Matching multiplications in Bit-Vector formulas

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

[1] IITB, India
[2] TIFR, India

Abstract. SMT solvers for the theory of fixed-width bit-vectors are widely used. Bit-vector formulas often involve word-level arithmetic operations. Empirical evidence shows that bit-vector formulas with multiplication are often hard for SMT solvers to reason about. Therefore, it is important that an SMT solver uses all the structure available in the problem, including the word-level reasoning. Sometimes multiplication operators are decomposed and implemented in alternative ways, and the solver fails to identify the word-level multiplication operation. In this paper, we present a pre-processing heuristic that identifies the decomposed multipliers, and adds special assertions to the input formula that encodes equivalence of the decomposed multiplication and the word-level multiplication. The pre-processed formulas are then solved using an available solver. We have implemented our pre-processing algorithms in the rewriting engine of Z3 and applied these on a suite of benchmarks. Our experiments with three state-of-the-art SMT solvers show that our heuristic allows several formulas to be solved quickly, while the same formulas time out without the pre-processing step.

1 Introduction

In recent years, SMT solving has emerged as a powerful technique for testing, analysis and verification of hardware and software systems. A wide variety of tools today use SMT solvers as part of their core reasoning engines; examples include bounded model checkers [1,2,3,4], static assertion checkers [5,6], word-level symbolic trajectory evaluators [7], constrained test generators [8,9,10], concolic simulators [11], among others. A common approach used by these tools is to model the behaviour of a system using formulas in a combination of first-order theories, and reduce the given problem to checking the (un)satisfiability of a formula in the combined theory. SMT solvers play a central role in this scheme of things, since they combine decision procedures of individual first-order theories to check the satisfiability of a formula in the combined theory. Not surprisingly, techniques to improve the performance of SMT solvers have attracted significant attention over the years. The literature contains a rich body of heuristic strategies for improving the performance of theory-specific solvers (see [12,13] for excellent expositions). In this paper, we add to the repertoire of such heuristics by proposing a pre-processing step that conjoins an input formula with specially constructed assertions, without changing its semantics. We focus on formulas in the quantifier-free theory of fixed-width bit-vectors with multiplication, and show by means of experiments that our pre-processing heuristic yields significant performance benefits in many cases for three state-of-the-art SMT solvers, namely Z3 [14], CVC4 [15] and Boolector [2]. Significantly, our heuristic helps reduce the solving time for multiple examples by upto several orders of magnitude when the input formula is unsatisfiable.

The primary motivation for our work comes from word-level bounded model checking (WBMC) [4,1] and word-level symbolic trajectory evaluation (WSTE) [7] of embedded hardware systems. Specifically, we focus on systems that process data, represented as fixed-width bit-vectors, using arithmetic operators. Examples of such systems include digital signal processing filters, graphics accelerators, encryption and decryption modules, custom datapath implementations etc. When reasoning about these systems, it is often necessary to check whether a high-level property, specified using bit-vector arithmetic operators (viz. addition, multiplication, division), is satisfied by a model of the system implementing a data-processing algorithm. For reasons related to performance, power, area, ease of design etc., complex arithmetic operators with large bit-widths are often implemented by composing several smaller, simpler and well-characterized blocks. For example, a 128-bit multiplier may be implemented using one of several multiplication algorithms, viz. long multiplication [16], Booth-encoded multiplication [17] or Wallace-tree multiplication [18], after partitioning its 128-bit operands into narrower, say 8-bit wide, blocks. SMT formulas resulting from WBMC/WSTE of such systems are therefore likely to contain terms with higher-level arithmetic operators (viz. 128-bit multiplication) encoding the specification, and terms that encode a lower-level implementation of these operators in the system (viz. a Wallace-tree multiplier). Efficiently reasoning about such formulas requires exploiting the semantic equivalence of these alternative representations of arithmetic operators. Unfortunately, our study, which focuses on systems using the multiplication operator, reveals that three state-of-the-art SMT solvers (Z3, CVC4 and Boolector) encounter serious performance bottlenecks in identifying these equivalences. This manifests dramatically when reasoning about the unsatisfiability of formulas.

A motivating example: To illustrate the severity of the problem, we consider the SMT formula arising out of WSTE applied to a pipelined serial multiplier circuit, originally used as a benchmark in [7]. The circuit reads in two 32-bit operands sequentially from a single 32-bit input port and stores them in internal registers, before multiplying them and making the 64-bit result available in an output register. The circuit also has several control signals that can be used to change the flow of control, effectively delaying the computation of the result.

The property to be checked asserts that if a and b denote the two operands that are read in, then after the computation is over, the output register indeed has the product $a *_{[32]} b$, where $*_{[32]}$ denotes 32-bit multiplication. The RTL design (i.e. system implementation), as used in [7], makes use of the $*$ operator in SystemVerilog, with 32-bit operands. The Language Reference Manual of SystemVerilog specifies that this amounts to using a 32-bit multiplication operation directly. The SMT formula resulting from a WSTE run on this example therefore contains terms with only 32-bit multiplication operators, and no terms encoding a lower-level multiplier implementation. This formula is shown to be unsatisfiable within a fraction of a second by Boolector (and also by CVC4 and Z3). Note that in WSTE (as also in WBMC), the SMT formula encodes violation of a property by a bounded run of the system. Hence, unsatisfiability of the formula implies the absence of any bounded violating runs.

We now change the RTL in the above example to reflect the implementation of 32-bit multiplication by the long-multiplication algorithm [16], where each 32-bit operand is partitioned into 8-bit blocks. The corresponding WSTE run yields an SMT formula that contains terms with 32-bit multiplication operator (derived from the property being checked), and also terms that encode the implementation of a 32-bit multiplier using long-multiplication. Surpisingly, none of Boolector, CVC4 and Z3 succeeded in deciding the satisfiability of the resulting formula even after 24 hours on the same computing platform. The heuristic strategies in these solvers failed to identify the semantic equivalence of terms encoding alternative representations of 32-bit multiplication, and proceeded to bit-blast the formulas, leading to this dramatic run-time blowup.

Problem formulation: The above example demonstrates that the inability to identify semantic equivalence of alternative representations of arithmetic operators plagues multiple state-of-the-art SMT solvers. Therefore, a heuristic that helps in this respect and is generic (not solver-specific) would be highly desirable. This motivates us to ask: *Can we pre-process an SMT formula containing terms encoding alternative representations of bit-vector arithmetic operators, in a solver-independent manner, so that multiple solvers benefit from it?* We answer this question positively in this paper for the multiplication operator. Note, however, that like all heuristics, we cannot guarantee that our pre-processing step helps in every case. The motivating example, that originally timed out after 24 hours on three solvers, is shown to be unsatisfiable by Z3 in 0.073s and by CVC4 in 0.017s, after application of our heuristic. However, Boolector does't benefit from our heuristic in this example. Nevertheless, there are several other examples where Boolector benefits significantly, as discussed in Section 5.

Term re-writing vs adding tautological assertions: Prima facie, the above problem can be solved by reverse-engineering the lower-level implementation of a bit-vector arithmetic operator, and by re-writing terms encoding the lower-level implementation with terms using the bit-vector operator. Indeed, variants of this approach have been used by earlier researchers in different contexts [19,20,21,22,23]. In the context of SMT solving, however, complications can potentially arise if we simply re-write a term encoding one representation of an arithmetic operator by another term encoding a different representation of the same operator. For example, as shown in Example 2 of Section 2.6, the same collection of sums-of-partial-products arising from long-multiplication can be obtained from two different bit-vector multiplication operations. This makes it difficult to determine which term re-writes should be used to help the SMT solver decide the satisfiability of the input formula. Furthermore, even if we are able to uniquely identify the equivalence of two terms representing the same arithmetic operation, re-writing one term with another may not correlate with improved solver performance for various reasons. Indeed, re-writing one term by another is a "peep-hole" transformation that is oblivious of the overall context in which the terms appear in the SMT formula. What appears beneficial locally may not be beneficial in the overall (un)satisfiability check. In addition, syntactially distinct terms that are semantically equivalent may play different roles when reasoning about different sub-formulas of an SMT formula. For example, one term may enable a re-write rule that helps simplify one sub-formula, while a syntactically distinct but semantically equivalent term may enable another re-write rule that helps simplify another sub-formula. Re-Writing one term by another precludes the possibility of both terms contributing to improving the performance of the overall satisfiability check.

Thus, re-writing a term encoding a specific implementation of a bit-vector arithmetic operator with another term encoding another implementation of the operator may not help in all cases. In this paper, we present a heuristic alternative to address the above problem, focusing only on bit-vector multiplication. Given a bit-vector formula φ that contains terms with two different representations of bit-vector multiplication, our heuristic searches for patterns corresponding to two multiplication algorithms (long multiplication and Wallace-tree multiplication) in the terms. Instead of re-writing the matched sub-terms directly, as has been done in earlier work, we conjoin the formula φ with additional assertions that equate a matched sub-term with the corresponding bit-vector arithmetic operator. Note that each added assertion is a tautology, and hence does not change the semantics of the formula. Significantly, since no re-writes are done, we can express multiple semantic equivalences without removing any syntactic term from the formula. This is an important departure from earlier techniques, such as [21], that rely on sophisticated re-writes of the formula. Our experiments show that the added tautological assertions succeed in preventing bit-blasting in several cases, while in other cases they help in pruning the search space even after bit-blasting. Both effects eventually translate to improved performance of the SMT solver. Since our heuristic simply pre-processes the input formula by adding assertions, it is independent of the internals of any specific solver, and can be used with multiple solvers. Our experiments show that the performance of different SMT solvers on the pre-processed formulas can vary. Hence, we propose a portfolio approach to solving the pre-processed formulas. We show experimentally that a portfolio solver using pre-processed formulas significantly outperforms a portfolio solver using the original formulas.

2 Preliminaries

In this section, we present some basics of the theory of quantifier-free fixed-width bit-vector formulas (QF_BV), and discuss two well-known multiplication algorithms of interest.

2.1 Theory of Bit-vectors

All computing systems operate by manipulating finite sequences of zeros and ones - bit-vectors. A computer system uses bit-vectors to encode information, for example numbers. Owing to the finite domain of these bit-vectors, the semantics of operations such as addition no longer matches what we are used to when reasoning about unbounded types, for example the natural numbers. As a simple example, consider the following formula:

$$(x - y > 0) \leftrightarrow (x > y)$$

This formula holds over the integers, but if x and y are interpreted as finite-width bit vectors, this equivalence no longer holds, owing to possible overflow of the subtraction operation.

A bit-vector is a vector of Boolean values with a given length l:

$$b : \{0, ..., l - 1\} \rightarrow \{0, 1\}$$

We denote bit-vectors by x,y,z, etc., and often refer to blocks of bits in a bit-vector. For example, we may declare that a bit-vector x is accessed in blocks of width w. Let x_i denote the ith block of bits, with the block containing the least significant bit (LSB) having index 0.

2.2 Syntax

Grammar for a subset of bit-vector arithmetic:

$formula ::= formula \wedge formula \mid \neg formula \mid (formula) \mid atom$
$atom ::= term\ rel\ term \mid Boolean\ Identifier \mid term[constant]$
$rel ::= <|=$
$term ::= term\ op\ term \mid x \mid\sim term \mid n^w \mid term[constant : constant] \mid$
$\quad ite(atom, term, term)$
$op ::= + \mid - \mid \cdot \mid / \mid<<\mid>>\mid \& \mid \mid \mid \otimes \mid \bullet$

where the operators have their usual meaning, x is a bit-vector variable, n^w is a binary constant represented using w bits. Note that we have only presented above parts of the grammar that are relevant to our discussion. For more details, the reader is referred to [24,12]. We assume that all variables and arithmetic operators are unsigned. Following the SMTLIB [25] convention, we also assume that arguments and results of an arithmetic operator have the same bit width. Let $len(t)$ denote the bit width of a term t. If $w \geqslant len(t)$, let $zeroExt(t, w)$ be a shorthand for $0^{w-len(t)} \bullet t$.

If an operator $\texttt{op}$ is commutative, when matching patterns, we will not make a distinction between $a \ \texttt{op}\ b$ and $b \ \texttt{op}\ a$. We use the notation "$t == s$" to denote that t and s are syntactically identical. Given bit-vector terms x, y, and t, suppose $w = max(len(x), len(y), len(t))$. We use "$[x * y = t]$" to denote term $x' * y' = t'$, where $x' = zeroExt(x, w)$, $y' = zeroExt(y, w)$, and $t' = zeroExt(t, w)$. Similarly, the notation $[x * y]$ is used to denote $x' * y'$, where $x' = zeroExt(x, len(x) + len(y))$ and $y' = zeroExt(y, len(x) + len(y))$.

2.3 Signature

A typical signature Σ_{bv} for selected functions and predicates of the bit-vector theory is presented below. The set of sorts Σ_{bv}^S contains infinitely many sort symbols $[n]$ where n is a strictly positive natural number. Formally, $\Sigma_{bv}^S = \{[n] \mid n > 0 \text{ and } n \in \mathcal{Z}\}$.

Symbol	Name	Arity	Sort Map τ
$[0]_{[1]}$	constant	0	$[1]$
$[01]_{[2]}$	constant	0	$[2]$
$\bullet$	concat	2	$([n], [m], [n+m])$
$[i : j]$	extract	1	$([n], [j-i+1])$ for $n > i > j \geqslant 0$
$\sim$	bitwise not	1	$([n], [n])$
$\vert$	bitwise or	2	$([n], [n], [n])$
$+$	plus	2	$([n], [n], [n])$
$>>$	right shift	2	$([n], [n], [n])$
$=_{[n]}$	equal	2	$([n], [n], Bool)$
$<$	less than	2	$([n], [n], Bool)$

2.4 Semantics

We now give a formal definition of the semantics of a fixed width bit-vector arithmetic formula. To help us define the semantics we first define three mappings:

- $bitToNat : [1] \rightarrow [0, 1]$ defined as: bitToNat(b) = if $b = 1$ then 1 else 0.
- $natBv_n : \mathbb{N} \rightarrow [n]$ defined as:
  $natBv_n(x) = \lambda i : [0, n-1].if(x \div 2^i)\text{mod } 2 = 0$ then 0 else 1
- $bvNat : [n] \rightarrow [0, 2^n - 1]$ defined as: $bvNat(f) = \sum_{i=0}^{n-1} bitToNat(f(i)).2^i$

$bitToNat$ converts one bit bit-vector to the corresponding natural number 0 or 1. $natBv_n$ converts natural numbers to their corresponding bit-vectors and $bvNat$ converts bit-vectors to their corresponding natural numbers. With these three mappings in place, the semantics can be given by:

- For $[n] \in \Sigma_{bv}^S$, $A_{[n]} = \{\lambda x : [0, n-1].f(x) \mid f : [0, n-1] \rightarrow \{0, 1\}\}$
- For constant symbols:
  - $[\![0_{[1]}]\!] = \lambda x : [0].0$
  - $[\![1_{[1]}]\!] = \lambda x : [0].1$
- For function symbols:
  - $[\![a_{[n]} \bullet b_{[m]}]\!] = \lambda x : [0, n+m-1].$if $x < m$ then $[\![b]\!](x)$ else $[\![a]\!](x-m)$
  - $[\![a_{[n]}[i : j]]\!] = \lambda x : [0, i-j].[\![a]\!](j+x)$
  - $[\![\sim a_{[n]}]\!] = \lambda x : [0, n-1].$ if $[\![a]\!](x) = 0$ then 1 else 0
  - $[\![a_{[n]} \mid b_{[n]}]\!] = \lambda x : [0, n-1].$ if $[\![a]\!](x) = 1$ or $[\![b]\!](x) = 1$ then 1 else 0
  - $[\![a_{[n]} + b_{[n]}]\!] = natBv_n(bvNat([\![a]\!]) + bvNat([\![b]\!]))$

    - $\llbracket a_{[n]} >> b_{[n]} \rrbracket = natBv_n(bvNat(\llbracket a \rrbracket) \div 2^{bvNat(\llbracket b \rrbracket)})$
- For predicate symbols:
    - $\llbracket a_{[n]} =_{[n]} b_{[n]} \rrbracket =$ if $(bvNat(\llbracket a \rrbracket) = (bvNat(\llbracket b \rrbracket)$ then $true$ else $false$
    - $\llbracket a_{[n]} < b_{[n]} \rrbracket =$ if $(bvNat(\llbracket a \rrbracket) < (bvNat(\llbracket b \rrbracket)$ then $true$ else $false$

The concatenation and extract operations are the only ones that can alter the length of bit-vectors. The arithmetic operations interpret the bit-vector as a base-2 natural number and are equivalent to the corresponding mod 2^n arithmetic operations, where n is the size of the bit-vectors.

2.5 Decision procedures for bit-vectors

The leading SMT solvers for `QF_BV` apply several simplification and rewriting passes to decide the satisfiability of the input formula. If these do not succeed in solving the problem, the solvers bit-blast the formula, i.e., translate the bit-vector formula to an equivalent propositional formula on the constituent bits of the bit-vectors. This reduces the bit-vector satisfiability problem to one of propositional satisfiability (SAT). The bit-blasted SAT problem is then solved using conflict driven clause learning (CDCL)[26,27] based SAT procedures.

Below we present a generic bit-blasting algorithm to decide the satisfiability of the input formula ϕ. Some terminology:

- $At(\phi)$ → set of atoms in ϕ
- $a \in At(\phi)$ is replaced by $e(a)$ → the propositional encoder of a
- Propositional skeleton $e(\phi)$ → resulting formula after propositional encoding of the atoms
- $T(\phi)$ → set of terms in ϕ
- $e(t)$ → a vector of new Boolean variables for each BV term $t \in T(\phi)$
- $e(t)_i$ → variable for the bit with index i of the term t

Algorithm 1 BIT BLASTING

```
B := e($\phi$);
for $t_{[l]} \in T(\phi)$ do
    for $i \in \{0, \ldots, l-1\}$ do
        set $e(t)_i$ to a new Boolean variable;
for $a \in At(\phi)$ do
    $B := B \wedge Constraint(e, a)$;
for $t_{[l]} \in T(\phi)$ do
    $B := B \wedge Constraint(e, t)$;
return $B$
```

For our work, we assume access to a generic `QF_BV` SMT solver, called SMTSOLVER, with the standard interface. We assume that this interface includes a function $add(F)$ that adds a formula F to the context of the solver, and a function *checkSat*() that checks the satisfiability of the conjunction of all formulas added to the context of the solver. Note that such interface is available with most state-of-the-art solvers, viz. BOOLECTOR, CVC4 and Z3.

2.6 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. 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.

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

$x_i * y_j$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.

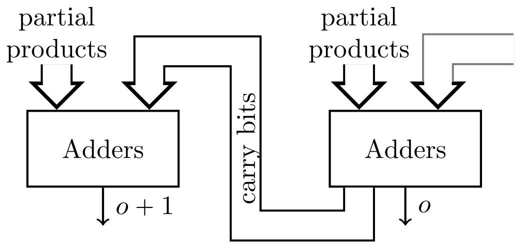
$$(0^4 \bullet v_1 * u_2 \bullet v_1 * u_1) + (v_2 * u_2 \bullet v_2 * u_1 \bullet 0^4) + (0^2 \bullet v_2 * v_2 \bullet v_1 * v_2 \bullet 0^2)$$

Example 2. As another interesting example, consider long multiplication applied to $v_2 \bullet 0^2 \bullet v_2$ and $0^2 \bullet v_1 \bullet v_1$. We obtain the following partial products.

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

Note that, if we had applied the long multiplication to $v_1 \bullet 0^2 \bullet v_1$ and $0^2 \bullet v_2 \bullet v_2$, we would have got the same partial products. This shows that simply knowing the collections of partial products at different indexes does not allow us to uniquely determine the operands. Recall that this problem was alluded to in Section 1

Wallace tree multiplier[18] 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.

Algorithm 2 MatchLong(t)

```
Require: $t$ : a term in QF_BV
Ensure: $M$ : matched multiplications := $\emptyset$
 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 $len(a) == len(b) == w$
 3:     $\Lambda := \lambda i.\emptyset$
 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 $len(a) == len(b) == w$ then
 8:             $\Lambda_o.insert(a * b)$
 9:         else return $\emptyset$
10:     return getMultOperands($\Lambda$,$w$)
11: return $\emptyset$
```

3 Pattern detection

In this section, we will present our method for solving formulas that contain implementations of multiplications. Our method attempts to match multiplications that are decomposed using long or Wallace tree multiplication. If we match some subterms of the input formula as 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 to the input and solve using an available solver. Let us first present our method of matching long multiplication.

3.1 Matching long multiplication

In Algorithm 2, 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 the form mentioned in the condition at line 7. Otherwise, t is declared unmatched at line 9. At line 5, we have computed the alignment o for s_{ij}. If s_{ij} happens to be a partial product, it is inserted in Λ_o at line 8. At line 10, we call getMultOperands to identify the operands of the long multiplication from Λ if t is indeed a long multiplication.

Algorithm 3 getMultOperands(Λ, w)

```
Require: $\Lambda$ : 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 $\Lambda_i \neq \varnothing$, respectively.
 2: $x, y$ : candidate operands that are accessed in block size $w$ and of size $hw$
 3: if $\Lambda_h == \{a * b\}$ 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:     $i := i - 1; C := \Lambda_i$
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
```

3.2 Partial products to operands

In Algorithm 3, 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. The algorithm proceeds by incrementally choosing a pair of operands with insufficient information and backtracks if the guess is found to be wrong.

At line 1, we compute h and l that establishes the range of the 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. We assign **ff** to $\mathit{backtrack}_h$, which states that no need of backtracking at index h. If Λ_h does not contain a single partial product, we declare the match has failed by returning $\emptyset$. The loop at line 8 iterates over index i from h to 1. In each iteration, it assigns values to x_i, y_i, and $\mathit{backtrack}_i$.

The algorithm may not have enough information at the ith iteration and the chosen value for x_i and y_i may 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 $\mathit{backtrack}_i$ 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 Λ_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 partial products in the previous iterations. Only the partial products 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 partial products. If any of the needed partial product 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 corresponding to $x_h * y_i$ and $x_i * y_h$. We match C with the five patterns at lines 14-19 and update x_i, y_i, and $\mathit{backtrack}_i$ 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 $\mathit{backtrack}_i$ 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 uncertainty.

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 $\Lambda_i \neq \emptyset$ 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 matchings exist.

Algorithm 4 MatchWallaceTree(t)

```
Ensure: $t$ : a term in QF_BV
 1: if $t == (t_k \bullet ... \bullet t_1)$ then
 2:     $\Lambda := \lambda i.\emptyset$; $\Delta$ : vector of multiset of terms := $\lambda i.\emptyset$
 3:     for $i \in 1..k$ do
 4:         if $len(t_i) \neq 1$ then return $\emptyset$
 5:         $S := \{t_i\}$;$\Delta_i := \{t_i\}$
 6:         for $S \neq \emptyset$ 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 $\emptyset$
17:         if $\Delta_{i-1} \neq \{t_{i-1}\}$ then return $\emptyset$
18:     return getMultOperands($\Lambda$,1)
19: return $\emptyset$
```

3.3 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.

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

The sum output of a half/full adders are the result of xor operations of inputs. To find the input to the cascaded 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 4, 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 a concatenation of single bit terms t_k,..,t_1. Similar to Algorithm 2, we maintain the partial product store Λ. For each i, we also maintain the multiset of terms Δ_i that were used as inputs to the adders for the ith bit. In the loop at line 6, we traverse down the subterms until a subterm is not the result of a xor. In the traversal, we also collect the inputs of the visited 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.

Algorithm 5 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 $M := \text{MatchLong}(t) \cup \text{MatchWallaceTree}(t)$ then
4:         for each $x * y \in M$ do
5:             SMTSolver.add($[x * y = t]$)
6: return SMTSolver.checkSat()
```

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} then 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, we return $\varnothing$. After the loop at line 3, we have collected the partial products in Λ. At line 18, we call getMultOperands($\Lambda, 1$) to get all the matching multiplications.

3.4 Our solver

Using the above pattern matching algorithms, we modify an existing solver SMTSolver, which is presented in Algorithm 5. 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.

4 Correctness

We need to prove that each $[x * y = t]$ added in OurSolver is a tautology. 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.

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

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.

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

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 $m * n \in$ MatchLong(t), $[m * n = t]$ is a tautology.*

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. getMultOperands(Λ, w) returns the matches that produces the sums. Therefore, $[m * n = t]$ is a tautology.

Theorem 3 *If $m * n \in$ MatchWallaceTree(t), $[m * n = t]$ is a tautology.*

Proof. All we need to show that t is summing the partial products stored in Λ. The rest of the proof follows the previous theorem.

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 and collects the intermediate results of the sum in Δ_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 has the expected partial products of a Wallace tree.

5 Experiments

We have implemented[3] our algorithms as a part of Z3 SMT solver. 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.

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

Implementation We have added about 1500 lines of code in the bit vector rewrite module of Z3 because it allows an easy access to the abstract syntax tree of the input formula. We call this version of Z3 as instrumented-Z3. 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. For example, 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 versions of OurSolver by varying the choice of SMTSolver. We used Z3, Boolector, and CVC4 for the variations.

In each case we stop the instrumented-Z3 solver 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 pre-processed formula. The time taken to run our matching algorithms and generate the pre-processed formula is less than one second across all our benchmarks, and hence is not reported. We also experimented by running instrumented-Z3 standalone and found the run times to be similar to the above Z3 case; hence the run times for instrumented-Z3 are not reported. 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 the motivating example described in Section 1 involving long multiplication that was not solved by any of the solvers in 24 hours. This 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 setting them to zero. Our benchmarks were written in SystemVerilog and fed to STEWord [7], 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. All our benchmarks correspond to the system implementation satisfying the specified property: in other words, the generated SMT formulas were un-satisfiable. For satisfiable formulas the solver was able to find satisfying assignments relatively quickly, both with and without our heuristic. Hence, we do not report results on satisfiable formulas.

Results We compare our tool with Z3, Boolector and CVC4. In the Tables 1-2, we present the results of the experiments. We chose timeout to be 3600 seconds. In Table 1, we present the timings of the long multiplication and Wallace tree multiplier experiments. The first 13 rows correspond to the long multiplication experiments. The columns under SMTSolver are the run times

Table 1: Multiplication experiments. Times are in seconds. Portfolio column is the least timing among the solvers. Bold entries are the minimum time.

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

of the solvers to prove the satisfiability of the input benchmark. The solvers timeout on most of the benchmarks.

The next three columns present 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. OurSolver with Boolector and Z3 does not make the above inference, leading to more running times. 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 syntactic 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 tautology has no ‘ands’. The difference in the syntactic structure between the input formula and the added tautology makes it difficult for Boolector and CVC4 to make the above inference.

We have seen that the solvers fail to apply the word level reasoning after adding the tautologies. In the case, the solvers bit blast the formula and run a sat solver. In Table 2, we present the number of conflicts and decisions within the sat solvers. The number of conflicts and decisions on running OurSolver with the three solvers, are considerably less than their SMTSolver counterparts in the most of the cases. This demonstrates that the tautologies also help in reducing the search inside the sat solvers. OurSolver with CVC4 has zero conflicts and decisions for all the long multiplication experiments, because the word level reasoning solved the benchmarks. Similarly, OurSolver with Z3 has zero conflicts and decisions for all the Wallace tree multiplier experiments.

6 Related Work

The quest for heuristic strategies for improving the performance of SMT solvers dates back to the early days of SMT solving. An excellent exposition on several important early strategies can be found in [12]. The importance of orchestrating different heuristics in a problem-specific manner has been highlighted in [13], which also makes a strong case for developing languages that enables users to choose their preferred heuristics and tactics for specific problems.

The works that come closest to our work are those developed in the context of verifying hardware implementations of word-level arithmetic operations. There is a long history of heuristics for identifying bit-vector (or word-level) operators from gate-level implementations (see, for example, [19,20,22,23] for a small sampling). The use of canonical representations of bit-vector arithmetic operations have also been explored in the context of verifying arithmetic circuits like multipliers (see [28,29], among others). However, these representations usually scale poorly with the bit-width of the multiplier. Equivalence checkers determine if two circuits, possibly designed in different ways, implement the same overall functionality. State-of-the-art hardware equivalence checking tools, like Hector [30], make use of sophisticated heuristics like structural similarities between sub-circuits, complex rewrite rules and heuristic sequencing of reasoning engines to detect equivalences between two versions of a circuit. The rewrite rules used in Hector [21] can detect different configurations of circuits implementing an arithmetic operator and replace them by the corresponding word-level operator. Since these efforts are primarily targeted at establishing the functional equivalence of one circuit with another, replacing one circuit configuration with another often works profitably. However, as argued in Section 1, this is not always desirable when checking the satisfiability of a formula obtained from word-level BMC or word-level STE. Hence, our approach differs from the use of rewrites used in hardware equivalence checkers, although there are close parallels between the two approaches.

Table 2: Conflicts and decisions in the experiments. M stands for millions. k stands for thousands.

	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

It is interesting to note that alternative representations of arithmetic operators are internally used in SMT solvers when bit-blasting high-level arithmetic operators. However, since an operator may be implemented in multiple ways, each solver chooses one (or a few) ways of bit-blasting an operator. For example, Z3 [14] uses a specific Wallace-tree implementation of multiplication when blasting multiplication operations. Since a wide multiplication operator admits multiple Wallace-tree implementation, this may not match terms encoding the Wallace-tree implementation of the same operator in another part of the formula. Similar heuristics for bit-blasting arithmetic operators are also used in other solvers like Boolector [2] and CVC4 [15]. However, none of these are intended to help improve the performance (run-time) of the solver when reasoning about bit-vector formulas. Instead, they are used to shift the granularity of reasoning from word-level to bit-level for the sake of completeness, but often at the price of performance.

7 Conclusion and future work

We have shown how adding tautological assertions that assert the equivalence of different representations of bit-vector multiplication can siginificantly improve the performance of SMT solvers. We are currently extending our procedure to support Booth multiplier and other more complex arithmetic patterns. We are also working to add proof generation support for the added tautological assertions. We could not include proof generation in this work, since the basic infrastructure of proof generation is missing in Z3 bit-vector rewriter module.

References

1. Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.
2. Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 174–177, 2009.
3. EBMC. `http://www.cprover.org/ebmc/`.
4. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.
5. Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, pages 427–443, 2012.
6. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, 2005.

7. Supratik Chakraborty, Zurab Khasidashvili, Carl-Johan H. Seger, Rajkumar Gajavelly, Tanmay Haldankar, Dinesh Chhatani, and Rakesh Mistry. Word-level symbolic trajectory evaluation. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 128–143, 2015.
8. Y. Naveh and R. Emek. Random stimuli generation for functional hardware verification as a cp application. In *Proc. of CP*, pages 882–882, 2005.
9. Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. Marcus, and G. Shurek. Constraint-based random stimuli generation for hardware verification. In *Proc. of AAAI*, pages 1720–1727, 2006.
10. Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.
11. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272, 2005.
12. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.
13. Leonardo de Moura and Grant Olney Passmore. *The Strategy Challenge in SMT Solving*, pages 15–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
14. Leonardo de Moura and Nikolaj Bjrner. Z3: An efficient smt solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.
15. Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.
16. Long multiplication. `https://en.wikipedia.org/wiki/Multiplication_algorithm#Long_multiplication`.
17. Booth's multiplication algorithm. `https://en.wikipedia.org/wiki/Booth's_multiplication_algorithm`.
18. Christopher S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Electronic Computers*, 13(1):14–17, 1964.
19. Dominik Stoffel and Wolfgang Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 23(5):586–597, 2004.
20. Cunxi Yu and Maciej J. Ciesielski. Automatic word-level abstraction of datapath. In *IEEE International Symposium on Circuits and Systems, ISCAS 2016, Montréal, QC, Canada, May 22-25, 2016*, pages 1718–1721, 2016.
21. Alfred Kölbl, Reily Jacoby, Himanshu Jain, and Carl Pixley. Solver technology for system-level to RTL equivalence checking. In *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, pages 196–201, 2009.
22. Pramod Subramanyan, Nestan Tsiskaridze, Kanika Pasricha, Dillon Reisman, Adriana Susnea, and Sharad Malik. Reverse engineering digital circuits using functional analysis. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, pages 1277–1280, 2013.

23. Wenchao Li, Adria Gascón, Pramod Subramanyan, Wei Yang Tan, Ashish Tiwari, Sharad Malik, Natarajan Shankar, and Sanjit A. Seshia. Wordrev: Finding word-level structures in a sea of bit-level gates. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust, HOST 2013, Austin, TX, USA, June 2-3, 2013*, pages 67–74, 2013.
24. Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer Publishing Company, Incorporated, 1 edition, 2008.
25. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.
26. João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
27. Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island.*, pages 203–208, 1997.
28. Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *DAC*, pages 535–541, 1995.
29. Amr A. R. Sayed-Ahmed, Daniel Große, Ulrich Kühne, Mathias Soeken, and Rolf Drechsler. Formal verification of integer multipliers by combining gröbner basis with logic reduction. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, pages 1048–1053, 2016.
30. HECTOR. `http://www.synopsys.com/Tools/Verification/FunctionalVerification/Pages/hector.aspx`.