The Effects of Arithmetic Encodings on SAT Solver Performance

Bryan Brady, Yang Yang
University of California, Berkeley
{bbrady,yangyang}@eecs.berkeley.edu

I. Introduction

As digital systems continue to grow, verification of these systems is becoming an increasingly important and difficult problem. To ensure a quick time-to-market, the verification problem must be addressed early in the design cycle. This requires the ability to verify system-level descriptions of hardware and embedded software systems, such as, C or Verilog, by proving assertions and proving functional equivalence. This can be done at the word-level by abstracting the word-level descriptions and proving equivalence or at the bit-level by converting the system-level descriptions into their equivalent boolean circuit representation and checking satisfiability.

When verifying system-level descriptions of arithmetic functions at the bit-level, a verification engineer must choose how to represent the arithmetic operations in these functions. For example, if the arithmetic function being tested uses addition, the verification engineer will have to decide what type of adder he/she will use to represent the addition operation. Due to the fact that systems are becoming increasingly larger and more complex, verification needs to be as fast as possible. The encoding of a simple element like an adder may have a large effect on the runtime of the verification engine. With the recent advances in SAT solving, using SAT to verify the correctness of arithmetic assertions or functions at the bit-level may be a viable option.

The goal of this project is to analyze and evaluate the performance of current state-of-the-art SAT solvers on arithmetic functions. More specifically, given assertions or functionality based on arithmetic operations, our goal is to determine which arithmetic encodings are "easier" to solve using a variety of SAT solvers.

II. Literature Survey

Bit-vectors ("words") are a critical abstraction for reasoning about arithmetic expressions. Intuitively, a bit-vector is a fixed-length string of individual bits. Operations on bit-vectors can be described in terms of their effect on each bit, or as transformations on the bit-vectors as a whole. Bit-vectors combined with arithmetic theory and boolean logic can be used to express properties or functions of a system. While deciding the equality of functions or validating properties of arithmetic expressions is NP-hard, efficient heuristics exist which allow most practical problems to be solved efficiently.

There are several approaches to word-level verification. Multiplicative Binary Moment Diagrams (*BMDs) [1] are a data structure that represent arithmetic expressions and handle arithmetic operations in word-level verification. It uses a decomposition of a linear function based on its "moments". The edge weights are combined multiplicatively. Like BDDs, *BMDs are canonical by construction. Unlike BDDs, however, *BMDs can represent most integer arithmetic operations with a size linear in the number of variables. In [2], Bryant et. al. propose a hierarchical methodology (using *BMDs) to perform word-level verification of arithmetic circuits focusing on circuits that are inefficient to represent at the bit-level, such as: multiplier, divider, and square root circuits. At the lowest level, the building blocks of the circuit are represented at the bit- and word-levels, while at higher levels, the circuits are represented as compositions of word-level descriptions. At the lowest level, it is necessary to verify the bit-level descriptions against the word-level specifications, however, at higher levels, it is only necessary to verify the composition of the word-level building blocks with the system specification. This hierarchical methodology makes it possible to efficiently represent and verify circuits at the word-level.

The Stanford Validity checker (SVC) [3] is a complete and automatic verification tool for deciding equality of a bit-vector arithmetic theory. It canonizes the arithmetic expressions at the word-level, and uses a solver to transform the atomic equations to a specific form, based on properties of the arithmetic theory. CVC Lite [4] (the successor to SVC and CVC) supports linear real arithmetic. It checks the validity of three-valued Kleene logic formulas by reducing the formulas into two-valued logic.

The approach in [5] combines structural word-level automatic test pattern generation (ATPG) and modular arithmetic constraint-solving techniques to solve the constraints imposed by assertion properties. First, the assertion property is inverted to produce a counter-example-generation problem, and the word-level ATPG is applied to solve the problem. A branch-and-bound algorithm is employed to justify the value requirements

on the control logic. After the constraints on the control logic are satisfied, an arithmetic constraint solver based on the modular number system is applied to solve the constraints on the data-path portion. Heuristics are used to convert the non-linear arithmetic constraints to linear constraints which are then solved by linear constraint solver.

In [6] Stoffel and Kunz describe a reverse-engineering techinque to verifying integer multipliers by extracting the adder networks present inside the multipliers. This methodology allows for the verification of multipliers with other multipliers with dissimilar internal structure.

Burch showed in [7] that it is possible use BDDs to verify multipliers while avoiding the exponential blowup. This is possible by the use of fan-out splitting. Essentially, fan-out splitting introduces new variables inside the multiplier whenever there is fan-out from one of the primary inputs. This technique requires that the specification of multiplication is changed according to the fan-out splitting. While fan-out splitting introduces extra variables, it allows the growth of the BDD to be cubic, which is much better than exponential.

This work attempts to analyze the various encodings of arithmetic circuits on bit- and word-level SAT solvers.

III. Arithmetic Circuits

For brevity, we will provide short descriptions of each adder and multiplier that we have implemented. Please refer to [1] and [2] for more details about the implementations of the adder and multiplier circuits. The other operations such as subtract, the boolean operators, and the comparison operators were all implemented in the usual way.

A. Adders

We have implemented five different types of adder circuits:

- ripple-carry adder (RC1): two-level carry logic,
- ripple-carry adder (RC2): multi-level carry logic,
- carry-lookahead adder (CLA): every four bits are grouped together. The carry bits inside each group are computed in parallel, while the carry bits between groups ripple through.
- parallel-prefix adder (PPA): uses a binary-tree like structure to compute the carry bits in time logarithmic to the size of the input word.
- carry-save adder (CSA): reduces the problem of adding three numbers to the problem of adding two numbers. This adder is used in many high speed multiplication algorithms.

The number of *XOR* gates varies between the ripple-carry implementations and the CLA and PPA implementations. However, this isn't a deciding factor for the runtime. The depths and numbers of *AND* and *OR* gates are different for these circuits, as well. A two-level ripple-carry adder has $5n$ *AND* and *OR* gates, and the depth of the circuit is $O(n)$. A multi-level ripple-carry adder has two fewer *AND* and *OR* gates for each pair of bits, i.e. $3n$ *AND* and *OR* gates. The depth of the circuit is still $O(n)$. An carry-lookahead adder has twenty more *AND* and *OR* gates for each four pairs of bits, i.e. $8n$ *AND* and *OR* gates. The depth of the circuit is $O(n)$, more precisely $O(n/4)$. Parallel-prefix adder compute the carry bits in $O(lgn)$ depths, so the depth of the circuit is $O(lgn)$, while the number of gates in this circuit is a lot more than the others, but the circuit is still $\Theta(n)$ size.

B. Multipliers

We have implemented three different combinational multiplication circuits:

- multiply by a constant: uses the grade-school method, however, each circuit is tailored specifically to the value of the constant, eliminating unnecessary logic
- grade-school multiplier: partial products are computed by iteratively shifting and adding
- Wallace-tree multiplier: partial products are added in parallel by using a carry-save adder tree. This reduces the problem into a series of carry-save additions and a final regular addition.

Grade-school multipliers operate in $\Theta(n)$ time and have $\Theta(n^2)$ size. Wallace-tree multipliers also have $\Theta(n^2)$ size, but they operate in $\Theta(lgn)$ time.

In addition to the multiplier circuits, it was necessary to account for the multiplication of negative numbers. This required additional logic. We implemented two different methods to handle this. One way is to double the size of both operands, then direct multiplication will provide the correct result. This is inefficient; by doubling the precision ahead of time, all additions must be double-precision and at least twice as many partial products are needed than for the efficient algorithms used in practice. The other method is commonly used. First, check to see if the multiplier is negative. If so, negate (i.e., take the two's complement of) both operands before multiplying. This is necessary because grade-school multiplication requires that the multiplier is positive. Since both operands are negated, the result will still have the correct sign and magnitude.

C. Sorting Networks

We have implemented sorting networks [8],[9], and [10] to handle the integer linear contraints. To be precise, an integer linear contraint is an inequality on a linear combination of interger variables: $C_0x_0 + C_1x_1 + ... + C_{n-1}x_{n-1} \geq C_n$.

Empirically it has been noted that SAT-solvers tend to perform poorly in the presence of parity. Because all but one variable must be set before anything can propagate, parity constraints generate few implications during unit propagation. Since full-adders (RC1, RC2, PPA, CLA,

CSA) contain XOR gates (a parity constraint), we might expect bad results from using them extensively.

To alleviate the problems inherent to full-adders, we represent numbers in unary instead of in binary in the following way. We first make all the coeffcients non-negative in preprocessing. Then we flatten out every m-bit integer variable x_i into a sequence of $m-1$ boolean variables $< b_{i,0}, ..., b_{i,m-2} >$, and every coeffient C_i into a sequenceon of boolean variables $< C_{i,0}, ..., C_{i,k_i-1} >$ on the base $B =< 2^0, 2^1, ..., 2^{m-2} >$. Then we group all the boolean variables into different groups according to the base and the coeffients. After that, we sort the boolean variables in every group by sorting network, which only contains *AND* and *OR* gates. The constraint is presented by a certain combinations of *AND* and *OR* operations on some of the outputs of the sorting networks.

By using sorting network, it is possible to handle the integer linear constraints without using XOR gates. The main part of circuits for integer linear contraints is sorting network. For a linear contraint with n m-bit integers and maximally k-bit coefficents, there are at most $m+k-2$ groups, each of which contains at most $(n \times m)$ boolean inputs. The depth of a sorting network for $O(n \times m)$ inputs is $\Theta(lg^2(n \times m))$. While we have sorting networks implemented, due to lack of time and appropriate benchmarks, we were unable to obtain meaningful results as to the performance of sorting networks. However, as stated in [10], sorting networks should alleviate the parity constraints, hence improving runtime.

IV. Solvers

Three bit-level SAT solvers were used to analyze the performace of the arithmetic encodings: Chaff [11], MiniSAT [8], and SatELite [9]. In addition, ABC [12] was used to convert from ISCAS bench format to CNF format. Unfortunately, we were unable to test any word-level solvers due to lack of time and non-functional software. However, we were able to run experiments on C-SAT, a circuit-SAT solver [13],[14].

V. Experiments and Results

A. Experiment 1: Satisfying Assignments for Addition

For this experiment, a large number of circuits of the following form $a +_{size} b = constant$(where size is 4, 8, 16, or 32) were generated using different combinations of adder implementations. Next, the SAT solvers were invoked on the test circuits and the satisfying assignments to a and b were extracted. For this experiment, only MiniSAT and Zchaff were used. (ABC uses MiniSAT and SatELite destroys the topology of the circuit.) The satisfying assignments generated by MiniSAT were $a, b > 0$, whereas Zchaff chooses either a or b positive, and the other negative. This behavior is consistent across adder implementations. Upon further exploration, we observed that the decision order in Zchaff is related to the value of the constant, in addition, for circuits with the same constant and different operator size, the decision order is similar. We have not been unable to come up with a theoretical reason for this behavior, however, we feel that this is an effect of a design decision made in the implementation of the SAT solvers.

B. Experiment 2: Multiplication

For this experiment, we generated numerous circuits of the following form: $a *_{32} b = p$, where a and b are 32-bit variables, and p is a prime. We tested on a set of different primes distributed in the range $[2, 2^{31}]$. In most cases, the runtimes of ABC and Minisat are more than two times longer than Zchaff, and there are slight differences in the runtimes between the two different multiplier circuits. We also tested on numerous circuits of the following form: $a *_{32} b = n$, where n is the product of two primes. The results of this experiment have similar characteristics as the previous multiplier experiment. One last observation, is that when we run this experiment with the grade-school multiplication algorithm and zchaff, the satisfying assigments are either $a = 1, b = p$ or $a = p, b = 1$. This is interesting because its the opposite of what was happening for addition.

C. Experiment 3: Miscellaneous Benchmarks

For this experiment, we ran all of the SAT solvers on a set of benchmark circuits. We ran each SAT solver on each benchmark once for each adder and multiplier type. Depending on the benchmark, some combinations of arithmetic encodings and SAT solvers are clearly better than others, however, there does not seem to be a general trend describing the runtime performance specific to an encoding, except that the number of XOR gates is positively correlated with the runtime. Figure 1 shows the lack of a trend among the various adder implementations using the grade-school multiplication algorithm. SatELite and Zchaff produce similarly trendless results, and for the sake of brevity, the figures will be left out.

D. Experiment 4: Additions in different orders

For this experiment, we generated a number of circuits of the following form $((x_1 +_{size} x_2) +_{size} x_3) +_{size} x_4 \neq (x_1 +_{size} x_2) +_{size} (x_3 +_{size} +x_4)$ (where size is 4, 8, 16, ..., 2048), using different adder implementations. Then we invoked both MiniSat and SatELite on the test circuits to check their satisfiability (Obviously, the answer is unsatisfiable). Table I gives the runtimes of MiniSat on different circuits.

In most cases, especially in the cases of large word-size, RC2 takes the shortest runtime, and the PPA takes the longest runtime (See 2. We noticed that the runtimes has the same trend as the circuits size, which increase in the order of RC2, RC1, CLA and PPA. Consequently,

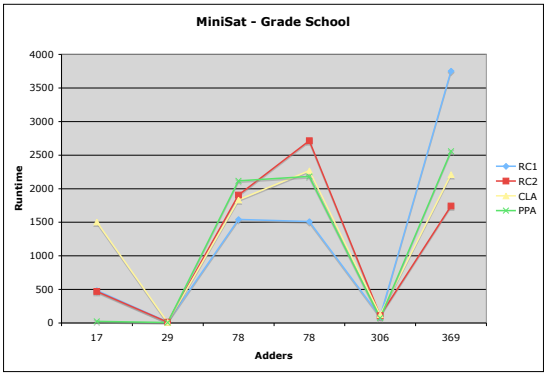

Fig. 1. MiniSAT runtimes on a number of benchmarks with varying adder implementations.

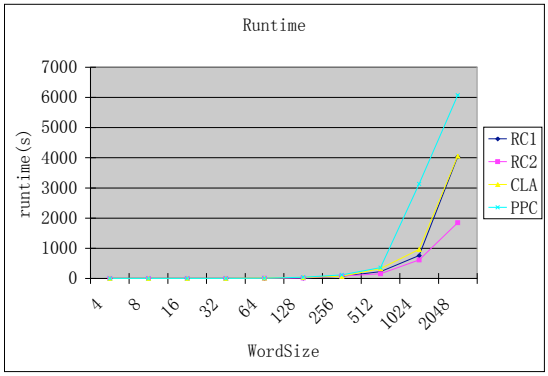

Fig. 2. MiniSAT runtimes for associativity test for various adder implementations.

Size	Runtimes(s)			
	RC1	RC2	CLA	PPC
4	0.005999	0.005999	0.007998	0.006998
8	0.054991	0.039993	0.086986	0.037994
16	0.119981	0.166974	0.247962	0.19597
32	0.833873	0.437933	0.619905	1.61375
64	2.19467	5.82411	12.3091	8.79766
128	11.8212	9.66253	32.1501	35.4066
256	72.456	78.671	67.3398	112.714
512	220.752	152.303	326.283	355.373
1024	754.105	609.23	960.821	3132.21
2048	4032.33	1844.11	4053.98	6071.45

TABLE I
RUNTIMES OF MINISAT FOR EXPERIMENT 4

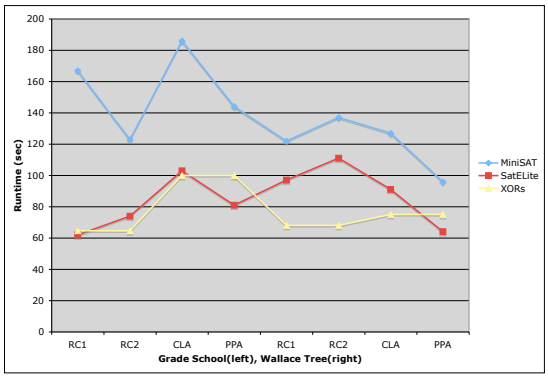

Fig. 3. Runtime for Karatsuba experiment with w=4

the number of variables and SAT-clauses for the different adder circuits increase in the same order, this could be one of the reasons for the runtime differences. Besides, the depths and maximal cuts of the circuits also vary depending on the adder circuits, but it seems that these factors do not effect the runtimes as much as the circuit sizes.

We also performed similar experiments on multplication. We generated a number of circuits of the following form $(x_1 *_{size} x2) *_{size} x_3 \neq x_1 *_{size} (x_2 *_{size} x_3)$, using different adder implementations. The runtimes are quite big even for the small sizes (it takes more than 5000 seconds when the word size is 8). The circuit sizes on these two multipliation circuits are close. Also, the results we got for the sizes 4, 5, 6, 7, 8 show that the runtimes for grade-school and wallce-tree multiplier are close too.

E. Experiment 5: Karatsuba Multiplication

For this experiment, the Karatsuba multiplication algorithm was verified against both the grade-school and Wallace Tree multiplication implementations. More precisely, we checked the validity of $x*y = (b^2+b)(x1*y1) - b((x1-x0)*(y1-y0)) + (b+1)(x0*y0)$ where x and y are $2w$ bits each, $b = 2^w$, $x = b*x_1 + x_0$ and $y = b^2*y_1 + y_0$. We tested this for $w = 4$. We had planned on testing this with $4 \geq w \leq 8$, however, with $w = 5$ the benchmarks timed out after one hour. As previously mentioned, the only trend we could find was a correlation between the number of XOR gates and the total runtime. See Figure 3. One thing to note about Figure 3 is for RC2, the runtime is somewhat higher than it is for RC1. We think this is because the multi-level carry logic of RC2 adds extra levels of depth to the circuit which degrades SAT solver performance.

F. Experiment 6: Benchmark Scaling

For this experiment, we chose to scale the word sizes of two benchmarks from 32 to 24 and 16 and subsequently analyze the runtimes. This experiment produced unexpected (and, so far, unexplainable results).

One would expect the runtimes of the circuits with varying sizes to increase as the word-size increases, however, this was not the case. Tables II and III show the runtimes for the various adder implementations for different word sizes. As you can see, the runtime for the benchmark with 24-bit words takes longer to solve than the benchmark with 32-bit words for RC2 and MiniSAT, however, for SatELite, the runtime for RC2 with 24-bit word size is lower than the runtime for 16-bit word sizes. Another unexpected result is the runtimes for the PPA implementations are faster for 32-bit word sizes than they are for 16-bit word sizes. MiniSAT and SatELite exhibit this behavior. Unfortunately, we have no explanation for these behaviors.

Adder Type	16-bit	24-bit	32-bit
RC1	47	105	93
RC2	55	431	109
CLA	81	98	122
PPA	146	N/A	76

TABLE II

TABLE II: MINISAT RUNTIMES FOR VARIOUS ADDER IMPLEMENTATIONS AND WORD SIZES.

Adder Type	16-bit	24-bit	32-bit
RC1	58	108	98
RC2	88	77	121
CLA	75	132	109
PPA	127	N/A	35

TABLE III

TABLE III: SATELITE RUNTIMES FOR VARIOUS ADDER IMPLEMENTATIONS AND WORD SIZES.

VI. CONCLUSIONS

As the results indicate, there is no clear answer as to which is the best arithmetic encoding to use. While it is well known that parity is a major bottleneck in SAT solver performance, there are other contributing factors such as circuit depth and overall circuit size. The runtimes for the various adders throughout the experiments seemed to be uncorrelated from one benchmark to another, which leads us to believe that the runtimes are benchmark dependent, and for the most part, independent of the adder implementation that is used. As for multiplier implementations, the Wallace Tree and Grade-school multiplication algorithms were comparible in runtime performance. The differences in runtime were not large enough or consistent enough to make a generalization as to which multiplier encoding is better.

One last observation is that, depending on the benchmark and adder implementation, SatELite's effectiveness varies. By this, we mean, for one adder, SatELite might outperform MiniSAT however, for the same benchmark and a different adder, MiniSAT will outperform SatELite. Unfortunately, there is no trend with this behavior. For instance, SatELite might outperform MiniSAT on benchmark X with adder A, however, on benchmark Y, MiniSAT might outperform SatELite with adder A.

As you may have noticed, we have not presented any results of the performance of C-SAT [13],[14]. This is due to the fact that C-SAT was extremely slow compared to bit-level SAT solvers. In fact, for all benchmarks except for the smallest one, C-SATs runtime was at least 1-2 orders of magnitude larger than the runtime for MiniSAT and SatELite.

VII. FUTURE WORK

One possible avenue for future work would be to test more circuit encodings on SAT solver performance, however, we feel that this may be a fruitless path judging from the results obtained heresofar. Another area for future work is to analyze the effect of various preprocessing techinques on different circuit encodings. Lastly, the various arithmetic encodings need to be tested on more word-level solvers.

VIII. ACKNOWLEDGEMENTS

Thanks to Sanjit Seshia for his guidance and advising throughout this project and thanks to Armando Solar-Lezama for his input on the Karatsuba experiments.

REFERENCES

[1] R. E. Bryant and Y.-A. Chen, "Verification of arithmetic circuits with binary moment diagrams," in *Design Automation Conference*, 1995, pp. 535–541. [Online]. Available: citeseer.ist.psu.edu/bryant95verification.html

[2] Y.-A. Chen and R. E. Bryant, "ACV: an arithmetic circuit verifier," in *ICCAD*, 1996, pp. 361–365. [Online]. Available: citeseer.ist.psu.edu/chen96acv.html

[3] C. W. Barrett, D. L. Dill, and J. R. Levitt, "A decision procedure for bit-vector arithmetic," in *DAC '98: Proceedings of the 35th annual conference on Design automation*. New York, NY, USA: ACM Press, 1998, pp. 522–527.

[4] S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill, "A practical approach to partial functions in cvc lite." [Online]. Available: citeseer.ist.psu.edu/berezin04practical.html

[5] C.-Y. Huang and K.-T. Cheng, "Assertion checking by combined word-level ATPG and modular arithmetic constraint-solving techniques," in *Design Automation Conference*, 2000, pp. 118–123. [Online]. Available: citeseer.ist.psu.edu/huang00assertion.html

[6] D. Stoffel and W. Kunz, "Verification of integer multipliers on the arithmetic bit level," in *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. Piscataway, NJ, USA: IEEE Press, 2001, pp. 183–189.

[7] J. R. Burch, "Using bdds to verify multipliers," in *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*. New York, NY, USA: ACM Press, 1991, pp. 408–412.

[8] N. Eén and N. Sörensson, "An Extensible SAT-solver," *Lecture Notes in Computer Science*, vol. 2919, January 2004.

[9] N. Eén and A. Biere, "Effective preprocessing in sat through variable and clause elimination," *SAT*, 2005.

[10] N. Eén and N. Sörensson, "Translating Pseudo-Boolean Constraints into SAT," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 1–26, February 2006.

[11] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zheng, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proceedings of the 38th Design Automation Conference*, pp. 530–535, 2001.

[12] A. Mischenko, "Abc." [Online]. Available: http://www.eecs.berkeley.edu/~alanmi

[13] F. Lu, L. Wang, K. Cheng, and R. Huang, "A circuit SAT solver with signal correlation guided learning," in *Design, Automation, and Test in Europe (DATE '03), Munich, Germany*, Mar. 2003, pp. 892–897. [Online]. Available: citeseer.ist.psu.edu/lu03circuit.html

[14] F. Lu, L.-C. Wang, K.-T. T. Cheng, J. Moondanos, and Z. Hanna, "A signal correlation guided atpg solver and its applications for solving difficult industrial cases," in *DAC '03: Proceedings of the 40th conference on Design automation*. New York, NY, USA: ACM Press, 2003, pp. 436–441.

[15] J. P. Uyemura, *Introduction to VLSI Circuits and Systems*. John Wiley and Sons, 2002.

[16] *Introduction to algorithms*. Cambridge, MA, USA: MIT Press, 2001.

[17] L. Zheng, "Searching For Truth: Techniques for Satisfiability of Boolean Formulas," Ph.D. dissertation, Princeton University, June 2003.