Efficiently Solving Bit-Vector Problems Using Model Checkers

Andreas Fröhlich, Gergely Kovásznai, Armin Biere
Institute for Formal Models and Verification
Johannes Kepler University, Linz, Austria*

Abstract

Bit-precise reasoning is essential in many applications of Satisfiability Modulo Theories (SMT). Most approaches for solving quantifier-free fixed-size bit-vector logics (QF_BV) rely on bit-blasting. In previous work, we have shown that bit-blasting is not polynomial in general [19], and later proposed QF_BV$_{\ll 1}$, a class of bit-vector problems that is PSPACE-complete [15]. In this paper, we give examples of how to create (polynomial) SMV specifications out of QF_BV$_{\ll 1}$ formulas. We then use various model checkers to solve those problems and give detailed experimental results. Our results show that BDD-based model checkers outperform current SMT solvers by several orders of magnitude on our benchmarks. Unrolling and using SAT-based model checking turns out to be the same as bit-blasting and gives worse results. In addition to this, our approach allows us to easily generate new challenging benchmarks for SMT solvers as well as for model checkers.

1 Introduction

Bit-precise reasoning over bit-vector logics is important for many practical applications of Satisfiability Modulo Theories (SMT), particularly for hardware and software verification. Examples of state-of-the-art SMT solvers with support for fixed-sized bit-vector logics are Boolector [6], MathSAT [8], STP [16], Z3 [11], and Yices [12]. All these solvers rely on *bit-blasting* in order to translate bit-vector formulas into propositional logic (SAT). The result is then checked by a SAT solver.

In practice, e.g. in the SMT-LIB [1], the BTOR [7], and the Z3 format, the bit-widths in bit-vector formulas are encoded as binary, decimal, or hexadecimal numbers, i.e., a *logarithmic encoding* is used. In [19], we proved that the encoding of bit-widths affects the complexity of the decision problem of bit-vector logics. In particular, logarithmic encoding makes the quantifier-free fragment QF_BV NEXPTIME-complete.[1] Thus, bit-blasting is *not polynomial* in general. Consider the following example (in SMT2 syntax):

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(declare-fun z () (_ BitVec 1000000))
(assert (= z (bvadd x y)))
(assert (= z (bvshl x (_ bv1 1000000))))
(assert (distinct x y))
```

This formula verifies that for an arbitrary bit-vector x of bit-width one million, there exists no bit-vector $y \neq x$ with $x + y = x \ll 1$. Written to a file, this formula can be encoded with 225 bytes. Using the SMT solver Boolector (even with all rewritings switched on), bit-blasting

*This work is partially supported by FWF, NFN Grant S11408-N23 (RiSE).

[1] In [19], we introduced the notation QF_BV1 resp. QF_BV2 for QF_BV using a *unary* resp. a *logarithmic*, actually without loss of generality, *binary encoding*. In this paper, QF_BV will always refer to the logarithmic/binary case.

produces a circuit of size 129 MB encoded in the actually rather compact AIGER format. Tseitin transformation results in a CNF in DIMACS format of size 843 MB.

In related work [20], we tried to avoid this growth in size by giving a translation from QF_BV to EPR and then using iProver to solve the problem. In most cases, this approach turned out to perform worse than Boolector on the original instance. Since QF_BV is NExpTime-complete, it is not clear if it is possible to solve the general case more efficiently. However, the given example only uses *addition*, *shift by one* and *equality*. In [15], we showed that this kind of formulas can be expressed by QF_BV$_{\ll 1}$, a subset of QF_BV which turned out to be PSpace-complete. In order to prove this, we gave a polynomial translation from QF_BV$_{\ll 1}$ to Sequential Circuits, similar to the one for linear arithmetic on *non-fixed-size* bit-vectors proposed in [22, 23].

In this paper, we show how model checkers can be used to solve *fixed-size* bit-bector problems of this class. In contrast to [15] which provided the theoretical background, we now focus on experimental evaluation and analyze the potential benefits for efficiently solving bit-vector formulas. First, in Sec. 2, we provide a short overview of our translation as described in [15] and give some examples to show how we used this concept to convert SMT2 files to SMV. In Sec. 3, we then describe some benchmarks that we generated to evaluate the performance of various model checkers compared to state-of-the-art SMT solvers with support for fixed-sized bit-vector logics. On most of our benchmarks, BDD-based model checkers turn out to be faster by several orders of magnitude. We provide experimental data and discuss the results in detail. Finally, in Sec. 4, we conclude the paper and discuss further topics for future work.

2 QF_BV$_{\ll 1}$ to SMV

In [22, 23], the authors gave a polynomial translation for linear arithmetic on *non-fixed-size* bit-vectors (QFPAbit) into Sequential Circuits. In contrast to [22, 23], we focus on *fixed-size* bit-vectors but share the goal of avoiding the exponential explosion due to explicit state representation as for example used in MONA [18]. We adapted this translation in [15] to deal with fixed-size bit-vectors and extended it by various other operators like *shift by one* and *indexing*.

Given $\Phi \in$ QF_BV$_{\ll 1}$ without nested equalities. Let n be a bit-width, $x^{[n]}, y^{[n]}$ denote bit-vector variables, $c^{[n]}$ a bit-vector constant, and $t_1^{[n]}, t_2^{[n]}$ bit-vector terms only containing bit-vector variables and bitwise operations. Following [22, 23], we assume w.l.o.g that Φ only consists of the following types of *atoms*: $t_1^{[n]} = t_2^{[n]}$, $x^{[n]} = c^{[n]}$, and $x^{[n]} = y^{[n]} \ll 1^{[n]}$. It is easy to check that any QF_BV$_{\ll 1}$ formula can be written like this with only a linear growth in the number of original variables.

We encode each atom in Φ separately into an atomic Sequential Circuit. The encoding itself is straightforward in most cases. A concrete example translating QF_BV to SMV is given after the theoretic part of this section. Compared to [22, 23], we have to consider the fact that all bit-vectors have a fixed bit-width.

Let n_{max} be the maximal bit-width of all bit-vectors in the formula. We construct an additional Sequential Circuit representing a counter. The counter initially is set to 0 and is incremented by 1 in each clock cycle. A counter like this can be realized with $\lceil \log_2(n_{max}) \rceil$ latches, i.e. polynomially in the size of Φ.

Now, for each atomic Sequential Circuit, we add a check whether the value of the counter reached the bit-width n of the bit-vector variables corresponding to the input streams of the circuit. Once this is the case, the individual circuit does not change its output value anymore.

Since $n_{max} \geq n$, this will always hold at some point.[2]

Finally, after constructing all atomic circuits, their outputs are combined by logical gates following the Boolean structure of Φ. Other operators, such as *addition* or *indexing*, can either be replaced by *shift by one* in a preprocessing step or directly encoded into a Sequential Circuit [15].

We now show the translation for the motivational example given in Sec. 1 to the concrete SMV-format. First of all, a counter for the bit-width of the variables has to be introduced. This can be done using logarithmic many variables:

```
init(counter_bit0) := FALSE;
next(counter_bit0) := counter_bit0 xor (TRUE);
init(counter_bit1) := FALSE;
next(counter_bit1) := counter_bit1 xor (counter_bit0);
...
init(counter_bit19) := FALSE;
next(counter_bit19) := counter_bit19 xor (counter_bit0 & ... & counter_bit18);
```

We then keep track of whether the counter already reached the value of a certain bit-width.[3] This variable later serves as a guard for all atoms containing variables of the given bit-width:

```
init(counter_gte_1000000) := FALSE;
next(counter_gte_1000000) := counter_gte_1000000 |
  (counter_bit0 & counter_bit1 & ... & !counter_bit6 & ... & counter_bit19);
```

After introducing those helper variables, the actual formula can now be translated. The *distinct* operator is first replaced by negation of an *equality*. The translation to SMV then is straightforward:

```
init(atom_equal) := TRUE;
next(atom_equal) := case
  counter_gte_1000000 : atom_equal;
  TRUE : atom_equal & (x <-> y);
esac;
```

For translating *addition*, two atoms have to be introduced since the carry bit has to be remembered in the next step:

```
init(atom_add) := TRUE;
next(atom_add) := case
  counter_gte_1000000 : atom_add;
  TRUE : atom_add & (z <-> (x xor y xor atom_cin));
esac;

init(atom_cin) := FALSE;
next(atom_cin) := case
  counter_gte_1000000 : atom_cin;
  TRUE : atom_add & ((x & y) | (x & atom_cin) | (y & atom_cin));
esac;
```

[2] In contrast to [22], we assume that the input streams for all variables start with the least significant bit.

[3] The counter bits in the *next*-statement correspond to the binary representation of $n - 1$ (i.e. $999999_{10} = 11110100001000111111_2$ in our example).

The *shift* operator can be translated in a very similar way but will not be given here explicitely to keep the example short. Another way would be to replace $(x \ll 1)$ by $(x + x)$ in the preprocessing step.

Finally, the specification is defined by the logical combination of the individual atoms and additionally respecting the bit-width:

```
AG(!counter_gte_1000000 | !atom_add | !atom_shift | atom_equal)
```

We also implemented our translation including various operators in a tool called BV2SMV. Binaries and source code are available for download at [9].

3 Experiments

We first describe our benchmark sets. We generated six different sets of QF_BV formulas in SMT2 format. All sets of benchmarks consist of 32 instances each and have two attributes: First, all benchmark sets are *not bit-width bounded* [15]. Because of this, bit-blasting is known to be exponential in general. Second, all benchmarks only contain *bitwise operators*, *addition*, *subtraction*, *shift by one*, *indexing* and *relational operators*. This ensures that a polynomial translation to SMV exists. The different instances in a particular set of benchmarks only differ in the bit-width of their variables and constants. The bit-widths n of the individual instances are of the form $n = 2^i$ and $n = 1.5 \cdot 2^i$ with $i \in \{5, \ldots, 20\}$ for all six sets. All benchmarks will be submitted to the `QF_BV` category of SMT-LIB.

`QF_BV/froehlichkovasznaibiere/ndist.a.`n: We verify that, for two bit-vector variables $x^{[n]}, y^{[n]}$, it holds that $x^{[n]} < y^{[n]}$ implies $(x^{[n]} + 1^{[n]}) \leq y^{[n]}$. The instances are unsatisfiable and use *addition* and *unsigned less/greater than operators*.

`QF_BV/froehlichkovasznaibiere/ndist.b.`n: We give a counter-example (due to overflow) to the claim that, for two bit-vector variables $x^{[n]}, y^{[n]}$, it holds that $(x^{[n]} + 1^{[n]}) \leq y^{[n]}$ implies $x^{[n]} < y^{[n]}$. The instances are satisfiable and use *addition* and *unsigned less/greater than or equal operators*.

`QF_BV/froehlichkovasznaibiere/power2bit.`n: We verify that, for a bit-vector variable $x^{[n]} = 2^j$, it is not possible for two different bits to be both set to 1. The instances are unsatisfiable and use *indexing*, *subtraction*, *bitwise operators*, and *(in)equality*.

`QF_BV/froehlichkovasznaibiere/power2eq.`n: We verify that, for two bit-vector variables $x^{[n]} = 2^j, y^{[n]} = 2^k$, with a certain identical bit set to 1, the bit-vectors cannot be distinct. The instances are unsatisfiable and use *indexing*, *subtraction*, *bitwise operators*, and *(in)equality*.

`QF_BV/froehlichkovasznaibiere/power2sum.`n: We verify that, for two bit-vector variables $x^{[n]} = 2^j, y^{[n]} = 2^k$, with $j \neq k$, $x^{[n]} + y^{[n]}$ cannot be a power of 2. The instances are unsatisfiable and use *addition*, *subtraction*, *bitwise operators*, and *(in)equality*.

`QF_BV/froehlichkovasznaibiere/shift1add.`n: We verify that for an arbitrary bit-vector $x^{[n]}$, there exists no bit-vector $y^{[n]} \neq x^{[n]}$ with $(x^{[n]} + y^{[n]}) = (x^{[n]} \ll 1)$. The instances are unsatisfiable and use *addition*, *shift by one*, and *(in)equality*. The example used throughout the paper is part of this benchmark family.

Out of the benchmark instances in SMT2 format, we generated SMV instances by using BV2SMV and the flattening tool smvflatten.[4] We used the state-of-the-art SMT solvers Boolector, MathSAT, Z3, and STP on the SMT2 instances, and NuSMV [10] on the corresponding SMV instances. In order to involve state-of-the-art model checkers like Tip [13] and IImc[5] (that

[4] `http://fmv.jku.at/smvflatten/`

[5] `http://ecee.colorado.edu/wpmu/iimc/`

uses techniques described in [2, 3]), we also converted all the SMV instances to AIGER format by using the translation tool smvtoaig that is part of the AIGER distribution.

All our experiments were run on the same cluster and with the same setup as the latest Hardware Model Checking Competition (HWMCC'12).[6] More precisely, we used a 32-node cluster with Intel Quad Core 2.6 GHz processors and 8 GB RAM. The wall clock time limit was set to 900 seconds and the memory limit to 7 GB. Each solver had full access to one node (4 cores).

In total, we used 19 different solvers (resp. configurations) on 6 different benchmark sets each consisting of 32 instances, yielding a total of 3648 runs. All our results are available on our web page at [9] together with generation scripts for all benchmarks in SMT2 format and our tool BV2SMV.

Tab. 1 provides an overview of the total number of solved instances and the average runtime (in seconds) and space requirement (in megabytes) on the solved instances. For BMC solvers, we used the knowledge that the counters in the generated specifications only allow the atomic circuits to change their value in the first number of steps equal to the bit-width n of the original SMT2 formula. We therefore set the bound for unrolling to be equal to $n + 1$ and, whenever a BMC solver reached the bound without timeout or out-of-memory, counted the instance to be shown unsatisfiable.

The solvers were executed with default settings if not stated otherwise explicitely. However, in some exceptional cases, we intentionally used some promising or interesting strategies. For instance, in Tab. 1, `Tip-BMC` references Tip using BMC-based strategy. Since we expected and later experienced that BDD-based techniques perform particularly well on our benchmarks, we intended to test model checkers with BDD-based strategies, those which offer such an option. Note that NuSMV uses BDD-based forward reachability analysis by default. We also tested NuSMV with backward reachability analysis, referenced by `NuSMV-bw`. IImc also offers BDD-based solving strategy, with both forward resp. backward reachability analysis; we reference IImc with default settings resp. with BDD-based forward resp. backward reachability analysis as `IImc` resp. `IImc-BDD-fw` resp. `IImc-BDD-bw`.

	STP	Boolector	MathSAT5	Z3	IImc-BDD-bw	NuSMV-bw	IImc-BDD-fw	IImc	NuSMV	Blimc‡	Tip-BMC‡	Aigbmc‡	Tip
solved	147	146	127	123	**192**	189	185	172	170	147	130	99	93
sat	23	**32**	13	23	**32**	29	**32**	**32**	27	9	31	21	17
unsat	124	114	114	100	**160**	**160**	153	140	143	138	99	78	76
time	206	190	310	171	12	30	79	132	148	233	266	295	496
space	1063	805	587	2180	8	24	9	74	38	95	1142	2073	6

Table 1: Overall results for all solvers

Apart from those in Tab. 1, we tested other models checkers as well, all submitted to HWMCC'12. We excluded some of them due to uncertain results: (a) Super_prove2 and Simple_sat, which employ ABC with improved strategies, produced discrepancies on some satisfiable

[6] `http://fmv.jku.at/hwmcc12/`

‡Versions submitted to HWMCC'12.

instances; (b) PdTrav, on some instances, threw exception about syntactical error in input.

In total, `IImc-BDD-bw` clearly performs best as it can solve all instances. Backward reachability analysis seems to produce better results than forward reachability for BDD-based model checkers in general. While this applies especially to unsatisfiable instances, `NuSMV-bw` only performs slightly better than `NuSMV` on the satisfiable ones. Interestingly, `Boolector` also gives very good results for the satisfiable instances. As expected, in particular the average space requirement of all SMT solvers is very large.

Fig. 1, 2, and 3 provide a detailed overview of the runtimes and space requirements of various solvers on the individual benchmark sets. We chose `Boolector` and `STP` representing the SMT solver class and `NuSMV`, `NuSMV-bw`, `IImc`, `IImc-BDD-bw`, and `Tip-BMC` as model checkers. Please consider that sampling memory is imprecise in case of low runtime, causing noise on the plots that show memory consumption.

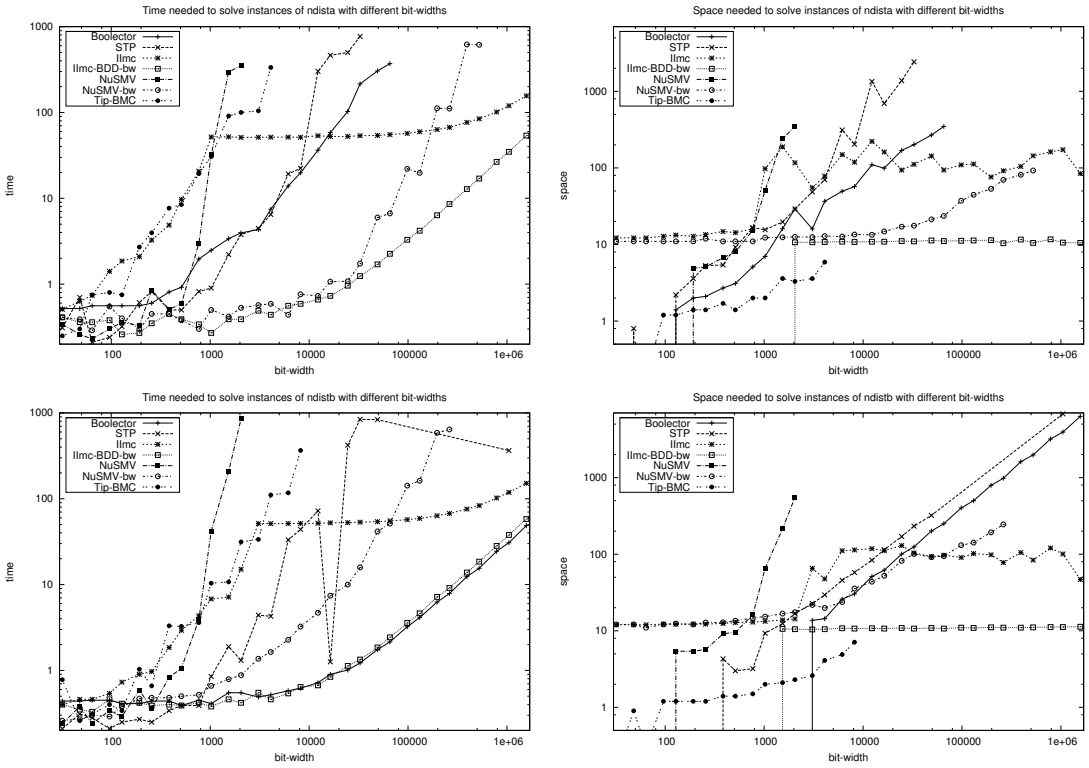

Figure 1: Detailed results of the `ndist.a` and `ndist.b` benchmark sets.

Fig. 1 shows the results of the solvers on the `ndist.a` and `ndist.b` benchmark sets. On the `ndist.a` instances, all BDD-based model checkers clearly outperform both SMT solvers considering time and space. `Tip-BMC` performs very similar to the SMT solvers. This is not surprising since unrolling up to a bound equal to the bit-width will in the end produce the same propositional formula as bit-blasting.

With `ndist.b` being satisfiable, SMT solvers show better runtimes while still requiring similar amounts of space. This can be explained by the fact that it is enough to guess the correct assignment which might be found as a consequence of good heuristics and at the same time

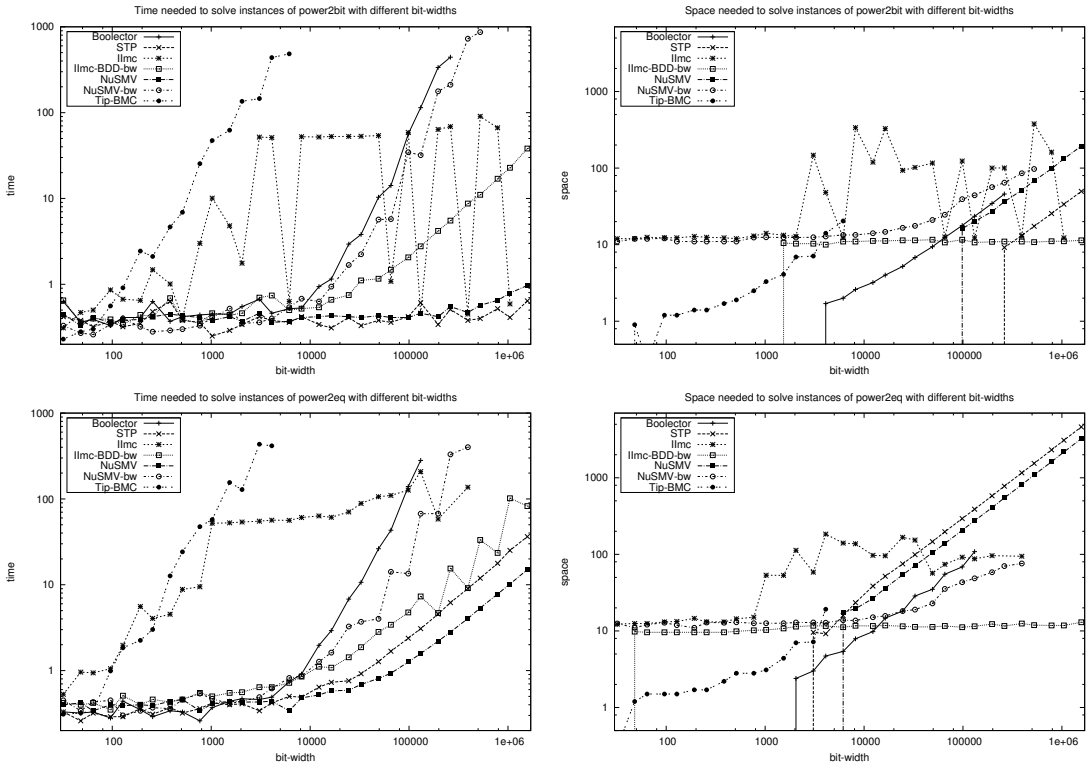

Figure 2: Detailed results of the `power2bit` and `power2eq` benchmark sets.

could cause the variation in the runtimes of `STP`. While backward reachability analysis seems to give a clear advantage on the unsatisfiable benchmark, it only slightly increases performance on the satisfiable one.

One interesting aspect in Fig. 2 is the fact that `STP` performs really well on both benchmarks. We suppose that this is connected to the fact that `power2bit` and `power2eq` both use indexing with relatively small indices. Interestingly, `Boolector` performs much worse on both instances. The good performance on this kind of formulas, therefore, does not seem to be a result of bit-blasting and applying SAT solvers but rather due to some special technique used in `STP`.

One might notice the typical shape of the runtime curves related to `IImc`: they start steep, but above a certain bit-width they show rather moderate ascent. The curves representing space consumption seem to grow slowly up to a certain point where, after a big jump, space usage almost seems to be fixed to a constant or, in some cases, even starts to decrease. We think that this strange behavior is due to the fact that `IImc` uses several scheduled approaches, such as IC3 [2], BMC, BDDs, etc. Probably due to the same fact, the `IImc` curves are even more hectic on the `power2bit` benchmark in Fig. 2. During our experiments we also tested IImc with IC3 strategy alone, resulting in timeouts on most instances. Therefore, we assume that above a certain bit-width IImc with default scheduling switches to BDDs, resulting in moderate ascent in memory consumption and runtime.

Probably Fig. 3 depicts most properly the distinction between BDD-based approaches and those which use SAT-based ones. Although SMT solvers and `Tip-BMC` time out quite soon on

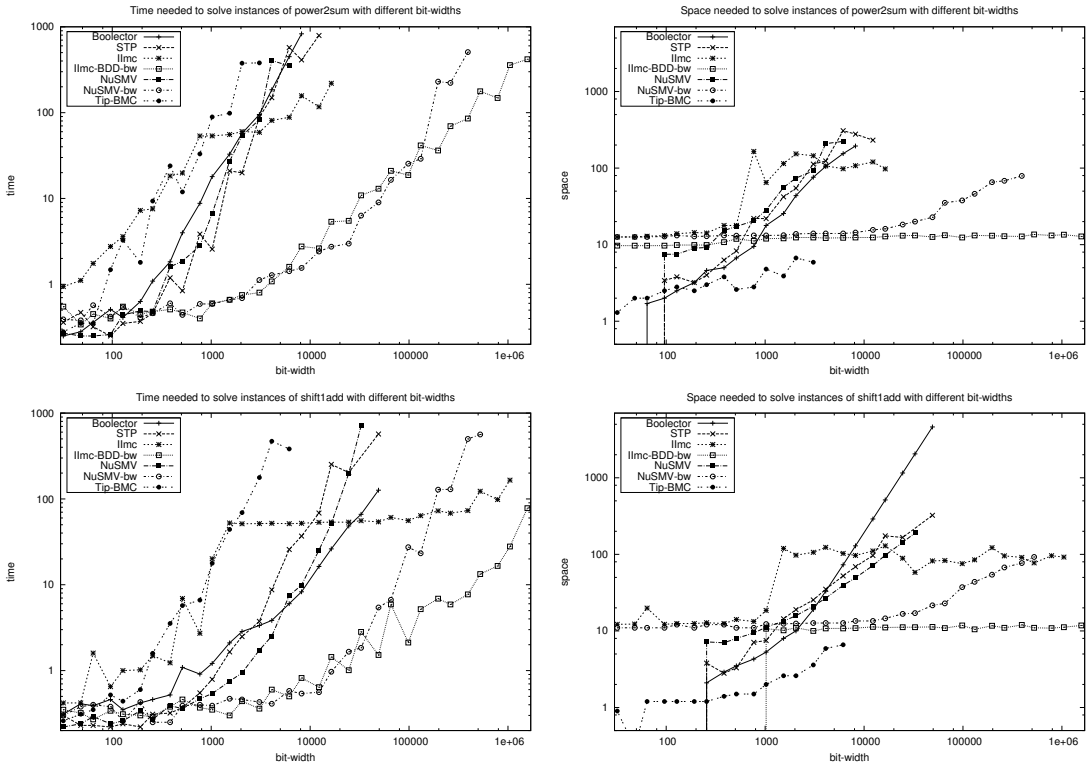

Figure 3: Detailed results of the `power2sum` and `shift1add` benchmark sets.

both problem sets, and, on the `power2sum` benchmark, the performance of `IImc` now is rather similar, BDD-based model checkers are able to deal even with very large bit-widths.

In general, looking at the runtimes, we can see that SMT solvers can compete well on instances with smaller bit-width, while BBD-based model checkers start to outperform their counter-parts with growing bit-width.

This effect becomes even stronger when we look at the space used during solving the formulas. Judging from the graphs, it might even be possible that the space requirement of BDD-based model checkers is logarithmic compared to that of SMT solvers. This could be the case due to the fact that SMT solvers apply bit-blasting, which is exponential for benchmarks that are not bit-width bounded, while our translation does not cause the problems to leave PSpace. However, this alone is not sufficient. BDD-based model checkers like NuSMV might create exponential sized BDDs nevertheless. More rigorous arguments or larger empirical analysis are needed.

4 Conclusion

In this paper, we efficiently solved quantifier-free bit-vector formulas using model checkers. While state-of-the-art SMT solvers usually apply bit-blasting to solve this kind of formulas, we already showed in previous work [19] that this can cause an exponential blowup in general. An approach for polynomially translating QF_BV to EPR exists [20] (as well as exponential

ones [14, 17]), but solving the resulting formulas also suffers from the NExpTime-completeness of EPR [20, 21]. Building on previous complexity results [15], however, we know that restricting QF_BV to only allowing *bitwise operators*, *shift by one*, *addition*, *subtraction*, *multiplication by constant*, *relational operators* and *indexing* leads to PSpace-completeness of the resulting logic. This allows us to polynomially translate bit-vector formulas to Sequential Circuits and use model checkers for reachability analysis.

In order to show the potential benefit of our approach, we created a set of benchmarks and used it to compare the performance of various model checkers on the translated instances to the one of current SMT solvers on the original files. We showed that on most of our problems, state-of-the-art model checkers like IImc and even older ones, such as NuSMV, performed better by several orders of magnitude considering runtime as well as space.

Our results also showed that BDD-based model checking techniques perform much better than SAT-based model checkers. This probably is the case because of the similarity between BMC and bit-blasting, and gives reason to investigate especially BDD-based solving techniques further.

Some of the best results were achieved by NuSMV. Considering the fact that NuSMV has seen relatively little development during the last years compared to current SMT solvers, this could lead to even better results if it is possible to improve the underlying techniques.

One of the main reasons we assume to be responsible for the good performance of model checkers on our benchmarks, is their better fit to the PSpace-nature of this problem class. Still, the resulting BDDs can of course be exponential in general.

While we did not pay special attention to the variable ordering during our translation, we ran NuSMV using `-dynamic` command, letting it figure out a good variable order during runtime. We also used the `-reorder` command to output the optimal variable order found by NuSMV and to look for patterns in it. When using this variable order in a second run instead of choosing the order dynamically, the runtimes usually decreased further.[7] Maybe our translation can be adapted using additional information to directly create variable orders that result in smaller BDDs. In order to do this, it might be interesting to look at the structure of the instances produced by our translation more closely. Especially the usage of counter definitions and constraints is similar throughout all formulas.

Sequential optimization techniques, such as those implemented in state-of-the-art model checkers like ABC [4], are useful even for bounded model checkers which otherwise only rely on unrolling. It is an interesting question whether it is possible to lift these techniques from model checking to bit-vector reasoning in combination or as a preprocessing step before bit-blasting.

Finally, only one model checker could solve all of our instances for the largest bit-widths. Constructing this kind of formulas, therefore, offers an easy way to provide challenging benchmarks for state-of-the-art SMT solvers and model checkers at the same time. For better solvers and future challenges, the difficulty of a problem can be adjusted by simply increasing the bit-width of the original SMT formula.

As a related classification problem, it will be interesting to investigate the complexity of Presburger arithmetic on fixed-size bit-vectors.[8] While the corresponding decision problem is known to be NP-complete for non-fixed-size bit-vectors, it is not clear whether we still remain in NP when considering fixed-size bit-vectors and whether translations as proposed in [5] are polynomial if a logarithmic encoding is used for the bit-widths.

[7] This is not included in our results since we did not analyze it in detail yet.

[8] The benchmark sets `ndist.a` and `ndist.b` are in this class.

References

[1] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[2] Aaron R. Bradley. Sat-based model checking without unrolling. In *Proc. VMCAI'11*, pages 70–87, 2011.

[3] Aaron R. Bradley, Fabio Somenzi, Zyad Hassan, and Yan Zhang. An incremental approach to model checking progress properties. In *Proc. FMCAD'11*, pages 144–153, 2011.

[4] Robert K. Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *Proc. CAV'10*, pages 24–40, 2010.

[5] Raik Brinkmann and Rolf Drechsler. Rtl-datapath verification using integer linear programming. In *Proc. ASP-DAC'02*, 2002.

[6] Robert Brummayer and Armin Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.

[7] Robert Brummayer, Armin Biere, and Florian Lonsing. BTOR: bit-precise modelling of word-level problems for model checking. In *Proc. BPR'08*, pages 33–38, 2008.

[8] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. The MathSAT SMT solver. In *Proc. CAV'08*, pages 299–303, 2008.

[9] BV2SMV project page. Website. `http://fmv.jku.at/bv2smv/`.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv version 2: An opensource tool for symbolic model checking. In *Proc. CAV02*, 2002.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proc. ETAPS'08*, pages 337–340, 2008.

[12] Bruno Dutertre and Leonardo de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[13] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4):543–560, 2003.

[14] Moshe Emmer, Zurab Khasidashvili, Konstantin Korovin, and Andrei Voronkov. Encoding industrial hardware verification problems into effectively propositional logic. In *FMCAD'10*, pages 137–144, 2010.

[15] Andreas Fröhlich, Gergely Kovásznai, and Armin Biere. More on the complexity of quantifier-free fixed-size bit-vector logics with binary encoding. In *Proc. CSR'13 (to appear)*, 2013.

[16] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.

[17] Zurab Khasidashvili, Mahmoud Kinanah, and Andrei Voronkov. Verifying equivalence of memories using a first order logic theorem prover. In *FMCAD'09*, pages 128–135, 2009.

[18] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. Mona implementation secrets. In *Proc. CIAA'00*, pages 182–194, 2000.

[19] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *Proc. SMT'12*, pages 44–55, 2012.

[20] Gergely Kovásznai, Andreas Fröhlich, and Armin Biere. Bv2epr: A tool for polynomially translating quantifier-free bit-vector formulas into epr. In *Proc. CADE'13 (to appear)*, 2013.

[21] Harry R. Lewis. Complexity results for classes of quantificational formulas. *J. Comput. Syst. Sci.*, 21(3):317–353, 1980.

[22] Andrej Spielmann and Viktor Kuncak. On synthesis for unbounded bit-vector arithmetic. Technical report, EPFL, Lausanne, Switzerland, February 2012.

[23] Andrej Spielmann and Viktor Kuncak. Synthesis for unbounded bit-vector arithmetic. In *Proc. IJCAR'12*, volume 7364 of *LNCS*, pages 499–513, 2012.