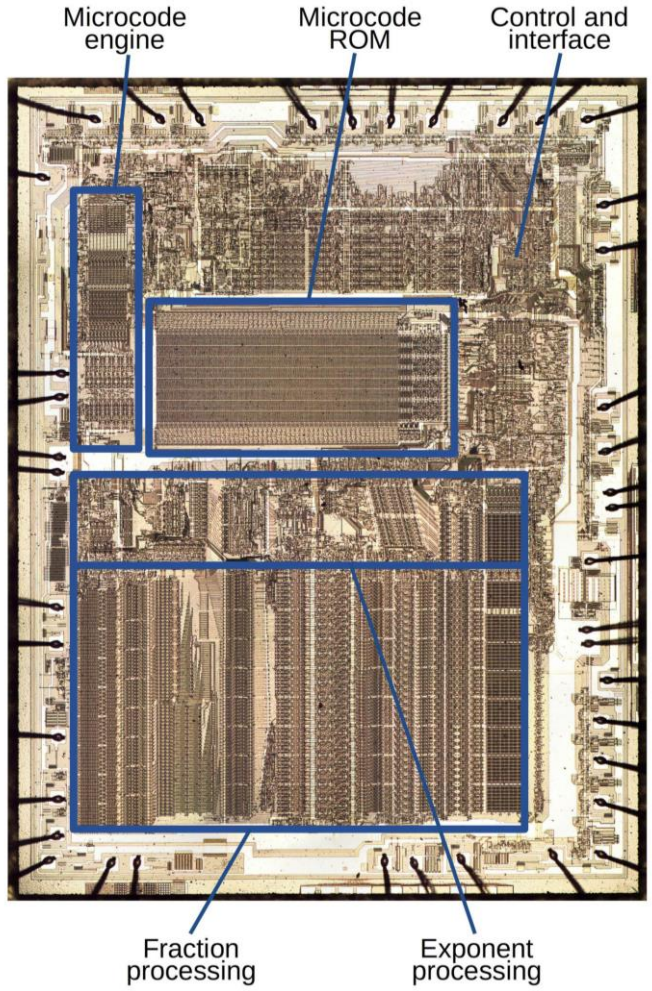
An Efficient Floating-Point Bit-Blasting API for Verifying C Programs

Mikhail R. Gadelha, Lucas C. Cordeiro, Denis A. Nicole

mikhail.gadelha@sidia.com

Motivation

- To prevent bugs!
- Ariane 5 rocket exploded mid-air in 1996 due to an exception thrown by an invalid floating-point conversion



Die photo of the Intel 8087 floating-point chip.

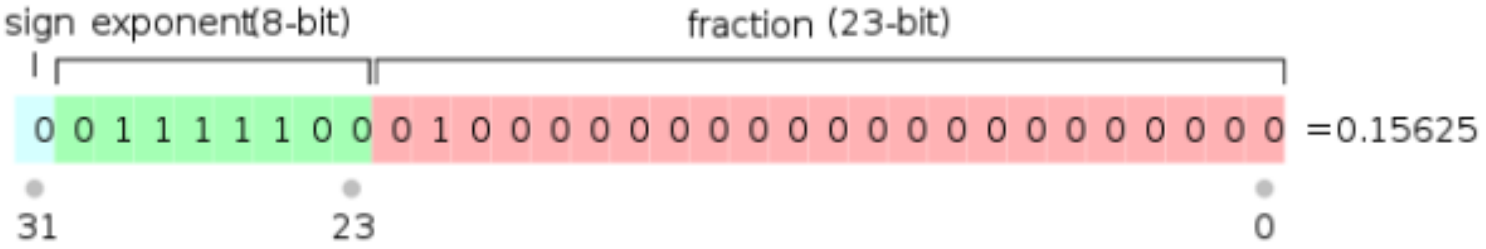
Source: Ken Shirriff

Floating-points

- The set of real numbers is uncountable, and some numbers cannot be represented with finite precision
- In particular, the IEEE 754 floating-point standard is used by many processors

IEEE Floating-points

- Floating-points are divided into three parts: one bit for the sign, an exponent, and a significant part which depends on the bit length of the type
- Floating-points are represented as $(-1)^{sgn} \times sig \times base^{exp}$



The number 0.15625 as a single-precision floating-point.
Source: wikipedia.org

IEEE Floating-points

- Five kinds:
  - ± infinity
  - ± zero
  - NaN (not a number)
  - Normal
  - Denormal or subnormal

IEEE Floating-points

- Five exceptions:
  - Invalid operation
  - Overflow
  - Division by zero
  - Underflow
  - Inexact

IEEE Floating-points

- Five rounding modes:
  - Round Toward Positive (RTP)
  - Round Toward Negative (RTN)
  - Round Toward Zero (RTZ)
  - Round to Nearest ties to Even (RNE)
  - Round to Nearest ties Away from zero (RNA)

```
1 int main()
2 {
3   double x = 0.1;
4   double y = 0.2;
5   double w = 0.3;
6   double z = x + y;
7   assert(w == z);
8   return 0;
9 }
```

Floating-points in C programs

- Famous floating-point “issue”
- Assertion on line 7 does not hold if the program is encoded using radix-2 floating-point arithmetic

```
1 int main()
2 {
3   double x = 0.1;
4   double y = 0.2;
5   double w = 0.3;
6   double z = x + y;
7   assert(w == z);
8   return 0;
9 }
```

x = 0.1000000000000000055511151231257827021181583404541015625;

Floating-points in C programs

- Famous floating-point “issue”
- Assertion on line 7 does not hold if the program is encoded using radix-2 floating-point arithmetic

```
1 int main()
2 {
3   double x = 0.1;
4   double y = 0.2;
5   double w = 0.3;
6   double z = x + y;
7   assert(w == z);
8   return 0;
9 }
```

x = 0.1000000000000000055511151231257827021181583404541015625;

y = 0.2000000000000000111022302462515654042363166809082031250;

Floating-points in C programs

- Famous floating-point “issue”
- Assertion on line 7 does not hold if the program is encoded using radix-2 floating-point arithmetic

```
1 int main()
2 {
3   double x = 0.1;
4   double y = 0.2;
5   double w = 0.3;
6   double z = x + y;
7   assert(w == z);
8   return 0;
9 }
```

Floating-points in C programs

x = 0.1000000000000000055511151231257827021181583404541015625;

y = 0.2000000000000000111022302462515654042363166809082031250;

w = 0.2999999999999999888977697537484345957636831909179687500;

- Assertion on line 7 does not hold if the program is encoded using radix-2 floating-point arithmetic

```
1 int main()
2 {
3   double x = 0.1000000000000000055511151231257827021181583404541015625;
4   double y = 0.2000000000000000111022302462515654042363166809082031250;
5   double w = 0.2999999999999999888977697537484345957636831909179687500;
6   double z = 0.3000000000000000044408920985006261616945266723632812500;
7   assert(w == z);
8   return 0;
9 }
```

Floating-points in C programs

- Assertion on line 7 does not hold if the program is encoded using radix-2 floating-point arithmetic

```
1 int main()
2 {
3   double x =
4   double y =
5   double w =
6   double z =
7   assert(w == z);
8   return 0;
9 }
```

x = 0.1000000000000000055511151231257827021181583404541015625;

y = 0.2000000000000000111022302462515654042363166809082031250;

w = 0.2999999999999999888977697537484345957636831909179687500;

z = 0.3000000000000000044408920985006261616945266723632812500;

w < z

Floating-points in C programs

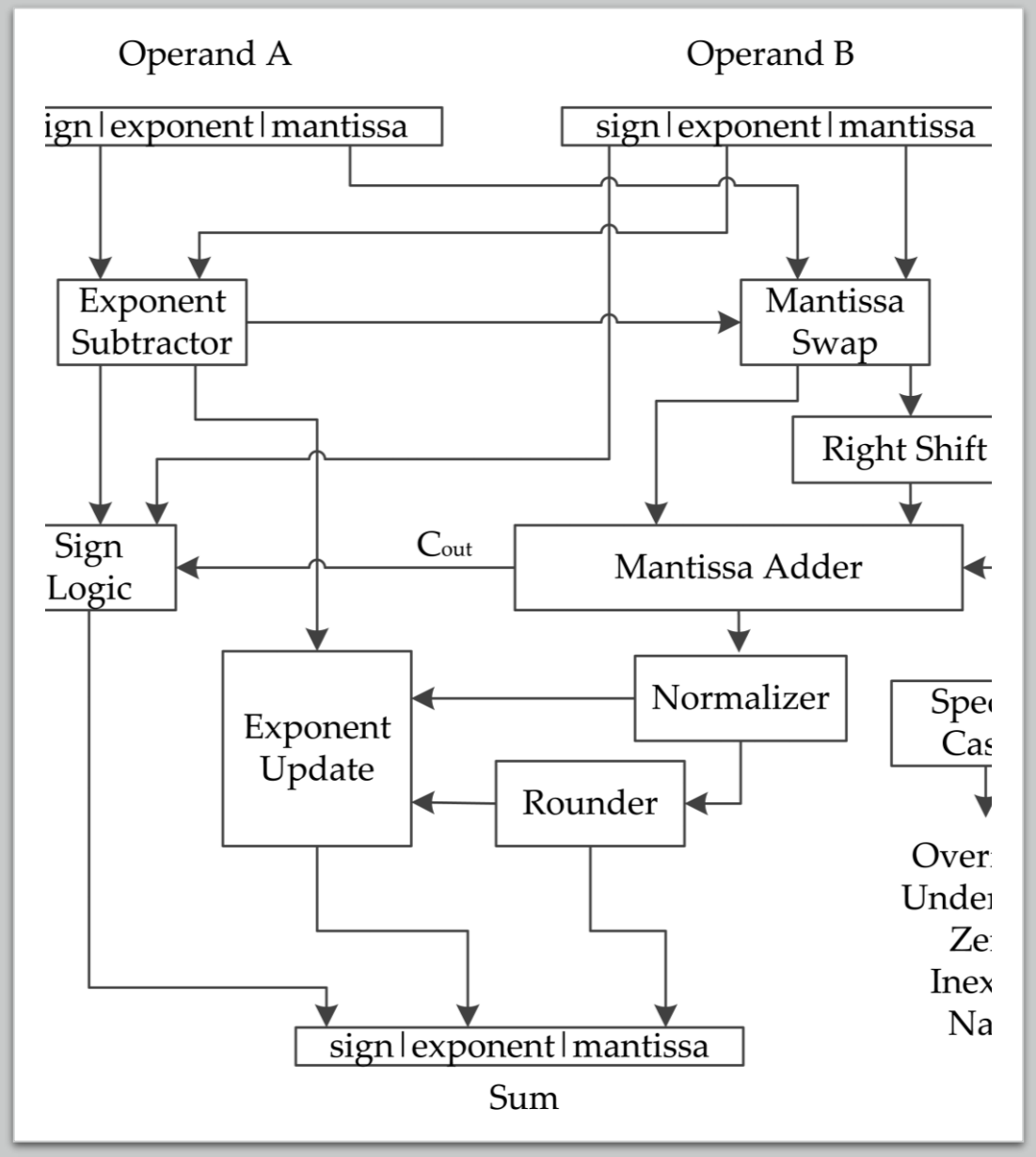
- Assertion on line 7 does not hold if the program is encoded using radix-2 floating-point arithmetic

SMT Floating-point Logic

- The SMT FP logic is an addition to the SMT standard, first proposed in 2010 by Rümmer and Wahl
- The current version of the theory largely follows the IEEE standard 754. It formalizes floating-point arithmetic, ±infinity and ±zero, NaNs, relational and arithmetic operators, and five rounding modes: RNE, RNA, RTP, RNP and RTZ.

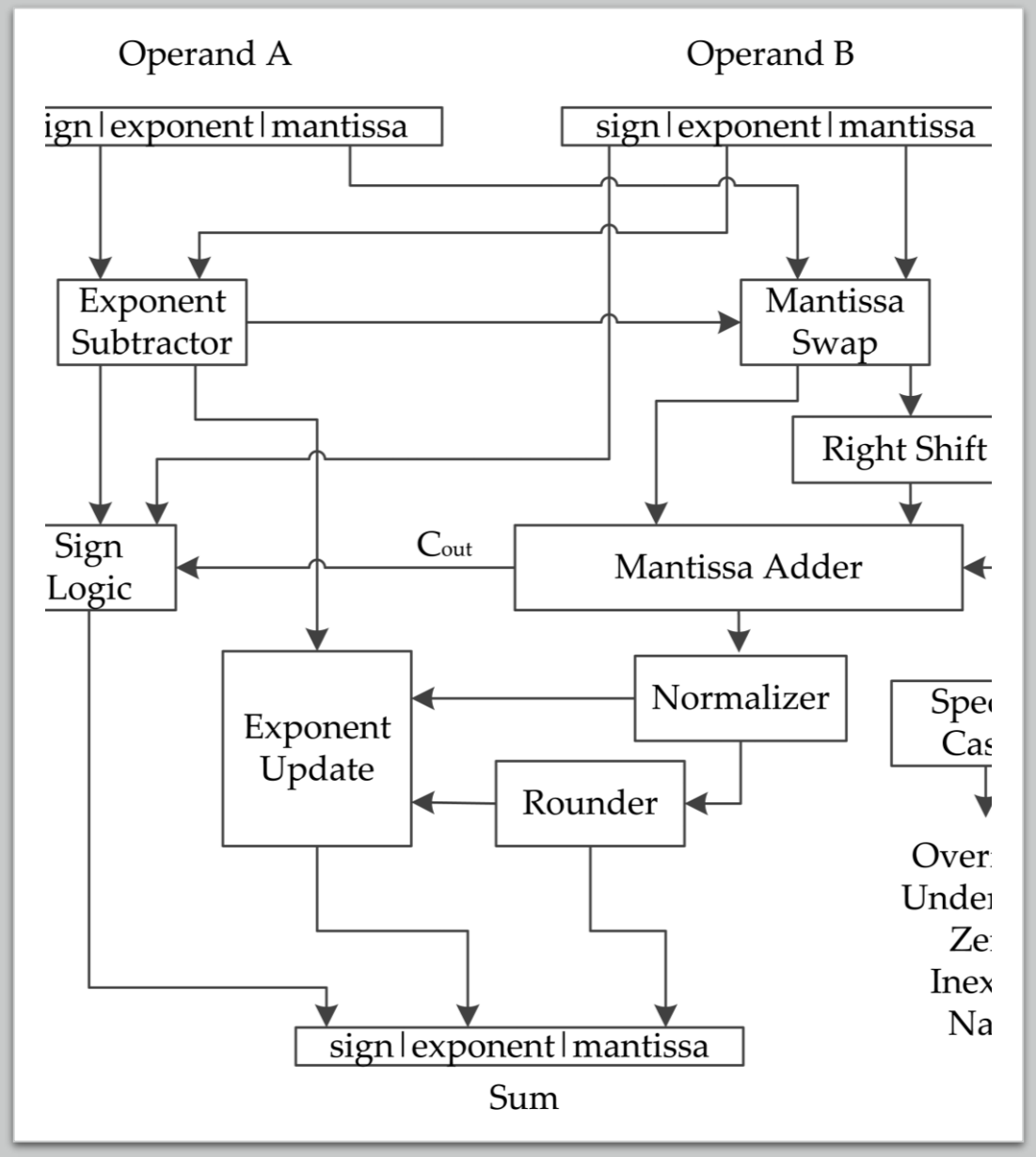
SMT Floating-point Logic

- Fully implemented by Z3, CVC4, Colibri, Solonar, UppSAT
- MathSAT partially implements it: no fp.rem (remainder operator) and no fp.fma (fused multiply-add)
- Non-standard calls to functions to reinterpret floating-points to and from bit-vectors are also implemented in the solvers



Bit-blasting Floating-point Arithmetic

- Usual four stage pipeline:
  - Unpack
  - Operate
  - Round
  - Pack



Bit-blasting Floating-point Arithmetic

- Seven operation groups:
  - Sort constructors
  - Rounding mode constructors
  - Value constructors
  - Classification operations
  - Comparison operations
  - Conversion operations
  - Arithmetic operations

Name	Common Name	Size (exponent + significand)
fp16	Half precision	16 (5 + 10)
fp32	Single precision	32 (8 + 23)
fp64	Double precision	64 (11 + 52)
fp128	Quadruple precision	128 (15 + 112)

Bit-blasting Floating-point Arithmetic

- Sort constructors: supports constructing 16, 32, 64 and 128 bits long floating-points (no support for 80-bit long double extended precision format)
- Rounding mode constructors: supports all five rounding modes even though the C standard does not support RNA; these are encoded as 3-bits long bit-vectors

Name	Common Name	Size (exponent + significand)
fp16	Half precision	16 (5 + 10)
fp32	Single precision	32 (8 + 23)
fp64	Double precision	64 (11 + 52)
fp128	Quadruple precision	128 (15 + 112)

Bit-blasting Floating-point Arithmetic

- Value constructors: Floating-point literals, ±infinity, ±zero and NaN can be created
- NaN are always created using the same bit-pattern (exponent all 1, significand is 000…01)
- Different from the standard, we support negative NaNs

Bit-blasting Floating-point Arithmetic

- Classification operators: Algorithms to classify normals, subnormals, zeros (regardless of sign), infinities (regardless of sign), NaNs, and negatives and positives.

- Comparison operators: The operators “greater than or equal to”, “greater than”, “less than or equal to”, “less than”, and “equality” are supported.

Bit-blasting Floating-point Arithmetic

- Conversion operators:
  - Floating-points to signed bit-vectors and floating-points to unsigned bit-vectors
  - Floating-points to another floating-point*
  - Signed bit-vectors to floating-points and unsigned bit-vectors to floating-points

* Different from the standard, we preserve NaN sign in these operations

Bit-blasting Floating-point Arithmetic

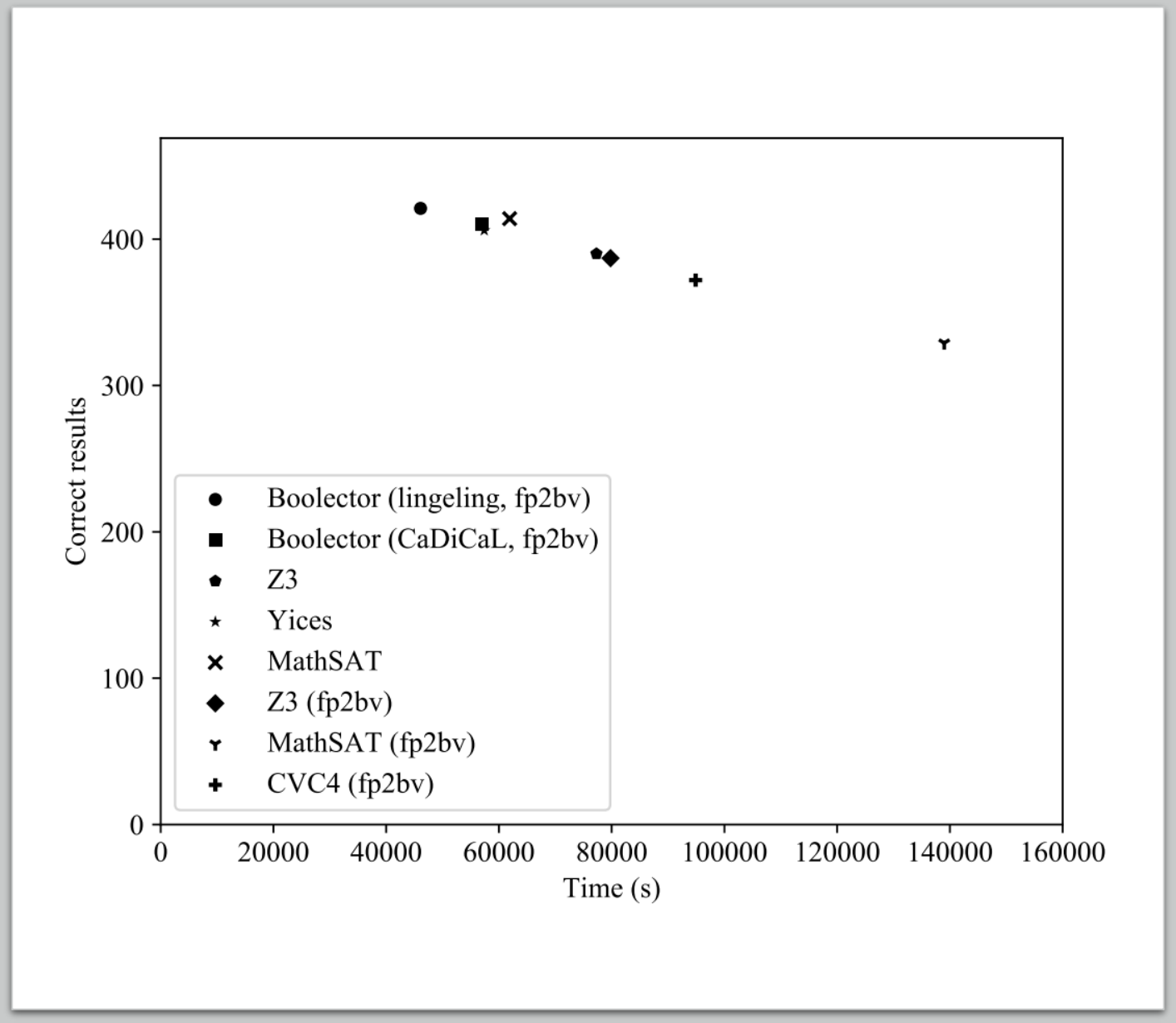
- Arithmetic operators:
  - Absolute value*
  - Negation*
  - Addition
  - Subtraction
  - Multiplication
  - Division
  - Fused multiply-add
  - Square root

* These operations handle the NaN sign accordingly (non-standard).

Experimental Evaluation

- First, we compare the verification results of 466 benchmarks of the sub-category ReachSafety-Floats from SV-COMP 2020.

- The programs are verified using ESBMC and the following solvers (fp2bv is our bit-blasting API):
  - Boolector (lingeling, fp2bv)
  - Boolector (CaDiCaL, fp2bv)
  - Z3
  - Yices (fp2bv)
  - MathSAT
  - Z3 (fp2bv)
  - MathSAT (fp2bv)
  - CVC4 (fp2bv)



Experimental Evaluation

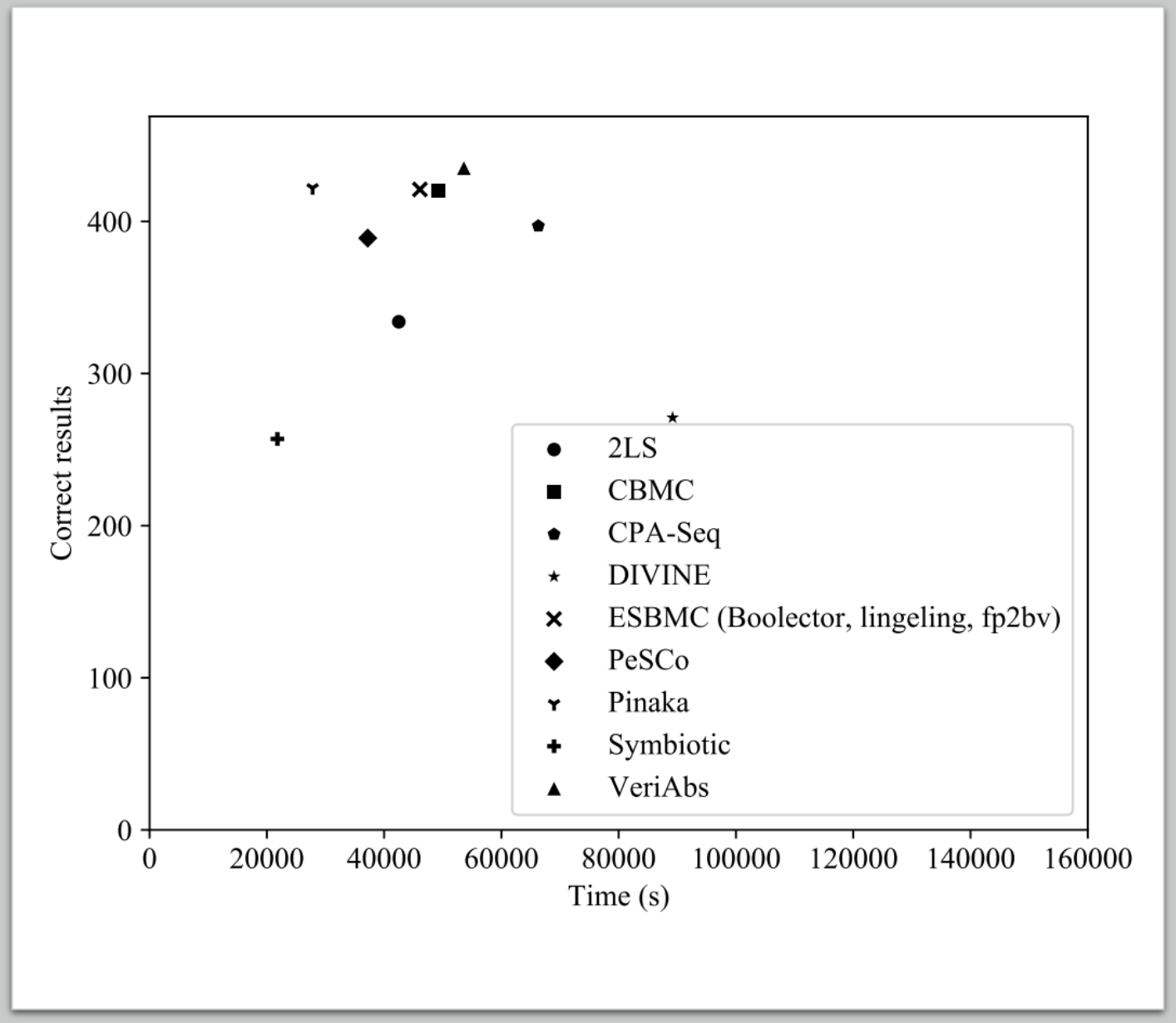
- Boolector (lingeling, fp2bv) reports the highest number of correct results (421), followed by MathSAT using their native floating-point API (414)

- Z3 with its native floating-point API and Z3 with our fp2bv API produce very similar results: 390 and 387, respectively. Our fp2bv API is based on the bit-blasting performed by Z3

ESBMC produced no incorrect result in this evaluation: although we can not formally prove that our algorithm is sound and complete, empirical evidence suggests it.

Experimental Evaluation

- We compare the implementation of our floating-point API with other software verifiers from SV-COMP 2020:
  - 2LS
  - CBMC
  - CPA-Seq
  - DIVINE
  - PeSCo
  - Pinaka
  - Symbiotic
  - VeriAbs



Experimental Evaluation

- Overall number of correct results and verification time:
  - VeriAbs 435 in 53600s
  - Pinaka 422 with 27800s
  - ESBMC 421 with 46100s

Our floating-point API is on par with other state-of-the-art tools. VeriAbs and Pinaka implement several heuristics to simplify the check for satisfiability using CBMC, while ESBMC using an incremental approach produced close results. ESBMC was also slightly faster and provided a few more results than CBMC, which lead us to believe that our tool would also greatly benefit VeriAbs and Pinaka if used as backend.

The Future of fp2bv: libcamada

- Given the great results we achieved using fp2bv in ESBMC, we decided to decouple it from the verifier.
- Work-in-progress C++11 libcamada: https://github.com/mikhailramalho/camada
- Version 1.0 to be released late summer
- It will be integrated in LLVM and Klee once the code is stable

Thank you!

mikhail.gadelha@sidia.com