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
http://fmv.jku.at

SMT 2013
July 8 - July 9, 2013
Helsinki, Finnland

JOHANNES KEPLER
UNIVERSITY LINZ | JKU

Motivation

- How does the *encoding* of the bit-widths affect the *complexity* of satisfiability checking for BV logics?
- In practice *logarithmic* (e.g. binary, decimal, hexadecimal) encoding is used (in contrast with unary encoding)

Example in SMT2

```
(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))
```

Using Boolector:

- input file of 225 bytes in SMT2 format
- 129 MB in AIGER format; 843 MB in DIMACS format

Motivation

- How does the *encoding* of the bit-widths affect the *complexity* of satisfiability checking for BV logics?
- In practice *logarithmic* (e.g. binary, decimal, hexadecimal) encoding is used (in contrast with unary encoding)

Example in SMT2

```
(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))
```

Using Boolector:

- input file of 225 bytes in SMT2 format
- 129 MB in AIGER format; 843 MB in DIMACS format

Previous Work

Let QF_BV be the set of bit-vector formulas with *binary* encoding and all common bit-vector operations, e.g. bitwise operations, arithmetic operations, concatenation, slicing, shifts, relational operations, ...

- QF_BV is NExpTime-complete

  G. Kovásznai, A. Fröhlich, A. Biere, *On the Complexity of Fixed-Size Bit-Vector Logics with Binary Encoded Bit-Width.* In: Proc. SMT'12.

Previous Work

How does restricting the set of operations affect the complexity?

- $\mathrm{QF\_BV}_{\ll c}$: bitwise operations, equality, and shift by any constant
  $\rightarrow$ $\mathrm{QF\_BV}_{\ll c}$ is $\mathrm{NExpTime}$-complete
- $\mathrm{QF\_BV}_{\ll 1}$: bitwise operations, equality, and shift by only 1
  $\rightarrow$ $\mathrm{QF\_BV}_{\ll 1}$ is PSpace-complete
- $\mathrm{QF\_BV}_{bw}$: bitwise operations and equality
  $\rightarrow$ $\mathrm{QF\_BV}_{bw}$ is NP-complete

  A. Fröhlich, G. Kovásznai, A. Biere, *More on the Complexity of Quantifier-Free Fixed-Size Bit-Vector Logics with Binary Encoding.* In: Proc. CSR'13.

Previous Work

How does restricting the set of operations affect the complexity?

- $\mathrm{QF\_BV}_{\ll c}$: bitwise operations, equality, and shift by any constant
  $\rightarrow$ $\mathrm{QF\_BV}_{\ll c}$ is NExpTime-complete
- $\mathrm{QF\_BV}_{\ll 1}$: bitwise operations, equality, and shift by only 1
  $\rightarrow$ $\mathrm{QF\_BV}_{\ll 1}$ is PSpace-complete
- $\mathrm{QF\_BV}_{bw}$: bitwise operations and equality
  $\rightarrow$ $\mathrm{QF\_BV}_{bw}$ is NP-complete

  A. Fröhlich, G. Kovásznai, A. Biere, *More on the Complexity of Quantifier-Free Fixed-Size Bit-Vector Logics with Binary Encoding.* In: Proc. CSR'13.

PSpace-inclusion also holds for addition, indexing, multiplication by constant, relational operations, ...

Complexity: $\mathrm{QF\_BV}_{\ll 1} \in \mathrm{PSPACE}$

- Existing work for non-fixed-size bit-vectors resp. quantifier-free Presburger arithmetic with bitwise operations ($\mathrm{QFPA_{BIT}}$):

  $\mathrm{QFPA_{BIT}} \xrightarrow{\text{polynomially}}$ Sequential Circuits

  A. Spielmann, V. Kuncak, *Synthesis for Unbounded Bit-Vector Arithmetic.* In: Proc. IJCAR'12.

- A flat normal form of the original formula is created:

  Logical combination of certain atomic expressions.

- For each atomic expression a direct translation into an atomic sequential circuit can be given.

  The result is the logical combination of the atomic circuits.

- Can be adopted for fixed-size bit-vectors of bit-width 2^n by introducing a n-bit counter.

Practical Implementation

- Bit-blasting can cause exponential growth.
- Translate original SMT2 input to a polynomial SMV specification.
- Use model checkers to solve the translated formulas.

  (additionally, smvflatten and smvtoaig are used to obtain inputs for model checkers that expect AIGER format)

Example in SMT2

```
(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))
```

Example

```
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(declare-fun z () (_ BitVec 1000000))
```

```
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);

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

Example

```
(assert (distinct x y))
```

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

Example

```
(assert (= z (bvadd x y)))
```

```
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 : (x & y) | (x & atom_cin) | (y & atom_cin);
esac;
```

Example

Shift is translated similar to addition. Finally, check:

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

Experiments: Benchmarks

`QF_BV/froehlichkovasznaibiere/shift1add.n`:

$$(x + y = x \ll 1) \rightarrow (x = y)$$

`QF_BV/froehlichkovasznaibiere/ndist.a.n`:

$$(x < y) \rightarrow (x + 1 \leq y)$$

`QF_BV/froehlichkovasznaibiere/ndist.b.n`:

$$(x + 1 \leq y) \rightarrow (x < y)$$

`QF_BV/froehlichkovasznaibiere/power2sum.n`:

$$(x = 2^k \wedge y = 2^l \wedge k \neq l) \rightarrow (x + y \neq 2^m)$$

`QF_BV/froehlichkovasznaibiere/power2bit.n`:

$$(x = 2^k \wedge x[c_1] = 1) \rightarrow (x[c_2] \neq 1)$$

`QF_BV/froehlichkovasznaibiere/power2eq.n`:

$$(x = 2^k \wedge y = 2^l \wedge x[c] = 1 \wedge y[c] = 1) \rightarrow (x = y)$$

Experiments: Setting

- Same cluster and setup as HWMCC'12.
  - 32 nodes, Intel Quad Core 2.6 GHz, 8 GB RAM
  - The wall clock time limit 900 seconds
  - Memory limit 7 GB
  - Each solver has full access to one node (4 cores)

- 3648 runs in total.
  - 19 solvers (resp. configurations)
  - 6 different benchmark sets each consisting of 32 instances

- All our results are available on our web page together with generation scripts for all benchmarks in SMT2 format and our translation tool bv2smv.

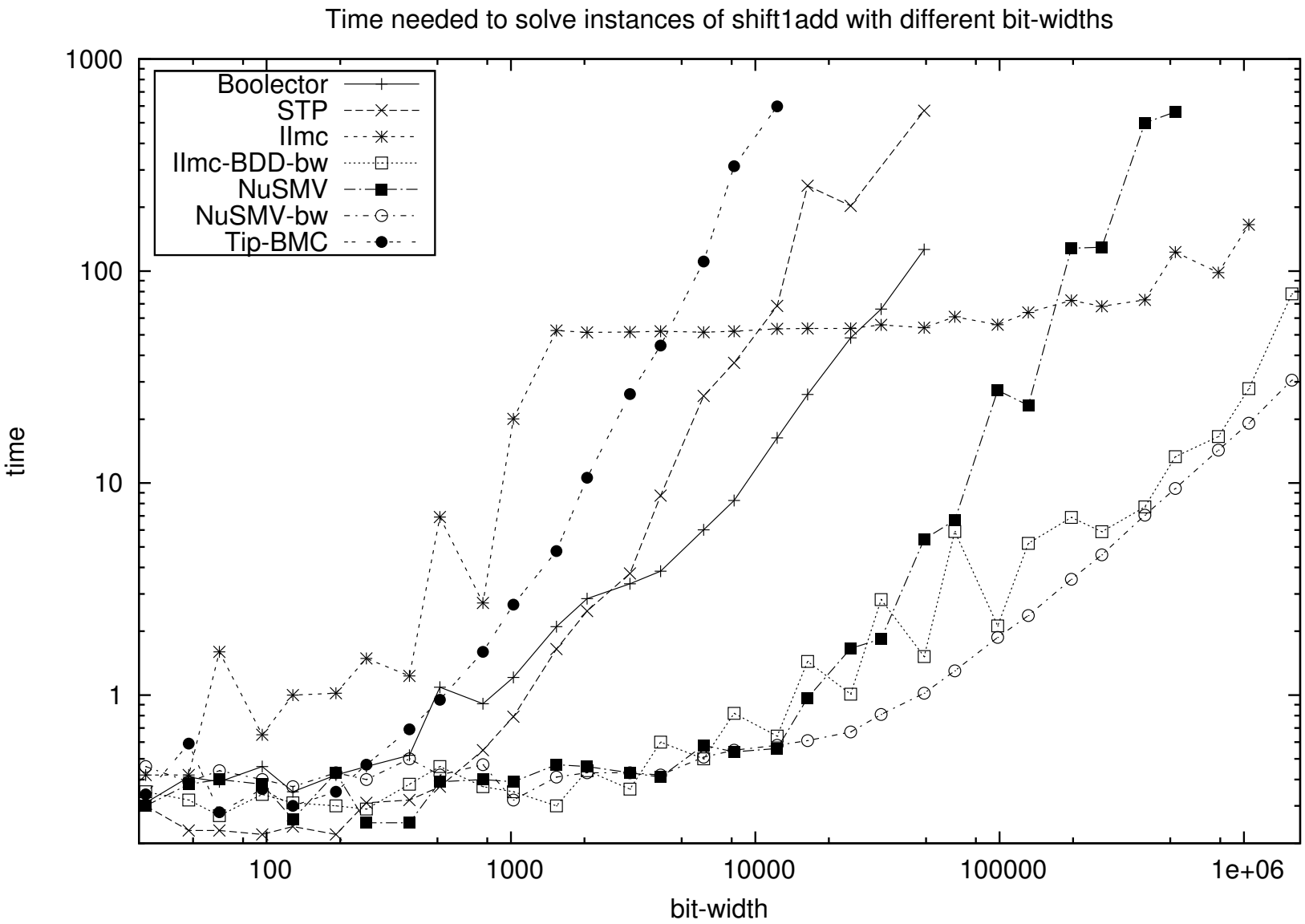
  http://fmv.jku.at/bv2smv/

Experiments: Results

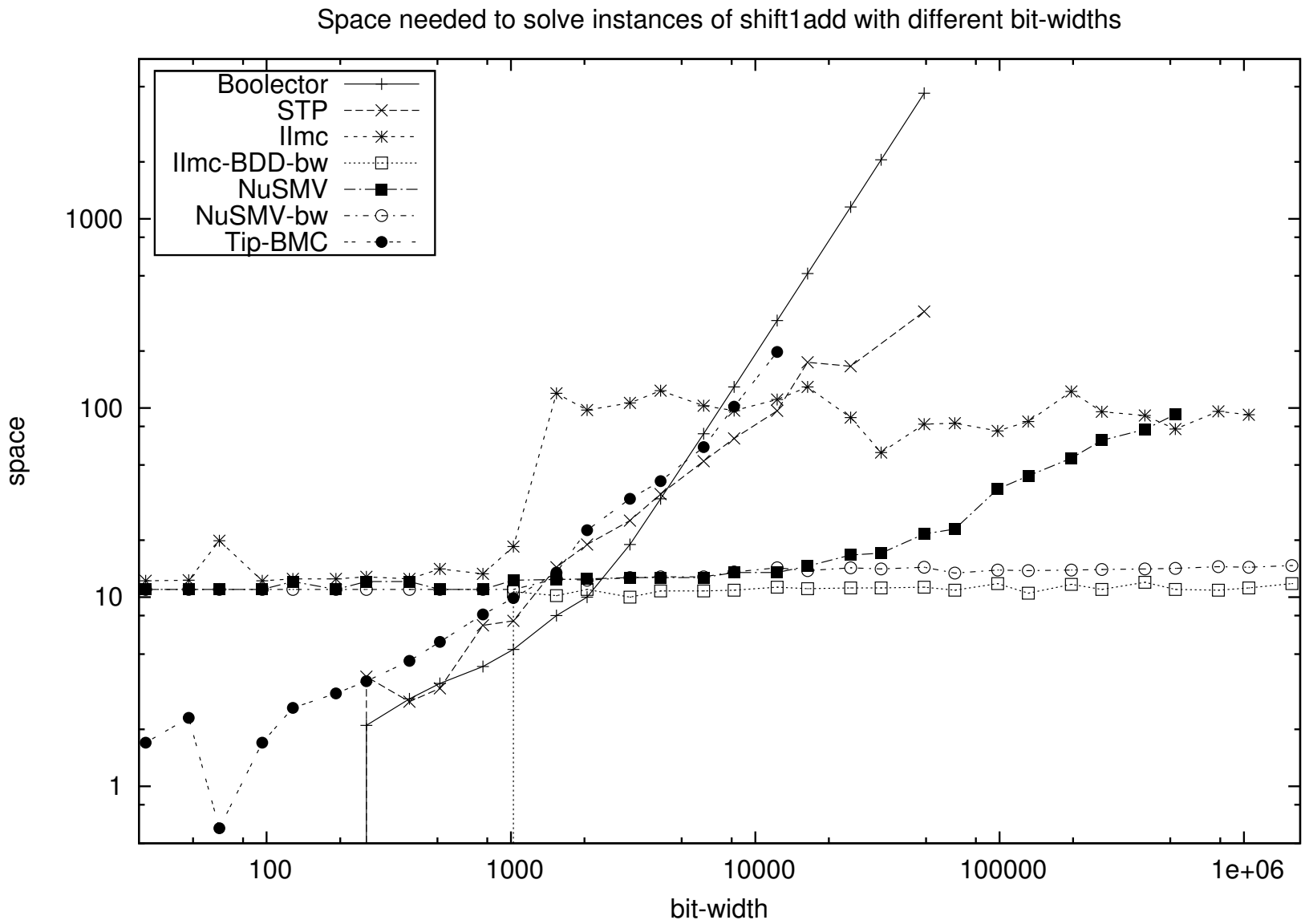
	STP	Boolector	MathSAT5	Z3	IImc-BDD-bw	NuSMV-bw	IImc-BDD-fw	IImc	NuSMV	Blimc	Tip-BMC	Aigbmc	Tip
solved	147	146	127	123	**192**	189	185	172	170	147	130	99	93
sat	23	**32**	13	23	**32**	29	**32**	**32**	27	9	31	21	17
unsat	124	114	114	100	**160**	**160**	153	140	143	138	99	78	76
time	206	190	310	171	12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

Table : Overall results for 13 solvers

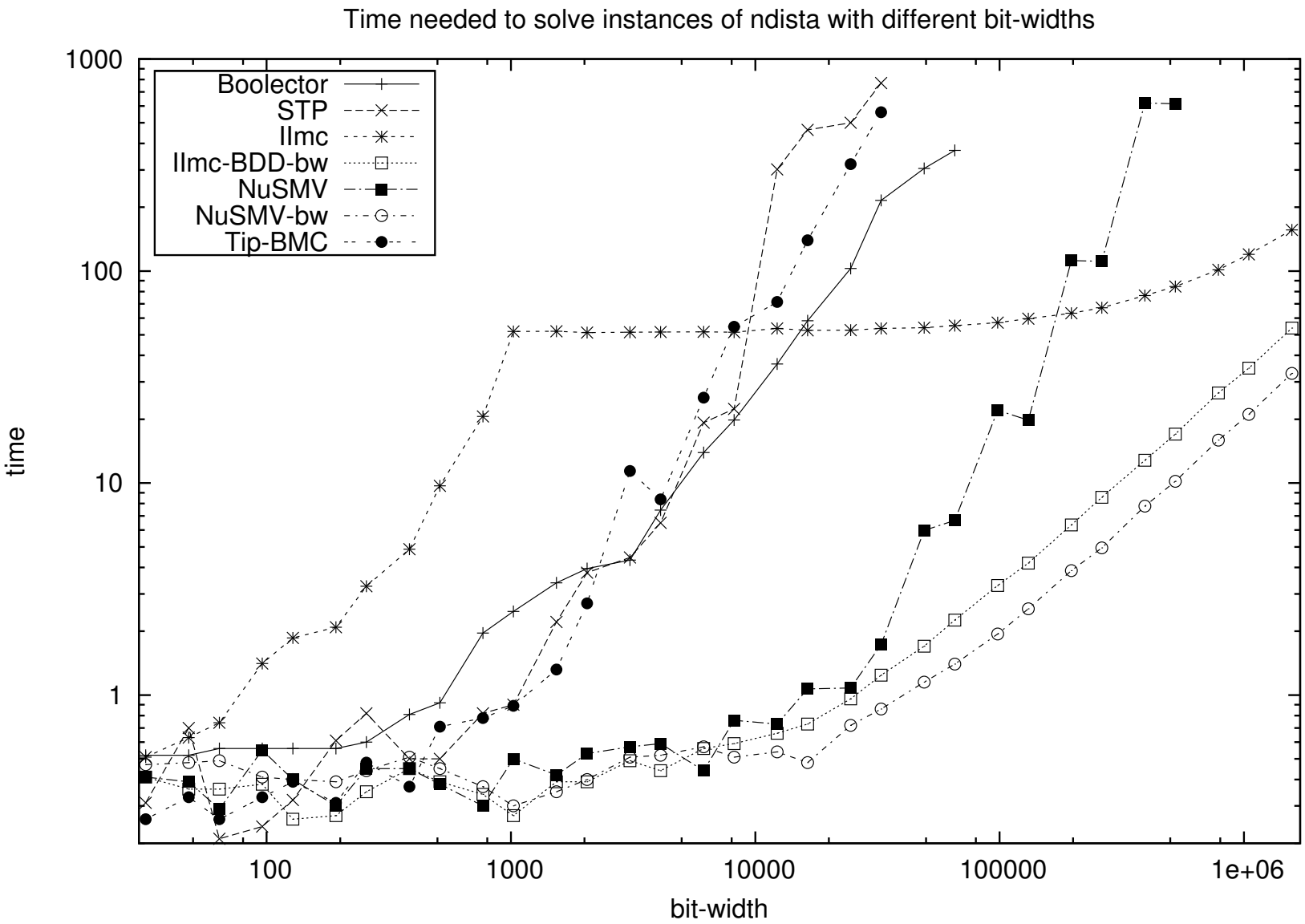
Experimental Results: shift1add - Time



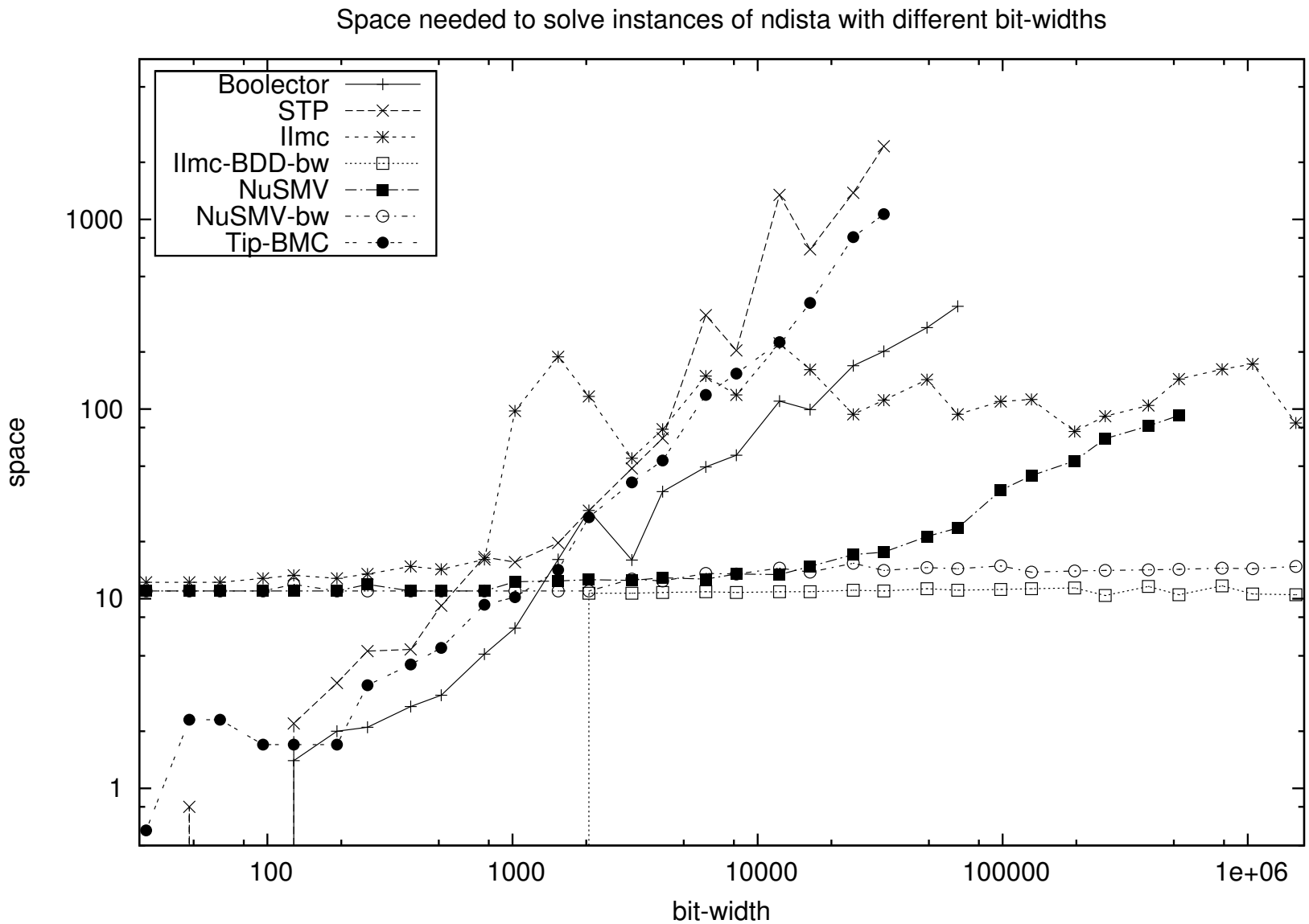
Experimental Results: shift1add - Space



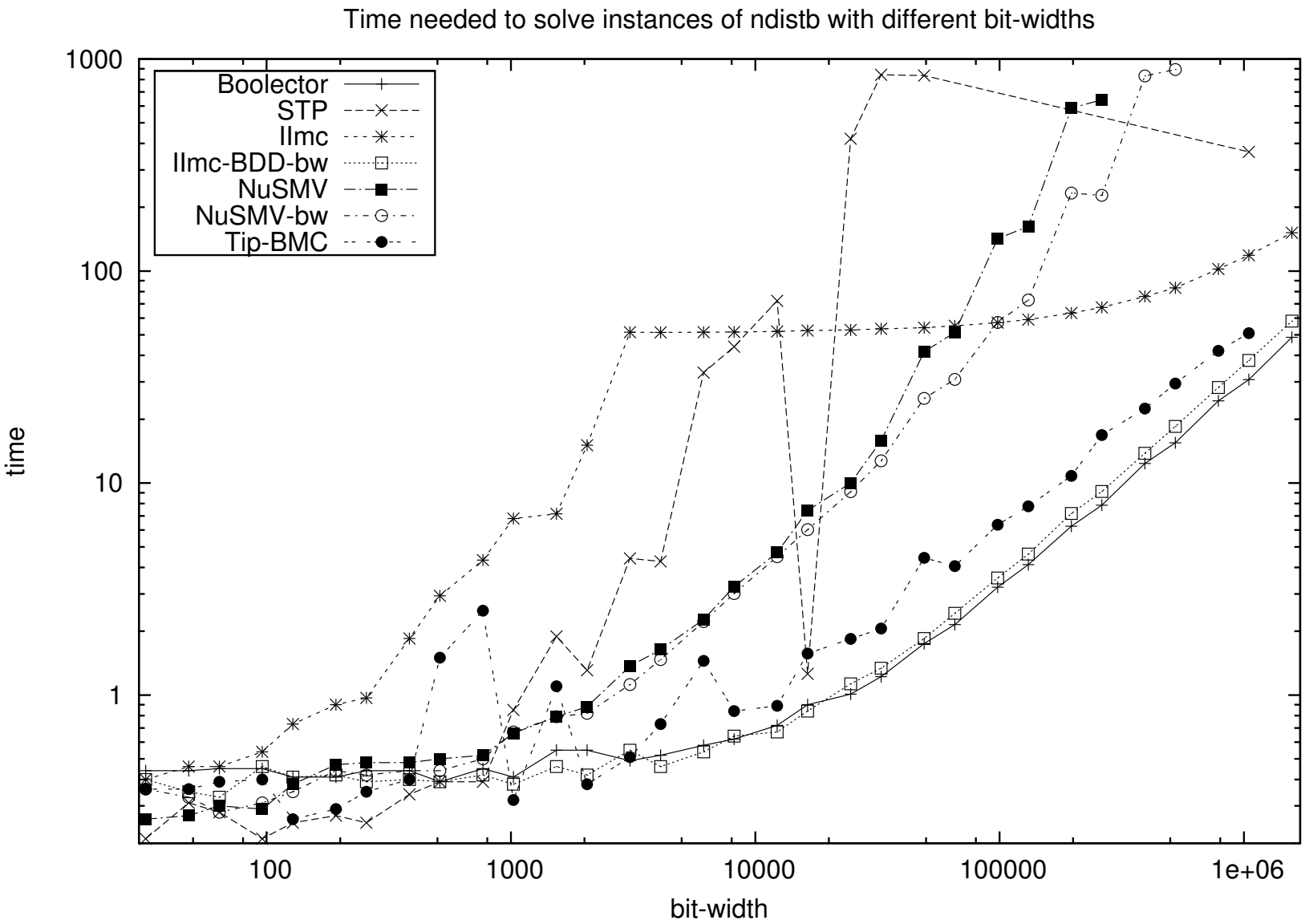
Experimental Results: ndist.a - Time



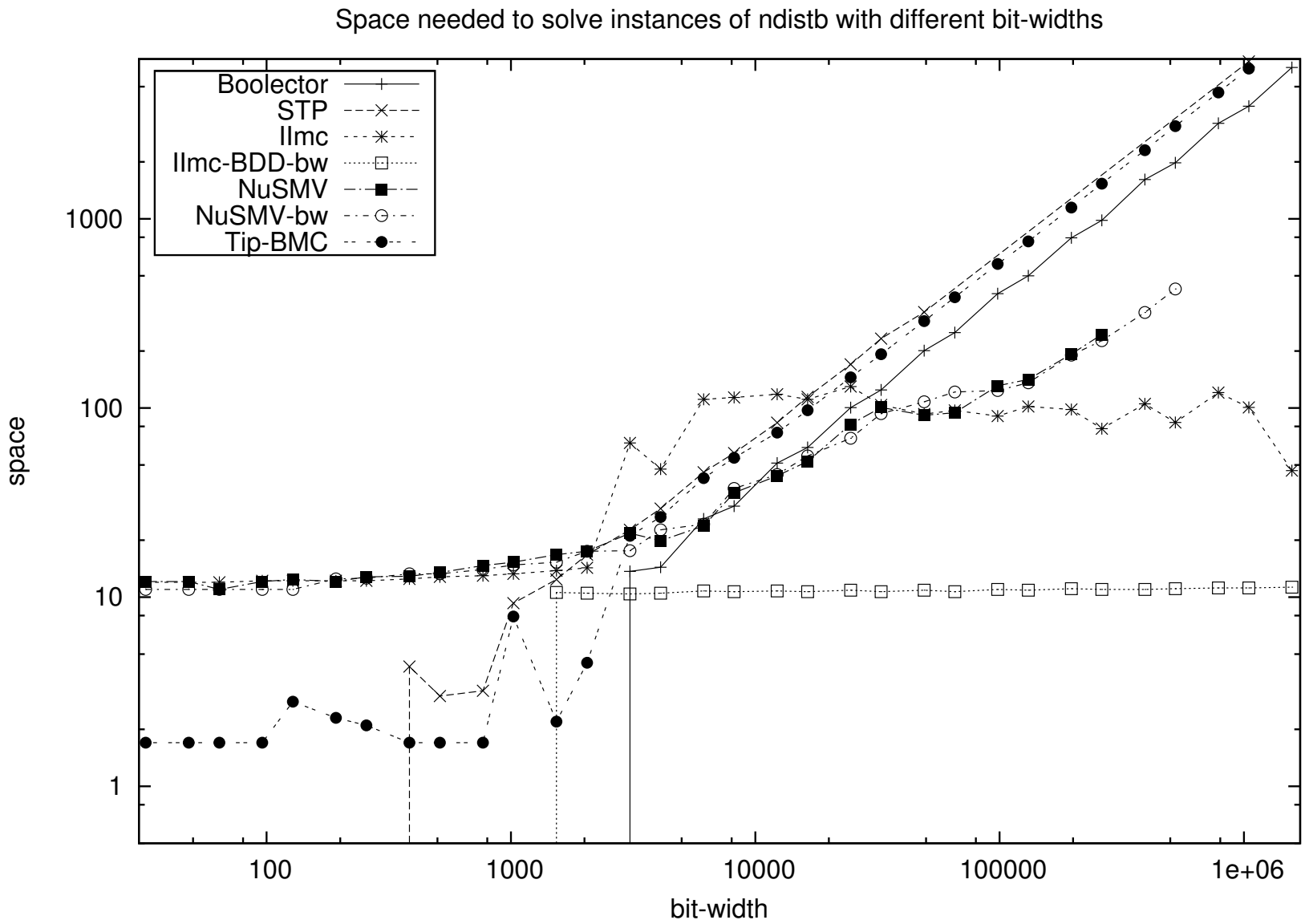
Experimental Results: ndist.a - Space



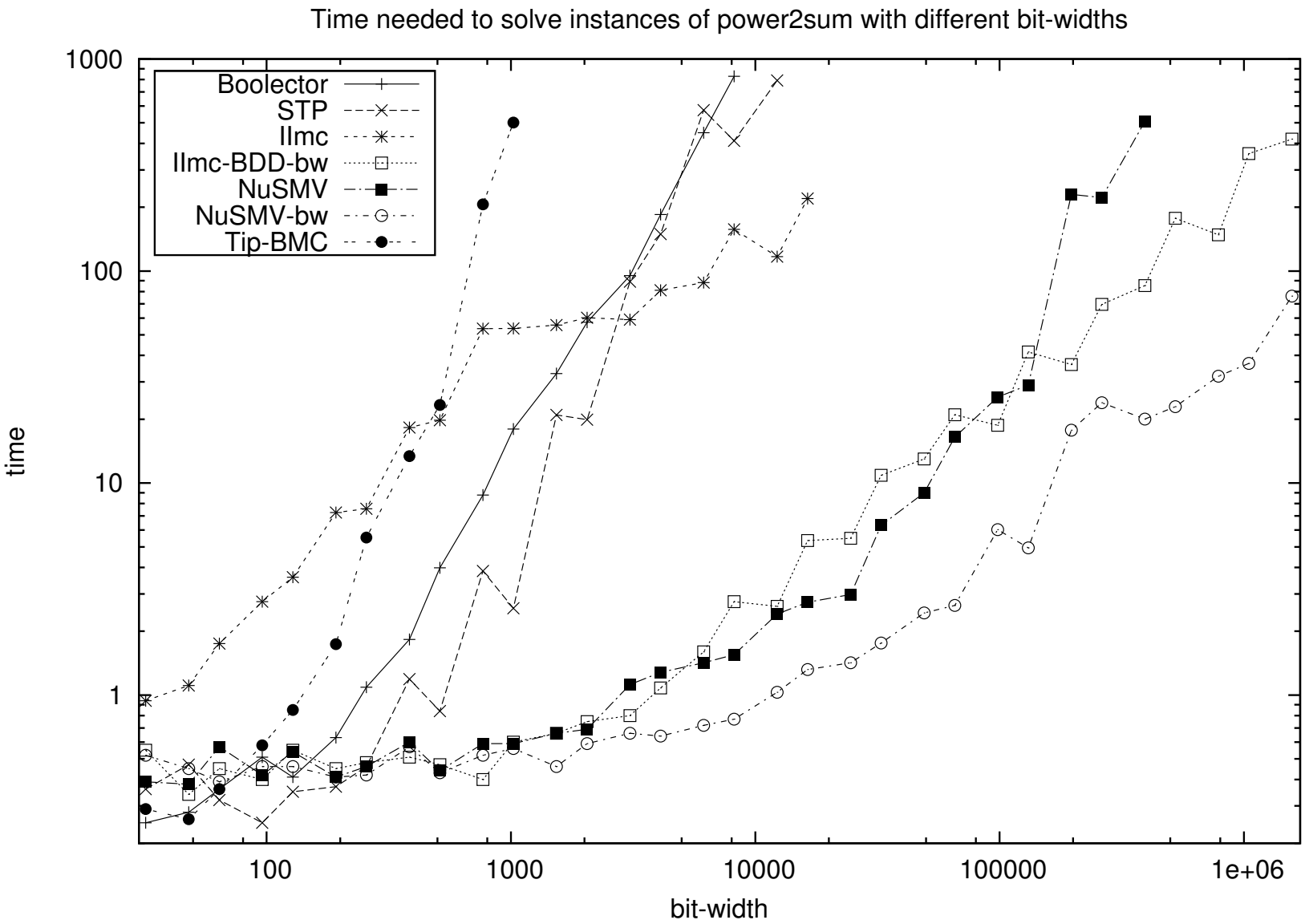
Experimental Results: ndist.b - Time



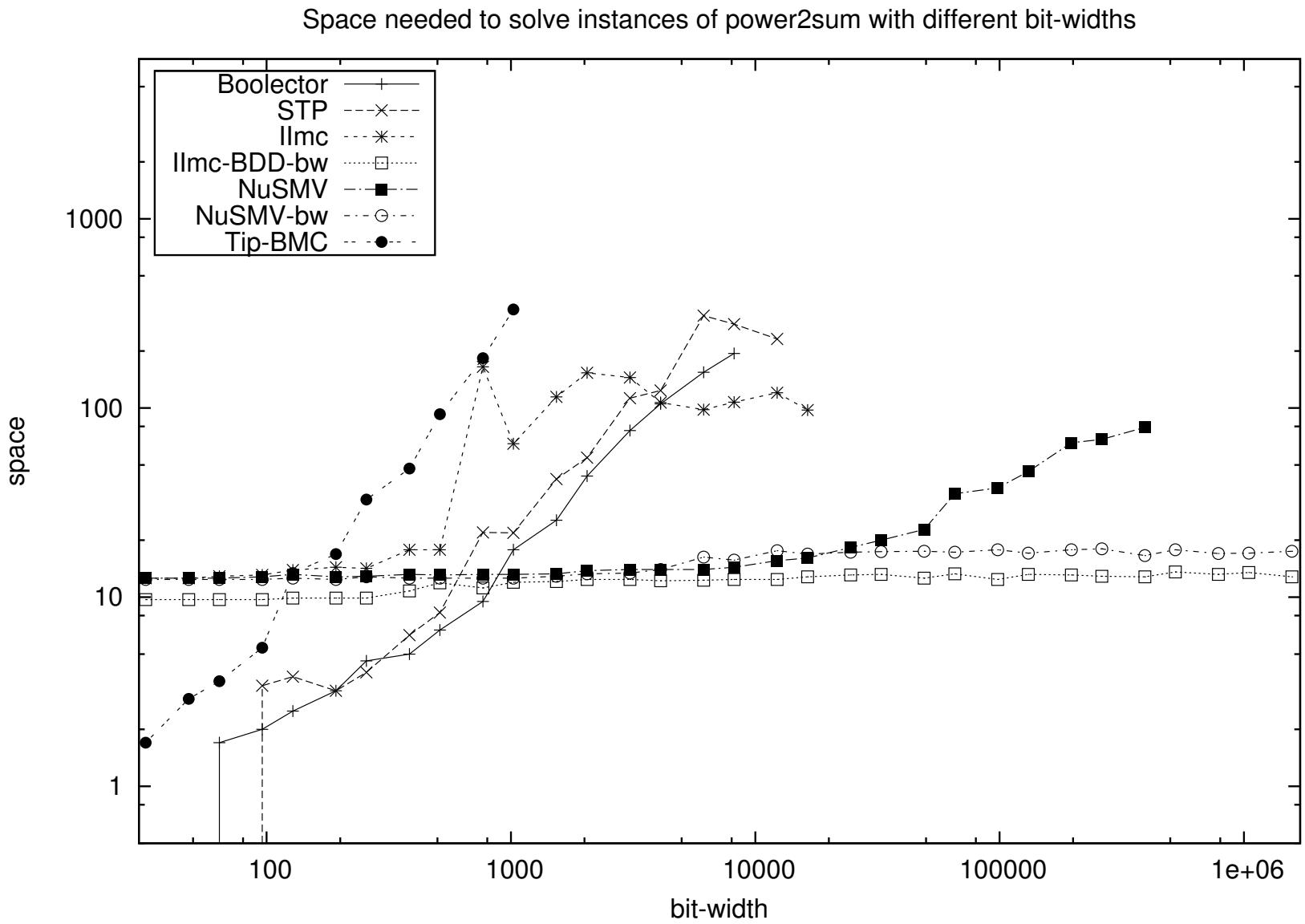
Experimental Results: ndist.b - Space



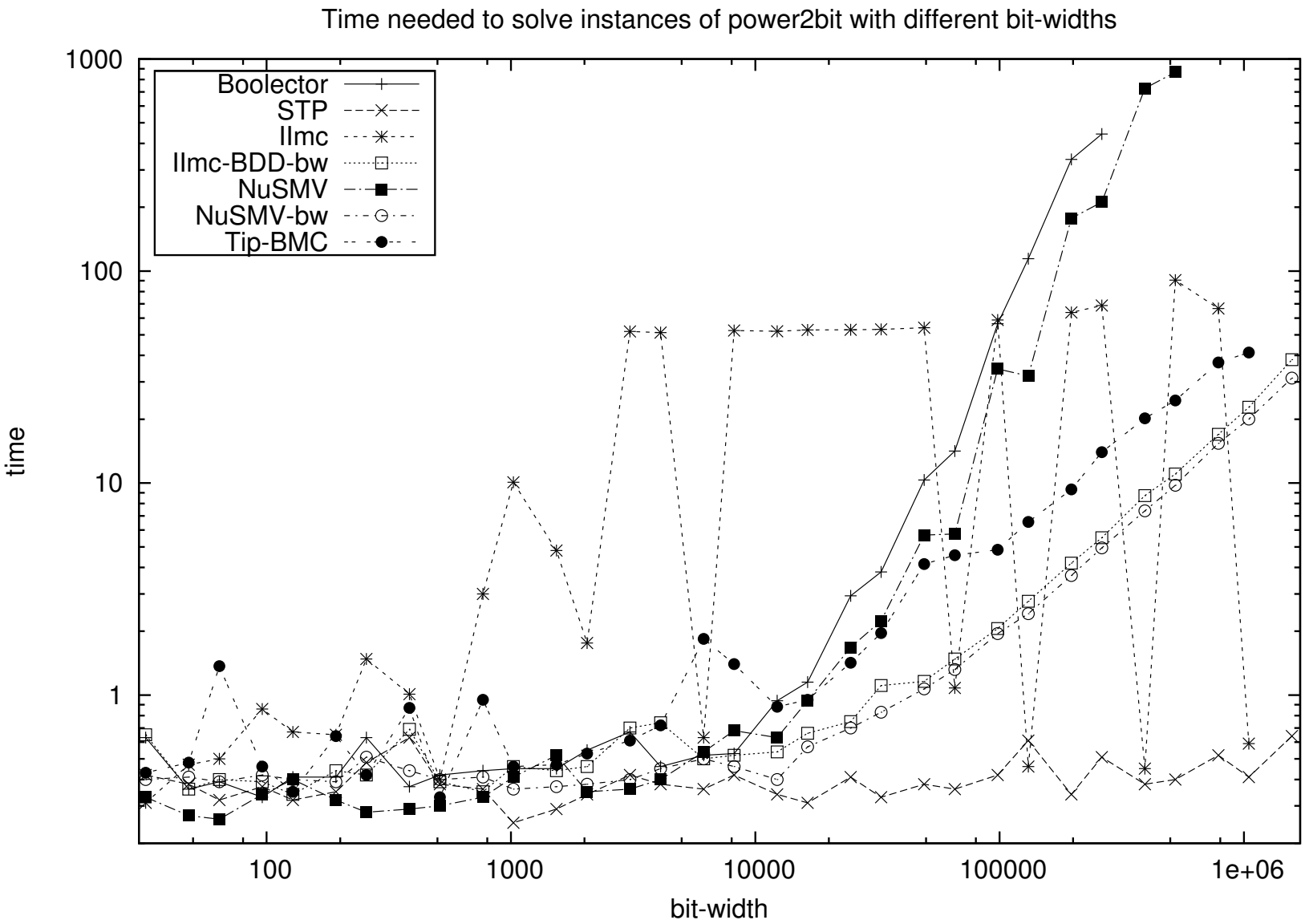
Experimental Results: power2sum - Time



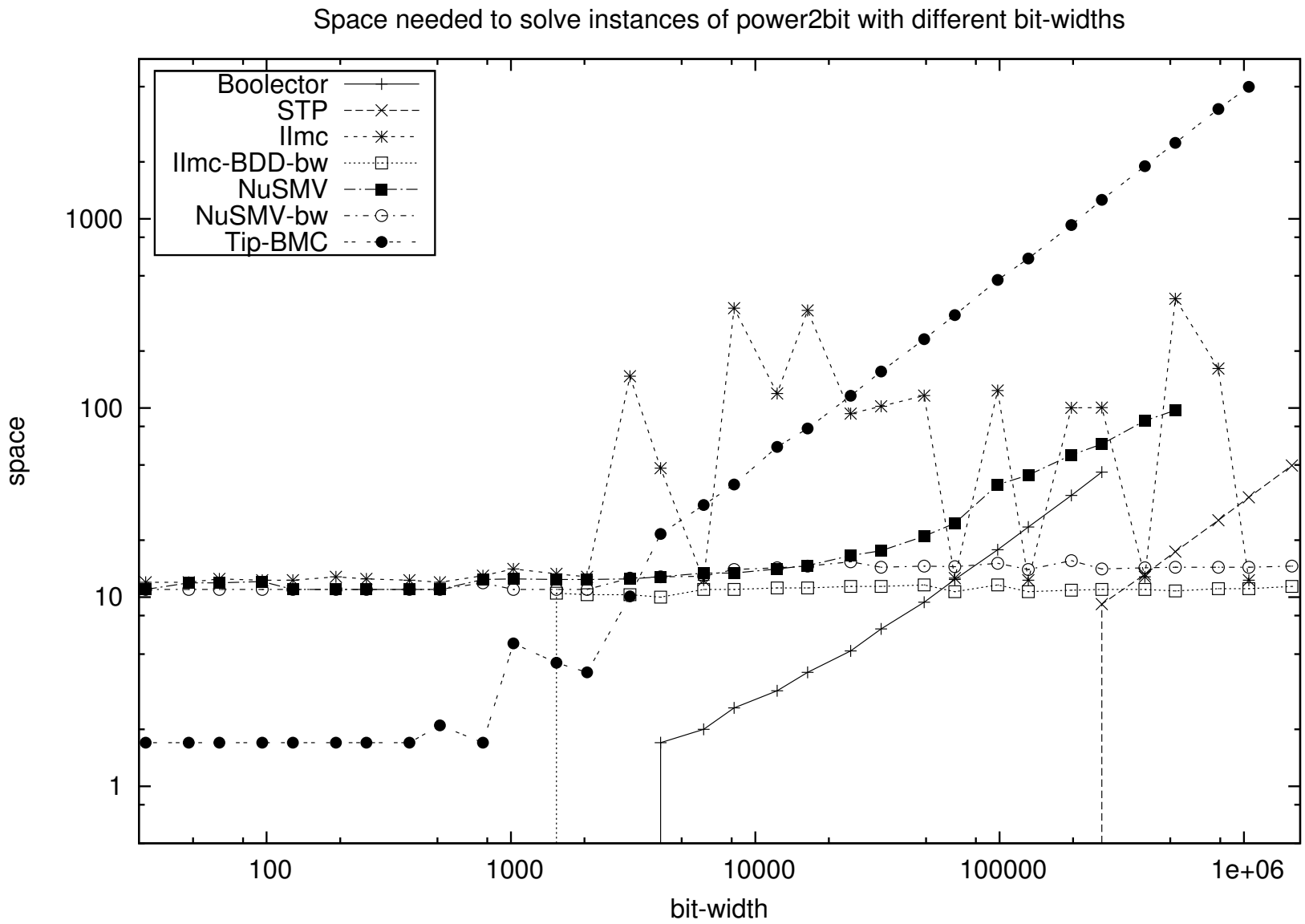
Experimental Results: power2sum - Space



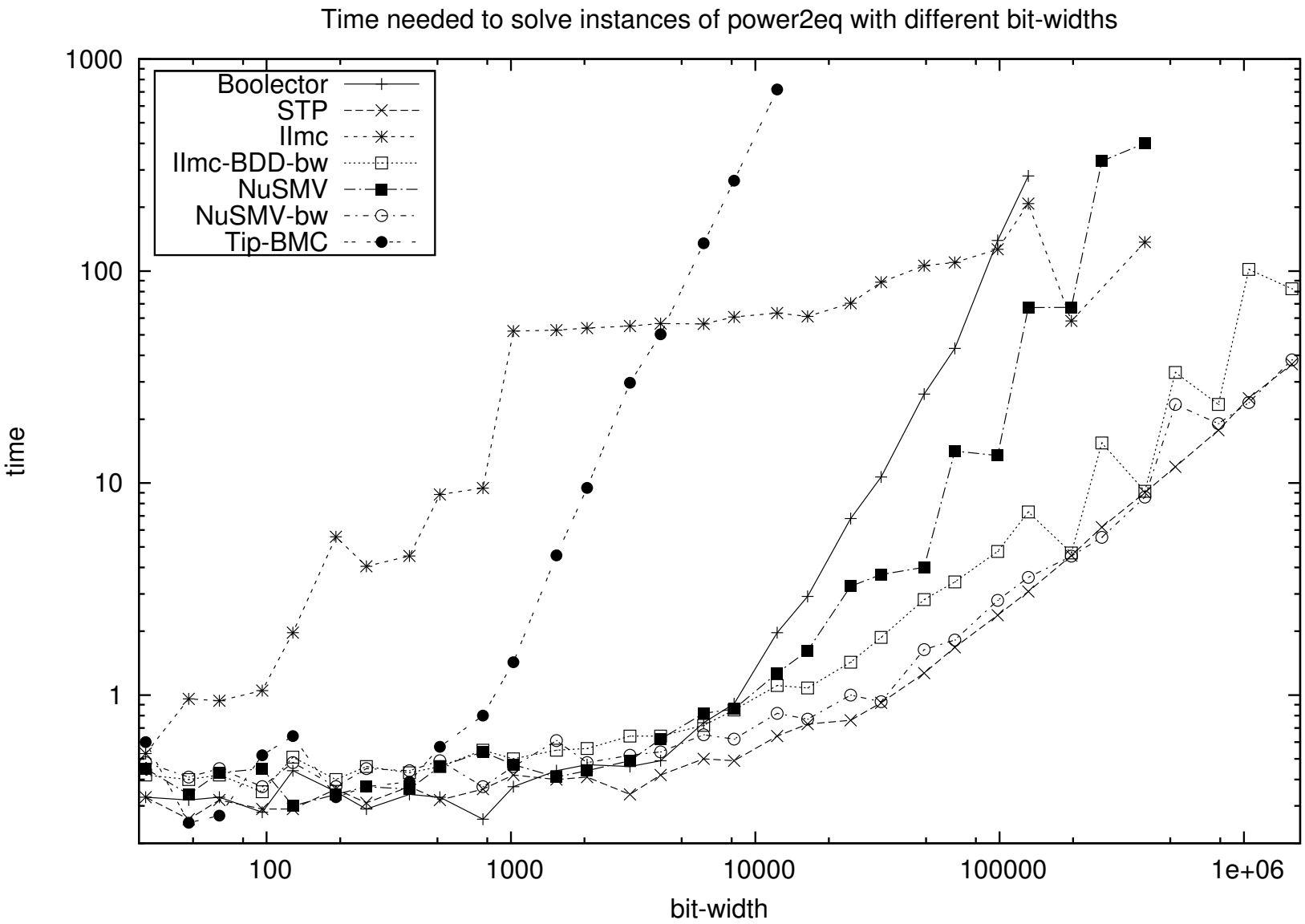
Experimental Results: power2bit - Time



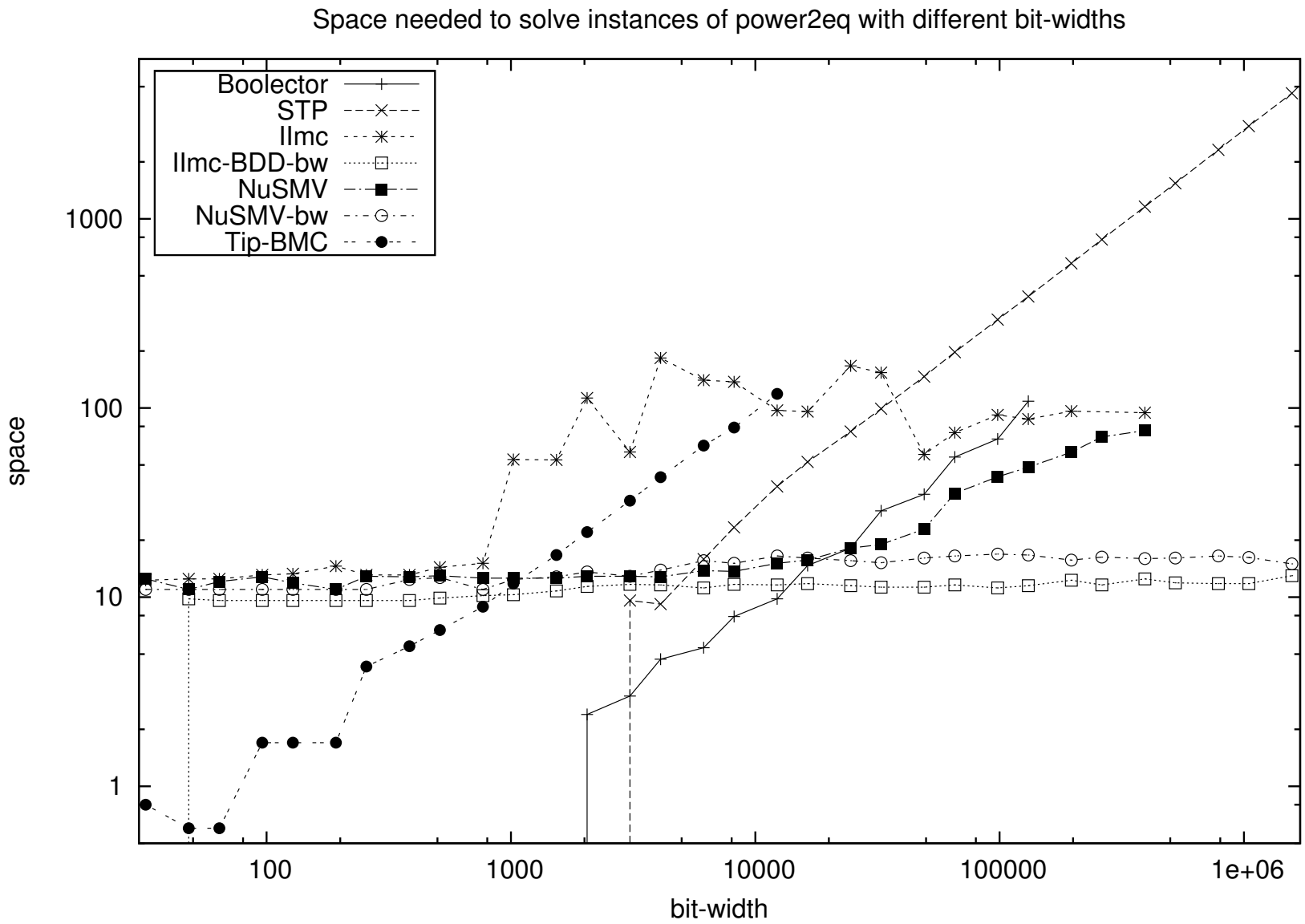
Experimental Results: power2bit - Space



Experimental Results: power2eq - Time



Experimental Results: power2eq - Space



Conclusion

Results:

- Translation $\mathrm{QF\_BV}_{\ll 1} \xrightarrow{\text{polynomially}}$ SMV
- Crafted benchmarks solved efficiently by model checkers.
- Space requirement reduced even more significantly.
- Best performance by using BDD backward reachability.

Future Work:

- Is it possible to solve industrial benchmarks more efficiently with our approach?
- Can SMT solvers profit from techniques used in model checkers?
- Is Presburger arithmetic on fixed-size bit-vectors still NP-complete?