Project Report: Decision Procedure for Theory of Quantifier-Free Bit Vectors

Abhishek Nair

Stanford University, Stanford CA 94305, USA aanair@stanford.edu

Keywords: Bit Vectors · Bit Blasting · Flattening · Incremental Bit Blasting

Abstract. In this report, we present an implementation of a decision procedure for the theory of Quantifier-free Bit-Vectors. We implement the standard Bit-Blasting algorithm along with incremental Bit-Blasting, an optimization to improve performance in the presence of expensive operations. We demonstrate results that show that incremental Bit-Blasting can significantly improve performance on some benchmarks.

1 Introduction

In recent years, there have been great theoretical and practical improvements in the field of *satisfiability modulo theories* (SMT). These advances have resulted in extensive use of SMT solvers for program analysis and verification.

Amongst the many theories supported by SMT solvers, the theory of Bit-Vectors is especially important for the analysis and verification of computer systems. This is because Bit-Vectors can naturally model computer programs which typically use bounded size data structures to store program state and data. In particular, using Bit-Vectors instead of integers to model computer programs enables precisely capturing behaviours such as arithmetic overflow or underflow - a common corner case in program analysis.

Several important verification problems can be modeled even without the use of quantifiers [2]. The current state-of-the-art technique for solving quantifier-free Bit-Vector SMT instances is to encode the Bit-Vector formula into propositional form and then use a SAT solver to determine satisfiability. Solvers implement a technique called Bit-Vector Flattening or Bit-Blasting to encode a Bit-Vector formula into propositional form. one example of a state-on-the-art solver that implements this technique is Boolector [3].

In this report, we describe an implementation of the Bit-Blasting procedure for quantifier-free Bit-Vector formulae. In addition to Bit-Blasting, an optimization called incremental Bit-Blasting is also described and implemented. We demonstrate that Bit Blasting performs poorly on certain arithmetic operations (Multiplication) and show how the aforementioned optimization can potentially improve the performance of the solver on formulae containing these arithmetic operations.

2 Background and Motivation

In order to motivate the need for a theory of Bit-Vectors [1], consider the following formula $(x - y > 0) \iff (x > y)$. This statement is self-evident when x, y are interpreted as integers. However, when x, y are interpreted as fixed size Bit-Vectors, this statement is no longer true because of possible overflow of the subtraction operation. The reason for this apparent discrepancy is that arithmetic operations such as addition and subtraction are defined modulo a certain constant unlike in the case of integers. The constant depends on the width of the operands and is typically 2^{n-1} where n is the width of the operands. Since computer systems represent all data using fixed size data elements, it is very useful to model them using the theory of Bit-Vectors. We now provide a brief overview on Bit-Blasting and incremental Bit-Blasting.

2.1 Bit-Blasting

To describe the Bit-Blasting procedure we use the definitions of `terms` and `atoms` as defined in [1]. At a high level, the Bit-Blasting procedure works by computing an equisatisfiable propositional formula and then relying on a SAT solver to produce a model or prove unsatisfiability. To produce the equisatisfiable propositional formula, the first step is to create a Boolean skeleton of the Bit-Vector formula. This is done by replacing each `atom` in the formula by a propositional variable. Next, each Bit-Vector `term` in the formula is assigned l propositional variables where l is the width of the term. Finally, constraints are derived for each atom and term based on their operators.

2.2 Incremental Bit-Blasting

The constraints derived for some operators can be quite expensive. For instance, a multiplication operator on operands of size l introduces $O(l^2)$ clauses to the resulting propositional encoding. Moreover, the symmetry and connectivity of the introduced clauses is often a burden on the decision heuristics of modern SAT solvers [1]. One method of speeding up the decision procedure is to simply not add the constraints related to expensive operations such as multiplication. The resulting propositional encoding represents an abstraction on the actual formula. If the abstracted encoding is unsatisfiable, then the original formula is also unsatisfiable (adding more constraints can never convert an unsatisfiable instance into a satisfiable instance). However, if the encoding is satisfiable we need to apply the model to the remaining constraints (some propositional variables may not appear in the model, these can be assigned a default value). If the model satisfies the abstracted away constraints, then the original formula is satisfiable. However, if some constraints are not satisfied, then we need to refine the abstraction by adding some of the previously unsatisfied constraints to the propositional encoding.

3 Decision Procedure: Flattening (Bit-Blasting)

My implementation closely follows the algorithm specified in Section 2. However, for ease of implementation, I do not treat terms and atoms as separate entities. Rather atoms are simply a special case of a term when the operator is a relational operator.

My implementation supports 24 operators including bitwise operators, logical operators, bit slicing operators and relational operators. Unfortunately, I did not have the bandwidth to implement the shift operator (logical and arithmetic) or other variants of operators that operate on more than two operands (e.g. multiple Boolean variable "and"). Since I am targeting quantifier-free logic, the keywords `forall` and `exists` are also not supported. Finally, I also do not support the `let` keyword (defined in [4]).

My code broadly consists of only two functions. The first function `preprocess` simply reads in the Bit-Vector formula and creates a list where each element corresponds to one assert statement. This function also returns a list of all the operators used in the Bit-Vector formula.

The second function called `process`, reads in the generated list and parses it to create an abstract data-type "Node" for each operator. The Node object stores information about the operator, the propositional variables corresponding to each operand and the width of the operands. The Node object also has a built-in method that will generate the constraints for that operator. The constraints for the different operators have been manually written in the code, in CNF form. Therefore, whenever the process function creates a new Node, it calls the generate_constraints() function and consequently the constraints for that node are added to a global list. Finally after the entire formula has been processed, this global list of constraints is written out into a file in the dimacs format. Then we call the `cadical` [5] SAT solver to either generate a model or prove unsatisfiability.

4 Optimization: Incremental Bit-Blasting

In order to implement the optimization described in section 2, we make a straightforward change to the code described above. In essence, when a new node is generated we check if the operation for that node is an expensive operation (only the multiplication operation is expensive). If it is, then instead of adding the constraint to the global list, we add the constraint to a separate list. Now we use the constraints generated by the inexpensive operations to test for satisfiability. If `cadical` returns an unsatisfiable result, we are done. If it returns a satisfiable result, then we check whether the model also satisfies the constraints in the separate list that we created. If it does, then the original instance is satisfiable. If it does not satisfy the separate constraints, then we add a few (hard-coded to 5) constraints from the separate list to the global list and regenerate the dimacs file and retest for satisfiablity.

5 Evaluation Methodology

We have tested our implementation on a total of 14 benchmarks. These benchmarks were selected from the `SMT-LIB` github repository [6] [7]. In 9 of the 14 benchmarks, there are a significant number of multiplication operations and the formula is unsatisfiable even when these multiplication operations are abstracted away, making them perfect to test the effectiveness of the incremental bit-blasting optimization. In the results section below, we report the total number of clauses in the final propositional formula for each benchmark and the time taken for `cadical` to solve the generated cnf. These numerics are reported both when optimization is turned on and turned off.

6 Results

The table below shows the number of propositional clauses and the running time for the benchmarks mentioned in the previous section.

Benchmark	No Optimization		Optimized	
	Number of clauses	Running Time (ms)	Number of Clauses	Running Time (ms)
cvs_vc71402	7822	70	7822	64
cvs_vc81758	13466	84	13466	81
cvs_vc81759	16704	124	16704	119
cvs_vc81760	16704	105	16704	108
cvs_vc81761	13476	119	13476	112
mult_ub_32x32	989590	4457	223718	1354
mult_ub_48x48	2957134	13077	50317	3395
mult_ub_64x64	6556422	29785	894374	6678
commute32	632415	2614	4575	264
commute48	1418591	6063	6815	542
commute64	2518111	11133	9055	925
distrib32	221457	967	5397	187
distrib48	488737	2056	8005	264
distrib64	860465	3634	10613	363

In the table above, there are two different classes of benchmarks. The first 5 benchmarks (named cvs*) do not contain any mutliplication operations. For these benchmarks, the running time with and without optimization is almost exactly the same (the small differences might be due to machine load and other random factors). However in the remaining 9 benchmarks, there are a significant number of multiplication operations and not all of these operations contribute towards determining the unsatisfiability of the formula. Therefore, performing incremental optimization leads to huge benefits (upto 10x for some benchmarks). These examples clearly indicate the usefulness of incremental optimization as an optimization strategy for solvers targeting the theory of Bit-Vectors.

7 Conclusions

The theory of Bit-Vectors is a very important theory in the domain of computer system analysis and verification. The state of the art decision procedure is Bit-Blasting. Our code implements Bit-Blasting and in addition also implements an optimization called incremental Bit-Blasting that may improve performance in the presence of expensive operations. Our results also show the effectiveness of the implemented optimization. An interesting future direction would be to analyze the drawbacks of incremental Bit-Blasting. In particular, it would be interesting to see how much of a performance impact incremental Bit-Blasting would cause if all the multiplication operations in the formula were required to prove unsatisfiability.

References

1. Kroening, Daniel and Strichman, Ofer: Decision Procedures. In: Springer, 2016
2. Jonas, Martin: SMT Solving for the Theory of Bit Vectors, September 2016
3. Niemetz, Aina and Preiner, Mathias and Biere, Armin: Boolector 2.0. In: Journal of Satisfiability Boolean Modeling and Computation
4. Barrett, Clark and Fontaine, Pascal and Tinelli, Cesare: The SMT-LIB Standard: Version 2.6. Available at: www.SMT-LIB.org
5. Biere, Armin and Fazekas, Katelin and Fleury, Mathias and Heisinger, Max: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In: Proc. of SAT Competition 2020
6. Benchmark Name: Wienand-2008, Available at: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/tree/master/wienand-cav2008, Accessed on: 1st December 2022
7. Benchmarks Name: CVS, Generated by Calysto static checker. Available at: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/tree/master/spear/cvs_v1.11.22. Accessed on: 1st December 2022