Matching multiplications in Bit-Vector formulas

Supratik Chakraborty[1], Ashutosh Gupta[2], and Rahul Jain[2]

[1] IITB
[2] TIFR

Abstract. The SMT solvers for the theory of bit-vectors are widely used. The bit-vector formulas often involve word-level arithmetic operations. The bit-vector formulas with multiplications are particularly hard for SMT solvers. Therefore, it is more important that a solver uses all the structure available in the problem including the word-level reasoning. Sometimes multiplications are decomposed and implemented in several ways and the solver fails to see the word-level multiplication. In this paper, we present a solver that identifies the decomposed multipliers, adds theory axioms to the input formula that encodes equivalence of the decomposed multiplication and the word-level multiplication, and solves the modified input using an available solver. We have added the algorithms in the rewriting engine of Z3 and applied on benchmarks. Our modification solves several formulas that are intractable by any other leading solver.

1 Introduction

In recent years, SMT (Satisfiability Modulo Theories) solving has emerged as a key technology for testing, analysis and verification of hardware and software systems. An SMT solver combines the reasoning power of multiple first-order theory solvers to check the satisfiability of a formula in a combination of theories. An important theory, formulas of which occur naturally when reasoning about systems with finite-precision data, is the theory of fixed-width bit-vectors (or words). Indeed, several important technologies, viz. bounded model checking of RTL designs [?,?,?] and of programs with bounded-width integers [?,?,?], symbolic trajectory evaluation of RTL designs [?], constrained test generation for programs and RTL designs [?,?,?] etc. rely crucially on checking the satisfiability of a formula in a combination of theories that includes the theory of bit-vectors. Improving the efficiency of bit-vector solvers can therefore help improve the performance of several technologies that use SMT solvers.

If the system under analysis involves word-level arithmetic operations, viz. multiplying the contents of two registers or adding two bounded-width integers, it is desirable to reason about the word-level arithmetic operations directly, i.e. without decomposing them into sub-operations. This allows the bit-vector theory solver to use simplification rules that encode properties of word-level arithmetic operators, viz. $x+y = y+x$ or $x*0 = 0$, when checking the satisfiability of a formula. Unfortunately, word-level arithmetic operators may not be implemented as monolithic operators, especially if the system under analysis is a hardware system. Depending on the complexity of the operation, the operator may be decomposed and implemented in one of several ways, guided by reasons related to performance, layout, component count etc. For example, addition of two words may be implemented using a ripple-carry adder, carry-propagate adder, carry-save adder, etc. Similarly, one can use a grade-school multiplier (implementing the standard long multiplication), Booth-encoded multiplier, Wallace-tree multiplier etc. to implement multiplication of two words. While bit-vector solvers in state-of-the-art SMT solvers [?,?,?,?] can reason about explicitly specified word-level arithmetic operators efficiently, empirical evidence [?]) shows that their performance degrades significantly when the operator is not specified explicitly, but as a composition of several sub-operators (as in a carry-save adder, or a grade-school multiplier). This motivates us to ask *if we can preprocess an SMT formula efficiently and provide "hints" to the SMT solver to help check satisfiability efficiently.*

At first sight, the above problem appears to be one of reverse-engineering a decomposed implementation (encoded in the SMT formula obtained from analyzing a system) of a complex word-level operation, which can be achieved by matching a sub-formula with a pre-specified "pattern". A closer inspection, however, reveals that there are several subtle complications. First, and contrary to intuition, the same decomposed implementation may correspond to multiple reverse-engineered word-level operations. A simple example of this is obtained by considering the high-school multiplication algorithm. Second, even if we are able to recover a complex word-level operation from a sub-formula corresponding to a decomposition in the given SMT formula, rewriting the sub-formula with the recovered operation may not correlate with an improved satisfiability check for the overall SMT formula. This can happen for two reasons:

- When multiple word-level operations can be recovered from the same sub-formula, one cannot decide a priori which word-level operation will be beneficial for the overall satisfiability check.
- Two sub-formulas that share further sub-formulas may match two pre-specified patterns, and thereby yield two different word-level operations. Rewriting one of the sub-formulas with its corresponding word-level operation removes the shared sub-formulas, and thereby precludes rewriting the other sub-formula with its corresponding word-level operation. In general, it may not be possible to determine a priori whether one of these re-writes helps the overall satisfiability checking more than the other. In fact, re-writing both of the sub-formulas may be desirable, but is precluded if we re-write one of the sub-formulas.
- The recovery is a local "peep-hole" operation that looks only at the sub-formula matching a specific "pattern". Specifically, the recovery is oblivious of the context in which the sub-formula appears in the overal SMT formula, and rewriting the sub-formula with the recovered word-level operation may not help (and may even hurt) the overall satisfiability check.

– In general, the sub-formula matching a pre-specified "pattern" may help in simplifying some part of the SMT formula, while the recovered word-level operation may help in simplifying some other part of the SMT formula. Re-writing the matched sub-formula with the recovered operation therefore precludes using the sub-formula to simplify the SMT formula.

2 Motivating example

add a figure with the example.

3 Preliminaries

In this section, we will present the syntax of the theory of quantifier-free foxed-width bit-vector formulas(`QF_BV`), the basics of solving methods used by SMT solvers, and the multiplication methods that are of our interest.

3.1 `QF_BV` syntax

A bit-vector is a fixed sequence of bits. We will denote bit-vectors by x,y,z, etc. We will refer to the blocks of bits in bit-vectors. Therefore, we may declare that a bit-vector x is accessed in blocks of size w. Let x_i denote the ith block from the least significant bit(LSB).

A `QF_BV` term t and formula F is constructed using the following grammar.

$$\begin{aligned} t ::=& t * t \mid t + t \mid x \mid n^w \mid t \bullet t.... \\ F ::=& t = t \mid t \bowtie t \mid \neg F \mid F \vee F \mid F \wedge F \mid F \oplus F \mid ... \end{aligned}$$

where x is a bit-vector variable, n^w is a constant number represented in w bits, $\bowtie \in \{\leqslant, <, \geqslant, >\}$, and $\bullet$ is a binary operator that concatenates bit-vectors. Here we are only presenting the parts of the theory that are relevant to our discussion. In this paper, all the variables and arithmetic operators are unsigned. All the inputs and outputs of arithmetic operators have same bit length. Let $len(t)$ denote the bit length of a term t. If $w \geqslant len(t)$, let $zeroExt(t, w)$ be a shorthand for $0^{w-len(t)} \bullet t$.

We know that $a*b$ is commutative and, in pattern matching, we will not make a distinction between $a * b$ and $b * a$. $t == s$ is true iff t and s are syntactically same. For bit-vectors x, y, and t, let $w = max(len(x), len(y), len(t))$. Let $[x*y = t]$ denote term $x' * y' = t'$, where $x' = zeroExt(x, w)$, $y' = zeroExt(y, w)$, and $t' = zeroExt(t, w)$. Similarly, $[x * y]$ is defined.

3.2 SMT solvers for `QF_BV`

SMT(satisfiability modulo theory) solvers are specialized solvers that solve formulas of a given theory. The leading SMT solvers for `QF_BV` apply several simplification passes followed by bit-blasting, i.e., translating input to a Boolean satisfiability problem(SAT problem). The bit-blasted SAT problem is solved using conflict driven clause learning(CDCL) based procedures. Some of the leading SMT solvers are Z3, Boolector, and CVC4.

For our algorithm, we assume that a `QF_BV` SMT solver SMTSolver with the standard interface is available. The interface includes a function *add*(F) that adds a formula into the solver and a function *checkSat*$()$ that checks the satisfiability of the conjunction of the added formulas.

3.3 Multipliers

Multiplication is an expensive operation to implement in hardware. There are several designs of multipliers for varying resource constraints. If one can have a large number of gates then Wallace tree multiplier can be used. Otherwise, one may decompose the multiplication task in small multiplications and combine the results appropriately. For example, long multiplier and Booth multiplier. Here, we will discuss long and Wallace tree multiplier.

Long multiplier Let us consider bit-vectors x and y that are accessed in the blocks of w bits and are of size kw. Ashutosh: Rahul: suggest a rewrite. The long multiplier decomposes the kw bits multiplication into chunks of w bits multiplications, called *partial products*. The partial products are summed with appropriate offsets to obtain the final result. The following notation is typically used to illustrate the long multiplication.

$$\begin{array}{cccccc}
 & & x_k & \dots & x_1 & \\
 & & y_k & \dots & y_1 & * \\
\hline
 & & x_k * y_1 & \dots & x_1 * y_1 & \\
 & \iddots & \vdots & \iddots & & \\
x_k * y_k & \dots & x_1 * y_k & & + & \\
\hline
\end{array}$$

$x_i * y_j$s are the partial products. Each $x_i * y_j$ is left shifted $(i+j-2)w$ bits. In the above scheme all the partial products that have same offset are aligned in single column. After the shifts, all the partial results are added in some order. The bit-width of the partial products is $2w$. In the `QF_BV` notation, $x_i * y_j$ is correctly denoted by $(0^w \bullet x_i) * (0^w \bullet y_j)$. Therefore, the bits of the partial products in neighbouring columns overlap and they can not be simply concatenated. The long multiplier does not specify the order of the addition of the shifted partial products. Therefore, there are several possible designs for a given k and w.

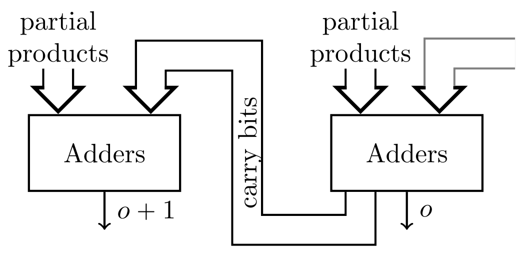
Example 1. Consider bit-vectors v_1, v_2, u_1, and u_2 of length 2. Let us apply long multiplication in multiplying $v_2 \bullet 0^2 \bullet v_1$ and $u_2 \bullet v_2 \bullet u_1$. We obtain the following partial products.

$$\begin{array}{cccccc}
 & & v_2 & 0^2 & v_1 & \\
 & & u_2 & v_2 & u_1 & * \\
\hline
 & & v_2 * u_1 & 0^4 & v_1 * u_1 & \\
 & v_2 * v_2 & 0^4 & v_1 * v_2 & & \\
v_2 * u_2 & 0^4 & v_1 * u_2 & & + & \\
\hline
\end{array}$$

We need to sum the partial products. However, if their non-zero bits do not overlap then we can simply concatenate them. And finally we may sum the concatenated vectors. The following is one of the combination of the concatenations and summations for the long multiplication.

$$(0^4 \bullet v_1 * u_2 \bullet v_1 * u_1) + (v_2 * u_2 \bullet v_2 * u_1 \bullet 0^4) + (0^2 \bullet v_2 * v_2 \bullet v_1 * v_2 \bullet 0^2)$$

Wallace tree multiplier Wallace tree decomposes the multiplication all the way down to single bits. Let us consider bit-vectors x and y that are accessed in the blocks of 1 bit and are of size k. In a Wallace tree, a partial product is the multiplication of single bits $x_i * y_j$. The multiplication of single bits is the conjunction of the bits, i.e., $x_i \wedge y_j$. There is no carry generated due to the multiplication of single bits. The partial product $x_i * y_j$ is aligned with the $(i+j-2)$th bit of output. Let us consider oth output bit. All the partial products that are aligned to o are summed using full adder and half adders. The full adders are used if more than two bits are available that are yet to be summed and half adders are used if there are only two bits that are left to be summed. The carry bits that are generated by the adders are aligned to $(o+1)$th output bit. The carry bits are summed to the partial products for $(o+1)th$ bit using adders as illustrated in the following figure.



Both the above multiplication methods do not fully specify the design. Therefore, there are several ways to implement the multiplications. Therefore, it is not trivial to verify that a hardware design indeed implements a multiplication.

4 Pattern detection

In this section, we will present our method for solving formulas that contain implementations of multiplications. Our method first attempts to match multiplications that are decomposed using long or Wallace tree multiplication. If we match some subterms of the input formula are indeed instances of the multiplications, we add tautologies stating that the terms are equal to the multiplications of the matched bit-vectors. Our matching method may find multiple matches for a subterm. We add a tautology for each match. Let us first present our method of matching long multiplication.

Algorithm 1 MatchLong(t)

```
Require: $t$ : a term in QF_BV
Ensure: $M$ : matched multiplications := $\varnothing$
 1: if $t == (s_{1k_1} \bullet ... \bullet s_{11}) + ... + (s_{pk_p} \bullet ... \bullet s_{p1})$ then
 2:     Let $w$ be such that for some $s_{ij} = (0^w \bullet a) * (0^w \bullet b)$ and $len(a) == w$
 3:     $\Lambda := \lambda i.\varnothing$
 4:     for each $s_{ij}$ do
 5:         $o := (\sum_{j'<j} len(s_{ij'}))/w$
 6:         if $s_{ij} == 0$ then continue;
 7:         if $s_{ij} == (0^w \bullet a) * (0^w \bullet b)$ and $len(a) == w$ then $\Lambda_o.insert(a * b)$
 8:         else return $\varnothing$
 9:     return getMultOperands($\Lambda$,$w$)
10: return $\varnothing$
```

4.1 Matching long multiplication

In Algorithm 1, we present a function MatchLong that takes a QF_BV term t and returns a set of matched multiplications. The algorithm and the subsequent algorithms are written such that as soon as it becomes clear that no multiplication can be matched then they return empty set. At line 1, we match t with a sum of concatenations and if the match fails then clearly t is not a long multiplication. At line 2, we find a partial product among s_{ij} and extract the block size w used by the long multiplication. The loop at line 4 populates the vector of the set of partial products Λ. Λ_i denotes the partial products that are aligned at the ith block. Each s_{ij} must either be 0 or a partial product of width w. Otherwise, t is declared unmatched at line 8. At line 5, we compute the alignment o for s_{ij}. If s_{ij} happens to be a partial product then it is inserted in Λ_o at line 7. At line 9, we call getMultOperands to identify the operands of the multiplication from Λ if t is indeed a long multiplication.

4.2 Partial products to operands

Ashutosh: Needs some discussions about the nature of the algorithm. In Algorithm 2, we present a function getMultOperands that takes a vector of multiset of partial products Λ and block width w, and returns a set of matched multiplications. At line 1, we compute h and l that establishes the range of search for the operands. We maintain two candidate operands x and y of size hw. We also maintain a vector of bits *backtrack* that encodes the possibility of flipping the uncertain decisions. Due to the scheme of the long multiplication, the highest non-empty entry in Λ must be a singleton set. If Λ_h contains a single partial product $a * b$, we assign x_h and y_h the operands of $a * b$ arbitrarily. Otherwise, we declare the match has failed by returning $\varnothing$. We assign **ff** to $backtrack_h$, which states that no need of backtracking at index h. The loop at line 8 iterates over index i starting from h and decreases i in each iteration. In each iteration, it assigns values to x_i, y_i, and $backtrack_i$.

Algorithm 2 GETMULTOPERANDS($\varLambda, w$)

```
Require: $\varLambda$ : array of multisets of the partial products
Ensure: $M$ : matched multiplications := $\varnothing$
 1: Let $l$ and $h$ be the smallest and largest $i$ such that $\varLambda_i \neq \varnothing$, respectively.
 2: $x, y$ : candidate operands that are accessed in block size $w$ and of size $hw$
 3: if $\varLambda_h == \{a * b\}$ then
 4:     $x_h := a; y_h := b$; $backtrack_h :=$ ff;
 5: else
 6:     return $\varnothing$
 7: $i := h; l_x := h; l_y = h$;
 8: while $i > 1$ do
 9:     $i := i - 1; C := \varLambda_i$
10:     for $j \in (h-1)..(i+1)$ do
11:         if $x_j \neq 0$ and $y_{h+i-j} \neq 0$ then
12:             if $x_j * x_{h+i-j} \notin C$ then goto BACKTRACK
13:             $C := C - \{x_j * x_{h+i-j}\}$
14:     match $C$ with
15:       | $\{x_h * b, y_h * d\} \rightarrow x_i := d; y_i := b$; $backtrack_i := (x_h == y_h)$;
16:       | $\{x_h * y_h\} \rightarrow x_i := 0; y_i := x_h$; $backtrack_i :=$ tt;
17:       | $\{x_h * b\} \rightarrow x_i := 0; y_i := b$; $backtrack_i := (x_h == y_h)$;
18:       | $\{y_h * b\} \rightarrow x_i := b; y_i := 0$; $backtrack_i :=$ ff;
19:       | $\{\} \rightarrow x_i := 0; y_i := 0$; $backtrack_i :=$ ff;
20:       | _ $\rightarrow$ goto BACKTRACK;
21:     if $x_i \neq 0$ then $l_x = i$
22:     if $y_i \neq 0$ then $l_y = i$
23:     if $h - l_x - l_y < 1$ then goto BACKTRACK;
24:     if $i == 1$ then
25:         for $o \in 0..(l-1)$ do
26:             $x'$ := Right shift $x$ until $o$ trailing 0 blocks in $x$
27:             $y'$ := Right shift $y$ until $l - o$ trailing 0 blocks in $y$
28:             $M := M \cup \{x' * y'\}$
29:     else
30:         continue;
31:     BACKTRACK:
32:       Choose smallest $i' \in h..(i+1)$ such that $backtrack_{i'} ==$ tt
33:       if no $i'$ found then return $M$
34:       $i := i'$; SWAP$(x_i, y_i)$; $backtrack_i :=$ ff
```

The algorithm may not have enough information at the ith iteration and the chosen value for x_i and y_i is be wrong. Whenever, the algorithm realizes that such a mistake has happened it jumps to line 31. It increases back the value of i to the latest i' that allows backtracking. It swaps the assigned values of x_i and y_i, and disables future backtracking to i by setting $backtrack_i$ to **ff**.

Let us look at the loop at line 8 again. We also have variables l_x and l_y that contain the index of the least non-zero entries in x and y, respectively. At line 9, we decrement i and $\varLambda_i$ is copied to C. At index i, the sum of the aligned partial products is the following.

$$x_h * y_i + \underbrace{x_{h-1} * y_{i+1} + \cdots + x_{i+1} * y_{h-1}}_{\text{operands seen at the earlier iterations}} + x_i * y_h$$

We have already chosen the operands of the middle multiplications in the previous iterations. Only the multiplications at the extreme ends have y_i and x_i that are not assigned yet. In the loop at line 10, we remove the middle multiplications. If any of the needed multiplication is missing then we may have made a mistake earlier and we jump for backtracking. After the loop, we should be left with at most two partial products in C. We match C with five patterns at lines 14-19 and update x_i, y_i, and $backtrack_i$ accordingly. If none of the pattern match, we jump for backtracking at line 20. In some cases we clearly determine the value of x_i and y_i, and we are not certain in the other cases. We set $backtrack_i$ to **tt** in the uncertain cases to indicate that we may return back to index i and swap x_i and y_i. In the following list, we discuss the uncertain cases.

line 15: If C has two elements $x_h * b$ and $y_h * d$, there is an ambiguity in choosing x_i and y_i if $x_h == y_h$.
line 16: If C has a single element $x_h * y_h$, there are two possibilities.
line 17: If $C = \{x_h * b\}$ and b is not y_h then similar to the first case there is an ambiguity in choosing x_i and y_i if $x_h == y_h$. Line 18 is similar.
line 19: If C is empty then there is no uncertainly.

At line 21-22, we update l_x and l_y appropriately. The condition at line 23 ensures that the expected least index i such that $\Lambda_i \neq \varnothing$ is greater than 0. At line 24, we check if $i == 1$, which means a match has been successful. To find the appropriate operands, we need to right shift x and y such that the total number of their trailing zero blocks is $l - 1$. We add the matched $x * y$ to the match store M. And, the algorithm proceeds for backtracking to find if more matching exists.

4.3 Matching Wallace tree multiplication

A Wallace tree has a cascade of adders that take partial products and carry bits as input to produce the output bits. In our matching algorithm, we find the set of inputs to the adders for an output bit and classify them into partial products and carry bits. The half and full adders are defined as follows.

$$\begin{aligned} &sumHalf(a,b) = a \oplus b && sumFull(a,b,c) = a \oplus b \oplus c \\ &carryHalf(a,b) = a \wedge b && carryFull(a,b,c) = (a \wedge b) \vee (b \wedge c) \vee (c \wedge a). \end{aligned}$$

The sum output of a half/full adders are the result of xor operations of inputs. To find the input to the adders, we start from an output bit and follow backward until we find the input that are not the result of some xor.

In Algorithm 3, we present a function MatchWallace that takes a `QF_BV` term t and returns a set of matched multiplications. At line 1, t is matched with

Algorithm 3 MatchWallaceTree(t)

```
Ensure: $t$ : a term in QF_BV
 1: if $t == (t_k \bullet ... \bullet t_1)$ then
 2:     $\Lambda := \lambda i.\varnothing$; $\Delta$ : vector of multiset of terms $:= \lambda i.\varnothing$
 3:     for $i \in 1..k$ do
 4:         if $len(t_i) \neq 1$ then return $\varnothing$
 5:         $S := \{t_i\}$;$\Delta_i := \{t_i\}$
 6:         for $S \neq \varnothing$ do
 7:             $t \in S$; $S := S - \{t\}$
 8:             if $t == s_1 \oplus .... \oplus s_p$ then
 9:                 $S := S \cup \{s_1, .., s_p\}$;$\Delta_i := \Delta_i \cup \{s_1, .., s_p\}$
10:             else if $t == carryFull(a, b, c)$ and $a, b, c, a \oplus b, a \oplus b \oplus c \in \Delta_{i-1}$ then
11:                 $\Delta_{i-1} := \Delta_{i-1} - \{a, b, c, a \oplus b\}$
12:             else if $t == carryHalf(a, b)$ and $a, b, a \oplus b \in \Delta_{i-1}$ then
13:                 $\Delta_{i-1} := \Delta_{i-1} - \{a, b\}$
14:             else if $t == a \wedge b$ then
15:                 $\Lambda_i.insert(a * b)$;
16:             else return $\varnothing$
17:         if $\Delta_{i-1} \neq \{t_{i-1}\}$ then return $\varnothing$
18:     return getMultOperands($\Lambda$,1)
19: return $\varnothing$
```

a concatenation of single bit terms t_1,..,t_n. Similar to Algorithm 1, we maintain the partial product store Λ. For each i, we also maintain the multiset of terms Δ_i that were used as input to the adders for the ith bit. In the loop at line 6, we traverse to the subterms until a subterm is not the result of a xor. In the traversal, we also collect the inputs of xors in Δ_i, which will help us in checking that all the carry inputs in adders for t_{i+1} are generated by the adders for t_i. If the term t is not the result of xors then we have the following possibilities.

line 10-13: If t is the carry bit of a half/full adder, and the inputs, the intermediate result of the sum bit, and the output sum bit of the adder are in Δ_{i-1}. We remove the inputs and intermediate result of the adder from Δ_{i-1}. We do not remove the output sum bit from Δ_{i-1}, since it may be used as input to some other adder.

line 14-15: If t is a partial product, we record it in Λ_i.

line 16: Otherwise, we return $\varnothing$.

At line 17, we check that $\Delta_{i-1} = \{t_{i-1}\}$, i.e., all carry bits from the adders for t_{i-1} are consumed by the adders for t_i exactly once. Again if the check fails then we return $\varnothing$. After the loop at 3, we have collected the partial products in Λ. At line 18, we call getMultOperands($\Lambda, 1$) to get all the matching multiplications.

Algorithm 4 OurSolver(F)

```
Require: $F$ : a QF_BV formula
Ensure: sat/unsat/undef
1: SMTSolver.add($F$)
2: for each subterm $t$ in $F$ do
3:     if $M := \textsc{MatchLong}(t) \cup \textsc{MatchWallaceTree}(t)$ then
4:         for each $x * y \in M$ do
5:             SMTSolver.add($[x * y = t]$)
6: return SMTSolver.checkSat()
```

4.4 Our solver

Using the above pattern matching algorithms, we modify an existing solver SMTSolver, which is presented in the Algorithm 4. OurSolver adds the input formula F in SMTSolver. For every subterm of F, we attempt to match with both long multiplication or Wallace tree multiplication. For each discovered matching $x * y$, we add a bit-vector tautology $[x * y = t]$ to the solvers, which is obtained after appropriately zero-padding x, y, and t.

5 Correctness

We need to prove that each $[x * y = t]$ added in OurSolver is a valid formula. First we will prove the correctness of getMultOperands. If either of x or y is zero, we assume term $x * y$ is also simplified to zero.

Theorem 1 *If $x * y \in$ getMultOperands(Λ, w), then*

$$\Lambda_i = \{x_1 * y_{i-1}, ..., x_{i-1} * y_1\}$$

where x_k and y_k are the kth block of x and y of size w, respectively.

Proof. After each iteration of the loop at line 8, if no backtracking triggered, the loop body ensures that the following holds at the end, which readers may easily check.

$$\Lambda_i = \{x_h * y_i, x_{h-1} * y_{i+1}, ..., x_i * y_h\} \quad (1)$$

Due to the above equation, if $x_j * y_k \in \Lambda_i$, $i = h - j - k$. If the program enters at line 25, it has a successful match and $i = 1$. Since $h - l_x - l_y \geqslant 1$, $\Lambda_l = \{x_{l_x} * y_{l_x}\}$ and $l = h - l_x - l_y$. We choose $o \leqslant l$, and shift x and y according to lines 26-27. After the shift, we need to write equation (1) as follows.

$$\Lambda_i = \{x_{h-(l_x-o)} * y_{i-(l_y-l+o)}, ..., x_{i-(l_x-o)} * y_{h-(l_y-l+o)}\}. \quad (2)$$

We can easily verify that the sum of the indexes in each of the partial products is i. Since all x_k is zero for $k > h - (l_x - o)$ and all y_k is zero for $k > h - (l_y - l + o)$, we may rewrite equation (2) as follows.

$$\Lambda_i = \{x_1 * y_{i-1}, ..., x_{i-1} * y_1\}.$$

Theorem 2 *If* $m * n \in \textsc{MatchLong}(t)$*,* $[m * n = t]$ *is valid.*

Proof. We collect partial products with appropriate offsets o at line 5. The pattern of t indicates that the net result is the sum of the partial products with the respective offsets. $\textsc{getMultOperands}(\Lambda, w)$ returns the matches that produces the sums. Therefore, $[m * n = t]$ is valid.

Theorem 3 *If* $m * n \in \textsc{MatchWallaceTree}(t)$*,* $[m * n = t]$ *is valid.*

Proof. All we need to show that t is intending to sum the partial products stored in Λ. Rest of the proof follows the previous theorem.

Ashutosh: needs tightening. Each bit t_i must be the sum of the partial products Λ_i and the carry bits produced by the sum for t_{i-1}. The algorithm identifies the terms that are added to obtain t_i. We only need to prove that the terms that are not identified as partial products are carry bits of the sum for t_{i-1}. Let us consider such a term t. Let us suppose the algorithm identifies t as an output of the carry bit circuit of a full adder (half adder case is similar) with inputs a, b, and c. The algorithm also checks that a, b, c, $a \oplus b$ and $a \oplus b \oplus c$ are the intermediate results of the sum for t_{i-1}. Therefore, t is one of the carry bits. Since a, b, $a \oplus b$ and c are removed from Δ_{i-1} after the match of the adder, all the identified adders are disjoint. Since we require that all the elements of Δ_{i-1} are eventually removed except t_{i-1}, all carry bits are added to obtain t_i. Therefore, Λ carry the expected partial products of a Wallace tree.

6 Experiments

We have implemented our algorithms as a part of Z3 [?] SMT solver[3]. We evaluate the performance of our algorithms using benchmarks that are industrial and handcrafted hardware verification problems. We compare our tool with Z3, Boolector[?] and CVC4[?]. Our experiments show that the solvers time out on most of the benchmarks and our tool produces results within the set time limit.

Implementation We implemented the algorithms inside Z3. We have added about 1500 lines of code in the bit vector rewrite module because it allows an easy access to the abstract syntax tree of the input formula. An important aspect of the implementation is the ability to exit as early as possible if the match is going to fail. We implemented various preliminary checks including the ones mentioned in Algorithm 1. We ensure that the size of Λ_i is upper bounded appropriately as per the scheme of long multiplication. We exit as soon as the upper bound is violated. We have implemented three version of OurSolver by varying SMTSolver. We used Z3, Boolector, and CVC4 for the variations. In the case of Z3, we insert the learned tautologies in the Z3 solver within the current execution. For Boolector and CVC4, we forcefully stop the Z3 solver

[3] `https://github.com/rahuljain1989/Bit-vector-multiplication-pattern`

after running our matching algorithms, print the learned tautologies in a file along with the input formula, and run the solvers in a separate process on the newly generated formula. We use the following versions of the solvers: Z3(4.4.2), Boolector(2.2.0), CVC4(1.4).

Benchmarks Our experiments include 20 benchmarks. Initially, we received an industrial hardware verification benchmark in SystemVerilog involving long multiplication that was not solved by any of the solvers in 24 hours. The example inspired our current work and to evaluate it we generated several similar benchmarks. For long multiplication, we generated benchmarks by varying three characteristics, firstly the total bit length of the input bit-vectors, secondly the width of each block, and thirdly assigning specific blocks as equal or set them to zero. Our SystemVerilog benchmarks are fed to STEWord [**?**], a hardware verification tool. STEWord takes SystemVerilog design as input and generates the corresponding SMT1 formula. We convert the SMT1 formula to SMT2 format using Boolector. In the process, Boolector extensively simplifies the input formula but retains the overall structure. We have generated benchmarks also for Wallace tree multiplier similar to the long multiplication. For n-bit Wallace tree multiplier, we have written a script that takes n as input and generates all the files needed as input by STEWord.

Results We compare our tool with Z3, Boolector and CVC4. Tables 1-2 present the results of the experiments. We chose timeout to be 600 seconds.

In table 1 we present the results of the long multiplication and Wallace tree multiplier experiments. The first 13 rows correspond to long multiplication experiments. The columns under SMTSolver are the run times of the solvers to prove the satisfiability of the input benchmark. The solvers timout on most of the benchmarks.

The last three columns are the run times of the three versions of OurSolver to prove the satisfiability of the benchmarks. OurSolver with CVC4 makes best use of the added tautologies. CVC4 is quickly able to infer that the input formula and the added tautologies are negations of each other justifying the timings captured. OurSolver with Boolector and Z3 does not make the above inference, leading to more running time. Boolector and Z3 bit blast the benchmarks having not been able to detect the structural similarity. However, the added tautologies help Boolector and Z3 to reduce the search space, after the sat solver is invoked on the bit blasted formula.

The last 7 rows correspond to the Wallace tree multiplier experiments. Since, the multiplier involves a series of half and full adders, the size of the input formula increases rapidly as the bit vector size increases. Despite the blowup in the formula size, OurSolver with Z3 is quickly able to infer that the input formula and the added tautology are negations of each other. However, OurSolver with Boolector and CVC4 do not make the inference, leading to larger run times. This is because of the synctactic structure of the learned tautology from our implementation inside Z3. The input formula has 'and' and 'not' gates as its building blocks, whereas Z3 transforms all 'ands' to 'ors'. Therefore, the added

Table 1: Multiplication experiments. Times are in seconds.

	SMTSolver				OurSolver			
Benchmark	Z3	Boolector	CVC4	Portfolio	Z3	Boolector	CVC4	Portfolio
base	184.3	42.2	16.54	16.54	0.53	43.5	0.01	0.01
ex1	2.99	0.7	0.36	0.36	0.33	0.8	0.01	0.01
ex1_sc	t/o	t/o	t/o	t/o	1.75	t/o	0.01	0.01
ex2	0.78	0.2	0.08	0.08	0.44	0.3	0.01	0.01
ex2_sc	t/o	1718	2826	1718	3.15	1519	0.01	0.01
ex3	1.38	0.3	0.08	0.08	0.46	0.7	0.01	0.01
ex3_sc	t/o	1068	t/o	1068	3.45	313.2	0.01	0.01
ex4	0.46	0.2	0.03	0.03	0.82	0.2	0.01	0.01
ex4_sc	287.3	62.8	42.36	42.36	303.6	12.8	0.01	0.01
sv_assy	t/o	t/o	t/o	t/o	0.07	t/o	0.01	0.01
mot_base	t/o	t/o	t/o	t/o	13.03	1005	0.01	0.01
mot_ex1	t/o	t/o	t/o	t/o	1581	13.8	0.01	0.01
mot_ex2	t/o	t/o	t/o	t/o	2231	13.7	0.01	0.01
wal_4bit	0.09	0.05	0.02	0.02	0.09	0.1	0.04	0.09
wal_6bit	2.86	0.6	0.85	0.6	0.28	0.8	14.36	0.28
wal_8bit	209.8	54.6	225.1	54.6	0.59	30.0	3471	0.59
wal_10bit	t/o	1523	t/o	1523	1.03	98.6	t/o	1.03
wal_12bit	t/o	t/o	t/o	t/o	1.55	182.3	t/o	1.55
wal_14bit	t/o	t/o	t/o	t/o	2.27	228.5	t/o	2.27
wal_16bit	t/o	t/o	t/o	t/o	2.95	481.7	t/o	2.95

tautology has no 'ands'. The difference in the synctactic structure between the input formula and the added tautology makes it difficult for Boolector and CVC4 to make the above inference.

7 Related Work

8 Conclusion and future work

We have presented two classes of pattern detection for the formulas with multipliers implemented using long school multiplier and Wallace tree. We have implemented the algorithms and applied to several handcrafted and industrial benchmarks. The experiments suggest a significant improvement in performance.

We are currently extending our procedure to support Booth multiplier and other difficult arithmetic patterns. We are also working to add proof generation support for the added valid formulas. We could not include proof generation in this work, since basic infrastructure of proof generation is missing in Z3 bit-vector rewriter module. We are also planning to request Z3 team to adopt our modification in their standard distribution.

We carried out many other experiments involving different combinations of multiplications. One such experiment was multiplication of three bit vectors.

Our observations indicate that none of the solvers are able to infer the structural similarity based on the two tautologies added for the two long multiplications. We carried out similar experiments involving multiple operands, expressions of the form $X * X + Y * Y$, $X * (Y + Z)$, $(X * (Y * Z)) * W$. None of the solvers were able to make use of the added tautologies(one for each long multiplication). The problem seems to be on two fronts: one is that to check equality of two terms, the solvers want exact structural similarity and secondly the infrastructure of using one assertion to infer the satisfiability of another does not seem to be well developed. We aim to bridge this gap as part of our future work.

Table 2: Conflicts and decisions in the experiments

	SMTSolver						OurSolver					
Benchmark	Z3		Boolector		CVC4		Z3		Boolector		CVC4	
	Conflicts	Decisions	Conflicts	Decisions	Conflicts	Decisions	Conflicts	Decisions	Conflicts	Decisions	Conflicts	Decisions
base	172k	203k	170k	228k	127k	148k	724	1433	148k	194k	0	0
ex1	7444	9065	7320	9892	8396	10k	474	890	7090	9558	0	0
ex1_sc	t/o —	t/o —	t/o 5.6M	t/o 7.7M	t/o 2.1M	t/o 2.3M	2564	5803	t/o 5M	t/o 6.8M	0	0
ex2	2067	2599	1789	2612	2360	3374	919	1420	1747	2526	0	0
ex2_sc	t/o —	t/o —	3.3M	4.9M	1.9M	2.3M	5076	8981	2.7M	4.3M	0	0
ex3	4109	5402	1682	3166	3374	4754	905	1321	3882	7305	0	0
ex3_sc	t/o —	t/o —	3.8M	5.9M	t/o 2.9M	t/o 3.6M	4814	9012	805k	1.4M	0	0
ex4	647	801	612	715	463	588	630	918	405	519	0	0
ex4_sc	143k	165k	130k	165k	110k	130k	115k	138k	67k	114k	0	0
sv_assy	t/o —	t/o —	t/o 5.3M	t/o 9.8M	t/o 1.8M	t/o 2.5M	0	0	t/o 4.7M	t/o 9.4M	0	0
mot_base	t/o —	t/o —	t/o 6M	t/o 10M	t/o 2.2M	t/o 2.9M	12k	30k	2.4M	5.5M	0	0
mot_ex1	t/o —	t/o —	t/o 4.4M	t/o 6.1M	t/o 1.7M	t/o 2M	280k	409k	30k	57k	0	0
mot_ex2	t/o —	t/o —	4.5M	6.3M	t/o 1.7M	t/o 1.9M	358k	496k	30k	57k	0	0
wal_4bit	363	435	283	343	396	479	0	0	300	378	442	486
wal_6bit	8077	9831	6887	9544	11k	12k	0	0	8523	12k	68k	54k
wal_8bit	180k	209k	177k	249k	1.2M	1.1M	0	0	94k	174k	5.9M	3.8M
wal_10bit	t/o —	t/o —	2.7M	3.7M	t/o 5.4M	t/o 2.2M	0	0	249k	519k	t/o 2.8M	t/o 1.2M
wal_12bit	t/o —	t/o —	t/o 5.2M	t/o 6M	t/o 4.1M	t/o 1.9M	0	0	416k	855k	t/o 2.3M	t/o 916k
wal_14bit	t/o —	t/o —	t/o 4.9M	t/o 6.5M	t/o 3M	t/o 907k	0	0	500k	999k	t/o 1.2M	t/o 412k
wal_16bit	t/o —	t/o —	t/o 4.8M	t/o 6.6M	t/o 1.9M	t/o 512k	0	0	941k	2M	t/o 672k	t/o 196k