of one of the assumptions is implied by other assumptions during BCP, it halts and returns that the problem is unsatisfiable. Whenever the solver unassigns one or more of the assumptions following a backtracking or a restart, it reassigns the unassigned assumptions in the user-given order (where each assignment is followed by BCP) before picking any other decisions.

An eager BV solver [11,17] works by preprocessing the given BV formula [11, 17,28], bit-blasting it to CNF and solving with SAT.

3 Modeling the Placement Fixer Problem

This section details the placement fixing problem, mentioned in Sect. 1, and shows how to reduce it to an optimization problem modulo either BV or LIA.

3.1 Problem Formulation

We start with the problem formulation. We will be using the example in Fig. 1 for illustration.

Initial Set Up. We are given a grid of size (X, Y) and a set of n non-overlapping (but possibly touching) rectangles $r_1, \ldots, r_n$ placed on the grid. Each rectangle r_i's initial placement is given as the coordinates of its bottom-left corner (x_i, y_i), height h_i and width w_i. The example in Fig. 1 has five rectangles.

A placement of rectangles in the grid might have violations between pairs of touching rectangles. A *violation* $v(b, t, \delta)$ between the *bottom* rectangle r_b and the *top* rectangle r_t, where $1 \leq b, t \leq n$ and $-w_b < \delta < w_t$, occurs when r_b's top side touches r_t's bottom side (that is, when $y_b + h_b = y_t$) and the relative horizontal position of the rectangles is $\delta = x_b - x_t$. Each violation $v(b, t, \delta)$ has a problem-induced *unique* priority $p(b, t, \delta) \in \mathbb{N}$. In other words, the problem causes all the violations to be ranked according to their priority.

In our example shown in Fig. 1, there exist three violations of priority: $p(1, 4, -2)$, $p(4, 3, 2)$, and $p(5, 2, 0)$.

Fixer Goal. Given the initial placement, the fixer may *shift* the rectangles horizontally or vertically (that is, move each rectangle horizontally or vertically), so as to reduce the number of violations according to their priority. Shifting the same rectangle both horizontally and vertically is allowed. The priority is *strictly followed* in the sense that fixing one violation of priority p should be preferred to fixing any number of violations of priorities lower than p. Note that shifting existing rectangles might create new violations.

The input problem induces additional constraints on the allowed shifts:

1. **Shift constraints:** some of the rectangles are non-shiftable (that is, they must not be shifted), while the greatest allowed horizontal and vertical shift for any shiftable rectangle is α and β, respectively.
2. **Parity preservation:** for each rectangle the y-coordinate at the new location must be even iff the original y-coordinate is even.

Consider our example in Fig. 1. Assume that all the rectangles are shiftable and that $\alpha = 2$ and $\beta = 2$. Violation 3 can be eliminated altogether by shifting r_5 down to $(6, 0)$ (shifting it down to $(6, 1)$ is disallowed by parity preservation). The other two violations v_1 and v_2 can be resolved by shifting r_4 to the right to $(4, 3)$. Note that if r_4 had been non-shiftable, violations v_1 and v_2 could have been resolved only at the expense of creating new violations, in which case the optimal solution to the problem would have depended on the actual priorities of the violations (unspecified in our example).

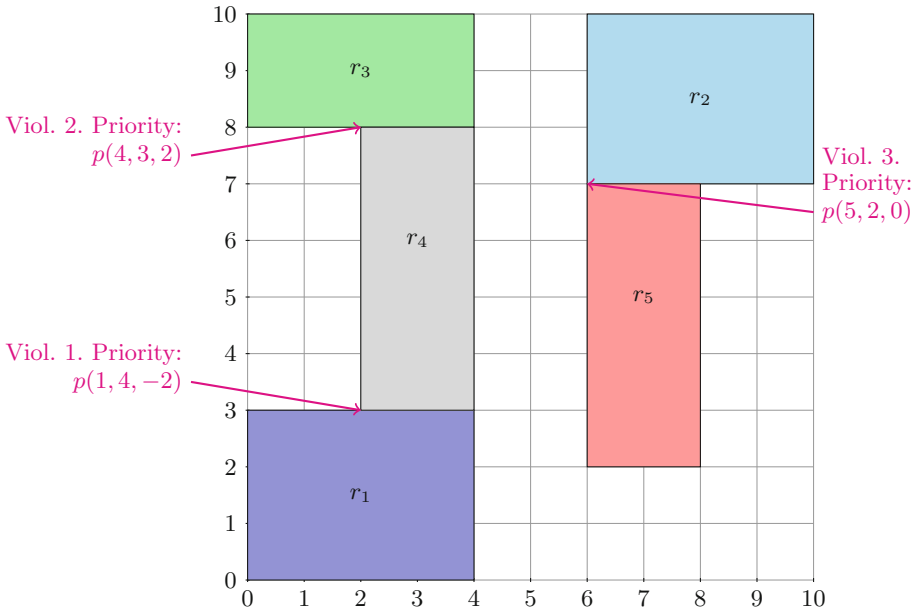

Fig. 1. Fixer placement problem modeling example.

3.2 Problem Encoding

The encoding is shown in Fig. 2. It can be applied to encode our problem into optimization modulo either BV or LIA.

First, the algorithm goes over all the shiftable rectangles. For each rectangle r_i, it creates two new variables x'_i, y'_i to represent r_i's location after the fix (the bit-width of the BV variables is chosen to accommodate the size of the grid). In addition, the algorithm ensures that the parity is preserved. For BV, the parity preservation constraint can be modeled by asserting $y_i \& 1 == y'_i \& 1$ (where & stands for bit-wise AND). For LIA, it can be modeled in either one of the two following ways: *(a)* using an auxiliary variable t to assert that $y_i - y'_i == 2t$, or *(b)* using LIA's native mod operator to assert that $y_i \bmod 2 == y'_j \bmod 2$.

Second, the algorithm ensures that the rectangles will not overlap after the fix. This can easily be done for both BV and LIA by adding inequalities for each pair of rectangles over the new variables x'_i, y'_i, x'_j, y'_j to ensure there is no overlap.

Third, the algorithm creates the target term to be used for BV maximization (adjusting our construction to LIA reasoning is explained in the next paragraph). It starts by creating an empty bit-vector u. It then goes over all the potential violations in a loop, in order of priority, starting with the violation of the lowest priority. It formulates a condition c which holds iff the violation occurs after the fix. Then the negation of c is inserted into u as the MSB (using concatenation).

for each shiftable $r_i \in R$ **do**
    Declare two variables x'_i, y'_i for r_i's location after the fix
    Assert $max(0, x_i - \alpha) \leq x'_i \leq min(x_i + \alpha, X - w_j)$ ▷ x-boundaries
    Assert $max(0, y_i - \beta) \leq y'_i \leq min(y_i + \beta, Y - h_j)$ ▷ y-boundaries
    Assert $y_i \bmod 2 == y'_j \bmod 2$ ▷ Parity preservation
for every pair $r_i, r_j \in R, i \neq j$ **do**
    if r_i and r_j can overlap **then** prevent overlapping between r_i and r_j by adding respective inequalities
Let u be an empty bit-vector
for each potential violation $v = (b, t, \delta)$ in increasing priority order **do**
    $c := (y'_b + h_b = y'_t) \wedge (\delta = x'_b - x'_t)$ ▷ violation v occurs after the fix
    $u = \neg c$ concat u
if Optimize for BV **then**
    Maximize u
else ▷ Optimize for LIA
    Maximize each bit of u in lexicographic order starting from the MSB and going towards the LSB

Fig. 2. Placement fixer: encoding

Our construction guarantees that the fixer accomplishes the task of generating a placement having as few violations as possible while strictly following the priority, iff u is given the maximal value. Hence, the solver is asked to maximize the value of u in the case where BV reasoning is applied. To achieve the same effect for LIA, the algorithm maximizes the bits of u lexicographically starting from the MSB and going towards the LSB (lexicographical maximization for LIA is available in both νZ and OptiMathSAT).

Note that one cannot use integer linear programming (ILP) to encode our problem efficiently, since our problem requires using *disjunctive constraints* to prevent overlaps between pairs of rectangles. Specifically, given any two rectangles r_1 and r_2, it is *either* that $x'_1 > x'_2 + w_2$ or $x'_2 > x'_1 + w_1$ (similar equations must be generated for y coordinates). One could, though, use Linear Disjunctive Programming (LDP) [2,3] to encode our problem. We have left the non-trivial work of reducing our problem to LDP to the future.

4 Optimization with Weak Assumptions

Our first OBV algorithm is based on a modification to Minisat's approach to SAT solving under assumptions, called SAT solver under *weak assumptions*. We call our algorithm `OBV-WA` (standing for Optimization modulo Bit-Vectors with Weak Assumptions). It can also be understood as a linear search for the t-maximal model starting with the highest possible value of t and going towards 0, where the algorithm stops at the first satisfying assignment. Section 6 will demonstrate that `OBV-WA` is substantially more efficient than the Naïve Linear Search (NLS) algorithm, depicted below (given a satisfiable formula F and the optimization target t):

1: Solve F with an SMT solver
2: **while** Solve returns SAT **do**
3: Assert t is greater than t's value in the last model returned by Solve
4: Solve F with an SMT solver
5: **return** last model returned by Solve

4.1 `OBV-WA` Algorithm

Assume an eager BV solver is provided with a satisfiable BV formula F and an optimization target $t^{[n]}$ and is requested to find a t-maximal model to F (one can verify that F is satisfiable by invoking a BV solver before applying our algorithm). First, `OBV-WA` translates F to CNF (following an optional invocation of word-level preprocessing). Then it applies a SAT solver, where literals corresponding to the bits $t_{n-1}, t_{n-2}, \ldots, t_0$ are provided to the solver as weak assumptions which are processed as follows. The SAT solver assigns the weak assumptions as the first decision variables in the specified order (from the MSB t_{n-1} towards the LSB t_0), where BCP follows each assignment. If the solver discovers that the negation of one of the assumptions is implied by other assumptions during BCP, it *continues to the next assumption* (in contrast to returning that the problem is unsatisfiable, as in Minisat's approach to SAT solving under assumptions).

This simple adjustment of Minisat's algorithm guarantees that the solver returns a t-maximal model. Indeed, `OBV-WA` checks the satisfiability of F under every t value starting from $t = 2^n - 1$ towards $t = 0$ in decreasing order. t is decreased by $\delta > 1$ only once the solver *proves* that there is no model in the range $[t, t - \delta + 1]$. Indeed, the bit t_i is flipped by the solver to 0 only if there is no model to F with $t_i = 1$.

The algorithm in Algorithm 1 is an implementation of `OBV-WA`. It contains the following three functions:

1. Solve: the main function invoked by the user: given a BV formula F and an optimization target $t^{[n]}$, it returns a t-maximal model. The function initializes an index i, which points to the next unassigned assumption, with $n-1$. It also initializes *dl_wa* to 0, where *dl_wa* is the highest decision level where a weak assumption is assigned as a decision literal. It then invokes a SAT solver with decision and backtrack strategies modified as specified below. The algorithm returns the model found by the SAT solver (we assume an implicit conversion from the Boolean model returned by the SAT solver to the corresponding BV model to the original formula).
2. OnDecision: invoked by the underlying SAT solver to get a decision literal when it has to take a decision. It receives the next decision level. OnDecision returns the next unassigned assumption, if any, and decreases the index i by 1. Assigned assumptions are skipped. If an unassigned assumption is found, the function stores the assumption's index in a decision level indexed array *SavedI* and updates *dl_wa*. This is required for proper backtracking. If all the assumptions are assigned, a standard SAT decision heuristic is applied.

3. OnBacktrack: invoked by the SAT solver whenever it backtracks. It receives the decision level to backtrack to. If the decision level is higher than dl_wa, nothing is done. Otherwise, the function updates the assumption index i so as to point to the next unassigned assumption. It also updates dl_wa accordingly.

Note that the decision level of an assigned weak assumption i might be different from $n - i$, since any assumption could entail other assumptions at the same decision level. For this reason, the algorithm must maintain the mapping *SavedI* from the decision level dl of each assigned weak assumption to its index i.

An approach similar to SAT solving under *unordered* weak assumptions has recently been used in [10] to reduce the number of faults in model-based safety analysis. The contribution of our work is in reducing OBV to SAT solving under weak assumptions, where the assumptions must correspond to the target variable bits, ordered from the MSB towards the LSB.

4.2 Incrementality

`OBV-WA` is incremental in the same sense as Minisat's algorithm: it can be invoked multiple times with different optimization targets, where the formula can be extended between the invocations. This type of incrementality is now supported in the new SMT-LIB format SMT-LIB 2.5 [4]. To support incremental push/pop, another type of incrementality inherited by SMT-LIB 2.5 from SMT-LIB 2.0, one can use selector literals as follows: following each push, add a fresh selector literal s to every clause in the bit-blasted formula and then add $\neg s$ as a (strong) assumption. To pop, add the unit clause $\neg s$. To use both strong and weak assumptions in one invocation, simply assign first the strong assumptions and then the weak ones.

5 Optimization with Inline Binary Search

In this section we present our second OBV algorithm, called `OBV-BS` (standing for Bit-Vector Optimization with Binary Search). We will see in Sect. 6 that `OBV-BS`'s is considerably more efficient than the Naïve Binary Search (NBS) algorithm performing a binary search for the maximal t value using the SMT solver as an oracle.

5.1 `OBV-BS` Algorithm

Like `OBV-WA`, this algorithm first translates the formula to CNF. It then applies a binary search-style algorithm implemented on top of an incremental SAT solver.

We need to extend our definitions for the subsequent discussion in the context of OBV solving given a formula F and the optimization target t. Let the *value* of an assignment α to F be $\alpha(t)$ (that is, the value assigned to the target t in α).

For a partial assignment α, we define its value $\alpha(t)$ to be equal to $\alpha_0(t)$, where α_0 extends α by assigning 0 to all the unassigned bits of t. Values of assignments

Algorithm 1. `OBV-WA` – OBV with Weak Assumptions

```
 1: function Solve(BV Formula F, Optimization Target t^[n])
Require: F is satisfiable
Ensure: A t-maximal model to F is returned
 2:     Pre-process and bit-blast F to CNF
 3:     i := n − 1                                   ▷ n − 1 is the MSB
 4:     dl_wa := 0
 5:     μ := SAT()
 6:     return μ

 7: function OnDecision(Decision level dl)
 8:     while i ≥ 0 and t_i is assigned do
 9:         i := i − 1
10:     if i < 0 then
11:         return StandardSATHeuristic(dl)
12:     SavedI(dl) := i
13:     dl_wa := dl
14:     return t_i

15: function OnBacktrack(Decision level dl)
16:     if dl ≤ dl_wa then
17:         i := SavedI(dl)
18:         dl_wa := dl − 1
```

induce an order between them. In particular, an assignment α is *higher*, *lower*, or *equal* to β, if $\alpha(t) > \beta(t)$, $\alpha(t) < \beta(t)$, or $\alpha(t) = \beta(t)$, respectively. We sometimes interpret assignments to F as Boolean assignments, assigning values to the bits of BV variables individually. Alternatively, we sometimes interpret assignments to F as sets of Boolean literals, where each assigned bit b of a BV variable appears as either b or $\neg b$.

Consider Algorithm 2 implementing `OBV-BS`. The algorithm maintains the current model μ, initialized with an arbitrary model to F at line 3, and a partial assignment α, which is empty in the beginning. The main loop of the algorithm (starting at line 5) goes over all the bits of the optimization target t starting from the MSB t_{n-1} down to t_0. Each iteration extends α with either t_i or $\neg t_i$, where t_i is preferred over $\neg t_i$ iff there exists a model where t_i is assigned 1 while bits higher than i have already been assigned in previous iterations. In other words, t_i is preferred whenever there exists a model whose value is greater than or equal to $\alpha(t) + 2^i$. Essentially, the algorithm implements a binary search over all the possible values of the optimization target t, where the search is automatically pruned based on the conclusions of the SAT solver's conflict analysis.

The algorithm is incremental in the same sense as `OBV-WA`, that is, it fully supports Minisat-style incremental solving under assumptions, while push/pop can be supported through selector variables.

5.2 Correctness Proof

Three invariants, which hold throughout the algorithm at the beginning of the algorithm's loop, are shown in Fig. 3. According to Inv. 1, μ must be a model. According to Inv. 2 and 3, α is always a subset of μ and any t-maximal model, respectively (where the assignments are interpreted as sets of Boolean literals). Note that if the invariants hold, then at the end of the algorithm μ is a t-maximal model, since: (a) by the end α will have assigned values to every bit of t, (b) Inv. 2 ensures that μ agrees with α on all bits of t and (c) Inv. 3 guarantees that α agrees on all bits of t with t-maximal models.

The invariants clearly hold just before the first loop iteration. Consider an arbitrary iteration of the loop. We assume that the invariants hold at its beginning.

First, the algorithm checks whether the current bit t_i is 1 in μ (at line 6). If it is, α is simply extended with t_i and the algorithm goes on to the next iteration. Let us verify that the invariants hold at the end of an iteration in this case. First, μ is not changed, hence Inv. 1 still holds. Second, α is extended with a μ literal, thus Inv. 2 is preserved. Inv. 3 and 2 hold in the beginning of the iteration, hence any t-maximal model ν agrees with α and μ on the values of the Boolean variables $t_{n-1}, \ldots, t_{i+1}$. Any such ν must also contain t_i positively, since otherwise μ's value would have been higher than that of ν. Thus, Inv. 3 is preserved.

Assume now that $t_i = 0$ in μ, that is $\neg t_i \in \mu$. In this case (the treatment of which starts at line 9), the algorithm checks whether there exists a model (different from μ) that extends α with t_i. It does this by invoking a SAT solver and providing it α and t_i as (strong) assumptions.

If the problem is satisfiable and a model τ is found, we update μ to τ and continue to the next iteration of the loop. Let us verify the invariants at the end of the loop for this case. μ is still a model after the update, so Inv. 1 holds. α still agrees with μ on all α values, since the α values have been provided to the SAT solver as assumptions, so the updated μ must contain them. Thus, Inv. 2 is preserved. Inv. 3 still holds, since α has not been changed.

In the only remaining case, if the SAT solver returns UNSAT, we extend α with $\neg t_i$. Let us verify the invariants. Inv. 1 is preserved, since μ is not changed. Inv. 2 is preserved, since μ must contain $\neg t_i$ according to our algorithm's flow (otherwise, the condition at line 6 would hold). Finally, any t-maximal model must still agree with α, preserving Inv. 3 for the following reasons. The only potential disagreement could be regarding the value of t_i, since Inv. 3 holds at the beginning of the loop. But the outcome of our SAT query guarantees that there is no model containing α and t_i, hence any t-maximal model must contain $\neg t_i$.

5.3 Performance Optimizations

We have implemented two important performance optimizations for Algorithm 2:

1. μ is a model.
2. $\alpha \subseteq \mu$.
3. $\alpha \subseteq \nu$ for every t-maximal model ν.

Fig. 3. `OBV-BS` invariants

Algorithm 2. `OBV-BS` – OBV with Inline Binary Search

```
 1: function SOLVE(BV Formula F, Optimization Target t^[n])
Require:  F is satisfiable
Ensure:  A t-maximal model to F is returned
 2:     Pre-process and bit-blast F to CNF
 3:     μ := SAT()
 4:     α := {}
 5:     for i ← n − 1 downto 0 step 1 do
 6:         if t_i ∈ μ then                          ▷ t_i ∈ μ ≡ t_i = 1 in μ
 7:             α := α ∪ {t_i}
 8:         else
 9:             τ := SATUNDERASSUMPTIONS(α ∪ {t_i})
10:             if SAT solver returned SAT then
11:                 μ := τ
12:             else
13:                 α := α ∪ {¬t_i}
14:     return μ
```

1. In *non-incremental mode*, one can add unit clauses instead of the assumptions at lines 7 and 13. This is expected to boost the performance, since it has been shown that using unit clauses instead of assumptions results in a substantial performance improvement in the context of incremental SAT solving under assumptions [28,30].
2. Modern SAT solvers apply phase saving [16,33,41] as their polarity selection heuristic. In phase saving, once a variable is picked by the variable decision heuristic, the literal is chosen according to its latest value, where the values are normally initialized with 0. In our implementation of `OBV-BS` we initialize the phase saving values of all the bits of the optimization target t to 1 in each invocation, encouraging the solver to prefer a higher value for t's bits by default. This optimization allows the algorithm to converge faster.

5.4 Comparing `OBV-WA` and `OBV-BS`

Let us compare `OBV-WA` and `OBV-BS` at a high-level. `OBV-WA` should work better when the t-optimal model's value has many 1's in it, since `OBV-WA` tries to assign 1's to all the bits of t whenever possible. Otherwise, `OBV-BS` is expected to perform better. In addition, `OBV-BS` has the advantage that it always has an approximation of the maximal model that can be returned to the user if optimality can be traded for performance. `OBV-WA` does not have intermediate non-optimal solutions.

6 Experimental Results

We have implemented our algorithms `OBV-WA` and `OBV-BS` in Intel's eager BV solver Hazel. This section studies the performance of `OBV-WA` and `OBV-BS` on industrial placement fixer benchmarks as well as publicly available placement fixer benchmarks crafted by us [29].

The crafted benchmarks consist of diversified instances of the generic problem of placing rectangles on a grid, described in Sect. 3. First, we created a number of families, where a family is defined per grid size $g \times g$, where $g \in \{10, 25, 50, 75, 100\}$. Each family consists of 40 benchmarks. Let the *density* of a benchmark $d \in \{0.2, 0.5, 0.7, 0.9\}$ be the fraction of occupied grid cells. Each family has 10 benchmarks for each of the four possible density values. The size and coordinates of the rectangles for each benchmark are drawn randomly, where the size of rectangles' sides is drawn from the set $\{1, 2, \ldots, \lceil g/10 \rceil\}$. Second, we crafted another family of high-density instances, called *HD (High-Density)*, for grid size 50×50. Each benchmark in the HD family was created by placing rectangles on the grid until all the room was exhausted.

For the comparison we used two publicly available OMT solvers: νZ [8,9] (version 4.3.3) in BV and LIA modes, and OptiMathSAT [36,37] (version 1.3.5) in LIA mode. νZ and OptiMathSAT are extensions of the leading SMT solvers Z3 and MathSAT, respectively, for OMT. Note that νZ is the only available solver that supports OBV.

Recall from Sect. 3.2 that we presented two ways of encoding the parity preservation constraint $y_i \bmod 2 == y'_j \bmod 2$: (a) using an auxiliary variable t to assert that $y_i - y'_i == 2t$, or (b) using LIA's native mod operator. We experimented with νZ in LIA mode on benchmarks generated with both encodings. νZ-BV, νZ-LIA, and νZ-LIA-m below stand for, respectively, νZ in BV mode, νZ in LIA mode using auxiliary variables to encode parity constraints, and νZ in LIA mode using the LIA's native mod operator to encode parity constraints. We used OptiMathSAT with only the auxiliary variable-based encoding, since OptiMathSAT does not support the mod operator.

We have also implemented the Naïve Linear Search (NLS) and Naïve Binary Search (NBS) algorithms (recall the beginning of Sects. 4 and 5, respectively) on top of Hazel.

We used machines with 32 GB of memory running Intel® Xeon® processors with 3 GHz CPU frequency. The time-out was set to 1800 s. Detailed experimental results are available in [29].

Consider Table 1. It presents the number of instances solved within the time-out per family, where a family is defined per grid size for all crafted instances, except for the HD family. In addition, we considered a family of 50 industrial instances. The family name is shown in column 1. Column 2 shows the average number of unsatisfied bits in the optimization target t (in the optimal solution), while column 3 provides the number of SAT calls within `OBV-BS` on average. The number of instances per family is shown in column 4. (Statistics are not available for the industrial instances because of IP considerations.) The subsequent columns present the number of instances solved for a particular solver.

Table 1. Comparing OBV algorithms

Grid size	UNSAT bits in t	#SAT calls in OBV-BS	#	OBV-WA	OBV-BS	Opti-MathSAT	νZ-BV	νZ-LIA	νZ-LIA-m	NLS	NBS
10×10	7	11	40	40	40	40	40	40	40	39	40
25×25	6	38	40	40	40	12	40	40	40	9	7
50×50	50	77	40	40	40	0	7	23	20	0	0
75×75	75	110	40	40	40	0	0	0	1	0	0
100×100	0.025	182	40	40	40	0	0	0	0	0	0
Industrial			50	50	50	0	0	0	0	0	0
HD	1324	889	54	1	54	0	0	0	0	0	0

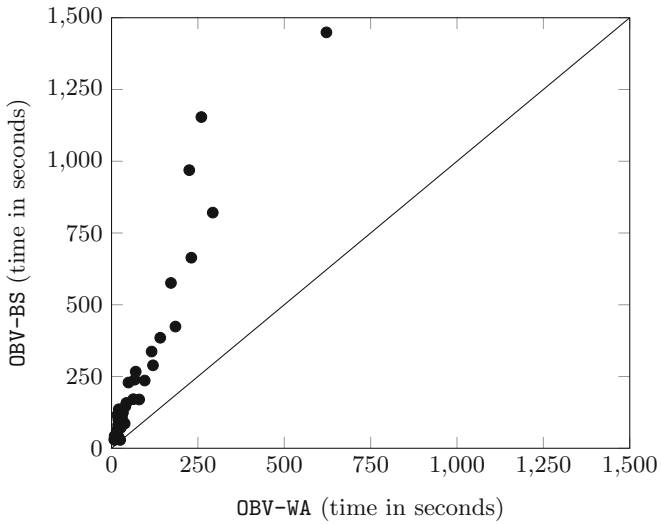

Fig. 4. Comparing `OBV-WA` to `OBV-BS` on 100×100 grids.

Consider the non-HD crafted instances and the industrial instances. Our algorithms clearly outperform the current state-of-the-art. Both `OBV-WA` and `OBV-BS` solve all the non-HD crafted instances and all the industrial instances. None of the other solvers can solve a single industrial instance. νZ, in each one of the three modes, solves only a portion of the crafted 50×50 instances, and can solve none of the crafted 100×100 instances. OptiMathSAT is outperformed by the other solvers on the crafted instances. The naïve binary and linear search algorithms (NBS and NLS) are not competitive.

Figures 4 and 5 compare `OBV-WA` to `OBV-BS` head-to-head on 100×100 grids and industrial instances, respectively. One can see that `OBV-WA` consistently outperforms `OBV-BS` on both the crafted and the industrial instances. In light of these results, `OBV-WA` is now applied for the placement fixing problem at Intel.

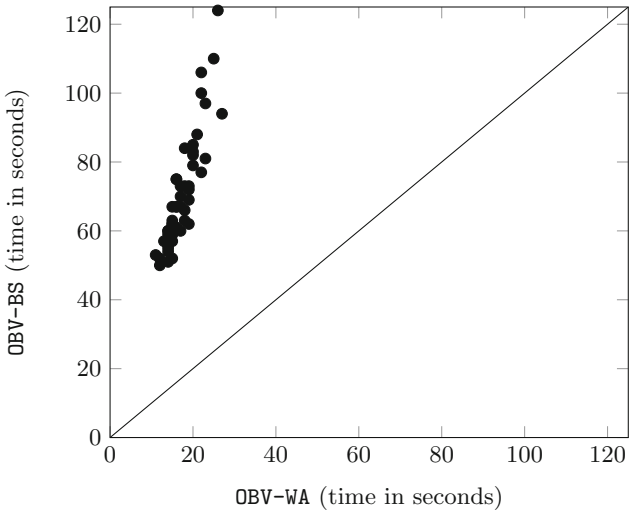

Fig. 5. Comparing `OBV-WA` to `OBV-BS` on industrial instances.

Strikingly, the apparent advantage of `OBV-WA` does not extend to the HD family. `OBV-BS` solves all the HD instances, while `OBV-WA` only solves a single HD instance (the other solvers solve none of the HD instances). This phenomenon is explained by the fact the number of unsatisfied bits in the maximal solution is significantly higher for the HD family. Our conclusion is that `OBV-BS` is more robust than `OBV-WA`, but in practice `OBV-WA` might still be preferred, if the instances are not too difficult.

7 Conclusion

This paper is the first full-blown work dedicated to the problem of Optimization modulo Bit-Vectors (OBV). We have presented two incremental OBV algorithms, which can easily be implemented in an eager Bit-Vector (BV) solver.

We have implemented our algorithms and studied their performance on real-world instances emerging in the industrial problem of fixing cell placement during the physical design stage of CAD process. The problem can be encoded as either optimization modulo BV or Linear Integer Arithmetic (LIA). We have also experimented with crafted, publicly-available instances that mimick the placement fixing problem.

Our algorithms have shown substantially better capacity than the state-of-the-art Optimization Modulo Theories (OMT) solvers νZ and OptiMathSAT, where OptiMathSAT has been applied in LIA mode and νZ in both BV and LIA modes.

As a future work we intend to study the integration of our algorithms with more recent approaches to incremental SAT solving under assumptions [31]. In addition, we are planning to apply our OBV algorithms to other problems.

Acknowledgments. The authors would like to thank Paul Inbar for editing the paper and Eran Talmor for providing useful suggestions that helped to improve this work.

References

1. Baier, C., Tinelli, C. (eds.): TACAS 2015. LNCS, vol. 9035. Springer, Heidelberg (2015)
2. Balas, E.: Disjunctive programming: properties of the convex hull of feasible points. Discrete Appl. Math. **89**(1–3), 3–44 (1998)
3. Balas, E., Bonami, P.: New variants of lift-and-project cut generation from the LP tableau: open source implementation and testing. In: Fischetti, M., Williamson, D.P. (eds.) IPCO 2007. LNCS, vol. 4513, pp. 89–103. Springer, Heidelberg (2007)
4. Barrett, C., Fontaine, P., Stump, A., Tinelli, C.: The SMT LIB standard. Version 2.5. http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf
5. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: version 2.0. In: Gupta, A., Kroening, D. (eds.) Proceedings of the 8th International Workshop on Satisfiability Modulo Theories, Edinburgh, UK (2010)
6. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
7. Bjørner, N.: Private communication, March 2015
8. Bjørner, N., Phan, A.: νz - maximal satisfaction with Z3. In: Kutsia, T., Voronkov, A. (eds.) 6th International Symposium on Symbolic Computation in Software Science, SCSS 2014. EPiC Series, vol. 30, Gammarth, La Marsa, Tunisia, 7–8 December 2014, pp. 1–9. EasyChair (2014)
9. Bjørner, N., Phan, A., Fleckenstein, L.: νz - an optimizing SMT solver. In: Baier and Tinelli [1], pp. 194–199
10. Bozzano, M., Cimatti, A., Griggio, A., Mattarei, C.: Efficient anytime techniques for model-based safety analysis. In: Kroening and Pasareanu [22], pp. 603–621
11. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Kowalewski and Philippou [21], pp. 174–177
12. Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R., Stenico, C.: Satisfiability modulo the theory of costs: foundations and applications. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 99–113. Springer, Heidelberg (2010)
13. Dillig, I., Dillig, T., McMillan, K.L., Aiken, A.: Minimum satisfying assignments for SMT. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 394–409. Springer, Heidelberg (2012)
14. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
15. Franzén, A., Cimatti, A., Nadel, A., Sebastiani, R., Shalev, J.: Applying SMT in symbolic execution of microcode. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 121–128. IEEE (2010)
16. Frost, D., Dechter, R.: In search of the best constraint satisfaction search. In: AAAI, pp. 301–306 (1994)
17. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
18. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)

19. Hadarean, L.: An Efficient and Trustworthy Theory Solver for Bit-vectors in Satisfiability Modulo Theories. Dissertation, New York University (2015)
20. Katelman, M., Meseguer, J.: vlogsl: a strategy language for simulation-based verification of hardware. In: Barner, S., Harris, I., Kroening, D., Raz, O. (eds.) HVC 2010. LNCS, vol. 504, pp. 129–145. Springer, Heidelberg (2011)
21. Kowalewski, S., Philippou, A. (eds.): TACAS 2009. LNCS, vol. 5505. Springer, Heidelberg (2009)
22. Kroening, D., Păsăreanu, C.S. (eds.): CAV 2015. LNCS, vol. 9206. Springer, Heidelberg (2015)
23. Li, Y., Albarghouthi, A., Kincaid, Z., Gurfinkel, A., Chechik, M.: Symbolic optimization with SMT solvers. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2014, San Diego, CA, USA, 20–21 January 2014, pp. 607–618. ACM (2014)
24. Manolios, P., Papavasileiou, V.: ILP modulo theories. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 662–677. Springer, Heidelberg (2013)
25. Marić, F., Janičić, P.: URBiVA: uniform reduction to bit-vector arithmetic. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 346–352. Springer, Heidelberg (2010)
26. Michel, R., Hubaux, A., Ganesh, V., Heymans, P.: An SMT-based approach to automated configuration. In: SMT Workshop 2012 10th International Workshop on Satisfiability Modulo Theories SMT-COMP 2012, p. 107 (2012)
27. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, 18–22 June 2001, pp. 530–535. ACM (2001)
28. Nadel, A.: Bit-vector rewriting with automatic rule generation. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 663–679. Springer, Heidelberg (2014)
29. Nadel, A., Ryvchin, V.: Bit-vector optimization: benchmarks and detailed results. https://goo.gl/epFbO1
30. Nadel, A., Ryvchin, V.: Efficient SAT solving under assumptions. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 242–255. Springer, Heidelberg (2012)
31. Nadel, A., Ryvchin, V., Strichman, O.: Ultimately incremental SAT. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 206–218. Springer, Heidelberg (2014)
32. Nieuwenhuis, R., Oliveras, A.: On SAT modulo theories and optimization problems. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 156–169. Springer, Heidelberg (2006)
33. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
34. Romano, A., Engler, D.: Expression reduction from programs in a symbolic binary executor. In: Bartocci, E., Ramakrishnan, C.R. (eds.) SPIN 2013. LNCS, vol. 7976, pp. 301–319. Springer, Heidelberg (2013)
35. Sebastiani, R., Tomasi, S.: Optimization in SMT with $\mathcal{LA}$ (Q) cost functions. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 484–498. Springer, Heidelberg (2012)
36. Sebastiani, R., Tomasi, S., Trentin, P.: Optimathsat. http://optimathsat.disi.unitn.it

37. Sebastiani, R., Trentin, P.: Optimathsat: a tool for optimization modulo theories. In: Kroening and Pasareanu [22], pp. 447–454
38. Sebastiani, R., Trentin, P.: Pushing the envelope of optimization modulo theories with linear-arithmetic cost functions. In: Baier and Tinelli [1], pp. 335–349
39. Sherwani, N.A.: Algorithms for VLSI Physical Design Automation, 3rd edn. Kluwer Press, Dordrecht (1998)
40. Silva, J.P.M., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
41. Strichman, O.: Tuning SAT checkers for bounded model checking. In: Allen Emerson, E., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
42. Wille, R., Große, D., Haedicke, F., Drechsler, R.: SMT-based stimuli generation in the SystemC verification library. In: Borrione, E. (ed.) Advances in Design Methods from Modeling Languages for Embedded Systems and SoCs. LNEE, vol. 63, pp. 227–244. Springer, Heidelberg (2010)