CS 257 Project - Mixing Integer Linear Programming and Bit-Blasting to Decide Bitvectors

Alex Lin, Jamie Song

December 2022

1 Abstract

Our project explores SMT solver implementations and optimizations for the theory of bit-vectors, which is useful in various industries such as modeling modern computer arithmetic datapaths to formally verify ASICs. We implemented both a lazy DPLL(T) SMT solver with a bit-blasting theory solver, and an optimization which models bit-vector constraints as mixed integer linear programming (MILP [3]) constraints [1][2], using Python as the programming language. We found that the optimization technique using MILP can be orders of magnitude more performant than state-of-the-art solvers, such as z3 [4], in arithmetic heavy workloads where arithmetic does not overflow.

2 Introduction

We implemented a decision procedure and optimization for the theory of bit-vectors. The baseline decision procedure is a lazy DPLL(T) SMT solver with a bit-vector theory solver using the bit-blasting technique (BB approach). The optimization technique (LP approach) is an optimization to the bit-vector theory solver aimed to take advantage of word-level reasoning. The LP approach solver will encode constraints into linear integer arithmetic constraints and use mixed integer linear programming to solve the constraints.

3 Motivation and Background

The theory of bit-vectors are particularly useful for modeling computer arithmetic due to the use of finite bit-length data types in computing, such as int, or uint_64. A great example of this is the practice of formal verification in the ASIC industry to verify digital logic circuits.

We define the theory of bit-vectors (with equality) we will be deciding. $\Sigma^S = \{W_n\}$ for all $n > 0$. $\Sigma^F = \{\&, |, \neg, \oplus, <<, >>, +, -, *, \text{Extract}, \text{Concat}, \leq, \geq, <, >\}$ and all possible bit-vector constants for any bit width. S is the class of structures which interpret W_n as a bit-vector of width n, and each function in the usual way for bit-vectors.

The motivation for a lazy approach as opposed to eager is the ability to better take advantage of word-level reasoning when using the LP approach, which is useful for modeling arithmetic-heavy RTL datapath calculations. The goal is to optimize bitvector arithmetic, which the bit-blasting technique often struggles with due to the need to encode complex adders and multipliers on the bit-level. By representing variables as word-level integers, we can effectively encode linear bitvector arithmetic as constant coefficients in LP constraints.

4 Overview of Approach

4.1 SAT Solver and DPLL(T)

To implement the SAT Solver, we divide it into two parts: Tseitin Transformation and the solver given inputs in conjunctive normal form (CNF).

4.1.1 Tseitin Tranformation

For the Tseitin Transformation part, we restrict the possible formulas to consist of variables, constants, NOTs, ANDs, and ORs for the sake of the implementation since details for transforming wffs into this form were presented in lecture. For example, implication, $A \rightarrow B$ can be converted into $\neg A \vee B$.

Following the scheme of the original Tseitin Transformation, we're transforming a variable or a constant to itself, and introducing a new auxiliary variable in the other three cases:

- NOT: $\neg B$ will be converted into CNF clauses $(A \vee B) \wedge (\neg A \vee \neg B)$ with auxiliary variable A.
- AND: $p \wedge q$ will be converted into CNF clauses $(A \vee \neg p \vee \neg q) \wedge (p \vee \neg A) \wedge (q \vee \neg A)$ with auxiliary variable A.
- OR: $p \vee q$ will be converted into CNF clauses $(\neg A \vee p \vee q) \wedge (\neg p \vee A) \wedge (\neg q \vee A)$ with auxiliary variable A.

After the Tseitin Transformation, the equisatisfiale CNF formula is generated, and can be decided by the SAT solver.

One thing to keep track of is to store all of the variables in the original formula. Even though the formula with a lot of newly introduced auxiliary variables are inputs to the SAT solver, we need track the original variables to send back the assignment to the Theory Solver.

4.1.2 DPLL

To implement the SAT solver, we use the DPLL algorithm described in the lecture, with some analysis of the implication graph to help identify the correct literal for backjump.

If no variables are assigned, the first step is to perform a unit propagation. If assigning boolean value b to a literal a value will directly result in falsifying a clause, then the literal should be be assigned to $\neg b$ in the current decision level without a decision point.

Afterwards, if there is a conflict, then perform the Explain step, which is the conflict analysis, to find the cause of the conflict and best level to backjump to. If there's no more decision point, then it means there's no where to backjump to, which means the original formula is not satisfiable.

If all variables are assigned at any point of the progress, directly return the assignment.

If no actions could be done, then increment the decision level and choose the next unassigned variable to decide, and update the implication graph by the newly assigned boolean value.

4.2 Bit-blast Theory Solver

To implement bit-blasting, consider the abstract syntax tree for a bitvector literal. For each function node, we introduce N new propositional variables for that node where N is the bit-width of that node as dictated by its sort. We constrain the new propositional variables using the semantic interpretation of the function and with the children of the node as inputs. For example, for 1-bit variables A, B, and the literal A & B, we introduce C where $C \leftrightarrow A \wedge B$. The semantic interpretation of all functions can be modeled as a digital logic circuit with AND, OR, and NOT gates, which can readily be transformed into propositional constraints. Details of implementations of these circuits can be found in any digital logic textbooks. In our implementation, we chose to use a ripple-carry adder and a dadda multiplier due to ease of implementation.

After accumulating propositional variables and constraints, we have an equi-satisfiable propositional formula which can be decided by a SAT solver.

4.3 MILP Theory Solver

For the LP approach optimization, we support a subset of functions: $\{\&, |, \neg, \oplus, +, -, *, \leq, \geq, <, >\}$.

To represent a bitvector literal as integer linear programming constraints, consider, again, the abstract syntax tree for the literal. For each function node, we introduce an integer variable for that node. Next, we describe how to encode each function as ILP constraints. Let A and B be N-bit bitvectors which can be constrained with $0 \leq A, B \leq 2^N - 1$.

Consider $A + B$. Introduce C where $C = A + B$. It is possible for the addition to overflow, so we also introduce a modulus variable, k, where $0 \leq k \leq 1$ and $C + k * 2^N = A + B$. The ILP constraint is $A+B \leq C+k*2^N$ and $C+k*2^N \leq A+B$. Subtraction, $A-B$, can be implemented similarly.

Consider $A = B$, $A \leq B$ and $A \geq B$. $A = B$ is equivalent to the conjunction of $A \leq B$ and $B \leq A$, all of which are already ILP constraints.

Consider $A*B$. It is possible for one multiplicand to be a constant bitvector, or both to be variables. First, let's consider the one constant case. Assume B is the constant. Introduce C as output and k for overflow, where $C+k*2^N = A*B$ and $0 \leq k \leq 2^N-1$. Because B is a constant, it can be represented as the coefficient of A, so we can directly encode this as ILP constraints: $C + k * 2^N \leq A * B$ and $A * B \leq C + k * 2^N$.

Next, let's consider $A * B$ where both A and B are variables. Let $C = A * B$ Because $A * B$ is not linear, to encode this as ILP constraints, we have to introduce the idea of bit-blasting word-level integers. We introduce N variables for the bits of A, $A_0, A_1, ..., A_N$. $A = \Sigma_{i=0}^{N} 2^i * A_i, 0 \leq A_i \leq 1$. These are literal constraints on A and A_i, which can be represented as ILP constraints. We also introduce N variables for the partial products which construct C, $P_0, P_1, ..., P_N$, $C = \Sigma_{i=0}^{N} 2^i * P_i$. We want to encode: if $A_i = 1$ then $P_i = B$ else $P_i = 0$. This can be done with the following ILP constraints [1].

$$L = 2^N - 1 \tag{1}$$

$$P_i - L * A_i \leq 0 \tag{2}$$

$$P_i - B + L * (1 - A_i) \geq 0 \tag{3}$$

$$0 \leq P_i \leq B \tag{4}$$

Consider $A\&B$. Let $C = A\&B$. We can bit-blast A, B, and C, then encode the bitwise semantics of & for each triplet of bits between A_i, B_i, and C_i, with the following ILP constraints [1]: $C_i \leq A_i$, $C_i \leq B_i$, $C_i \geq A_i + B_i + 1$, $0 \leq C_i \leq 1$.

Consider $C = \neg A$. We can bit-blast A, and C, and use the following constraints: $C_i = 1 - A_i$. All other bitwise operators can be constructed from & and $\neg$.

The process of constructing all the ILP constraints given the abstract syntax tree of a bitvector literal will be traversing each node in a bottom up manner, and introducing new integer variables to represent that node and accumulate the conjunction of ILP constraints generated by the current node's functions using the node's immediate children as inputs and the constraints presented above. We used the MILP solver by HiGHS to solve integer linear programming [3].

5 Evaluation Methodology

5.1 Baseline - Bit-blast Approach

We compared the SAT Solver in the Baseline Bit-blasting approach with the state-of-the-art z3 SAT solver [4], using the same bit-blasting algorithm.

The test cases we consider are: AND, OR, Logical Shift Left, Logical Shift Right, Subtract, Simple Addition, Simple Multiplication, and Equality. AND / OR is solving the result after anding / oring two bitvectors with the same width. Shift is solving the result of a bitvector with width n left/right shift by k bits. It is the same as multiplying / integer dividing by 2^k. Subtraction / Addition with Equality is to solve the linear equation $C = A \pm B$ where A, B, C have the same width. Simple multiplication is the solve the equation $C = A * B$ where A, B have width m, n, respectively, and C has width mn, assuming without overflow.

Each test is randomly generated each run, and the results for 10 runs were averaged for each approach. The CPU used was a 2.3 GHz Dual-Core Intel Core i5 processor.

5.2 Optimization - LP Approach

We compared the LP approach against the baseline bit-blast approach and the state-of-the-art z3 solver. For the purposes of this evaluation, we assume the function * does not overflow. This is because not doing so would explode the ILP search space due to having to search for the residual value. This is a reasonable assumption because in most RTL datapath workloads, we do not want arithmetic operations to overflow the bitvector's width.

The test cases we consider are: square unsat, square sat, quadratic, mult unsat, mult sat, and matmul 2 through 19. Square is solving $C^2 = A^2 + B^2$ where A and B are fixed 24-bit bitvectors and Z is solved for; sat is when Z exists as an integer. Quadratic is solving $X^2 - (A+B) * X + A * B = 0$ for fixed A and B which are 16-bit bitvectors. Mult is solving $A * B = C$ where C is a fixed 22-bit bitvector with $A, B \geq 2$; sat is when C is not prime. Finally, matmul N is solving matrix multiplication $Ax = B$ where A is an NxN invertible matrix, B is a Nx1 matrix and each matrix element is of bitvector width 32.

Each test case is randomly generated each run, and the results for 10 runs were averaged for each approach.

6 Results

6.1 Baseline - Bit-blast Approach

Test Case	# of clauses (baseline)	Baseline Time (ms)	z3 SAT Solver Time (ms)
AND	104	4.499427	26.4442111
OR	115	2.736399	15.262957
Shift Left	275	19.8625863	20.3015207
Shift Right	275	24.4246003	20.5512976
Simple Subtract	1127	145.0382498	41.0259654
Simple Addition	751	92.7529576	30.7519252
Simple Multiplication 1	126779	241013.644530	2890.64453
Simple Multiplication 2	76838	130829.1454413	2267.4436499

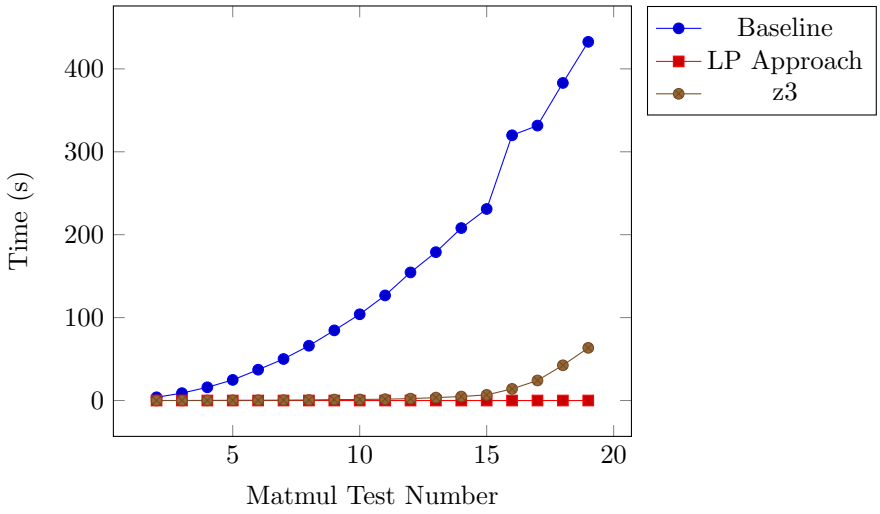
The results show that our implemented SAT solver favors smaller sets of clauses. The runtime grows exponentially as the number of clauses and literals grows. When there are not many clauses needed to solve the BitVector Theory, our SAT solver is much faster than the z3 solver, as seen in the

AND and OR cases. However, as the number of clauses increases, our SAT performance drops down exponentially, which is most notably seen in the multiplication cases.

6.2 Optimization - LP Approach

Test Case	Baseline Time (ms)	LP Approach Time (ms)	Z3 Time (ms)
square unsat	2373.96	8.65	11.96
square sat	2380.08	2.70	16.46
quadratic	624.41	0.79	9.72
mult sat	711.60	165.71	16.74
mult unsat	721.48	77.67	22.00
matmul 2	3903.55	0.86	6.02
matmul 3	8914.04	1.22	68.11
matmul 4	15960.27	1.33	140.55
matmul 5	24853.82	1.53	313.82
matmul 6	37182.19	1.75	362.37
matmul 7	50062.04	2.43	480.82
matmul 8	66052.49	2.46	675.37
matmul 9	84618.13	2.85	1142.18
matmul 10	104014.07	3.41	1268.11
matmul 11	126768.69	3.76	1623.48
matmul 12	154509.41	4.59	2281.57
matmul 13	178937.32	5.00	3547.02
matmul 14	207997.00	5.98	4816.18
matmul 15	231025.08	6.36	6804.18
matmul 16	319902.55	7.47	14022.92
matmul 17	331601.34	8.13	24172.02
matmul 18	383032.10	9.37	42503.63
matmul 19	432595.99	10.63	63531.82

Figure 1. Time Taken for Matmul Tests



These results show that the LP approach is significantly faster than the baseline bit-blast approach and the z3 solver in a variety of cases. The LP approach is much faster than the baseline in all these

cases. It is also significantly faster than z3 in most cases, but only slightly slower in the mult sat and mult unsat cases.

The advantages of the LP approach can clearly be demonstrated by the matmul cases in Figure 1. The LP approach scales much better than the baseline and z3 solvers in addition and multiplication heavy test cases such as matrix multiplication, and is orders of magnitude faster.

These results show promise in the use of the LP approach for modeling and the verification of non-overflow datapath computation, as it may be orders of magnitude more performant than bit-blasting and state-of-the-art implementations in arithmetic-heavy cases.

7 Conclusion

The study of bit-vector SMT solvers is important and has a lot of real-world applications. In this project, we have implemented: a baseline SAT solver with Tseitin Transformation, a theory solver using bit-blasting, and a theory solver optimization using mixed integer linear programming [5]. We evaluated the results using different benchmarks and find that without optimizations, the time to solve a SAT problem increases exponentially with the number of clauses and literals. It is not surprising because SAT is an NP-complete problem, and it is expected to grow exponentially. Even with the state-of-the-art z3 tool (with our bit-blasting algorithm connected), the runtime is still quite long. However, optimizations in domain-specific problems can perform orders of magnitude better. Based on our results, the advantages of the LP approach optimization is quite significant and may be useful in practice for modeling arithmetic-heavy systems, such as datapaths.

8 References

[1] Z. Zeng, P. Kalla, M. Ciesielski. LPSAT: A Unified Approach to RTL Satisfiability DATE'01. 2001.

[2] Brinkmann, R., Drechsler, R.: RTL-datapath verification using integer linear programming. In: Proceedings of the ASPDAC/VLSI Design Conference 2002, pp. 741–746. IEEE Computer Society Press (2002)

[3] https://highs.dev

[4] De Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS, Lecture Notes in Computer Science, vol. 4963, pp. 337–340. Springer, Berlin (2008).

[5] https://github.com/VoidMercy/CS-257-Project