A Bit-Vector Solver with Word-Level Propagation

Wenxi Wang, Harald Søndergaard, and Peter J. Stuckey

Department of Computing and Information Systems,
The University of Melbourne, Victoria 3010, Australia

Abstract. Reasoning with bit-vectors arises in a variety of applications in verification and cryptography. Michel and Van Hentenryck have proposed an interesting approach to bit-vector constraint propagation on the word level. Each of the operations except comparison are constant time, assuming the bit-vector fits in a machine word. In contrast, bit-vector SMT solvers usually solve bit-vector problems by bit-blasting, that is, mapping the resulting operations to conjunctive normal form clauses, and using SAT technology to solve them. This also means generating intermediate variables which can be an advantage, as these can be searched on and learnt about. Since each approach has advantages it is important to see whether we can benefit from these advantages by using a word-level propagation approach with learning. In this paper we describe an approach to bit-vector solving using word-level propagation with learning. We provide alternative word-level propagators to Michel and Van Hentenryck, and give the first empirical evaluation of their approach that we are aware of. We show that, with careful engineering, a word-level propagation based approach can compete with (or complement) bit-blasting.

1 Introduction

Since most time-critical and safety-critical software is built on fixed-width integers, it is vital to reason about fixed-width integers correctly and accurately in a software verification context. We consider the problem of how to support this reasoning with modern constraint solving techniques.

SAT Modulo Theory (SMT) solvers are the most common tools in this area, and almost all the modern SMT solvers ultimately rely on bit-blasting [4, 5, 9, 13, 17] to solve bit-vector constraints, that is, translating constraints to propositional logic form. But bit-blasting tends to cause two problems. First, it may result in very large propositional formulas that even the most powerful current SAT solvers struggle to handle. Second, it disperses important word level information during the encoding—much is obscured in translation. Here we investigate alternatives to bit-blasting, replacing it with word-level propagation entirely to produce a pure word-level bit-vector SMT solver.

Using word-level propagation was suggested by Michel and Van Hentenryck [18] who viewed the problem as a Constraint Satisfaction Problem (CSP).

Each variable is associated with a "bit-vector domain" which is progressively tightened using word-level constraint propagation rules (we will make these clear shortly). The idea is appealing, as the propagation rules can be made to run in constant time (as long as the bit width of the bit-vector is less than or equal to the size of machine registers). An additional rule to check if a tightened domain remains valid also runs in constant time. However, we are not aware of any experimental evaluation of the method. Moreover, there is no "learning" mechanism in Michel and Van Hentenryck's proposal. We show that real improvement relies on the communication of explanations for the propagated bits. We also propose alternative ("decomposed") word-level propagators for some operations, based on insights in Warren's compendium [22] and we investigate the relative strengths and weaknesses of decomposed and composed propagators.

Additionally we use our solver to investigate different algorithmic possibilities. In a learning solver we can generate explanations in a "forward" manner, as propagation progresses, as is done in a SAT solver, or we can generate them in a "backward" manner during conflict analysis, as in an SMT solver. Forward explanation is simpler to implement, while backward explanation may require less explanation work overall. In our experiments we compare the two approaches.

Another potential benefit of word-level propagation is deeper conflict analysis. Normally, using bit-blasting, conflict analysis starts as soon as the first conflict is found. In the word-level solver, we could do the same, to find the first conflict clause and backtrack to the level indicated by this conflict (we call this "standard backjumping"). With word-level propagation, since we can discover several conflicts at once, we may find several learnt clauses at once, corresponding to several backtrack levels. We choose the smallest level from them in order to backtrack to the highest level of the search tree and add all the learnt clauses along the way to prevent all the conflicts from happening again (we call this "multi-conflict backjumping"). We also offer a comparison of these two approaches.

To construct the solver we have extended MiniSAT [6] so that it can keep track of opportunities for word-level propagation and intersperse this kind of propagation with unit propagation. Our word-level propagators contribute to MiniSAT's powerful search and learning mechanism by providing clauses as explanations for word-level propagated bits. In this way, the word level propagators become lazy clause generators [20] for a SAT solver extended with constraint programming technology [21]. In summary the main contributions of this paper are:

- a word-level propagating (but bit-level explaining) constraint solver;
- algorithms for generation of explanations for word-level propagators;
- an investigation of the algorithmic design space in building word-level propagation with explanation;
- the first (as far as we know) empirical evaluation of the proposal by Michel and Van Hentenryck [18], and comparison with the standard bit-blasting approach to these problems;
- results suggesting that, with careful engineering, a word-level propagation approach can be competitive with, or a useful supplement to, bit blasting.

The remainder of the paper is arranged as follows. Section 2 introduces bit-vector constraints and notation. Section 3 outlines the architecture of MiniSAT extended with word-level propagation. Section 4 introduces the propagators used in our solver. Section 5 explores several options for the design of the word-level solver and Section 6 evaluates these options and also compares to pure bit-blasting through experiments using standard benchmarks. Section 7 outlines related work and Section 8 concludes. The reader is assumed to have a basic understanding of modern SAT-solving technology.

2 Bit-Vector Constraints

In the following we shall need to distinguish word-level logical operations from Boolean operations carefully. As bit-wise operations we use $\sim$, $\&$, $|$, and $\oplus$ for bit complement, conjunction, disjunction, and exclusive or, respectively. As Boolean connectives, we use $\neg$, $\wedge$ and $\vee$ for negation, conjunction, and disjunction, respectively.

A *bit-vector* $x_{[w]}$ is a sequence of w binary digits (bits) and x_i denotes the ith bit in this sequence. The elements of the sequence are indexed from right to left, starting with index $0 : x = x_{w-1}...x_1x_0$. Here we take Boolean variables as bit-vectors of length 1. A "trit-vector" (for bit-width w) is a sequence of w elements taken from $\{0, 1, *\}$. Here the $*$ represents an undetermined bit, so a trit-vector x corresponds to the cube $(\bigwedge_{i \in I_0} \neg x_i) \wedge (\bigwedge_{i \in I_1} x_i)$, where I_0 is the set of index positions that hold a 0, and I_1 is the set of index positions that hold a 1.

In an implementation, the trit-vector can be represented by a pair of bit-vectors: $\langle \mathsf{lo}(x), \mathsf{hi}(x) \rangle$, where $\mathsf{lo}(x)$ and $\mathsf{hi}(x)$ are bit-vectors representing the lower and upper bound of x respectively, with

$$\mathsf{lo}(x)_i = \begin{cases} 0 & \text{if } x_i = * \\ x_i & \text{otherwise} \end{cases} \qquad \mathsf{hi}(x)_i = \begin{cases} 1 & \text{if } x_i = * \\ x_i & \text{otherwise} \end{cases}$$

For example, trit-vector z= 011*0*11 is written $\langle 01100011, 01110111 \rangle$ in this "lo-hi" form. The advantage of this form is that, as long as the bit width of a trit-vector x is less than or equal to the size of machine registers, $\mathsf{lo}(x)$ and $\mathsf{hi}(x)$ can be treated as unsigned integers, that is, z is $\langle 99, 119 \rangle$. Supported by an implementation language (such as C) that can utilise word-level operations, we can rephrase bit-propagation on a trit-vector as word-level operations on its bounds. For an example, consider y=*1110*** corresponding to $\langle 01110000, 11110111 \rangle$, and the constraint $y = z$. We can utilize the word-level rule: $\mathsf{lo}(y) = \mathsf{lo}(z) = \mathsf{lo}(y) \mid \mathsf{lo}(z)$, $\mathsf{hi}(y) = \mathsf{hi}(z) = \mathsf{hi}(y) \;\&\; \mathsf{hi}(z)$ to obtain the new lo-hi form of y (and z): $\langle 01110011, 01110111 \rangle$ representing 01110*11. Instead of propagating the bits one by one, we effectively fix the bits y_7, y_1, y_0 and z_4 simultaneously with the word-level operations on the bounds.

The lo-hi form allows for invalid representations of trit-vectors. That happens when, for some x, a bit in $\mathsf{lo}(x)$ is 1 while the corresponding bit in $\mathsf{hi}(x)$ is 0. As

will be seen, propagation can produce such invalid forms, but this happens when, and only when, an inconsistency is present in the current set of constraints. The validity checking rule is simple:

$$\mathsf{valid}(x) = {\sim}\,\mathsf{lo}(x) \mid \mathsf{hi}(x) \tag{1}$$

The result for a valid bit-vector lo-hi form should be a bit-vector of all 1 bits with the same bit width of the bit-vector variable; otherwise it is invalid, and the 0 bits in the result are the bits that cause the invalidity.

The following predicates on trit-vectors will prove useful:

$$\mathsf{fixed}(x) \equiv \mathsf{lo}(x) = \mathsf{hi}(x) \qquad \mathsf{pos}(x) = \{\mathsf{lit}(x_i) \mid \mathsf{lo}(x_i) = 1\}$$
$$\mathsf{msb}(x_{[w]}) = x_{w-1} \qquad \mathsf{neg}(x) = \{\mathsf{lit}(x_i) \mid \mathsf{hi}(x_i) = 0\}$$
$$\mathsf{lit}(b) = \begin{cases} \ulcorner b \urcorner & \text{if } \mathsf{lo}(b) = 1 \\ \ulcorner \neg b \urcorner & \text{if } \mathsf{hi}(b) = 0 \end{cases} \qquad \mathsf{lits}(x) = \mathsf{pos}(x) \cup \mathsf{neg}(x)$$

We use $\mathsf{fixed}(x)$ to return a Boolean value indicating whether every bit in bit-vector x is fixed. We use $\mathsf{msb}(x_w)$ to denote the most significant bit of bit-vector x. We use $\mathsf{lit}(b)$ to return the *literal* corresponding to the Boolean bit b under the condition that b is fixed (hence the use of Quine corners). We use $\mathsf{pos}(x)$ ($\mathsf{neg}(x)$) to return the set of literals fixed to 1 (resp. 0) in bit-vector x, and $\mathsf{lits}(x)$ to return the set of fixed literals in x. In the later algorithms, we take the set of literals to mean the conjunction of the literals.

Our solver handles all operations in the QF_BV category of SMT-LIB2 except for multiplication, division, modulus and remainder. The operations have the usual semantics [15]. We summarize the most important constraints:

Logical Constraints. Logical constraints include bitwise equality $x = y$, bitwise negation $x = {\sim}\,y$, bitwise conjunction $z = x \,\&\, y$, bitwise disjunction $z = x|y$, bitwise exclusive-or $z = x \oplus y$, bitwise nand, bitwise nor, reified equality $b \Leftrightarrow x = y$, and if-then-else operation $ite(b, x, y) = z$ where b is Boolean. The semantics of $ite(b, x, y) = z$ is $(b \wedge (z = x)) \mid (\neg b \wedge (z = y))$.

Arithmetic Constraints. Arithmetic constraints include (fixed-width) addition $x + y = z$, two's complement unary minus $y = -x$ which is equivalent to $y = {\sim}\,x + 1$, subtraction $z = x - y$ which is equivalent to $z = x + ({\sim}\,y + 1)$, unsigned inequality $b \Leftrightarrow x \leq_u y$, $b \Leftrightarrow x <_u y$, $b \Leftrightarrow x \geq_u y$, $b \Leftrightarrow x >_u y$, and the corresponding signed inequality constraints. Signed inequality constraints can be translated into unsigned inequality constraints. For instance, $b \Leftrightarrow x \leq_s y$ is equivalent to $b \Leftrightarrow (x \leq_u y) \oplus x_{w-1} \oplus y_{w-1}$.

Structural Constraints. Structural constraints include left shift($\ll$), unsigned and signed right shift ($\gg_u$, $\gg_s$), left and right rotate ($rotl$, $rotr$), concatenation ($::$), extraction ($extract(x, n, m) = y$) where y is the extraction of bits n down to m from x, signed and unsigned extension (ext_u, ext_s), and repeat ($repeat(x, n) = y$) where y is the concatenation of n copies of x.

Algorithm 1 General algorithm for MiniSAT and word-level solver

```
add the input into the system          ▷ initialization; CNF or word-level formulas
if PROPAGATE() ≠ true then             ▷ unit/word-level propagation; top level conflict
    return UNSAT
while true do
    if PROPAGATE() = true then                                          ▷ no conflict
        if all variables are assigned then
            return SAT
        else
            decide()
    else                                                           ▷ conflict happens
        if top-level conflict found then
            return UNSAT
        else
            learnt_clause := conflict_analyze()
            backjump(learnt_clause)
```

3 Extending MiniSAT

MiniSAT [6] is a small, complete, and efficient SAT solver which was designed with domain specific extension in mind. The general algorithm for both the MiniSAT and word-level SAT based solver is suggested in Algorithm 1 [15, 6], based on the architecture shown in Figure 1.

3.1 The Architecture and SAT Solving Process in MiniSAT

The input to MiniSAT is a CNF formula, that is, the conjunction of clauses. Each clause is the disjunction of literals, that is, Boolean variables or their negation. The output is either the assignment of all the variables that satisfies the input CNF formula, or "UNSAT" if the formula is unsatisfiable.

First, a literal ℓ is dequeued from the propagation queue, to see if any new literal can be propagated based on this literal, by looking up the Boolean watch list of this literal ($BWatch(\ell)$) and sending the related clauses to do the unit propagation. The unit propagation is the only propagation method applied in MiniSAT which finds clause C where all literals except for one literal ℓ have been made false, then propagates ℓ to *true*. After each round of unit propagation, either a new literal may be propagated in which case this literal will be enqueued into the Boolean propagation queue ($BPQueue$), and the clause C will be added to the explanation database as the reason for this variable b ($Reasons(b)$); or a conflict happens in which case the clause C becomes the conflict clause. If clause C is at the top-level then it means the whole problem is unsatisfiable; otherwise the conflict clause is analyzed based on the explanations of the fixed literals and a learnt clause is synthesized to direct "back-jumping". In addition, the learnt clause is added into the clause database to avoid the same conflict from occurring in the future, which is known as "no-good learning".

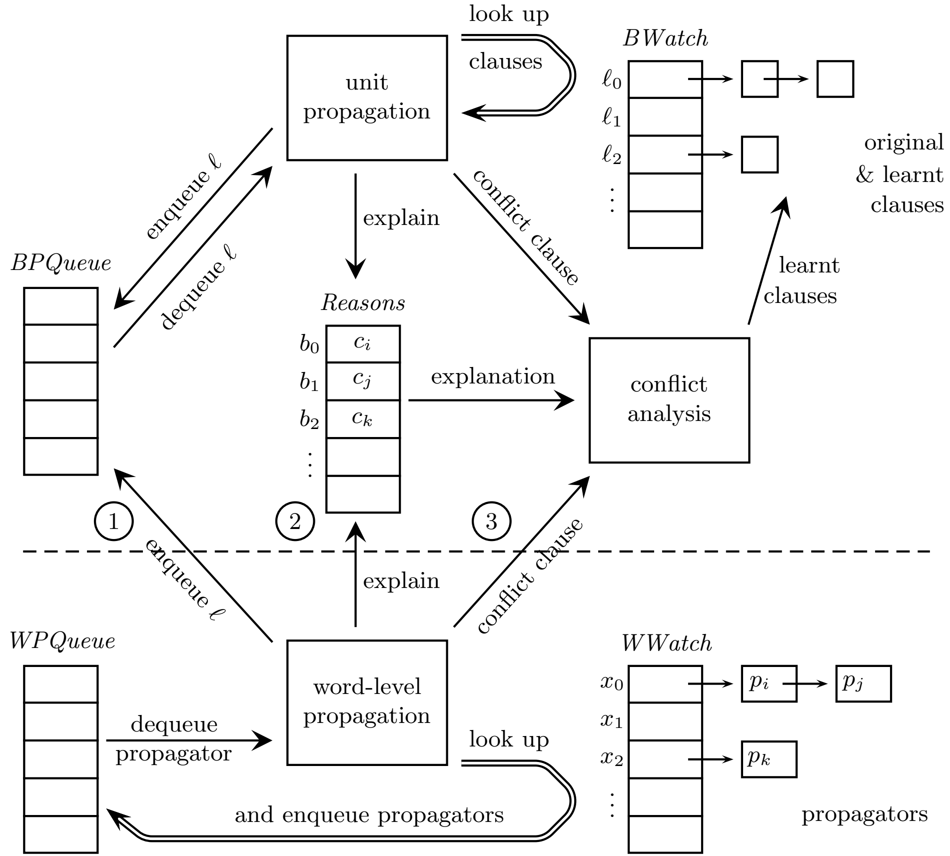

Fig. 1. Overall architecture: MiniSAT (top) and word-level mechanism (bottom)

3.2 Architecture and Solving Process in Word-Level Solver

The extended architecture for our word-level SAT based solver is shown in the bottom part of Figure 1. The input of the word-level solver is both the word-level formulas which are for bit-vector operations, and the CNF formulas which are for Boolean operations. A separate static watch list ($WWatch(x)$) for the word-level propagators of each related bit-vector is added to MiniSAT. Correspondingly, a separate word-level propagator queue (*WPQueue*) for the word-level propagators is added (it will have a lower priority than the Boolean propagation queue). Note that at the beginning, we put all the propagators into the propagator queue and run them to the fix-point.

The extended solving process is shown in Algorithm 2. Once a bit ℓ of an integer x is newly propagated, both this literal is enqueued into the Boolean

Algorithm 2 Extended solving process in word-level SAT based solver

```
function ENQUEUE(literal ℓ, clause C)
    BPQueue.enqueue(ℓ)
    b := var(ℓ)                          ▷ get the corresponding boolean variable b
    Reasons[b] := C     ▷ add the explanation C for b to the explanation database
    if ℓ is in a bit-vector then
        x := word(b)                          ▷ get the corresponding bit-vector x
        for p in WWatch(x) do
            if p is not in WPQueue then
                WPQueue.enqueue(p)        ▷ enqueue propagators not in WPQueue
function PROPAGATE( )
    clause confl := true
    while confl = true do                                          ▷ no conflict
        while ¬BPQueue.isEmpty() ∧ confl = true do
            ℓ := BPQueue.dequeue()
            confl := unit_prop(ℓ)
        if confl = true then                     ▷ BPQueue is empty, no conflict
            p := WPQueue.dequeue()
            confl := word_prop(p)
    return confl
```

propagation queue, and all the related word-level propagators of integer x in the word-level watch list are enqueued into the propagator queue. When a literal is dequeued from the Boolean propagation queue, the corresponding Boolean constraints in the Boolean watch list are invoked to do the unit propagation. Only when the Boolean propagation queue is empty do we start to dequeue propagators from the propagator queue. We thus favour unit propagation since it is faster but weaker, while the word level propagation is more powerful but slower. In addition, since the previously learnt clauses are in the priority queue, previous conflicts can be avoided earlier.

As can be seen from Figure 1, the interactions[1] between MiniSAT (top part) and the word-level mechanism (bottom part) are the propagated literals ①, explanations ②, and the conflict clauses ③. The word-level propagators are required to return explanations for the literals they propagate and return conflict clauses when they detect conflict. Without these capabilities, word-level propagators cannot benefit from the learning capabilities of the SAT solver, including back-jumping and powerful autonomous search.

4 Word-Level Propagators with Bit-Level Explanation

Bit-blasting is the most common approach to bit-vector constraint solving. Bit-blasting rewrites all the word level formulas into large number of low-level propositional formulas although many of them may be redundant and never used in

[1] The algorithms below point out where/when the interactions ①, ②, ③ occur.

the solving process. Instead of doing bit-blasting, we use word-level propagation. The propagators perform propagation, and they also generate explanations in the form of clauses, for literals fixed by propagation. They can be seen as lazy clause generators, generating clauses only as these are needed.

The input of the word level propagators are all bit-vectors. Inside the propagator, we utilize and also extend the propagation rules introduced in [18] to do the propagation on the word level. At the same time we explain every propagated bit at the bit level. After each round of propagation for the bit-vector interval, validity checking (1) is applied on the new intervals. After the checking, either some bits are propagated, or a conflict happens which means a conflict clause (or several conflict clauses) should be returned. The explanation for the propagated bit is a set of literals which are the reason for making the propagated bit fixed. Note that the explanation for each fixed bit can also explain the conflict which happens because of this bit.

Logical Constraints. The detailed word-level propagation rules for the logical constraints can be found in [18]. The explanations for the basic logical constraints are as following. We take the bitwise equality $(x = y)$ as an example: when the ith bit of integer x_i is fixed to 1, we know that the reason is that y_i is already fixed to 1. So the clause $c_2 : \neg y_i \vee x_i$ is the explanation that could explain why x_i is fixed to 1. The explanation for reified equality constraint $b \leftrightarrow x = y$ is introduced in Section 5.1.

- Bitwise Equality $(x = y)$:
  $c_1 : \neg x_i \vee y_i$ $\quad c_2 : \neg y_i \vee x_i$
- Bitwise Conjunction $(z = x \wedge y)$:
  $c_1 : \neg z_i \vee x_i$ $\quad c_2 : \neg z_i \vee y_i$ $\quad c_3 : \neg x_i \vee \neg y_i \vee z_i$
- Bitwise Negation $(x = \sim y)$:
  $c_1 : \neg x_i \vee \neg y_i$ $\quad c_2 : x_i \vee y_i$
- Bitwise Disjunction $(z = x \vee y)$:
  $c_1 : \neg x_i \vee z_i$ $\quad c_2 : \neg y_i \vee z_i$ $\quad c_3 : x_i \vee y_i \vee \neg z_i$
- Bitwise Exclusive Or $(z = x \oplus y)$:
  $c_1 : x_i \vee y_i \vee \neg z_i$ $\quad c_2 : x_i \vee \neg y_i \vee z_i$ $\quad c_3 : \neg x_i \vee y_i \vee z_i$
  $c_4 : \neg x_i \vee \neg y_i \vee \neg z_i$
- Bitwise Conditional $(ite(b, x, y) = z)$:
  $c_1 : \neg b \vee \neg x_i \vee z_i$ $\quad c_2 : \neg b \vee x_i \vee \neg z_i$ $\quad c_3 : b \vee \neg y_i \vee z_i$
  $c_4 : b \vee y_i \vee \neg z_i$ $\quad c_5 : \neg x_i \vee \neg y_i \vee z_i$ $\quad c_6 : x_i \vee y_i \vee \neg z_i$

Arithmetic Constraints. A constraint $z = x + y$ is translated into constraints that introduce two new variables: c for the sequence of carry-ins, and d for carry-outs. As pointed out by Michel and Van Hentenryck [18], a full adder can then be captured with the constraints

$$\begin{aligned} &z = x \oplus y \oplus c \\ &d = (x \,\&\, y) \mid (c \,\&\, (x \oplus y)) \\ &c = d \ll 1 \end{aligned}$$

where the last constraint connects the carry-in bit-vector c and carry-out bit-vector d. By adding the intermediate variables into the addition constraint, the propagator for addition can be divided into several *decomposed* propagators which solve the basic constraints individually. The explanations for the addition constraint simply combines the explanations of these basic constraints. The propagation rules and explanations for inequality constraints will be introduced in Section 5.1.

Structural Constraints. We have extended the propagation rules mentioned in [18] to solve all the structural constraints in the QF_BV category of SMT-LIB2. We can take the structural constraints as the variants of the bitwise equality constraints for bit manipulation. Therefore, the propagation rules for structural constraints are based on the propagation rules of the bitwise equality constraints but with different "masks" to fixing the particular bits to be 1 or 0. The explanations for the structural constraints are also similar to the bitwise equality constraints but with some bit shift ($\ll$,$\gg_u$, $\gg_s$, $rotl$, $rotr$), or fixing some bits value ($\ll$, $\gg_u$, $\gg_s$, ext_u, ext_s).

5 Word-Level Propagation Solving

5.1 Propagators: Composed vs Decomposed

To solve a complicated constraint, one way we can proceed is to create a single "composed" propagator. This propagator may be complex to implement, and may end up finding long explanations. In many cases it can be worth splitting the complicated constraint into several smaller constraints thus decomposing it. Not only are the decomposed components easier to implement, but more importantly, in a learning solver, the intermediate variables introduced may be useful for both search as well as making explanations shorter. Of course the end line for this approach is effectively full bit-blasting. On the other hand, the composed propagators are compact, while the decomposed propagators need the communication among the components. We propose both single propagators and decompositions to implement the reified equality constraint $b \Leftrightarrow x = y$ and the reified inequality constraint $b \Leftrightarrow x \leq_u y$.

Composed Propagators. Propagators return a conflict clause ("*true*" indicates no conflict) and enqueue the propagated literals together with their explanations.

We can implement a propagator for the reified equality constraint: $b \Leftrightarrow x = y$ as shown in Algorithm 3. The propagator reuses the implementation of the propagators for $x = y$ and $x \neq y$, or checks that $x = y$ in the current domain in which case it explains b, or that $x \neq y$ in the current domain, in which case it explains $\neg b$.

The propagator for $x \neq y$ (Algorithm 4) first checks whether x and y are known to be equal and if so, returns a failure explanation. If x and y are known

Algorithm 3 Propagator for $b \Leftrightarrow x = y$

```
function PROP_REIFEQ(bit b, bit-vec x, y)
    if lo(b) = 1 then
        return PROP_EQ(x, y) ③
    else if hi(b) = 0 then
        return PROP_DISEQ(x, y) ③
    else
        if fixed(x) ∧ fixed(y) ∧ lo(x) = lo(y) then                ▷ x = y
            Explanation := lits(x) ∧ lits(y) → b
            ENQUEUE(b, Explanation) ① ②
        else
            z := lo(x) & ~ hi(y) | ~ hi(x) & lo(y)
            if z ≠ 0 then                                          ▷ x ≠ y
                choose i with z_i = 1
                Explanation := lit(x_i) ∧ lit(y_i) → ¬b
                ENQUEUE(¬b, Explanation) ① ②
        return true
```

to differ, it simply returns *true*. Otherwise if there is at most one unfixed bit, and they are otherwise equal it explains why the unfixed bit should be set to the opposite value of the corresponding fixed bit in the other variable. For example, if $x = 11010$, $y = 110 * 0$, we propagate bit $y_1 = 0$, the explanation is $x_4 \wedge x_3 \wedge \neg x_2 \wedge x_1 \wedge \neg x_0 \wedge y_4 \wedge y_3 \wedge \neg y_2 \wedge \neg y_0 \rightarrow \neg y_1$.

Algorithm 4 Propagator for $x \neq y$

```
function DISEQ(bit-vec x, y)
    if fixed(x) ∧ (lo(x) = lo(y) ∨ lo(x) = hi(y)) then   ▷ x fixed; y possibly not fixed
        f := lo(y) ⊕ hi(y)
        if unique 1 bit in f then                        ▷ only one bit of y is unknown
            find i with f_i = 1
            if lit(x_i) = x_i then ℓ := ¬y_i else ℓ := y_i
            Explanation := lits(x) ∧ lits(y) → ℓ
            ENQUEUE(ℓ, Explanation) ① ②
    return true

function PROP_DISEQ(bit-vec x, y)
    if fixed(x) ∧ lo(x) = lo(y) then                             ▷ x = y
        return lits(x) ∧ lits(y) → false ③
    if lo(x) & ~ hi(y) | ~ hi(x) & lo(y) then                    ▷ x ≠ y
        return true
    DISEQ(x, y)
    DISEQ(y, x)
    return true
```

Algorithm 5 Propagator $x \leq_u y$

```
if lo(x) >_u hi(y) then
    f := lo(x) & ~ hi(y)
    i := first 1 bit position in f        ▷ find first bit pair: 1 bit of x and 0 bit of y
    return pos(x) \ {x_j | j < i} ∧ neg(y) \ {¬y_j | j < i} → false ③
for i := w − 1 downto 0 do                                    ▷ propagate the bits in x
    if ¬fixed(x_i) then
        xl := lo(x) | (1 ≪ i)                                  ▷ pretend ith bit is fixed to 1
        if xl >_u hi(y) then
            ℓ := ¬x_i                                           ▷ fix x_i to 0
            f := lo(xl) & ~ hi(y)
            i := first 1 bit position in f
            Explanation := pos(x) \ {x_j | j < i} ∧ neg(y) \ {¬y_j | j < i} → ℓ
            Enqueue(ℓ, Explanation) ① ②
        else
            break
/* the similar algorithm to propagate the bits in y */
return true
```

Similar to the way of solving the equality constraint, for the inequality constraint $b \Leftrightarrow x \leq_u y$, we also need two propagators, one for the constraint $x \leq_u y$ and one for $x >_u y$. Since the propagation rules and the way of generating the explanations of these two constraints are similar, we show only the propagator for $x \leq_u y$, as Algorithm 5. First, we still need to check if there is a conflict, that is, if the lower bound of x is (unsigned) greater than the upper bound of y, in which case a conflict clause needs to be returned. To generate the conflict clause, we go through every bit of x and y bit by bit from the most significant bits to find the first "bit pair" of 1 bit in x and 0 bit in y, and add all the 1 bits in x and 0 bits in y before the "bit pair" (included) to the conflict clause. For example, if $x = 100\mathbf{1}0{*}{*}$, $y = 100\mathbf{0}{*}11$ then the conflict clause is $x_6 \wedge x_3 \wedge \neg y_5 \wedge \neg y_4 \rightarrow \mathit{false}$.

After the conflict checking, we start the propagation which utilizes the propagation rules introduced in [18]. We take the propagation for the bits in variable x as an example. In the propagation of constraint $x \leq_u y$, we can only fix x to 0. But we pretend to fix the first free bit (from left) of x to 1 to see if there is a conflict in which case we know that this free bit must be fixed to 0; otherwise we cannot propagate anything. The way of generating the explanation for this fixed bit is similar to how the conflict clause was generated, but with the pretend lower bound of x. For example, if $x = 1100{*}1{*}$ and $y = 1100\mathbf{0}{*}{*}$ then $xl = 1100\mathbf{1}10$, and the explanation is $x_6 \wedge x_5 \wedge \neg y_4 \wedge \neg y_3 \wedge \neg y_2 \rightarrow \neg x_2$.

The explanations generated by the composed propagators are often large, especially when the bit width of the involved bit-vectors is large. In comparison, each explanation for a basic constraint introduced in Section 4 contains at most three literals.

Decomposed Propagators. The decomposed propagator for equality constraint $b \Leftrightarrow x = y$ is based on this observation [22]:

$$b = \mathsf{msb}(\sim((x - y) \mid (y - x)))$$

We add intermediate variables to split this constraint into several basic constraints which can be processed by the word level propagators already introduced. Note that the $m_1 = x - y$ constraint will be further split as the arithmetic constraint introduced in Section 2 and Section 4. The explanation for the reified equality constraint $b \Leftrightarrow x = y$ is made up by those of the basic constraints—several small explanations with the intermediate literals involved.

$$m_1 = x - y; \quad m_2 = -m_1; \quad m_3 = m_1 \mid m_2; \quad m_4 = \sim m_3; \quad b = \mathsf{msb}(m_4)$$

The decomposed propagator for inequality constraint $b \Leftrightarrow x < y$ is also based on this observation:

$$b = \mathsf{msb}((\sim x \mid y) \;\&\; ((x \oplus y) \mid \sim(y - x)))$$

The way to solve an inequality constraint with decomposed propagators is the same as for the equality constraint.

It is worth pointing out that the two kinds of propagator do not lead to identical search trees. The presence of intermediate variables introduced by the decomposition makes a considerable difference to activity based search, since there are new variables to search on and different initial activities.

5.2 Explanation: Forward vs Backward

Normally in a SAT solver, for every fixed Boolean literal, a reason why it became *true* is required for conflict analysis. Therefore, normally when we fix a Boolean literal in our word-level propagator, we return an explanation for it eagerly, so-called "forward explanation." Another approach, standard for SMT theory solvers [19] and discussed by Gent et al. [10], is to generate the explanation only during conflict analysis where the reason for a propagated literal is required. Compared to the forward explanation method, this has the advantage that explanations are only generated as needed. Furthermore, the "backward explanation" is especially good for our word-level propagator. Our propagators have two parts: one is the propagation part, the other is the explanation generation part which is the more time consuming. Therefore, backward explanation makes propagation faster, but possibly makes conflict analysis slower.

5.3 Conflict Analysis: First vs Highest Level

As already mentioned we can detect conflicts in many bit positions simultaneously. But to choose which one to do the conflict analysis on remains a question.

With bit-blasting, as soon as the first conflict is found, conflict analysis is started, returning a learnt clause of the form $C \vee \ell$, where ℓ is the unique literal

Table 1. Forward explanation vs backward explanation and standard backjumping vs multi-conflict backjumping (times are in seconds)

Problem		F + S		B + S		B + M	
name	number	time	TO	time	TO	time	TO
sage: app1	1176	727	0	432	0	**416**	0
sage: app2	475	8	0	**5**	0	**5**	0
sage: app5	990	44	0	**27**	0	29	0
sage: app6	245	0	0	0	0	0	0
sage: app7	339	4	0	**3**	0	**3**	0
sage: app8	1760	662	1	447	1	542	**0**
sage: app9	2096	370	1	732	**0**	587	**0**
sage: app12	4905	2118	0	**1226**	0	1262	0
stp_samples	424	18	0	**13**	0	**13**	0
bench_ab	284	0	0	0	0	0	0
Total	12694	3951	2	2885	1	**2857**	**0**
Overall time		4951		3385		**2857**	
brummayerbiere3	42	495	33	310	33	**271**	33
spear: cvs_v1.11.22	5	0	4	0	4	0	4
spear: openldap_v2.3.35	6	0	6	0	6	0	6
spear: samba_v3.0.24	4	0	4	0	4	494	**3**
rubik	7	524	2	308	2	407	**1**
uclid_contrib_smtcomp09	7	149	6	90	6	**72**	6
Total	71	1168	55	708	55	1244	**53**
Overall time		28668		28208		**27744**	

(UIP) at the current decision level, and the maximum decision level in the remainder of the clause C determines the level to backjump to. One way to manage conflict analysis for word-level propagation is to choose the first conflict to do the conflict analysis as usual for SAT. We call this "standard backjumping".

An alternative approach is to generate a conflict clause for each bit position that is in conflict. We can then add all the learnt clauses generated to the clause database and then jump to the highest decision level indicated by one of them. This has the advantage of generating more information from the failure, and potentially higher backjumps. We call this "multi-conflict backjumping".

6 Experimental Evaluation

For the experimental data, we pick the folders from the QF_BV category of SMT-LIB2 benchmarks which do not make use of multiplication, division, modulus and remainder, and the bit width for the bit-vector operations is no greater than 64 (the size of our machine register). In total there are more than 12000 test cases. We split them into two categories: easy and difficult, according to the per-problem solve time of the bit-blaster baseline solver. We use a time limit

Table 2. Resource consumption: bit-blaster vs word-level bit-vector solver and composed propagators vs decomposed propagators (memory is in MB)

Problem	bit-blaster			Deq + Dle			Ceq + Dle			Deq + Cle			Ceq + Cle		
name	time	TO	mem	time	TO	mem	time	TO	mem	time	TO	mem	time	TO	mem
app1	393	0	27	416	0	25	381	0	23	1985	32	16	905	32	**12**
app2	9	0	**7**	5	0	11	60	1	11	252	18	10	6311	1	8
app5	49	0	23	29	0	20	271	1	15	1505	15	16	1025	17	**10**
app6	0	0	**7**	0	0	8	0	0	8	0	0	8	0	0	8
app7	3	0	**7**	3	0	8	3	0	8	1	14	8	1	14	8
app8	1127	0	16	542	0	16	828	1	14	2062	2	13	1585	2	**10**
app9	863	0	15	587	0	14	303	2	13	2122	1	12	1387	3	**9**
app12	1052	0	19	1262	0	20	994	2	16	972	6	17	595	8	**11**
stp_sam	37	0	39	13	0	31	8	0	23	13	0	30	7	0	**20**
bench_ab	1	0	**7**	0	0	8	0	0	8	0	0	8	0	0	8
Total	12694	0	167	2857	0	161	2848	7	139	8912	88	138	11816	77	**104**
Overall time	3534			**2857**			6348			52912			50316		
brumm3	402	31	33	271	33	41	422	33	31	228	32	27	208	32	**16**
cvs	688	2	9	0	4	10	0	4	9	0	5	10	0	5	**8**
openldap	176	5	353	0	6	240	0	6	**46**	0	6	238	0	6	47
samba	0	4	1005	494	3	676	24	0	124	0	4	751	5	0	**87**
rubik	87	2	**7**	407	1	16	838	1	11	589	1	13	56	2	10
uclid	0	7	**7**	72	6	109	710	3	25	0	7	138	393	4	25
Total	1353	51	1414	1244	53	1092	1994	47	246	817	55	1177	662	49	**193**
Overall time	26853			27744			25494			28317			**25162**		

of 500 seconds. In Tables 1 and 2, "time" means the total time in seconds for all the successful test cases in the folder; "TO" is the number of cases that timed out; "Total" is the total time of all successful test cases; "Overall time" is "Total" plus 500 seconds penalty for each unsuccessful case, which gives an overall "score" similar to what is used in SMT competitions. All the experiments were performed on a commodity computer with a Core-i7 CPU (2.7 GHz) and 5 GB RAM.

The first experiment compares forward explanation (F) versus backward explanation (B), as well as standard backjumping (S) versus multi-conflict backjumping (M). We implemented three variants of the word-level bit-vector solvers which all use the decomposed word-level propagators for equality and inequality constraints. The reason we only look at three variants is that the two parameters (F/B, S/M) do not interact with each other. Table 1 shows that, first, backward explanation outperforms the forward explanation significantly, especially when the test cases becomes time consuming. Second, the multi-conflict backjumping outperforms the standard backjumping considerably in both categories.

The second experiment compares bit-blasting with word-level bit-vector solving using composed and decomposed propagators. We implemented a vanilla bit-blaster as a baseline to compare against, which uses the decomposition of equality and inequality applied in the decomposed word-level solver Deq+Dle.

Table 3. Conflicts and inspections during the search (a '-' indicates a time of 0)

Problem	bit-blaster		Deq + Dle	
name	fail/sec	insp(k)/sec	fail/sec	insp(k)/sec
app1	2.6	14.4	3.1	12.6
app2	4.8	20.3	9.2	24.0
app5	1.0	6.3	2.1	9.1
app6	-	-	-	-
app7	76.3	66.0	61.3	36.7
app8	2.6	21.3	4.5	11.7
app9	2.2	16.0	3.8	10.4
app12	1.5	4.2	1.4	3.2
stp_sam	0.3	10.1	1.1	10.4
bench_ab	2.0	23.0	-	-

Problem	bit-blaster		Ceq + Cle	
name	fail/sec	insp(k)/sec	fail/sec	insp(k)/sec
brumm3	41.5	507.0	109.9	297.0
cvs	1507.7	6041.5	11134.8	6645.2
openldap	211.5	2404.3	930.7	1833.7
samba	84.0	2675.5	200.0	1800.0
rubik	280.6	4155.5	875.0	2141.1
uclid	198.9	335.1	3506.5	7354.8

Since Table 1 suggests the B+M combination has merit, all word-level bit-vector solvers listed in Table 2 use backward explanation and multi-conflict backjumping. However, they use different combinations of composed propagators (C) and decomposed propagators (D) for equality (eq) and inequality constraints (le). Table 2 shows the resource consumption including the running time and average memory usage (mem) in MB. The results show that the Deq+Dle word-level propagator is typically faster than bit-blasting on the easy cases, using less memory. For the difficult cases, the Ceq+Cle word-level propagator outperforms the bit-blasting in some cases and also uses much less memory. But in general the bit-blasting method is more robust.

Table 3 shows the average number of conflicts per second (fail/sec), and the average number of inspections[2] in thousand per second (insp(k)/sec) that occur during the search. We compare bit-blasting only against the best word-level solver as identified above, that is, Deq+Dle for easy cases and Ceq+Cle for difficult cases. Note that the bit-blasting often finds fewer conflicts during the search with more propagation, while the word-level solvers often find more conflicts with less propagation. That is because propagating and checking the conflicts at word-level is parallel in some sense, resulting in a higher rate of conflict-finding as well as the reduction in inspections.

[2] A call to a unit or word-level propagator (which may or may not result in fixing new bits).

7 Related Work

Word-level reasoning on bit-vector logic is NEXPTIME-complete [14]. In spite of this, the problem has received much attention recently, albeit with limited progress. Current related work falls into one of or the combination [1] of three categories:

Word-level reasoning based on lazy SMT techniques: Hadarean et al. [12] propose two word-level solvers an equality solver and inequality solver as the theory solver in their lazy bit-vector solver. But they cannot express the conflict at the bit-level which significantly affects the efficiency of the method as they showed in [12].

Word-level reasoning based on constraint programming: Bardin et al. [2] propose two word-level propagators based on the Constraint Logic Programming framework. One is called Is/C which is to solve linear arithmetic constraints, and the other is the BL (Bit-List) propagator which runs in linear time to solve the linear bitwise constraints. Constraint propagators for modular arithmetic constraints have been proposed by Gotlieb et al. [11] who utilize so-called clockwise intervals in a linear fragment of modular integer constraints. None of these CP approaches support learning, or compare with bit-blasting.

Word-level reasoning based on linear programming: This approach is to transform the problem into linear programming constraints [3, 23]. For RTL verification, the performance of LP solvers are often no better than SMT solvers as reported in [16].

8 Conclusion

We have extended word-level propagation algorithms of Michel and Van Hentenryck [18] to produce an explaining solver. We have introduced decomposed counterparts to the proposed propagators, as these were not constant time. We also utilize a concept of multi-conflict backjumping, capitalizing on the fact that word-level propagation can detect multiple failures simultaneously. We have given an empirical comparison of word-level propagation versus bit-blasting, the standard approach to these problems. Our solver is a prototype, still to be tuned. Nevertheless it shows that, with careful engineering, a word-level propagation solver can compete with bit-blasting, particularly on easier problems.

For future work, it may be advantageous to apply some word-level simplification as done with the linear solver in STP [8, 9]. We also need to deal with non-linear arithmetic operations, one way or other. Finally, an interesting line of research would be to combine word-level propagation with word-level search, especially stochastic local search as recently suggested by Fröhlich *et al.* [7].

Acknowledgment

This work is supported by the Australian Research Council under ARC grant DP140102194.

References

1. Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. Constraint integer programming: A new approach to integrate CP and MIP. In L. Perron and M. A. Trick, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5015 of *Lecture Notes in Computer Science*, pages 6–20. Springer, 2008.
2. Sébastien Bardin, Philippe Herrmann, and Florian Perroud. An alternative to SAT-based approaches for bit-vectors. In J. Esparza and R. Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6015 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2010.
3. Raik Brinkmann and Rolf Drechsler. RTL-datapath verification using integer linear programming. In *VLSI Design*, pages 741–746. IEEE Computer Society, 2002.
4. Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.
5. Byron Cook, Daniel Kroening, and Natasha Sharygina. Cogent: Accurate theorem proving for program verification. In *Computer Aided Verification (CAV'05)*, volume 3576 of *Lecture Notes in Computer Science*, pages 296–300. Springer, 2005.
6. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing (SAT'04)*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer, 2004.
7. Andreas Fröhlich, Armin Biere, Christoph M. Wintersteiger, and Yussef Hamadi. Stochastic local search for satisfiability modulo theories. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the 29th AAAI Conference on Artificial Intelligence*, pages 1136–1143. AAAI Press, 2015.
8. Vijay Ganesh. *Decision Procedures for Bit-Vectors, Arrays and Integers*. PhD thesis, Stanford University, 2007.
9. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In W. Damm and H. Hermanns, editors, *Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531. Springer, 2007.
10. Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy explanations for constraint propagators. In M. Carro and R. Pena, editors, *Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2010.
11. Arnaud Gotlieb, Michel Leconte, and Bruno Marre. Constraint solving on modular integers. In *Proceedings of the Ninth International Workshop on Constraint Modelling and Reformulation (ModRef'10)*, 2010.
12. Liana Hadarean, Clark Barrett, Dejan Jovanović, Cesare Tinelli, and Kshitij Bansal. A tale of two solvers: Eager and lazy approaches to bit-vectors. In A. Biere and R. Bloem, editors, *Computer Aided Verification (CAV'14)*, volume 8559 of *Lecture Notes in Computer Science*, pages 680–695. Springer, 2014.
13. Frank Hutter, Domagoj Babic, Holger H. Hoos, and Alan J. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design (FMCAD'07)*, pages 27–34. IEEE Comp. Soc., 2007.
14. Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *Proceedings of SMT12*, pages 44–55, 2012.

15. Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, 2008.
16. Sarvani Kunapareddy, Sriraj D. Turaga, and Solomon S. T. M. Sajjan. Comparison between LPSAT and SMT for RTL verification. In *Proceedings of the 2015 International Conference on Circuit, Power and Computing Technologies*, pages 1–5. IEEE Computer Society.
17. Rhishikesh S. Limaye and Sanjit A. Seshia. Beaver: An SMT solver for quantifier-free bit-vector logic. Master's thesis, EECS Department, University of California, Berkeley, May 2010.
18. Laurant D. Michel and Pascal Van Hentenryck. Constraint satisfaction over bit-vectors. In M. Milano, editor, *Constraint Programming: Proceedings of the 2012 Conference*, volume 7514 of *Lecture Notes in Computer Science*, pages 527–543. Springer, 2012.
19. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.
20. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
21. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, 2008.
22. Henry S. Warren Jr. *Hacker's Delight*. Addison Wesley, 2003.
23. Z. Zeng, P. Kalla, and M. Ciesielski. LPSAT: A unified approach to RTL satisfiability. In *Design, Automation and Test in Europe (DATE'01)*, pages 398–402. IEEE Press, 2001.