Matching multiplications in Bit-Vector formulas

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

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

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 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] for an excellent exposition). 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 strategy yields significant performance benefits for three state-of-the-art SMT solvers, namely Z3 [13], CVC4 [14] and

BOOLECTOR [2]. Our experiments demonstrate that our heuristc can achieve reduction in solving times for multiple examples, by upto several orders of magnitude, especially when the input formula is unsatisfiable.

Our primary motivation 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 implementng 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 cleverly in embedded systems, using a composition of several smaller and well-characterized blocks. For example, a 128-bit multiplier may be implemented using one of several multiplication algorithms, viz. long multiplication [15], Booth-encoded multiplication [16] or Wallace-tree multiplication [17], 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 shows up conspicuously 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 serial multiplier circuit, originally used as a benchmark in [7]. The circuit reads in two 32-bit operands sequentially 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 $a * b$, where "*" denotes 32-bit multiplication. The RTL design (i.e. system implementation), as used in [7], makes use of the "*" operator in System-Verilog, effectively specifying a 32-bit multiplication operation directly. The SMT formula resuting from a WSTE run on this example therefore contains terms with only 32-bit multiplication operators, and no terms encoding a lower-level implementation of the multiplication operator. 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 systems; hence unsatisfiability of the formula implies the absence of any bounded violating runs.

We now change the RTL in the above example to represent the implementation of 32-bit multiplication by the long-multiplication algorithm, where each 32-bit operand is partitioned into 8-bit blocks. The corresponding WSTE run now yields an SMT formula that contains terms with 32-bit multiplication operators (derived from the property being checked), and also terms that encode the implementation of 32-bit multiplication using the long-multiplication algorithm. Interestingly, 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 all the three 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 spectacular performance degradation.

The above example demonstrates that the problem of not being able to identify equivalence of alternative representations of arithmetic operators plagues multiple state-of-the-art SMT solvers. Therefore, a heuristic that is generic (not solver-specific) would be highly desirable. This motivates us to ask: *Can we pre-process an SMT formula with 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 all solvers will benefit from our heuristic on all problem instances. For example, the problem described in the motivating example is shown to be unsatisfiable by Z3 in 0.073s and by CVC4 in 0.017s, after application of our heuristic. Interestingly, Boolector didn't benefit from our heuristic in this case, although there are other examples where Boolector benefits significantly, as shown in Section 5. This is not surprising, since it is well-known that a heuristic that works well on one problem instance may not work well for another, even for the same solver [12].

At first sight, identifying equivalence between terms encoding alternative representations of arithmetic operators appears to be one of reverse-engineering an implementation of the operator. Indeed, this reverse-engineering problem has been addressed to varying extents by earlier researchers in different contexts [18]. A closer inspection, however, reveals that there are subtle complications that arise if we simply replace a term representing an arithmetic operation at one level by another term representing the same operation at a different level. First, and contrary to intuition, the same decomposed implementation may correspond to multiple reverse-engineered bit-vector operations. 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 improved performance when checking the (un)satisfiability check of the overall SMT formula. This can happen for various 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.
– 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 (un)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.

Thus, naively replacing a pattern corresponding to a decomposed implementation with the bit-level operator is unlikely to help SMT solving in any but the simplest of cases.

In this paper, we present a pre-processing heuristic 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 rewriting the matched sub-terms directly, as in earlier work, we conjoin the formula φ with additional assertions that equate a matched sub-term with the corresponding bit-vector operator. Since no re-writes are done, this allows us to express multiple equivalences in a convenient way. Our experiments show that the added assertions succeed in preventing bit-blasting in several cases, while in other cases they significantly help in pruning the search space even after bit-blasting. Both benefits 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 SMT solver, and can be combined with multiple solvers. Indeed. our experiments show that the performance of different SMT solvers on the pre-processed formula can vary. Hence, we propose a portfolio approach to solving the pre-processed formulas, and 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 will present the syntax of the theory of quantifier-free fixed-width bit-vector formulas(`QF_BV`), the basics of solving methods used by SMT solvers, and the multiplication methods that are of our interest.

2.1 `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.

$$t ::= t * t \mid t + t \mid x \mid n^w \mid t \bullet t ....$$
$$F ::= t = t \mid t \bowtie t \mid \neg F \mid F \vee F \mid F \wedge F \mid F \oplus F \mid ...$$

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.

2.2 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)[19,20] based procedures. Some of the leading SMT solvers are Z3[13], Boolector[2], and CVC4[14].

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.

2.3 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. 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.

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

$x_i * y_j$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.

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

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.

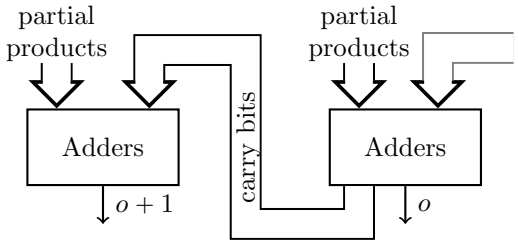
$$(0^4 \bullet v_1 * u_2 \bullet v_1 * u_1) + (v_2 * u_2 \bullet v_2 * u_1 \bullet 0^4) + (0^2 \bullet v_2 * v_2 \bullet v_1 * v_2 \bullet 0^2)$$

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.

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

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 the introduction.

Wallace tree multiplier[17] 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.

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 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

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 $len(a) == len(b) == w$
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 $len(a) == len(b) == w$ **then**
8: $\Lambda_o.insert(a * b)$
9: **else return** $\varnothing$
10: **return** getMultOperands(Λ,w)
11: **return** $\varnothing$

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.

3.2 Partial products to operands

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. 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 $backtrack_h$, 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 $\varnothing$. 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 $backtrack_i$.

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

Algorithm 2 GETMULTOPERANDS(Λ, w)

Require: Λ : 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: $\quad x_h := a; y_h := b;$ $backtrack_h :=$ **ff**;
5: **else**
6: $\quad$ **return** $\varnothing$
7: $i := h; l_x := h; l_y = h;$
8: **while** $i > 1$ **do**
9: $\quad i := i - 1; C := \Lambda_i$
10: $\quad$ **for** $j \in (h-1)..(i+1)$ **do**
11: $\quad\quad$ **if** $x_j \neq 0$ and $y_{h+i-j} \neq 0$ **then**
12: $\quad\quad\quad$ **if** $x_j * x_{h+i-j} \notin C$ **then goto** BACKTRACK
13: $\quad\quad\quad C := C - \{x_j * x_{h+i-j}\}$
14: $\quad$ **match** C **with**
15: $\quad\quad | \{x_h * b, y_h * d\} \rightarrow x_i := d; y_i := b;$ $backtrack_i := (x_h == y_h);$
16: $\quad\quad | \{x_h * y_h\} \rightarrow x_i := 0; y_i := x_h;$ $backtrack_i :=$ **tt**;
17: $\quad\quad | \{x_h * b\} \rightarrow x_i := 0; y_i := b;$ $backtrack_i := (x_h == y_h);$
18: $\quad\quad | \{y_h * b\} \rightarrow x_i := b; y_i := 0;$ $backtrack_i :=$ **ff**;
19: $\quad\quad | \{\} \rightarrow x_i := 0; y_i := 0;$ $backtrack_i :=$ **ff**;
20: $\quad\quad | \_ \rightarrow$ **goto** BACKTRACK;
21: $\quad$ **if** $x_i \neq 0$ **then** $l_x = i$
22: $\quad$ **if** $y_i \neq 0$ **then** $l_y = i$
23: $\quad$ **if** $h - l_x - l_y < 1$ **then goto** BACKTRACK;
24: $\quad$ **if** $i == 1$ **then**
25: $\quad\quad$ **for** $o \in 0..(l-1)$ **do**
26: $\quad\quad\quad x' :=$ Right shift x until o trailing 0 blocks in x
27: $\quad\quad\quad y' :=$ Right shift y until $l - o$ trailing 0 blocks in y
28: $\quad\quad\quad M := M \cup \{x' * y'\}$
29: $\quad$ **else**
30: $\quad\quad$ **continue;**
31: $\quad$ BACKTRACK:
32: $\quad\quad$ Choose smallest $i' \in h..(i+1)$ such that $backtrack_{i'} ==$ **tt**
33: $\quad\quad$ **if** no i' found **then return** M
34: $\quad\quad i := i'$; SWAP(x_i, y_i); $backtrack_i :=$ **ff**

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 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 $backtrack_i$ 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 $\mathtt{tt}$ 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 $\Lambda_i \neq \varnothing$ 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 matchings exist.

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.

$$sumHalf(a,b) = a \oplus b \qquad sumFull(a,b,c) = a \oplus b \oplus c$$
$$carryHalf(a,b) = a \wedge b \qquad carryFull(a,b,c) = (a \wedge b) \vee (b \wedge c) \vee (c \wedge a)$$

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 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 a concatenation of single bit terms t_k,..,t_1. 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 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

Algorithm 3 MATCHWALLACETREE(t)

Ensure: t : a term in `QF_BV`

```
1: if t == (t_k • ... • t_1) then
2:     Λ := λi.∅; Δ : vector of multiset of terms := λi.∅
3:     for i ∈ 1..k do
4:         if len(t_i) ≠ 1 then return ∅
5:         S := {t_i};Δ_i := {t_i}
6:         for S ≠ ∅ do
7:             t ∈ S; S := S − {t}
8:             if t == s_1 ⊕ .... ⊕ s_p then
9:                 S := S∪{s_1, .., s_p};Δ_i := Δ_i∪{s_1, .., s_p}
10:            else if t == carryFull(a, b, c) and a, b, c, a ⊕ b, a ⊕ b ⊕ c ∈ Δ_{i−1} then
11:                Δ_{i−1} := Δ_{i−1} − {a, b, c, a ⊕ b}
12:            else if t == carryHalf(a, b) and a, b, a ⊕ b ∈ Δ_{i−1} then
13:                Δ_{i−1} := Δ_{i−1} − {a, b}
14:            else if t == a ∧ b then
15:                Λ_i.insert(a ∗ b);
16:            else return ∅
17:        if Δ_{i−1} ≠ {t_{i−1}} then return ∅
18:    return GETMULTOPERANDS(Λ,1)
19: return ∅
```

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} 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 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

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** $M :=$ MATCHLONG(t) $\cup$ MATCHWALLACETREE(t) **then**
4: **for** each $x * y \in M$ **do**
5: SMTSOLVER.add($[x * y = t]$)
6: **return** SMTSOLVER.checkSat()

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.

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 newly generated formula. The time taken to run our matching algorithms and generate the new 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).

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

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 [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.

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 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

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

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. The list of papers that describe heuristics for SMT solving is a long and illustrious one. Instead of citing these individual papers, we point the reader to an excellent exposition on this topic in [12], which also makes a strong case for developing languages that enables users to choose their preferred heuristics and tactics in SMT solvers.

The works that come closest to our work are those developed in the context of verifying hardware implementations of word-level arithmetic. Heuristics

for identifying bit-vector (or word-level) operators from gate-level implementations of these operators have been developed and optimized by the hardware verification community for long [21,22,23,18]. The use of canonical representations of bit-vector arithmetic operations have also been explored, as in [24,25], among others. These efforts were primarily intended to simplify word-level reasoning about circuits, and to verify that implementations of complex arithmetic operators like multiplication indeed implement the semantics of multiplication. None of these works were, however, really targeted at developing SMT solving heuristics for the theory of fixed-width bit-vectors.

The use of alternative representations of arithmetic operators has typically been used in SMT solvers when bit-blasting high-level operators. For example, Z3 [13] uses a specific Wallace-tree implemention of multiplication when bit-blasting multiplication operations. Similar heuristics are also followed in other solvers like BOOLECTOR [2] and CVC4 [14]. However, none of these are intended to help improve the performance (run-time) of the solver when reasoning about bit-vector formulas.

7 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 tautologies. 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 plan to extend our work to problems involving different combinations of multiplications. For example, $(X * (Y * Z)) * W$ involves three multiplications. We could chose to specify the implementation of the three multiplications to be either of Long, Wallace, Booth multiplication or an unspecified multiplication. To enable the pattern detection in arbitrary arithmetic formulas, we are working to modify the word level reasoning in the solvers.

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. Daniel Kroening. 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. Leonardo de Moura and Grant Olney Passmore. *The Strategy Challenge in SMT Solving*, pages 15–44. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
13. 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.
14. 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.
15. Long multiplication.
16. Booth's multiplication algorithm.
17. Christopher S. Wallace. A suggestion for a fast multiplier. *IEEE Trans. Electronic Computers*, 13(1):14–17, 1964.
18. 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.

19. João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.
20. 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.
21. 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.
22. 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.
23. 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.
24. Randal E. Bryant and Yirng-An Chen. Verification of arithmetic circuits with binary moment diagrams. In *DAC*, pages 535–541, 1995.
25. 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.

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