# Improving Local Search for Bit-Vector Logics in SMT with Path Propagation

Aina Niemetz, Mathias Preiner, Andreas Fröhlich, and Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria

***Abstract*—Bit-blasting is the main approach for solving word-level constraints in SAT Modulo Theories (SMT) for bit-vector logics. However, in practice it often reaches its limits, even if combined with sophisticated rewriting and simplification techniques. In this paper, we extended a recently proposed alternative based on stochastic local search (SLS) and improve neighbor selection based on down propagation of assignments. We further reimplemented the previous SLS approach in our SMT solver Boolector and confirm its effectiveness. We then added our novel propagation-based extension and provide an extensive experimental evaluation, which suggests that combining these techniques with Boolector's bit-blasting engine enables Boolector to solve substantially more instances.**

## I. Introduction

SAT Modulo Theories (SMT) procedures for deciding the satisfiability of first order formulas w.r.t. the theory of fixed-size bit-vectors usually employ a so called *bit-blasting* approach, where the input formula is eagerly reduced to propositional logic (SAT). While efficient in practice, it heavily relies on rewriting to simplify the input prior to bit-blasting, and consequently, on the underlying SAT solver. This method, however, does in general not scale if the input size can not be reduced sufficiently. Lazy approaches based on DPLL(T) as in [2][7], on the other hand, aim to improve solver performance by employing a layered approach—in the latter case in a parallel portfolio setting together with the standard bit-blasting approach. Attacking the problem from a different angle, in [5], Fröhlich et al. proposed a stochastic local search (SLS) procedure to solve bit-vector formulas directly on the theory level, i.e., on the word-level, without the need for a SAT solver. In contrast to [6], where Griggio et al. attempted to reproduce previous successful applications of SLS in the SAT domain (e.g. [8]) by integrating a bit-level SLS solver with the SMT solver MathSAT [3], lifting SLS to the theory level delivered promising initial results. However, we argue that neighborhood exploration, as suggested in [5], does not yet fully exploit the advantage of working on the theory level. In essence, it mostly simulates bit-level local search by focusing on single bit flips.

In this paper, we first reimplemented the word-level local search approach introduced in [5] in our SMT solver Boolector, the winner of the QF_BV track of the SMT competition 2015, and confirm its effectiveness as presented in [5]. We then aim at improving neighbor selection as in [5] by introducing so called *propagation moves*. That is, rather than almost solely relying on bit flips of bit-vector and Boolean variables (driven by a scoring function), we introduce an additional strategy to satisfy lines by propagating assignments from the outputs to the inputs. We extended the SLS engine in Boolector with our propagation-based strategy and provide an extensive experimental evaluation which shows that using these techniques in combination with Boolector's bit-blasting engine in a sequential portfolio manner [10] considerably improves its performance.

## II. SLS for QF_BV at a glance

The core SLS engine as implemented in Boolector is similar to the SLS architecture presented in [5] with some exceptions, which are mainly due to implementation issues. In the following, we will highlight all relevant differences and give an overview of the general workflow corresponding to the algorithm depicted in Figure 1. Note that, as in bit-level local search, the given word-level local search procedure is incomplete in the sense that it is not able to determine unsatisfiability.

Given a bit-vector formula $\phi$, procedure `sat` initially applies several rewriting and simplification techniques to yield a simplified formula $\pi$ (line 5), which in the following serves as the input to the actual SLS procedure. In contrast to [5], we do not impose restrictions to the bit-vector logic definition, i.e., we do not require $\pi$ to be in Negation Normal Form (NNF). However, without loss of generality, we do restrict the set of Boolean expressions

This work was funded by the Austrian Science Fund (FWF) under NFN Grant S11408-N23 (RiSE).

```
1  procedure sat(φ)
2    global α  // assignment
3    global s  // score
4    global π  // simplified formula
5    π := simplify(φ)
6    init_constraint_weights()
7    for i=1 to ∞
8      α := init_inputs(π)
9      s := compute_score()
10     for j=0 to max_moves(i)
11       if all_constraints_sat()
12         return SAT
13       root := select_constraint()
14       var, val := select_move(root)
15       update_assignments(var,val)
16       if is_randomized_move(var,val)
17           update_constraint_weights()
18       update_score()
```

Fig. 1. The core SLS procedure in pseudo-code.

```
1  procedure select_move(root)
2    choose depending on a ratio n:m
3      var, val := select_prop_move(root)
4    else
5      var, val := select_sls_move(root)
6    return var, assignment
```

Fig. 2. Procedure `select_move` in pseudo-code. Move selection depends on a ratio $n : m$ of propagation to sls moves.

to the set of unary and binary operators $\{\neg, \wedge, =, <\}$. We further represent formula $\pi$ as a directed acyclic graph (DAG) with (possibly) multiple roots, which we also refer to as *constraints*.

Given a set of constraints $\{a_1, ..., a_m\}$ in $\pi$, we adopt the constraint weighting scheme in [5] and associate each constraint $a_i$ with a weight $w_i$, which is initialized with 1 and updated whenever no propagation and no regular SLS move could be found, and a random variable/value pair is chosen. As in [5], we then define the notion of states of a local search algorithm for an SMT bit-vector problem based on the values of the constraint weights and the assignments to its inputs, i.e., a set of Boolean and bit-vector variables. In the following, we refer to 0-arity bit-vector function symbols as *bit-vector variables*, and to numerical constants (e.g. `#bvX` in SMT-LIB notation [1]) as *bit-vector constants*. We further implicitly treat Boolean variables as bit-vector variables of bit-width one and include them in all definitions over bit-vector variables if not otherwise noted.

Given the simplified formula $\pi$, as in [5], we define the initial state of the SLS procedure by initializing all constraint weights with one (line 6) and all bit-vector variables with zero (line 8), and yield an initial assignment $\alpha$. Starting from this initial assignment, the SLS procedure then iteratively moves to neighboring states until a satisfying assignment is found. The actual local search procedure consists of two loops (line 7-12), where the inner loop (line 10-12) represents a single round of search, and the outer loop realizes restarts after a certain number of moves has been performed. Given a constant $c_2$ (we choose $c_2 = 100$), the maximum number of moves in a single search round is defined as in [5] as

$$\texttt{max\_moves}(i) = \begin{cases} c_2 & \text{if } i \text{ is odd} \\ c_2 \cdot 2^{\frac{i}{2}} & \text{if } i \text{ is even.} \end{cases}$$

In each iteration of the outer loop, we compute the score for all Boolean expressions (line 9) as a floating value, and recursively define a scoring function $s$ to drive the search and assess the quality of an assignment as follows.

Given $\alpha$ and a Boolean variable $v$, its score $s(v, \alpha)$ is defined as in [5] as its assignment in $\alpha$, i.e.,

$$s(v, \alpha) = \alpha(v).$$

Analogously, given $\alpha$ and the negation of a Boolean variable $v$, the score of $\neg v$ is defined as

$$s(\neg v, \alpha) = \neg\alpha(v).$$

Given two Boolean expressions $a$ and $b$, the score of an and-expression over $a$ and $b$ is adopted from [5] as

$$s(a \wedge b, \alpha) = \tfrac{1}{2} \cdot (s(a, \alpha) + s(b, \alpha))$$

$$s(\neg(a \wedge b), \alpha) = max(s(\neg a, \alpha), s(\neg b, \alpha)).$$

The score of an equality over bit-vector expressions $a$ and $b$ of bit-width $n$ is defined as in [5] via the hamming distance $h$ of their assignments in $\alpha$, i.e. given a constant $0 \leq c_1 \leq 1$ (we choose $c_1 = 0.5$), we define

$$s(a = b, \alpha) = \begin{cases} 1.0 & \text{if } \alpha(a) = \alpha(b) \\ c_1 \cdot \left(1 - \dfrac{h(\alpha(a),\, \alpha(b))}{n}\right) & \text{otherwise} \end{cases}$$

$$s(a \neq b, \alpha) = \begin{cases} 1.0 & \text{if } \alpha(a) \neq \alpha(b) \\ 0.0 & \text{otherwise.} \end{cases}$$

The score definition of an inequality over bit-vector expressions $a$ and $b$ of bit-width $n$ significantly differs

from [5] due to implementation issues and is defined via a function $m$, which is a cheap heuristic to determine an upper bound on the number of bit flips required such that $\alpha(a)$ and $\alpha(b)$ match the given inequality relation. Note that $m$ is pessimistic in the sense that the actual number of required bit-flips might be smaller. The score of a bit-vector inequality is then defined as

$$s(a < b, \alpha) = \begin{cases} 1.0 & \text{if } \alpha(a) < \alpha(b) \\ c_1 \cdot \left(1 - \frac{m_<(\alpha(a), \alpha(b))}{n}\right) & \text{otherwise} \end{cases}$$

$$s(a \geq b, \alpha) = \begin{cases} 1.0 & \text{if } \alpha(a) \geq \alpha(b) \\ c_1 \cdot \left(1 - \frac{m_\geq(\alpha(a), \alpha(b))}{n}\right) & \text{otherwise.} \end{cases}$$

Lastly, given $\alpha$, a set of constraints $\{a_1, \cdots, a_m\}$ in $\pi$, and its corresponding set of weights $\{w_1, \cdots, w_m\}$, we define the overall score of formula $\pi$ as in [5] as $s(\pi, \alpha) = w_1 \cdot s(a_1, \alpha) + \cdots + w_m \cdot s(a_m, \alpha)$. Note that the score of a constraint $a_i$ is normalized and therefore bound to $0 \leq s(a_i, \alpha) \leq 1$. The overall score of formula $\pi$, however, may be greater than 1.

As in [5], we perform score computation bottom-up, i.e., starting from the inputs. However, due to the fact that we do not require formula $\pi$ to be in NNF, it is not possible to employ what [5] refers to as "early pruning". Hence, in order to still minimize the overhead for score computation, in contrast to [5] we do not recompute the score for all nodes in a formula's DAG representation on update. Rather, we identify the cone of influence, i.e., those parts of the formula affected by changing a given input, and update its score accordingly (line 18).

In each iteration of the inner loop (lines 10-12), depending on a yet unsatisfied constraint `root` and assignment $\alpha$, a variable `var` and a value `val` is selected (line 14), and assignment $\alpha$ and all scores are updated accordingly (lines 15 and 18). Note that given constraints $\{a_1, \cdots, a_m\}$ in $\pi$, constraint `root` is selected as in [5] as the unsatisfied constraint $a_i$ that, given a constant $c_3$ (we choose $c_3 = 20$), maximizes

$$s(a_i, \alpha) + c_3 \cdot \sqrt{\frac{\log selected(a_i)}{nmoves}},$$

where $selected(a_i)$ is the number of times $a_i$ has already been selected, and $nmoves$ is the overall number of moves performed so far.

In the general SLS case, i.e., without enabling our additional propagation strategy, we adopt the notion of (extended) neighborhood for regular SLS moves from [5],

```
1 procedure select_sls_move(root)
2   V := select_vars(root)
3   choose with probability wp
4     var, val := random_walk(V)
5   else
6     var, val := find_best_move(V)
7     if (var, val) = none
8       var, val := randomize(V)
9   return var, val
```

Fig. 3. A regular SLS move in pseudo-code.

which includes single bit flips, increment, decrement, and bitwise negation. Move selection is then performed as in [5] corresponding to Figure 3 as follows. Given an unsatisfied constraint `root`, all bit-vector variables reachable while traversing from the root to the inputs are collected into a set of candidate variables `V` (line 2). Out of all possible combinations in `V` and its extended neighborhood, the variable/neighbor pair with the most improvement of the overall score $s(\pi, \alpha)$ is determined as the best move (line 6). If no best move is found, a random variable and value is chosen (line 8), and the weights of all constraints are updated as in [5] (Figure 1, line 17). That is, with a probability $sp$ (we choose $sp = 0.95$), the weights of all unsatisfied constraints are increased by 1. Otherwise, the weights of all satisfied constraints are decreased to a minimum of 1.

Note that in the general SLS case, all moves performed are regular SLS moves as described in Figure 3, which corresponds to a ratio $0 : \infty$ of propagation moves to regular SLS moves in Figure 2.

Further, as in [5], `select_sls_move` optionally supports so called *random walks*, i.e., if enabled, with a probability $wp$ (we choose $wp = 0.1$) a random move out of all variable/neighbor combinations is chosen.

## III. Propagation Moves

The notion of (extended) neighborhood in [5] combines a simple SAT-style SLS neighborhood relation, given by flipping single bits of a variable assignment, with three additional bit-vector moves: increment, decrement, and bitwise negation. With a neighborhood size of $n + 3$ for a single bit-vector variable of bit-width $n$, it is easy to see that exploring its neighborhood is dominated by bitwise flips. As a consequence, neighborhood exploration in [5] mainly simulates bit-level local search without fully exploiting possible benefits of the word-level. Further, for variables with increasing bit-width, SLS moves as described in Section II become increasingly expensive.

Even though the approach in [5] showed promising results on the *Sage2* benchmark family, it still struggles in general in comparison to state-of-the-art bit-blasting approaches. Especially when dealing with a certain type of problem, the shortcomings of the chosen neighborhood relations become evident. Consider the following example in SMT-LIB notation:

```
(set-logic QF_BV)
(declare-fun v () (_ BitVec 65))
(assert (= (_ bv18446744073709551617 65)
           (bvmul (_ bv274177 65) v)))
(check-sat)
(exit)
```

Assuming that we disable possible simplification techniques, it is not possible to determine the (single) solution $v = 67280421310721$ within a time limit of 1200s on a 3.4GHz Intel Core i7-2600 machine (355837 moves, 21 restarts) with the SLS procedure as described in section II. With our propagation-based strategy, however, the example above is solved instantly within one single propagation move. In this section, we will introduce this strategy and its application in detail as follows.

### A. Propagation-Based Move Selection

When enabling our propagation-based strategy, within the core procedure as described in Figure 1, we support three different scenarios, depending on the type of move to be selected (line 14):

- $(i)$ all moves performed are propagation moves (i.e., propagation and regular SLS moves are performed with a ratio $\infty : 0$)
- $(ii)$ propagation and regular SLS moves are performed with a ratio $n : m$

In case of propagation moves, move selection is performed corresponding to Figure 4 as follows. Given an unsatisfied constraint `root`, a simplified formula $\pi$, and assignment $\alpha$, we force `root` to be true (line 3) and iteratively propagate its new assignment along one path towards the inputs while assuming all other paths to be fixed with respect to assignment $\alpha$.

In each iteration, path selection (lines 15-26) is implemented as choosing one of the current node's inputs, which in general happens randomly except for two cases. If the current node is an IF-THEN-ELSE (ITE) node (line 15), with probability $ip$ (we choose $ip = 0.1$), the first (and else, the second) out of two options is chosen:

(1) flip the condition, or
(2) assume the assignment of the condition to be fixed and follow the enabled branch.

```
1  procedure select_prop_move(root)
2    cur := root
3    val := 1
4
5    loop
6      if is_bv_var(cur)
7        return cur, val
8
9      // conflict
10     if not has_non_const_input(cur)
11       // recover w. regular SLS move
12       return select_sls_move(root)
13
14     // path selection
15     if is_ite(cur)
16       if flip_cond()
17         cur := get_cond_node(cur)
18         val := flip_bv(α(cur))
19       else
20         cur := get_enabled_branch(cur)
21       continue
22     if is_boolean_and(cur) and
23        has_exactly_one_ctrl_input(cur)
24       inp := get_ctrl_input(cur)
25     else
26       inp := select_random_input(cur)
27     // path propagation
28     oth := get_other_inputs(cur,inp)
29     val := compute_value(cur,val,oth)
30
31     // conflict
32     if val = none
33       // recover w. regular sls move
34       return select_sls_move(root)
35
36     cur := inp
```

Fig. 4. A propagation move in pseudo-code.

In (1), we reset `cur` to the condition and `val` to the flipped value of its assignment in $\alpha$ (lines 17-18). In (2), we reset `cur` to the enabled branch (line 20) before continuing with the next iteration.

If the current node is a Boolean AND node (line 22), on the other hand, path selection is based on the notion of (a posteriori) observability don't cares as defined in the context of ATPG [9]. Given an unsatisfied constraint `root`, while propagating its flipped assignment $\alpha(\texttt{root}) = 1$ along one path towards the inputs, as a

consequence, the assignments of all AND-gates along this path are flipped as well. Given a concrete assignment to the inputs of an AND-gate, however, we can determine lines that do not influence its output under the current assignment. Consequently, if the output of an AND node is currently assigned to 0 (i.e., to be flipped to 1), we follow its controlling input, i.e., the input with controlling value 0, if only one of its inputs is controlling. Else, we choose randomly. Note that the procedure in Figure 4 aims at finding an assignment for a bit-vector variable. As a consequence, during path selection, we do not choose bit-vector constants as input `inp`. If a node `cur` has only bit-vector constants as inputs, assignment `val` conflicts with the inputs of `cur` and we recover with a regular SLS move (line 12).

After selecting a path, we implement down propagation of assignments by computing a new assignment for input `inp` as the inverse of the current node given its assignment and the assignment to its other inputs, which are assumed to be fixed (line 29). Note that in general, if the other inputs are not bit-vector constants and such an inverse value does not exist, procedure `compute_value` concludes with an assignment for the chosen input that matches the assignment of the current node, disregarding its other inputs. However, if its other inputs are bit-vector constants, it concludes with `none` and we again recover with a regular SLS move (line 34). Finally, if we were able to successfully propagate all assignments along a path from `root` to a bit-vector variable, we conclude with this variable and its new assignment (line 6).

Note that procedure `select_prop_move` optionally supports to force a random walk rather than performing a regular SLS move (where random walks, if enabled, occur only with probability $wp$, otherwise) when recovering from a conflicting assignment (lines 12 and 34).

### *B. Propagating Assignments via Inverse Computation*

Down propagation of assignments is implemented by procedure `compute_value` via computing the inverse of a given node (representing a bit-vector operation) given its assignment and the assignment of all but one of its inputs. Without loss of generality, we restrict the set of bit-vector operations to {=, `bvnot`, `bvult`, `bvshl`, `bvshr`, `bvadd`, `bvand`, `bvmul`, `bvudiv`, `bvurem`, `concat`, `extract`} as defined in the SMT-LIB standard v2[1], where all but `bvnot` and `extract` are binary operations. Note that for some operations in this set no well-defined inverse operation exists. In that case, procedure `compute_value` in general produces non-unique values via randomization (of bits or bit-vectors). Further note that given the assignment of the bit-vector operation and one of its inputs, if the input is not a bit-vector constant and no inverse value could be found, `compute_value` disregards the assignment of the given input and chooses an inverse value that matches the assignment of the given operation. However, if the input is a bit-vector constant, its assignment conflicts with the assignment of the given operation, and `compute_value` concludes with `none`.

In the following, we assume that given a bit-vector $v$, its least significant bit (LSB) is positioned at index 0 (LSB $= v[0]$). We further denote a bit-vector slice expression (`extract` in SMT-LIB notation) from index $i$ to $j$ (incl.) with $v[i:j]$ and assume that $i$ and $j$ are numerical constants. Procedure `compute_value` determines the assignment $x$ of an input of a given operation, given its assignment $c$ and the assignment $a$ of its other input (if any), as follows:

**=** Given a bit-vector expression $c := a = x$ or $c := x = a$, its inverse w.r.t. $x$ is defined as $x := a$ if $c = 1$, and a random value other than $a$, otherwise.

**bvnot** Given a bit-vector expression $c :=\sim x$, its inverse w.r.t. $x$ is defined as $x :=\sim c$.

**bvult** Given a bit-vector expression $c := a < x$, we determine its inverse w.r.t. $x$ as follows:

1) If $c = 1$ and $a \neq 2^n - 1$, we choose a random value for $x$ such that $a < x$.
2) If $c = 0$, we choose a random value for $x$ such that $a \geq x$.
3) If $a$ is not a bit-vector constant, and neither 1) nor 2) apply, we disregard $a$ and choose a random value for $x$ such that $x > 0$.

If $a$ is a bit-vector constant and neither 1) nor 2) apply, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

Given a bit-vector expression $c := x < a$, we determine its inverse w.r.t. $x$ as follows:

1) If $c = 1$ and $a \neq 0$, we choose a random value for $x$ such that $x < a$.
2) If $c = 0$, we choose a random value for $x$ such that $x \geq a$.
3) If $a$ is not a bit-vector constant, and neither 1) nor 2) apply, we disregard $a$ and choose a random value for $x$ such that $x < 2^n - 1$.

If $a$ is a bit-vector constant and neither 1) nor 2) apply, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

**bvshl** Given a bit-vector expression $c := a << x$, its inverse w.r.t. $x$ is defined as the number $n_0$ of least significant bits set to 0 in $c$. If $a$ is a bit-vector constant and $n_0$ is equal to the bit-width of $a$, or if $a << n_0$ and $c$ do not match, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

Given a bit-vector expression $c := x << a$, its inverse w.r.t. $x$ is defined as $x := c >> a$. Note that the bits shifted in may be set arbitrarily. If $a$ is a bit-vector constant and the $a$ least significant bits in $c$ are not set to 0, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

**bvshr** is defined analogous to `bvshl`.

**bvadd** Given a bit-vector expression $c := a + x$ or $c := x + a$, its inverse w.r.t. $x$ is defined as $x := c - a$.

**bvand** Given a bit-vector expression $c := a \,\&\, x$ or $c := x \,\&\, a$, its inverse w.r.t. $x$ is defined depending on the bits set in both $a$ and $c$, i.e., given position $i$,

- if $c[i] = 1$, then $x[i] := 1$
- if $c[i] = 0$ and $a[i] = 1$, then $x[i] := 0$
- if $c[i] = 0$ and $a[i] = 0$, then $x[i] := dc$

Note that don't care values ($dc$) may be set arbitrarily. If $a$ is a bit-vector constant, and $a[i] = 0$ for any position $i$ where $c[i] = 1$, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

**bvmul** Given a bit-vector expression $c := a \cdot x$ or $c := x \cdot a$ of bit-width $n$, we determine its inverse w.r.t. $x$ as follows.

1) If $a$ is a divisor of $c$, then $x := c/a$.
2) If $gcd(a, 2^n) = 1$, we determine the multiplicative inverse $a^{-1}$ of value $a$ via the Extended Euclidean algorithm and define $x := a^{-1} \cdot c$.
3) If neither applies and $a$ is not a bit-vector constant, we try if any $v \in \{7, 5, 3, 2\}$ is a divisor of $c$. If so, we choose $x := c/v$, and $x := 1$, otherwise.

If $a$ is a bit-vector constant and neither 1) nor 2) apply, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

**bvudiv** Given a bit-vector expression $c := a/x$ of bit-width $n$, we determine its inverse w.r.t. $x$ as follows:

1) If $a = c = 0$, we choose a random value for $x$.
2) If $a \neq 0$, and $c$ is a divisor of $a$, then $x := a/c$.
3) If $a$ is not a bit-vector constant, and neither 1) nor 2) apply, we disregard $a$, and choose a random value $x$ such that $x \cdot c$ does not overflow.

If $a$ is a bit-vector constant and neither 1) nor 2) apply, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

Given a bit-vector expression $c := x/a$ of bit-width $n$, we determine its inverse w.r.t. $x$ as follows:

1) If $a = 0$ and $c = 2^n - 1$, we choose a random value for $x$. This is due to the fact, that given a value $v$ of bit-width $n$, Boolector handles division by zero as $v/0 = 2^n - 1$.
2) If $a \neq 0$, and $c \cdot a$ does not overflow, then $x := c \cdot a$.
3) If $a$ is not a bit-vector constant, and neither 1) nor 2) apply, we disregard $a$, choose a random value $v$ such that $v \cdot c$ does not overflow, and define $x := c \cdot v$.

If $a$ is a bit-vector constant and neither 1) nor 2) apply, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

**bvurem** Given a bit-vector expression $c := a \,\%\, x$ of bit-width $n$, we determine its inverse w.r.t. $x$ as follows:

1) If $a = c$ and $a \neq 2^n - 1$, we choose a random value for $x$ such that $x > a$.
2) If $a > c$ and $a - c > c$, since $c + m \cdot x = a$ we choose $m = 1$ and define $x := a - c$.
3) If $a$ is not a bit-vector constant, and either $a > c$ and $a - c \leq c$, or $a < c$ and $c < 2^n - 1$, we disregard $a$ and choose a random value for $x$ such that $x > c$.

If $a = c = 2^n - 1$, or if $a$ is a bit-vector constant and 2) does not apply, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

Given a bit-vector expression $c := x \,\%\, a$ of bit-width $n$, we determine its inverse w.r.t. $x$ as follows:

1) If $a > c$, with probability 0.5 we either choose
   a) $x := c$, or,
   b) since $c + m \cdot a = x$ we choose $m$ such that $c + m \cdot a$ does not overflow and define $x := c + m \cdot a$. If $c + a$ overflows, but a is not a bit-vector constant, we choose a).
2) If $a \leq c$ and $a$ is not a bit-vector constant, we choose $x := c$.

If $a$ is a bit-vector constant and 1) does not apply, current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

**concat** Given a bit-vector expression $c := a \circ x$ where $a$ is of bit-width $m$ and $c$ of bit-width $n$, its inverse w.r.t. $x$ is defined as the slice $x := c[0 : n - m - 1]$. If $a$ is a bit-vector constant and $c[n - m : n - 1] \neq a$, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

Given a bit-vector expression $c := x \circ a$ where $a$ is of bit-width $m$ and $c$ of bit-width $n$, its inverse w.r.t. $x$ is

defined as the slice $x := c[m : n-1]$. If $a$ is a bit-vector constant and $c[0 : m-1] \neq a$, the current value of $c$ is conflicting with $a$ and we can not find a value for $x$.

**extract** Given a bit-vector expression $c := x[l : u]$, where $x$ is of bit-width $n$ (and $c$ of bit-width $u - l + 1$), we determine its inverse w.r.t. $x$ given a position $i$ as follows. If $l \geq i \leq u$, then $x[i] = c[i - l]$. Else, we choose a random value for $x[i]$.

## IV. Experiments

We implemented the core SLS engine and its propagation-based extension in our SMT solver Boolector as described in Sections II and III, and provide an evaluation of the following configurations.

1) **BB**: The core Boolector engine, which uses a bit-blasting approach. This configuration is a version close to the version that won the QF_BV track of the SMT competition 2015.
2) **BSLS**: The SLS Boolector engine, optionally with random walks enabled (+RW).
3) **BPROP**: The SLS Boolector engine with our propagation-based strategy enabled. This configuration by default uses propagation moves only. It optionally supports the configuration of a ratio $n : m$ of propagation to regular SLS moves (+n:m), and conflict recovery via random walk rather than performing a regular SLS move (+FORCERW).

We further compare our Boolector configurations BSLS(+RW) against the original implementation of [5] in Z3 [4] and refer to version 4.4.0 of Z3 with its SLS engine enabled as configuration Z3SLS. Note that Z3SLS enables random walks by default.

We compiled a benchmark set from all benchmarks with status sat and unknown in the QF_BV category of the SMT-LIB[1] benchmark library, and excluded all benchmarks that configuration BB proved to be unsatisfiable within a time limit of 1200 seconds (16434 instances in total). We excluded 449 benchmarks from the *Sage2* benchmark family that were not SMT-LIB v2 compliant due to non-compliant operators.

All experiments were performed on a cluster with 30 nodes of 2.83GHz Intel Core 2 Quad machines with 8GB of memory using Ubuntu 14.04.2 LTS. The results in [5] indicate that even though there still exists a considerable gap between the performance of state-of-the-art bit-blasting and word-level local search as introduced in [5], the latter significantly outperforms bit-blasting on several instances. Based on these findings, we evaluated our BSLS and BPROP configurations with regard to an application within a sequential portfolio setting and set a time limit of 10 seconds for all solver instances of these and the Z3SLS configurations. The memory limit for each solver instance was set to 7GB. In case of a time or memory out, a penalty of the given time limit was added to the total CPU time.

[1] http://www.smtlib.org

| | BB | | BSLS | | BSLS+RW | | Z3SLS | |
|---|---|---|---|---|---|---|---|---|
| **Family** | **Slvd** | **[s]** | **Slvd** | **[s]** | **Slvd** | **[s]** | **Slvd** | **[s]** |
| asp (376) | **46** | **3543** | 0 | 3760 | 0 | 3760 | 0 | 3760 |
| bench (223) | 223 | 0.1 | **223** | **0.0** | 223 | 0.0 | 223 | 0.0 |
| bmc (22) | **21** | **57** | 11 | 118 | 12 | 115 | 7 | 150 |
| brubiere2 (10) | 3 | 72 | 10 | 4.3 | **10** | **3.1** | 1 | 90 |
| brubiere3 (6) | 4 | 46 | 6 | 4.0 | **6** | **0.2** | 2 | 40 |
| brubiere4 (10) | 0 | 100 | 9 | 10 | 9 | 10 | **10** | **0.0** |
| calypto (13) | 4 | 93 | **4** | **92** | 3 | 100 | 3 | 100 |
| check2 (1) | **1** | **0.0** | 1 | 0.0 | 1 | 0.0 | 0 | 10 |
| crafted (1) | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 |
| dwp (103) | 103 | 0.3 | **103** | **0.0** | 103 | 0.0 | 103 | 0.9 |
| fft (19) | **4** | **160** | 0 | 190 | 0 | 190 | 0 | 190 |
| float (126) | **23** | **1095** | 2 | 1253 | 0 | 1260 | 0 | 1260 |
| gulwani (6) | **5** | **17** | 1 | 50 | 1 | 51 | 1 | 50 |
| mcm (155) | **14** | **1452** | 8 | 1508 | 5 | 1517 | 8 | 1474 |
| pspace (21) | 0 | 210 | **21** | **16** | 21 | 16 | 0 | 210 |
| rubik (3) | **2** | **17** | 0 | 30 | 0 | 30 | 0 | 30 |
| RWS (20) | **13** | **98** | 0 | 200 | 0 | 200 | 0 | 200 |
| sage (6236) | **6236** | **2179** | 5315 | 10497 | 5275 | 10626 | 5969 | 5261 |
| Sage2 (6981) | 1037 | 63023 | 604 | 64307 | 620 | 64289 | **1597** | **56890** |
| spear (1675) | **1524** | **7435** | 1188 | 6348 | 1187 | 6336 | 456 | 13289 |
| stp (1) | 0 | 10 | 0 | 10 | 0 | 10 | 0 | 10 |
| stp_s (149) | **149** | **3.6** | 127 | 410 | 128 | 421 | 149 | 6.1 |
| tacas07 (3) | 2 | 20 | 2 | 10 | 2 | 10 | **2** | **10** |
| uclid (262) | 258 | 426 | 29 | 2533 | 23 | 2551 | **259** | **297** |
| VS3 (10) | 0 | 100 | 0 | 100 | 0 | 100 | 0 | 100 |
| wienand (4) | 0 | 40 | 0 | 40 | 0 | 40 | 0 | 40 |
| totals (16436) | **9673** | **80197** | 7665 | 91491 | 7630 | 91635 | 8791 | 83262 |

TABLE I
RESULTS BY BENCHMARK FAMILY FOR CONFIGURATIONS BB, BSLS, BSLS+RW, AND Z3SLS WITH A TIME LIMIT OF 10 SECONDS.

Table I summarizes the results of configurations BB, BSLS, BSLS+RW, and Z3SLS on our benchmark set, grouped by family, within a time limit of 10 seconds. Configuration BSLS+RW corresponds to the configuration of Z3SLS as both enable random walks, whereas configuration BSLS disables random walks by default. On family *pspace*, Z3SLS terminated with an error, for which a penalty of the given time limit was added to the total CPU time. Compared to the bit-blasting configuration BB, as in [5], the overall results of configuration BSLS+RW confirm both the effectiveness of the SLS approach on certain instances, and the overall gap in performance. The performance of configurations BSLS+RW and Z3SLS, however, considerably differs on some benchmark families. Z3SLS outperforms BSLS+RW by 694 and 977 instances on the *sage* and *Sage2* benchmarks, whereas on the *spear* family, with

| | BSLS | | BPROP+1:100 | | BPROP+1:10 | | BPROP+1:1 | | BPROP+10:1 | | BPROP+100:1 | | BPROP | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Family** | **Slvd** | **[s]** | **Slvd** | **[s]** | **Slvd** | **[s]** | **Slvd** | **[s]** | **Slvd** | **[s]** | **Slvd** | **[s]** | **Slvd** | **[s]** |
| asp (376) | 0 | 3760 | 0 | 3760 | 0 | 3760 | 0 | 3760 | 0 | 3760 | 0 | 3760 | 0 | 3760 |
| bench (223) | **223** | **0.0** | 223 | 0.1 | 223 | 0.1 | 223 | 0.0 | 223 | 0.0 | 223 | 0.0 | 223 | 0.0 |
| bmc (22) | 11 | 118 | 12 | 116 | **12** | **110** | 12 | 118 | 10 | 123 | 10 | 122 | 11 | 118 |
| brubiere2 (10) | 10 | 4.3 | 10 | 4.5 | 10 | 4.2 | **10** | **3.2** | 10 | 4.1 | 10 | 4.2 | 10 | 4.2 |
| brubiere3 (6) | 6 | 4.0 | 6 | 19 | **6** | **3.8** | 6 | 13 | 6 | 13 | 6 | 9.5 | 6 | 12 |
| brubiere4 (10) | 9 | 10 | 9 | 10 | 9 | 10 | **10** | **5.1** | 10 | 7.7 | 10 | 7.7 | 10 | 7.7 |
| calypto (13) | 4 | 92 | 4 | 96 | 3 | 100 | **4** | **90** | 4 | 90 | 3 | 100 | 3 | 100 |
| check2 (1) | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 |
| crafted (1) | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 |
| dwp (103) | 103 | 0.0 | 103 | 0.0 | 103 | 0.0 | 103 | 0.0 | 103 | 0.0 | 103 | 0.0 | 103 | 0.0 |
| fft (19) | 0 | 190 | 0 | 190 | 0 | 190 | **1** | **183** | 0 | 190 | 0 | 190 | 0 | 190 |
| float (126) | 2 | 1253 | 3 | 1246 | 0 | 1260 | 3 | 1248 | 7 | 1217 | 7 | 1209 | **8** | **1206** |
| gulwani (6) | **1** | **50** | 1 | 50 | 1 | 51 | 1 | 52 | 1 | 58 | 0 | 60 | 0 | 60 |
| mcm (155) | 8 | 1508 | 8 | 1508 | 7 | 1506 | **9** | **1500** | 6 | 1513 | 5 | 1519 | 5 | 1519 |
| pspace (21) | 21 | 16 | 21 | 16 | **21** | **16** | 0 | 210 | 0 | 210 | 21 | 94 | 0 | 210 |
| rubik (3) | 0 | 30 | 0 | 30 | 0 | 30 | 0 | 30 | 0 | 30 | 0 | 30 | 0 | 30 |
| RWS (20) | 0 | 200 | 0 | 200 | 0 | 200 | 0 | 200 | 0 | 200 | 0 | 200 | 0 | 200 |
| sage (6236) | **5315** | **10497** | 5278 | 10604 | 5256 | 10868 | 5217 | 11330 | 5097 | 11925 | 5088 | 11984 | 5133 | 11827 |
| Sage2 (6981) | 604 | 64307 | 626 | 64212 | **635** | **64205** | 571 | 64614 | 615 | 64456 | 589 | 64536 | 606 | 64420 |
| spear (1675) | 1188 | 6348 | 1188 | 6339 | 1189 | 6324 | 1316 | 5190 | 1482 | 3087 | 1483 | 3039 | **1485** | **3029** |
| stp (1) | 0 | 10 | 0 | 10 | 0 | 10 | 0 | 10 | 0 | 10 | 0 | 10 | 0 | 10 |
| stp_s (149) | 127 | 410 | 127 | 410 | 130 | 407 | **132** | **366** | 124 | 413 | 127 | 398 | 127 | 400 |
| tacas07 (3) | 2 | 10 | 2 | 10 | 2 | 10 | 2 | 10 | 2 | 10 | 2 | 10 | 2 | 10 |
| uclid (262) | **29** | **2533** | 28 | 2534 | 13 | 2553 | 1 | 2618 | 9 | 2557 | 17 | 2560 | 21 | 2555 |
| VS3 (10) | 0 | 100 | 0 | 100 | 0 | 100 | 0 | 100 | 0 | 100 | 0 | 100 | 0 | 100 |
| wienand (4) | 0 | 40 | 0 | 40 | 0 | 40 | 0 | 40 | 0 | 40 | 0 | 40 | 0 | 40 |
| totals (16436) | 7665 | 91491 | 7651 | 91505 | 7622 | 91759 | 7623 | 91690 | 7711 | 90015 | 7706 | 89983 | **7755** | **89808** |

TABLE II

RESULTS BY BENCHMARK FAMILY FOR CONFIGURATIONS BSLS, BPROP (PROPAGATION MOVES ONLY), AND BPROP+N:M (WITH RATIO 1:100, 1:10, 1:1, 10:1, AND 100:1 OF PROPAGATION MOVES TO REGULAR SLS MOVES) WITH A TIME LIMIT OF 10 SECONDS.

a difference of 731 instances, it is vice versa. Running BSLS+RW with different seeds for initializing the random number generator, however, has almost no influence on the number of solved instances. Out of 10 randomly seeded runs of configuration BSLS+RW on our benchmark set, we observed a maximum deviation of 0.04% in the number of solved instances. We therefore believe that the difference in performance between BSLS+RW and Z3SLS is mainly due to the following reasons. Similar to the influence of rewriting and simplification techniques on the performance of state-of-the-art bit-blasting approaches, rewriting and the choice of rewriting and simplification techniques considerably influence the performance of the actual SLS procedure. Given our benchmark set and a time limit of 10 seconds, Boolector's SLS engine in configuration BSLS+RW with rewriting and simplification disabled solves 2166 less instances. The choice of rewriting and simplification techniques employed by Boolector and Z3 differs significantly, which might be one reason for the difference in performance on certain benchmark families. Further, even though BSLS+RW corresponds to Z3SLS as far as implementation issues allowed, both still differ in several implementation aspects, in particular, the scoring function for the $<$ operator, which might influence the performance of the SLS approach considerably.

The overall results in Table I imply that enabling random walks in Boolector's SLS engine does not improve its performance. We therefore use configuration BSLS as base for our further experiments.

Further note that even though the last two authors of this paper were also involved in [5], the reimplementation of the approach in [5] within Boolector by the first author should be considered as independent. The results of Table I, as a consequence, should be considered as an independent confirmation of the results in [5].

Table II compares the results of our base SLS configuration BSLS to configuration BPROP (propagation moves only), and BPROP+100:1, BPROP+10:1, BPROP+1:1, BPROP+1:10, and BPROP+1:100 (with ratios 100:1, 10:1, 1:1, 1:10, and 1:100 of propagation moves to regular SLS moves) on our benchmark set, grouped by family, within a time limit of 10 seconds. Overall, the results suggest that configurations with a higher ratio of propagation to regular SLS moves perform better in terms of solved instances and run time. With an additional 90 solved instances, configuration BPROP shows the most improvement in comparison to configuration

| | BSLS | | BPROP | | BPROP+FRW | |
|---|---|---|---|---|---|---|
| **Family** | **Slvd** | **[s]** | **Slvd** | **[s]** | **Slvd** | **[s]** |
| asp (376) | 0 | 376 | 0 | 376 | 0 | 376 |
| bench (223) | **223** | **0.0** | 223 | 0.0 | 223 | 0.2 |
| bmc (22) | 10 | 13 | **10** | **13** | 9 | 14 |
| brubiere2 (10) | **9** | **2.7** | 9 | 3.3 | 3 | 7.0 |
| brubiere3 (6) | 4 | 2.0 | 4 | 3.7 | **6** | **2.1** |
| brubiere4 (10) | **9** | **1.0** | 8 | 2.0 | 8 | 2.0 |
| calypto (13) | 3 | 10 | 3 | 10 | **5** | **9.0** |
| check2 (1) | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 |
| crafted (1) | 1 | 0.0 | 1 | 0.0 | 1 | 0.0 |
| dwp (103) | 103 | 0.0 | 103 | 0.0 | 103 | 0.0 |
| fft (19) | 0 | 19 | 0 | 19 | 0 | 19 |
| float (126) | 0 | 126 | **3** | **124** | 2 | 125 |
| gulwani (6) | **1** | **5.3** | 0 | 6.0 | 0 | 6.0 |
| mcm (155) | 1 | 155 | **2** | **154** | 1 | 155 |
| pspace (21) | **20** | **16** | 0 | 21 | 0 | 21 |
| rubik (3) | 0 | 3.0 | 0 | 3.0 | 0 | 3.0 |
| RWS (20) | 0 | 20 | 0 | 20 | 0 | 20 |
| sage (6236) | **5038** | **1385** | 4971 | 1418 | 4769 | 1579 |
| Sage2 (6981) | **509** | **6605** | 473 | 6657 | 452 | 6614 |
| spear (1675) | 819 | 1249 | 1299 | 604 | **1662** | **187** |
| stp (1) | 0 | 1.0 | 0 | 1.0 | 0 | 1.0 |
| stp_s (149) | **74** | **94** | 73 | 94 | 23 | 127 |
| tacas07 (3) | 2 | 1.2 | 2 | 1.2 | 2 | 1.2 |
| uclid (262) | 0 | 262 | 0 | 262 | 0 | 262 |
| VS3 (10) | 0 | 10 | 0 | 10 | 0 | 10 |
| wienand (4) | 0 | 4.0 | 0 | 4.0 | 0 | 4.0 |
| totals (16436) | 6827 | 10361 | 7185 | 9807 | **7270** | **9544** |

TABLE III
RESULTS BY BENCHMARK FAMILY FOR CONFIGURATIONS BSLS, BPROP, AND BPROP+FRW WITH A TIME LIMIT OF 1 SECOND.

| | BB | | BB+BPROP+FRW | |
|---|---|---|---|---|
| **Family** | **Slvd** | **[s]** | **Slvd** | **[s]** |
| asp (376) | **285** | **147790** | 285 | 148075 |
| bench (223) | **223** | **0.1** | 223 | 0.2 |
| bmc (22) | **22** | **59** | 22 | 72 |
| brubiere2 (10) | **10** | **843** | 10 | 848 |
| brubiere3 (6) | 6 | 49 | **6** | **2.1** |
| brubiere4 (10) | 1 | 10981 | **8** | **2400** |
| calypto (13) | 7 | 9308 | **7** | **8358** |
| check2 (1) | 1 | 0.0 | 1 | 0.0 |
| crafted (1) | 1 | 0.0 | 1 | 0.0 |
| dwp (103) | 103 | 0.3 | **103** | **0.0** |
| fft (19) | **5** | **17014** | 5 | 17020 |
| float (126) | **94** | **53264** | 94 | 53356 |
| gulwani (6) | **6** | **34** | 6 | 40 |
| mcm (155) | **56** | **135932** | 56 | 135983 |
| pspace (21) | **21** | **688** | 21 | 709 |
| rubik (3) | **3** | **94** | 3 | 98 |
| RWS (20) | **16** | **5032** | 16 | 5048 |
| sage (6236) | **6236** | **2179** | 6236 | 3527 |
| Sage2 (6981) | 5621 | 2213660 | **5648** | **2148247** |
| spear (1675) | 1671 | 12747 | **1675** | **327** |
| stp (1) | 1 | 16 | 1 | 16 |
| stp_s (149) | **149** | **3.6** | 149 | 131 |
| tacas07 (3) | 3 | 29 | **3** | **20** |
| uclid (262) | **262** | **434** | 262 | 696 |
| VS3 (10) | **3** | **8844** | 3 | 8847 |
| wienand (4) | 0 | 4800 | 0 | 4800 |
| totals (16436) | 14806 | 2623801 | **14844** | **2538620** |

TABLE IV
RESULTS BY BENCHMARK FAMILY FOR CONFIGURATIONS BB AND BB+BPROP+FRW WITH A TIME LIMIT OF 1200 SECONDS.

BSLS. Out of the 7755 instances solved by BPROP, more than 56% (4366 instances) were solved with propagation moves only, i.e., no recovery by means of a regular SLS move was necessary; and for roughly 85% (6568 instances), less than 8 recovery moves were required. When recovering with a random walk rather than a regular SLS move, i.e., in configuration BPROP+FRW, out of 7540 solved instances roughly 72% (5427 instances) required less than 8 recovery moves. This suggests that regular SLS recovery moves, even though expensive, yield a better performance. Given a time limit of 1 second, though, as depicted in Table III, configuration BPROP+FRW outperforms BPROP by 85 instances. In particular on benchmark family *spear*, within 1 second and compared to configurations BB, BSLS, and BPROP, configuration BPROP+FRW solves 1586, 843, and 363 more instances. These results suggest a combination of configurations BB and BPROP+FRW into configuration BB+BPROP+FRW in a sequential portfolio manner [10], where configuration BPROP+FRW is run prior to bit-blasting for a given time limit of 1 second. In the following, configuration BB+BPROP+FRW is a virtual configuration as it has not been realized within Boolector yet. All results attributed to BB+BPROP+FRW have been determined based on the results of configuration BB and BPROP+FRW on our benchmark set with a time limit of 1200 and 1 seconds, respectively, as follows. On instances where configuration BPROP+FRW timed out within 1 second, a penalty of 1 second has been added to the result of configuration BB, else the result of configuration BPROP+FRW was chosen.

Table IV summarizes the results of configurations BB and BB+BPROP+FRW on our benchmark set, grouped by family, within a time limit of 1200 seconds. Even though invoking the BPROP+FRW engine prior to bit-blasting produces an overhead of 1 second for instances that BPROP+FRW can not solve within the given time limit, the combination of both engines considerably improves the performance of the bit-blasting approach both in terms of solved instances and total run time. Overall, configuration BB+BPROP+FRW is able to solve 38 more instance than configuration BB, while requiring only 96% of its total runtime. In particular benchmark families *brubiere4*, *Sage2*, and *spear* show the most improvement with speed-ups of up to a factor of 39. Even when considerung the 1 second overhead of BB+BPROP+FRW on each benchmark of the full QF_BV benchmark set of the SMT-LIB, which is a total of 50429 instances

including all unsat benchmarks, the performance gain of 85181 seconds on the benchmarks of our set of 16434 instances still yields a performance improvement in terms of the overall run time.

The results in Table IV are further illustrated in Figure 5 by means of a scatter plot. It shows that configuration BB+BPROP+FRW solves benchmarks up to orders of magnitude faster than the pure bit-blasting approach BB. In fact, 117, 582, and 1320 instances are solved more than 1000, 100, and 10 times faster when combining BPROP+FRW with BB. The overhead generated by unsuccessful BPROP+FRW invocations, i.e., for instances that cannot be solved by BPROP+FRW within the 1 second time limit, does not outweigh the performance gain by the successful ones.

Finally, we tried to further increase the time limit for BPROP+FRW of configuration BB+BPROP+FRW to 2 and 3 seconds. Compared to the 1 second time limit, with a time limit of 2 seconds, BB+BPROP+FRW solves an additional 8 instances while requiring 3500 seconds less run time. The best results, however, are achieved with a time limit of 3 seconds, where BB+BPROP+FRW solves an additional 14 instances requiring 3700 seconds less run time compared to the 1 second time limit.

## V. CONCLUSION

In this paper, we reimplemented the word-level local search procedure presented in [5] within our SMT solver Boolector and independently confirmed its effectiveness. We further introduced an extension based on propagating assignments from the outputs to the inputs and evaluated our approach in particular with respect to a combination with Boolector's bit-blasting engine, which considerably improves its performance on satisfiable instances.

We chose all parametric values either as in [5], or such that they provide a good overall performance if newly introduced. Further optimizing those values with respect to performance is left to future work.

Our technique currently still relies on regular SLS tactics in certain conflict scenarios. However, this might not be necessary if path propagation via inverse computation guarantees that search space is not inadvertently pruned on conflict. This might currently be the case due to short cuts introduced for some bit-vector operations when e.g. choosing the simplest (and not some randomized) valid solution. Extending inverse computation to eliminate this kind of short cuts, and the implementation of the sequential portfolio style combination of our techniques and Boolector's state-of-the-art bit-blasting engine is left to future work.

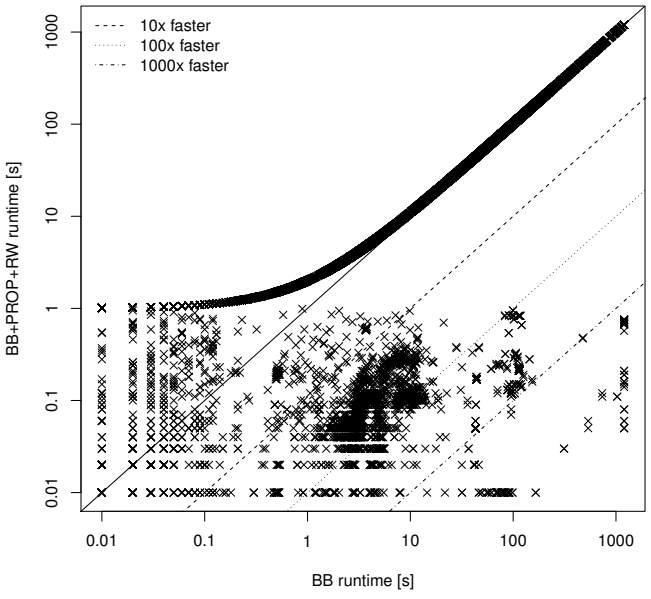


Fig. 5. BB vs. BB+BPROP+FRW on our benchmark set with a time limit of 1200 seconds and a 1 second time limit for BPROP+FRW.

Binaries of Boolector and all log files of our experimental evaluation can be found at http://fmv.jku.at/difts15-prop.

## REFERENCES


[1] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at www.SMT-LIB.org.

[2] R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In *Proc. CAV*, volume 4590 of *LNCS*, pages 547–560. Springer, 2007.

[3] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In *Proc. TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

[4] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*. Springer, 2008.

[5] A. Fröhlich, A. Biere, C. M. Wintersteiger, and Y. Hamadi. Stochastic local search for satisfiability modulo theories. In *Proc. AAAI*, pages 1136–1143. AAAI Press, 2015.

[6] A. Griggio, Q. Phan, R. Sebastiani, and S. Tomasi. Stochastic local search for SMT: combining theory solvers with walksat. In *Proc. FROCOS*, LNCS, pages 163–178. Springer, 2011.

[7] L. Hadarean, K. Bansal, D. Jovanovic, C. Barrett, and C. Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In *Proc. CAV*, volume 8559 of *LNCS*, pages 680–695. Springer, 2014.

[8] F. Hutter, H. H. Hoos, and T. Stützle. Efficient stochastic local search for MPE solving. In *Proc. IJCAI*, pages 169–174. Professional Book Center, 2005.

[9] Z. Navabi. *Digital Sytem Test and Testable Design*. Springer, 2011.

[10] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.