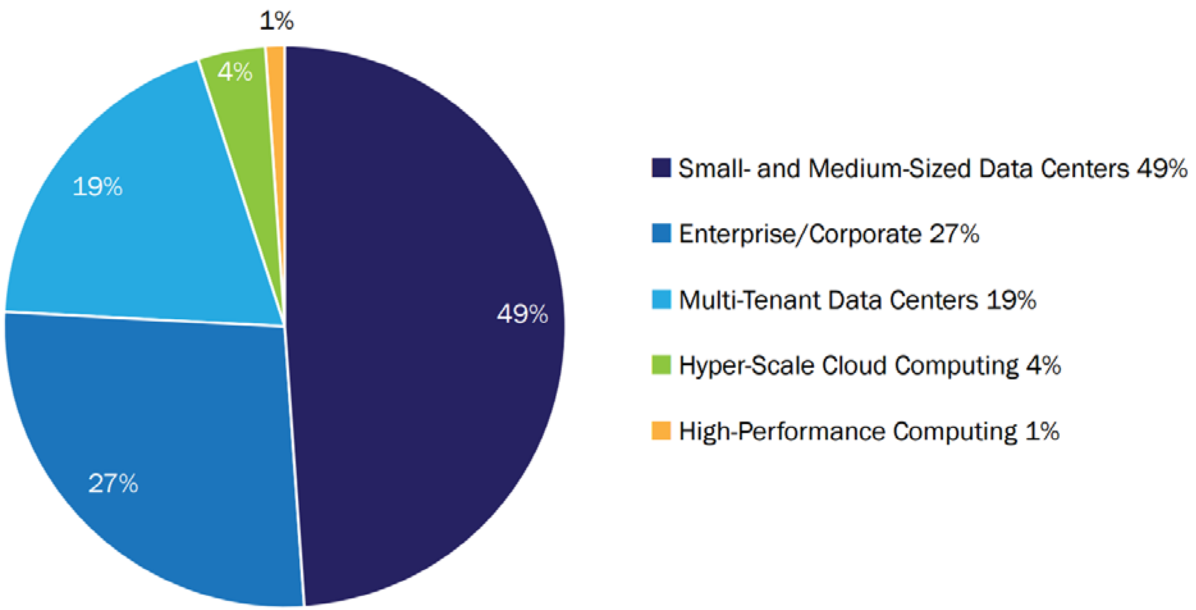
Efficient floating point precision tuning for approximate computing

Nhut-Minh Ho [1], Elavarasi Manogaran [1], Weng-Fai Wong [1] and Asha Anoosheh [2]

[1] National Univ. of Singapore, Singapore
[2] Univ. of California, Berkeley, U.S.A.

Energy concern in computing

Top 500 supercomputers cost ≈$400 million/year for energy consumption:



Estimated U.S. data center electricity consumption by market segment

Other devices

Green computing: "FLOPS-per-Watt"

http://www.green500.org/

Reduce application energy consumption

Error tolerant applications

- Big data analytics
- Media data processing/classification
- Simulations
- Non-critical functions in each program
- …

Approximate Computing

Approximate computing

- Sacrifice **accuracy** for **performance** => also increase energy efficiency
- Various approaches:
  - Hardware:
    - Low-power circuit with uncertainty
    - Fine-grain floating-point bitwidth hardware
  - Software:
    - Task skipping: loop perforation
    - **Floating point precision tuning**

Floating point numbers

- Appear almost in every computer program

```
float a = 999.0;
float b = 0.0001;
for (int i = 0; i < 10000; i++){
        a += b;
}
```

Expected : a = 1000.0

Actual : a ≈ 1000.220703

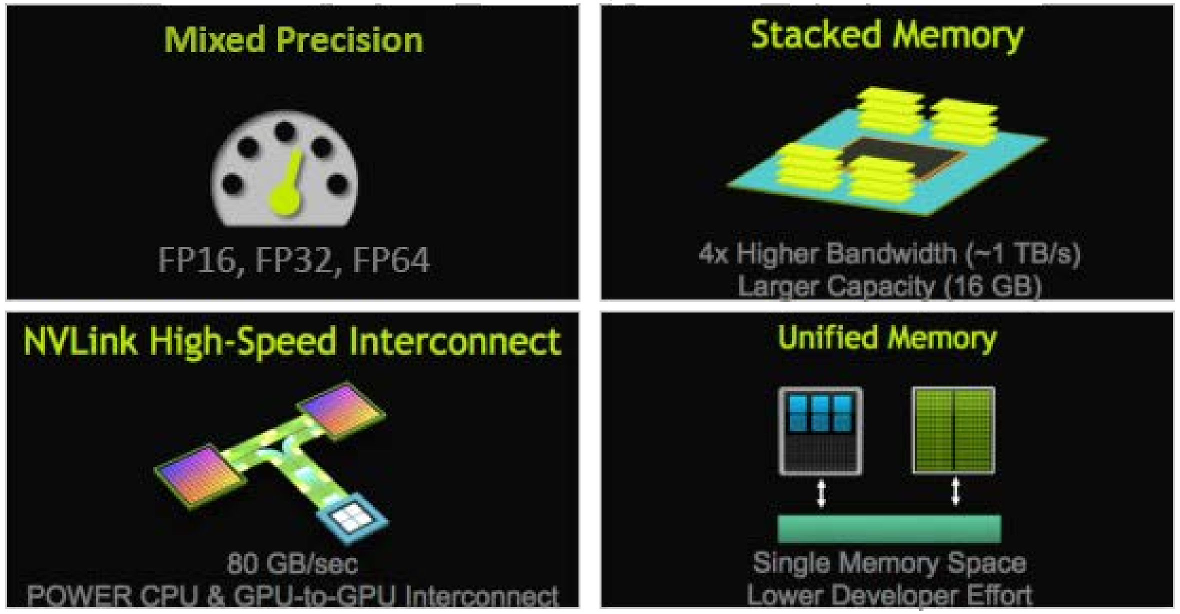
$$\text{Error} \approx \frac{1000.220703 - 1000}{1000} \approx 2.2 \times 10^{-4}$$

Precision tuning, previous work

- Arbitrary-precision fixed point tuning for DSP programs
  - Many techniques: search-based, error analysis based.
  - None of them can scale to real-world floating-point programs nowadays.
- 2-type floating point precision tuning
  - Search for variables can be converted: *double* → *float.*
  - Can analyze real-world applications.
  - Recent works: *Floatwatch* 2007, *Precimonious* 2013, *Enhanced Precimonious* 2016.
  - Cannot work on finer grained precision in different architecture without modification.

2-type floating point is enough ?

- Modern CPU architecture: sufficient.
- FPGA (Field-programmable gate array) prototype showed advantages of using finer grain floating point unit.
- Nvidia's GPU (graphics processing unit) newest architecture supports half-precision.



Motivation

Current techniques for x86 applications:

- Moderately fast
- Limited precision support (float & double)

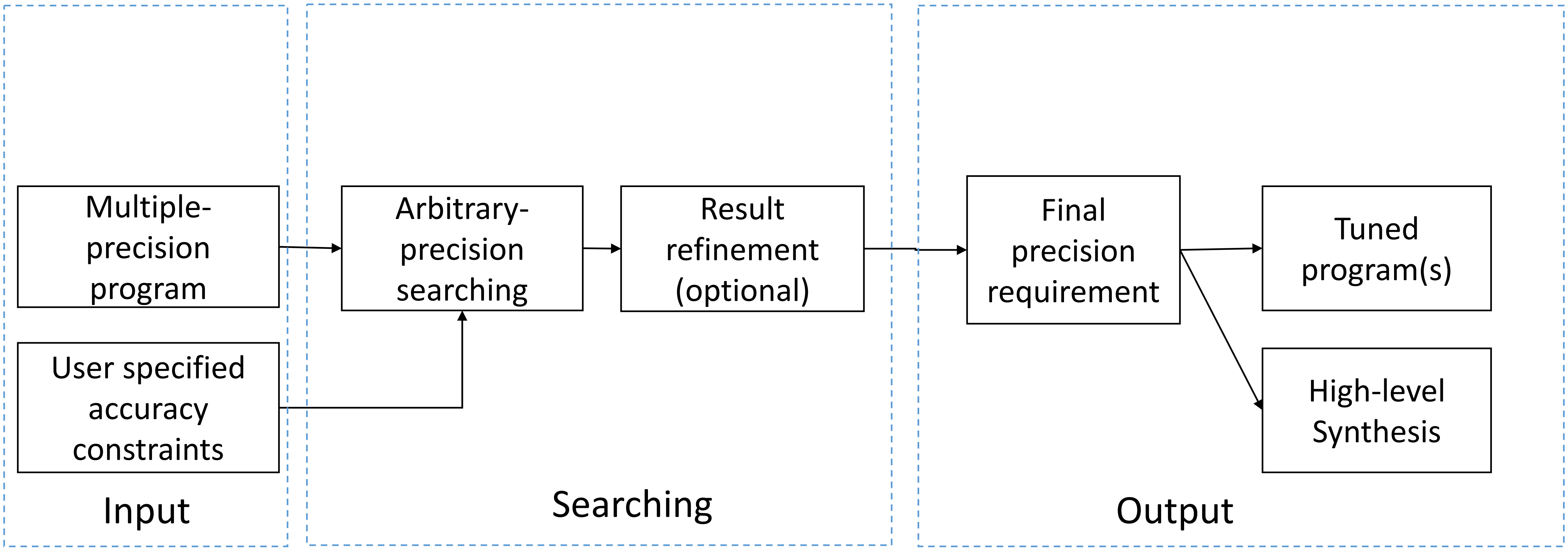
Current techniques for FPGA community:

- Slow when processing complex applications
- Fine-grained precision (any number of bits)
- HLS paves the way for big and complex applications on FPGA

Our goal:

- Fast
- Scale well
- Fine-grained precision support
- The result can be used on HLS process, as well as migrated to GPU, Xeon Phi

Overview



Multiple-precision program
User specified accuracy constraints
Input
Arbitrary-precision searching
Result refinement (optional)
Searching
Final precision requirement
Tuned program(s)
High-level Synthesis
Output

MPFR transformation

```
function(){
    double var1 = 1.0;
    double var2 = var1 + 1.0;
    ...
}
```

Original code

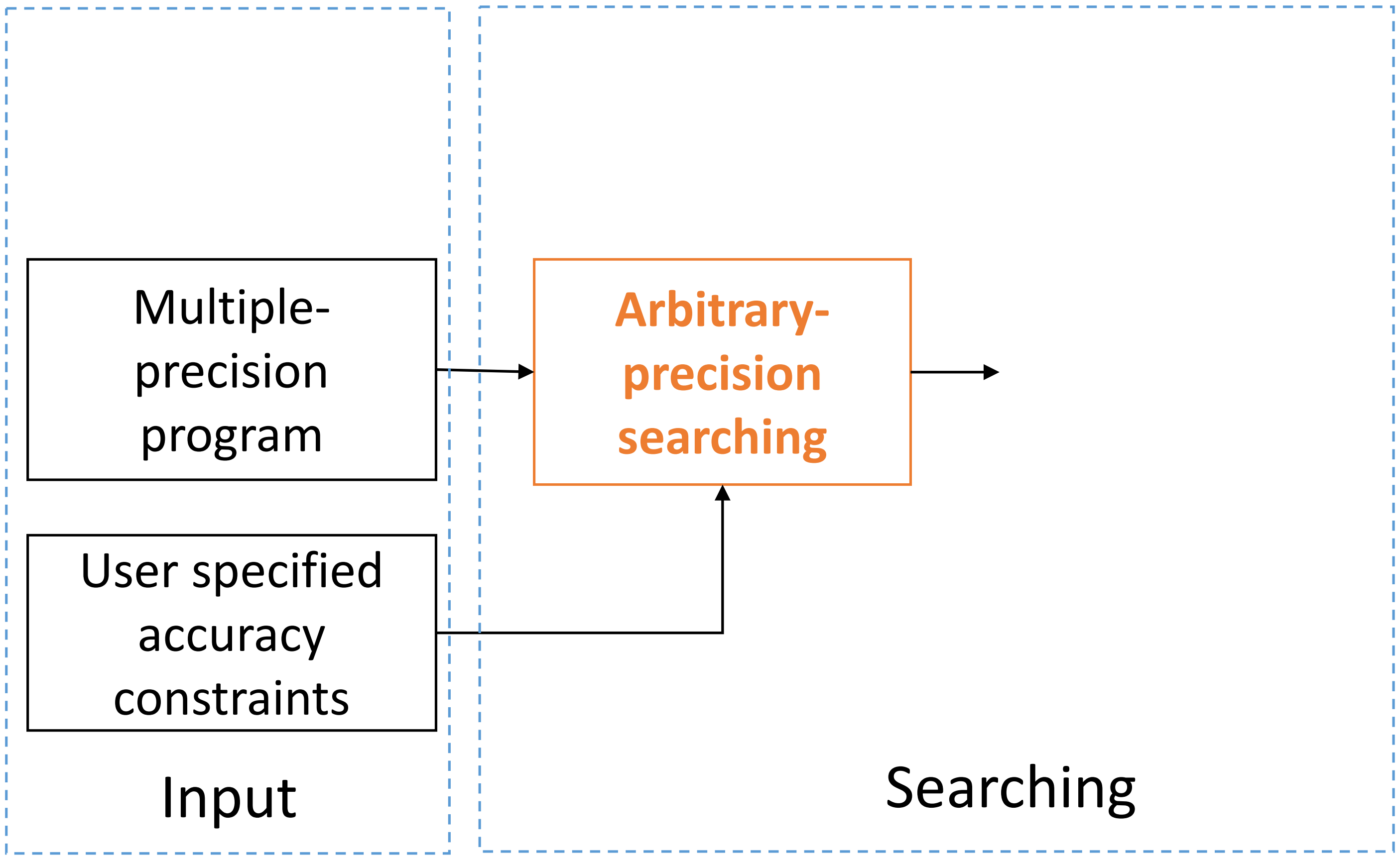
Precision assigned at runtime

```
function(){
    mpfr_t var1;
    mpfr_init2(var1, 53);
    mpfr_set_d(var1, 1.0, MPFR_RNDN);
    mpfr_t var2;
    mpfr_init2(var2, 53);
    mpfr_add_d(var2, var1, 1.0, MPFR_RNDN);
    ...
}
```

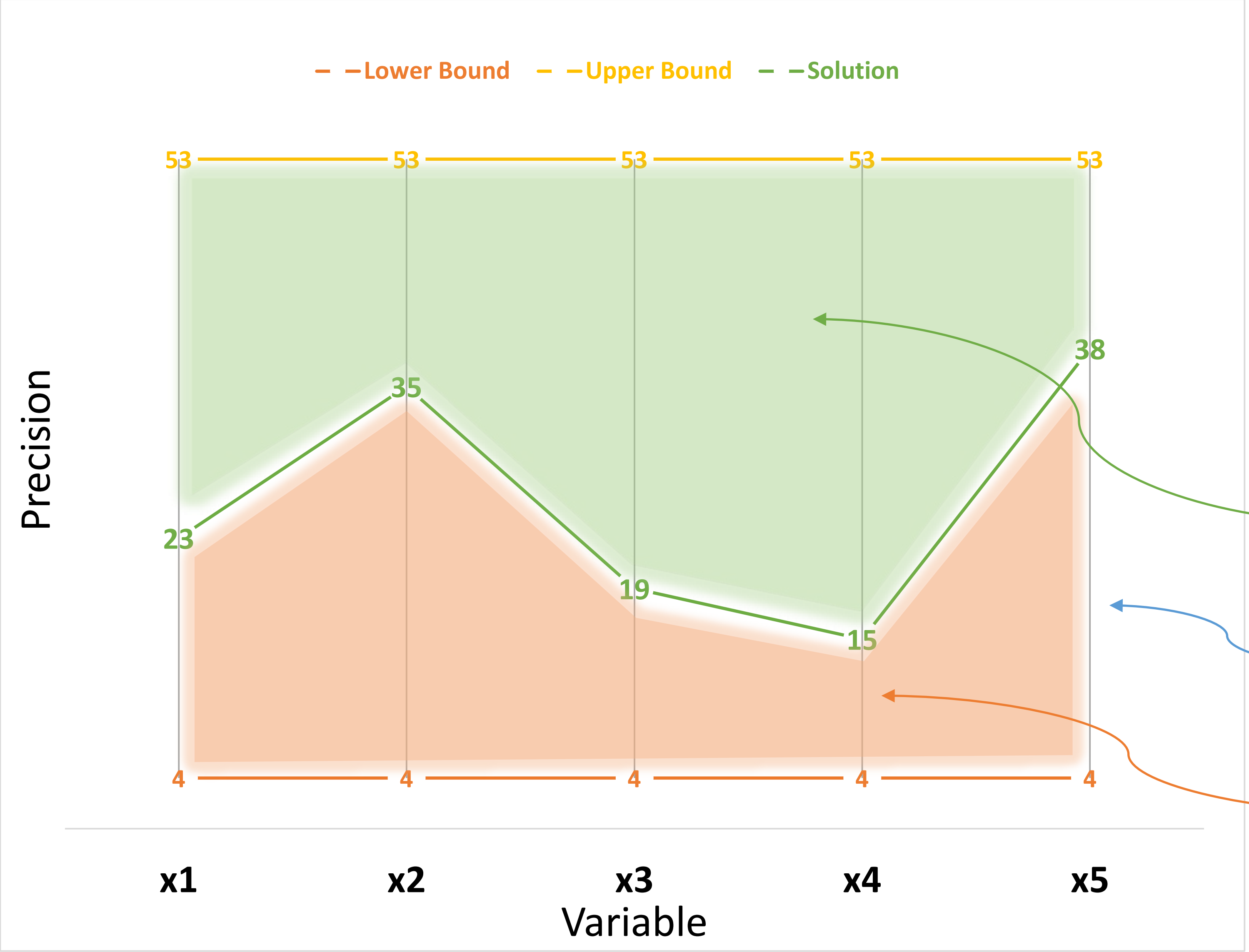
Rewritten code

Use Multiple Precision Floating-Point Reliable (MPFR) library to create the Multiple-precision program for searching

Arbitrary-precision tuning



Arbitrary-precision tuning




```
function(){
      mpfr_t  x1;
      mpfr_t  x2 = …. ;
      ….
      mpfr_t  x5 = ….;
}
```

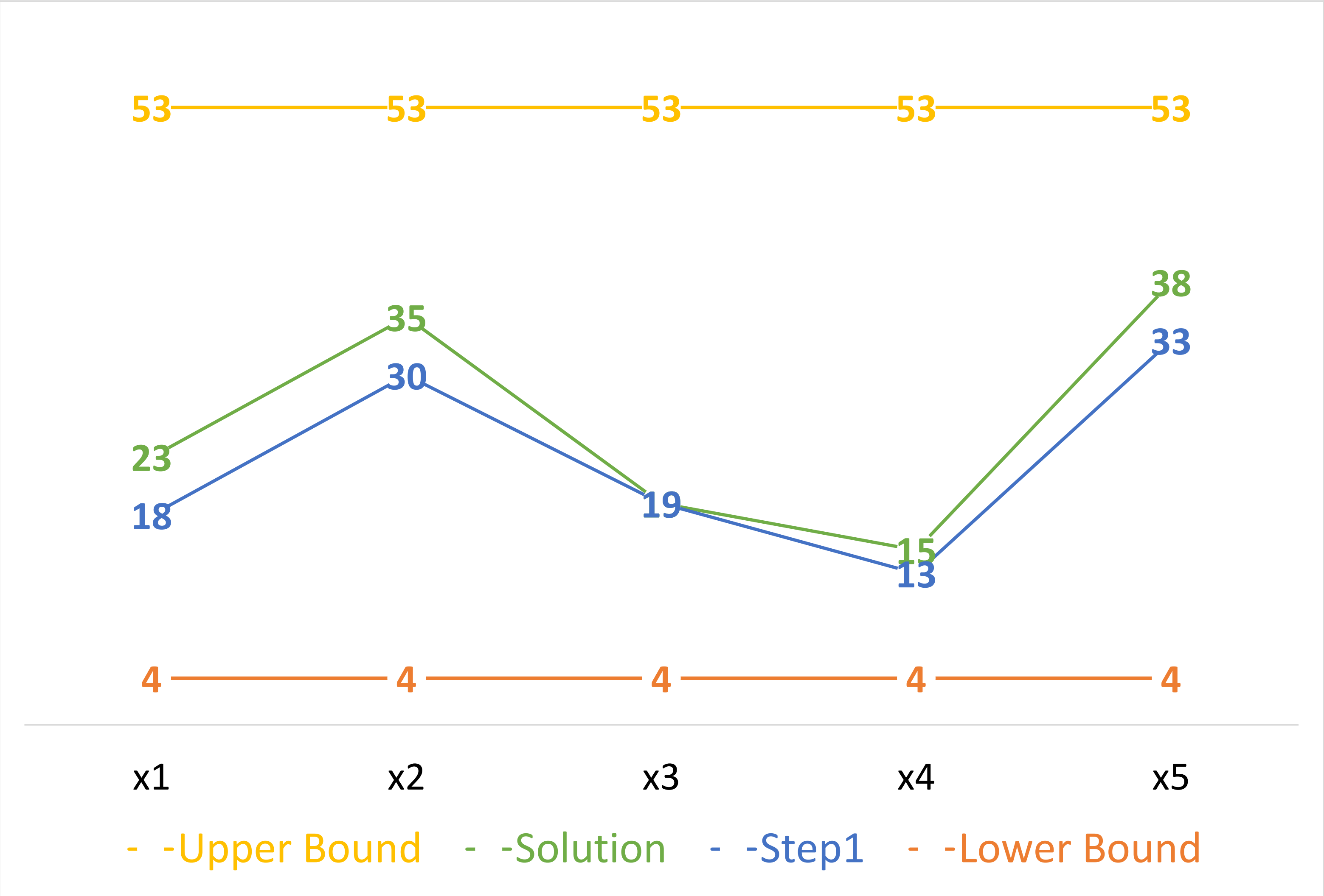
- User defined accuracy constraint ϵ
- **Err** : error of the output

Region 1: $Err \leq \epsilon$

One slice of the search space

Region 2: $Err > \epsilon$

Step 1: Isolated downward

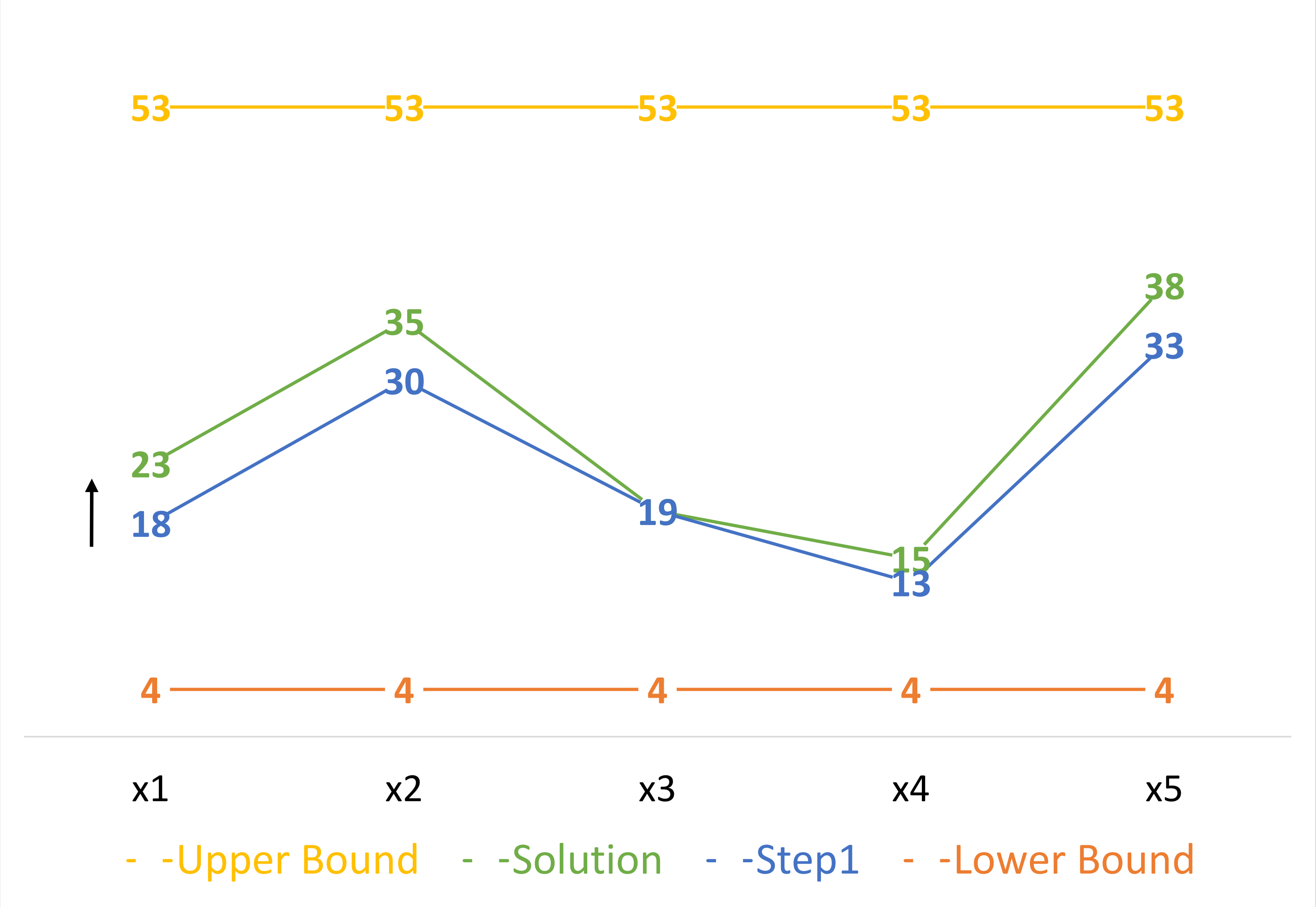


Find the minimum possible precision for each variable while keeping others at highest precision.

Binary search + parallelism at variable level.
Run-time $\leq \log_2(53-4)$

Step1 result usually causes $Err > \epsilon$

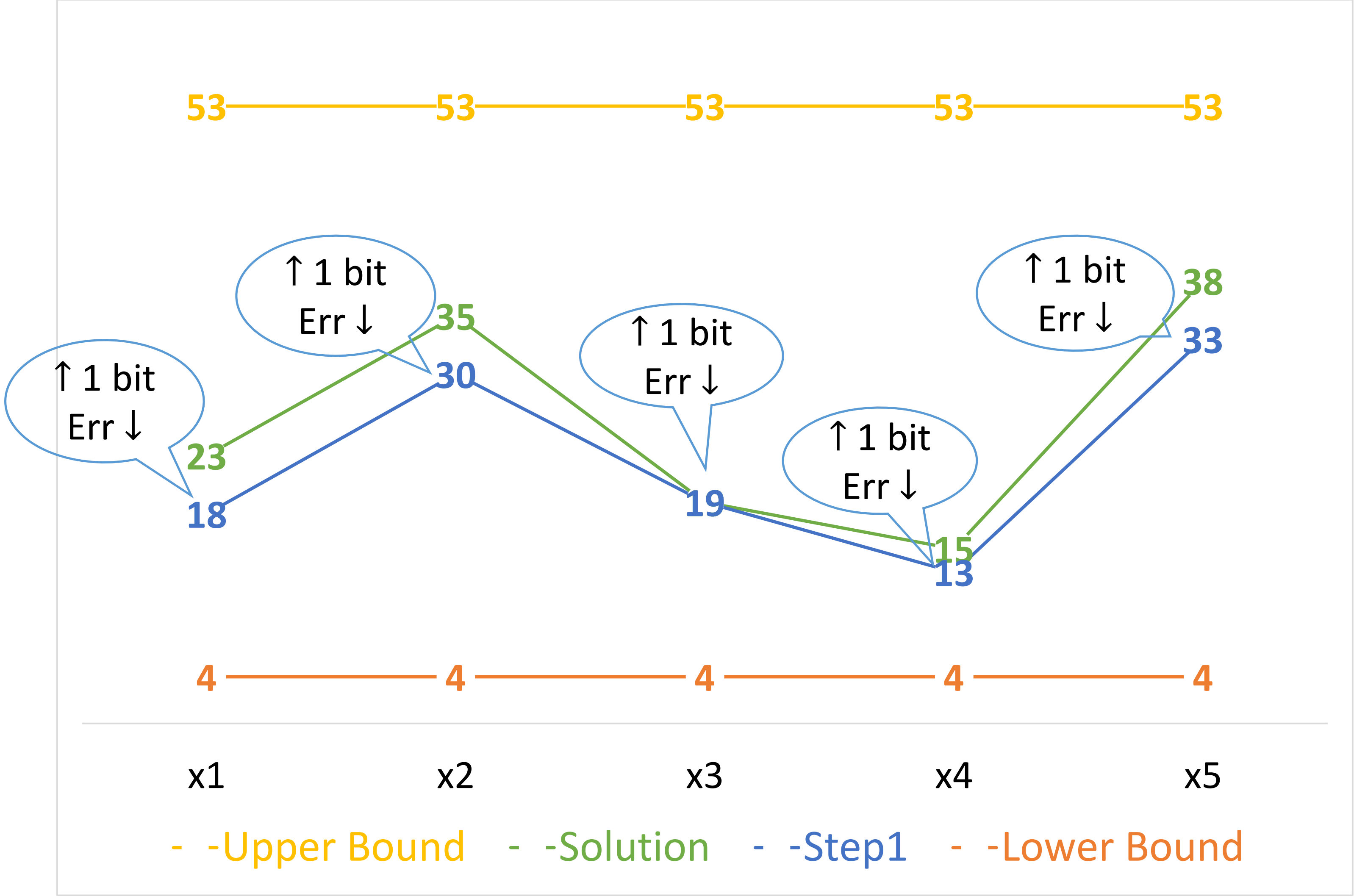
Step 2: Grouped upward



From Step1 result, try to get back to the solution

Strategy: Get back to the point where $Err \leq \epsilon$ as fast as possible.

Step 2: Grouped upward



Strategy: Get back to the point where $Err \leq \epsilon$ as fast as possible.

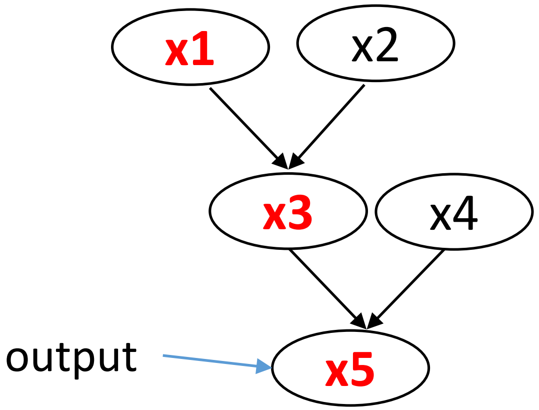
Greedily shift the blue line upward:

- 1 variable at a time: good but stuck when no variables can reduce *Err*

The effect may not propagate to the output => no change in Err

Step 2: Grouped upward (cont)

- Our approach: “grey-box” distributed search



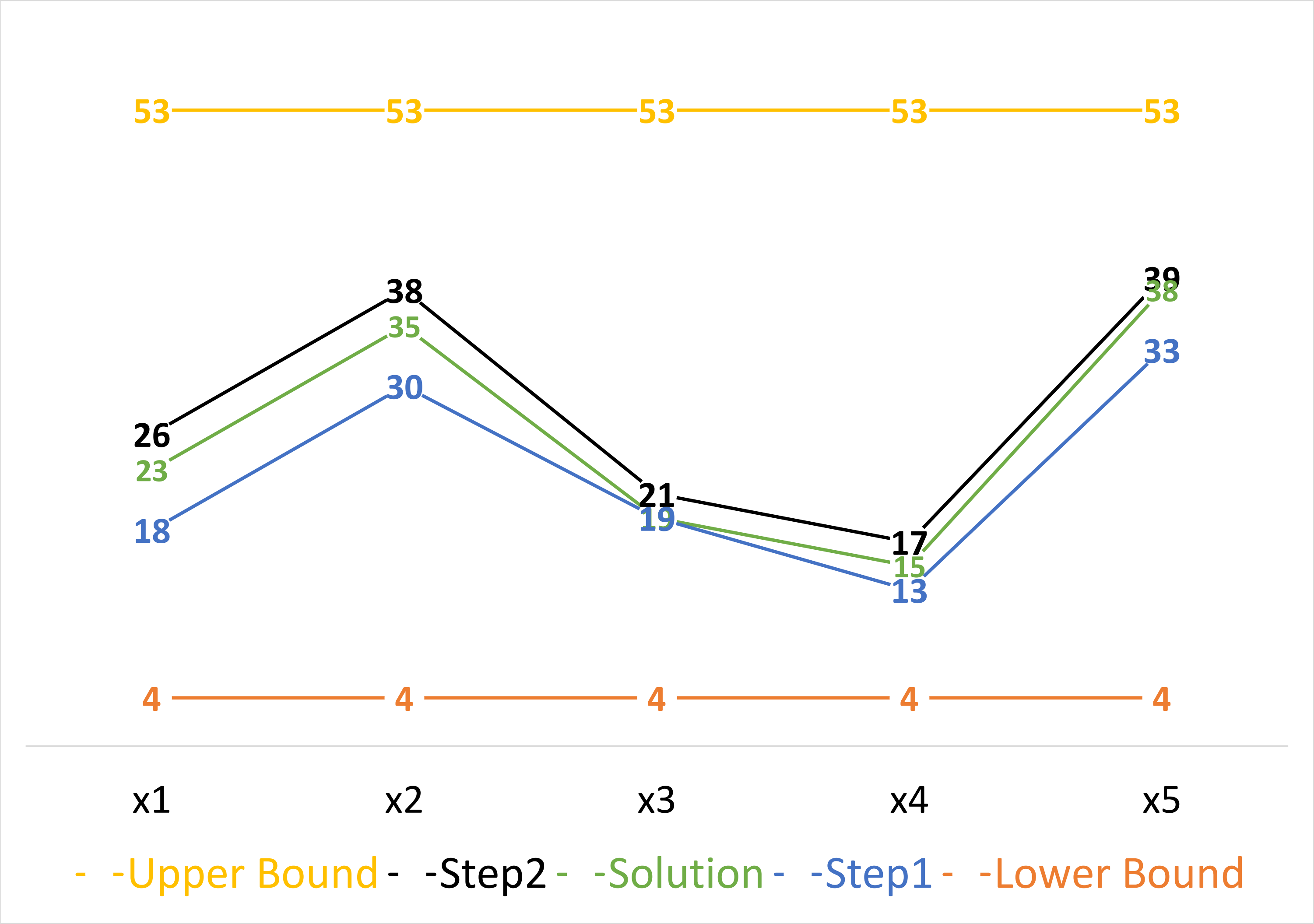

```
x3 = x1 + x2
x5 = x3 * x4
```

Dataflow graph

When increasing precision of a variable, should increase all other variables in the path from it to the output.

Increase precision of the whole dependence group, not single variable.
{x1,x3,x5}, {x2,x3,x5}, {x3,x5}

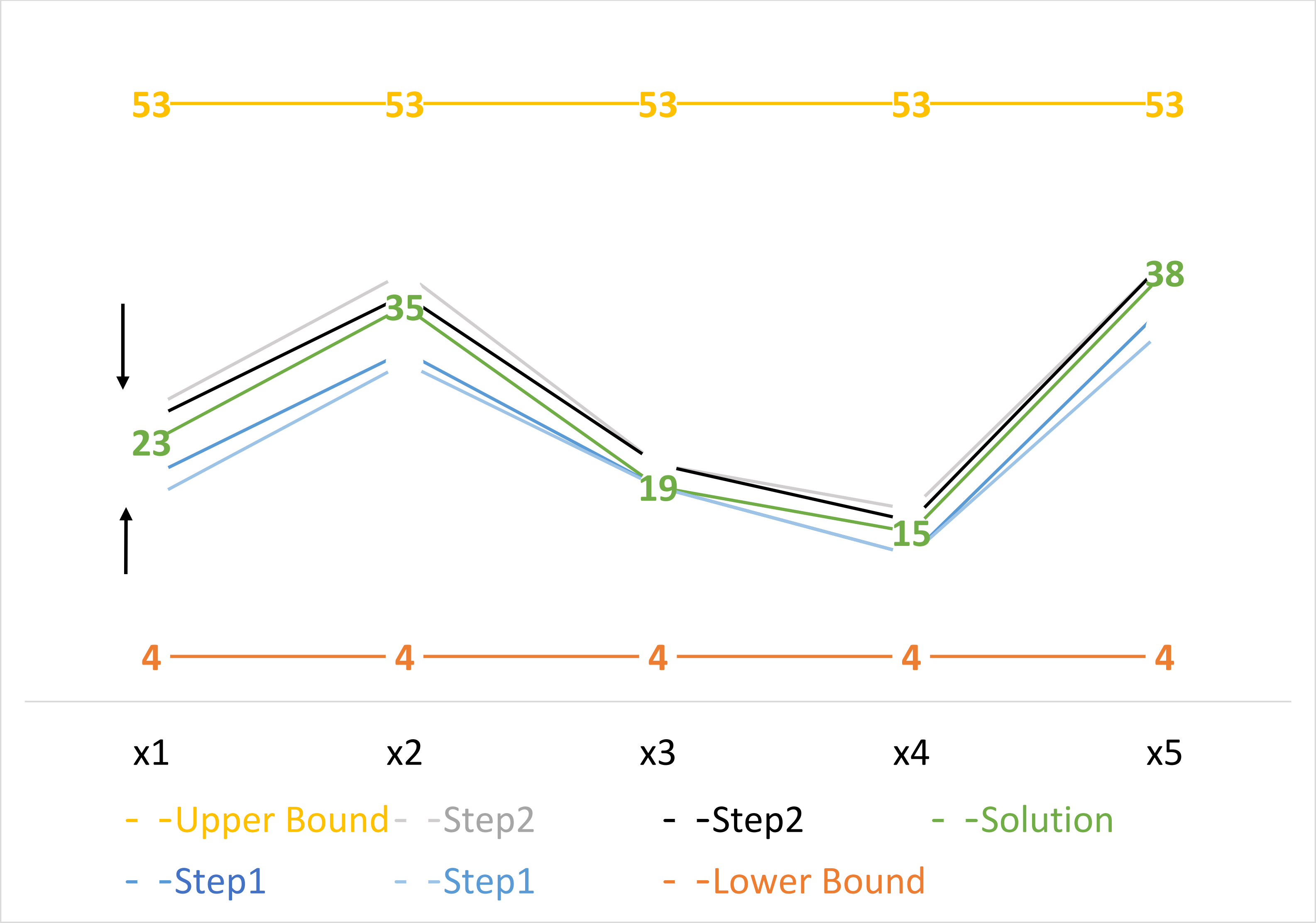
Step 2: Grouped upward



- Shift *Step1 result* upward by competition between groups of variables.
- Group reduces most error will win 1 bit for all members.
- Parallelize at group level (5 groups)

Step 2 gives an acceptable result higher than the solution.

The iterative process



- Reuse step 1 to find another result closer to the solution.
- Then reuse step 2 to move upward to the solution.
- The algorithm converges after a few epochs.

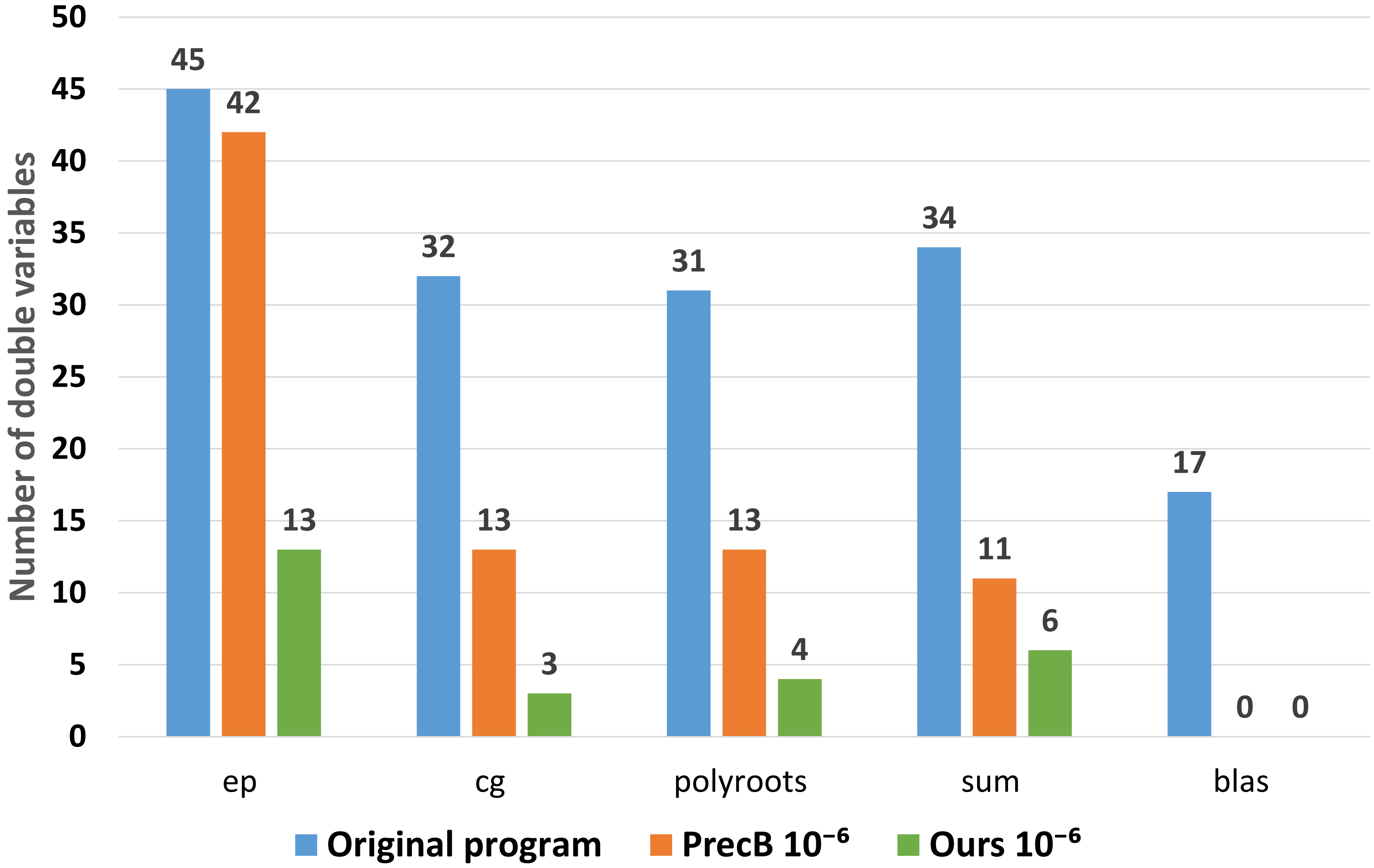
Result

- Quality: $\approx$6% fewer in number of bits compared to an established algorithm *Max-1*(for small programs).
- Complexity:
  - T_{mpfr} = time to run the input program (multiple-precision version):
  - Average: 25.9 x T_{mpfr} , for programs have 10-45 variables
  - Large program (417 variables): 110.5 x T_{mpfr}

Compare to state-of-the-art

- ***Precimonious*** searches for the mixed use of 2 types : *float* and *double*.
- The fine-grain results are mapped to 2 types for comparison.

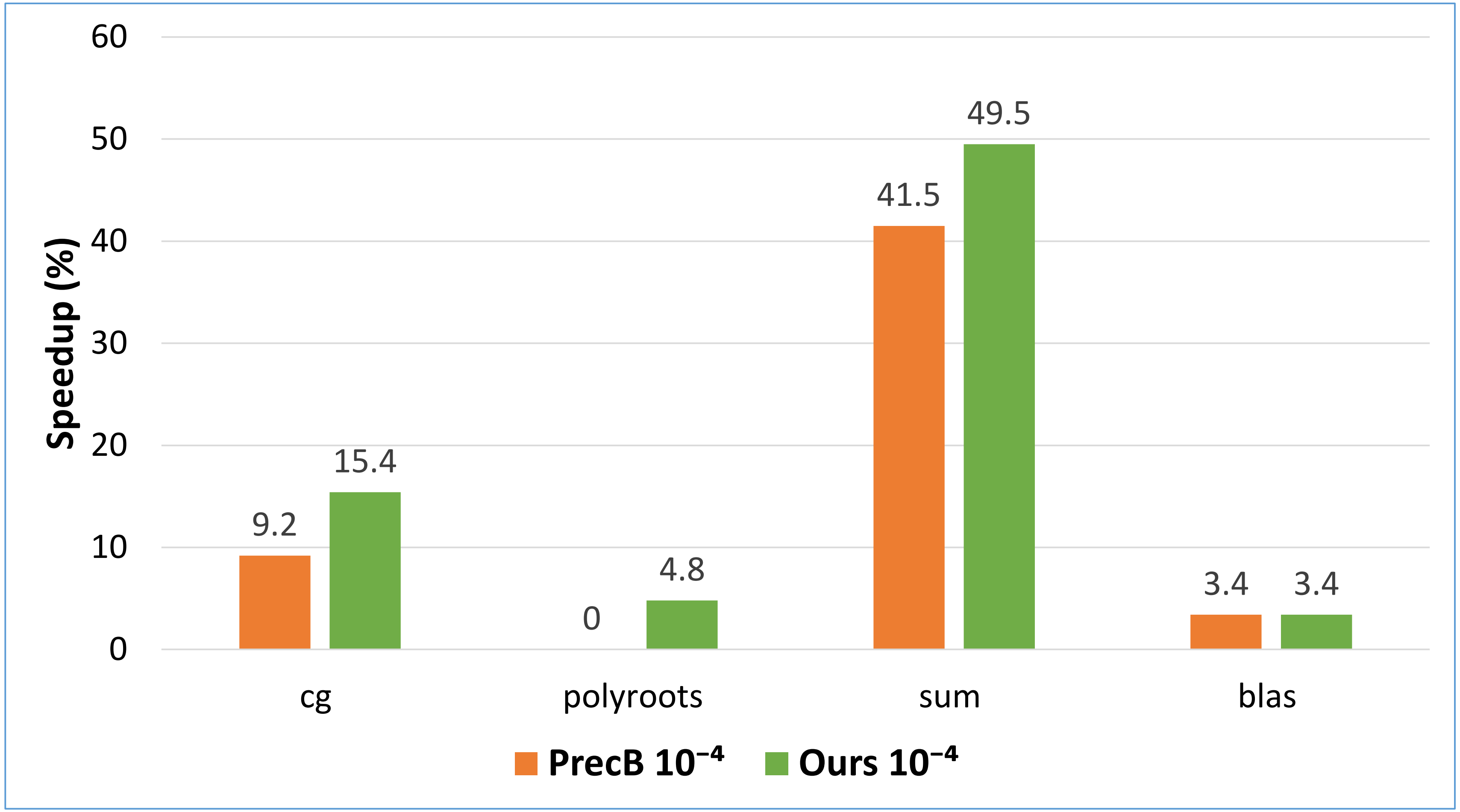
Number of double variables required for $\epsilon = 10^{-6}$



PrecB: tuned by *Precimonious* 2016 [7].

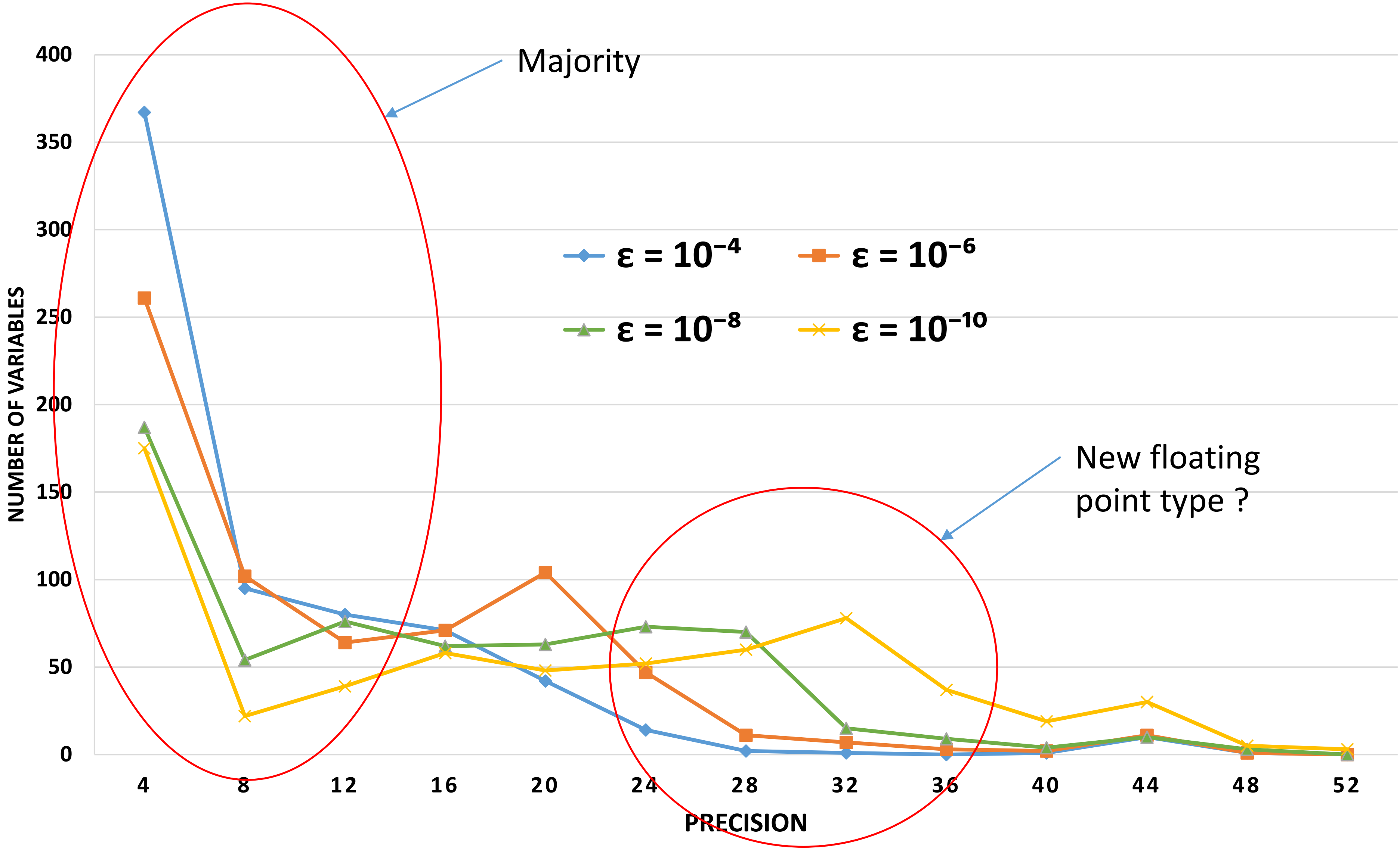
Ours: tuned by our tool chain

Speedup (%) compared to the original version



$$\epsilon = 10^{-4}$$

Aggregated result across 11 programs

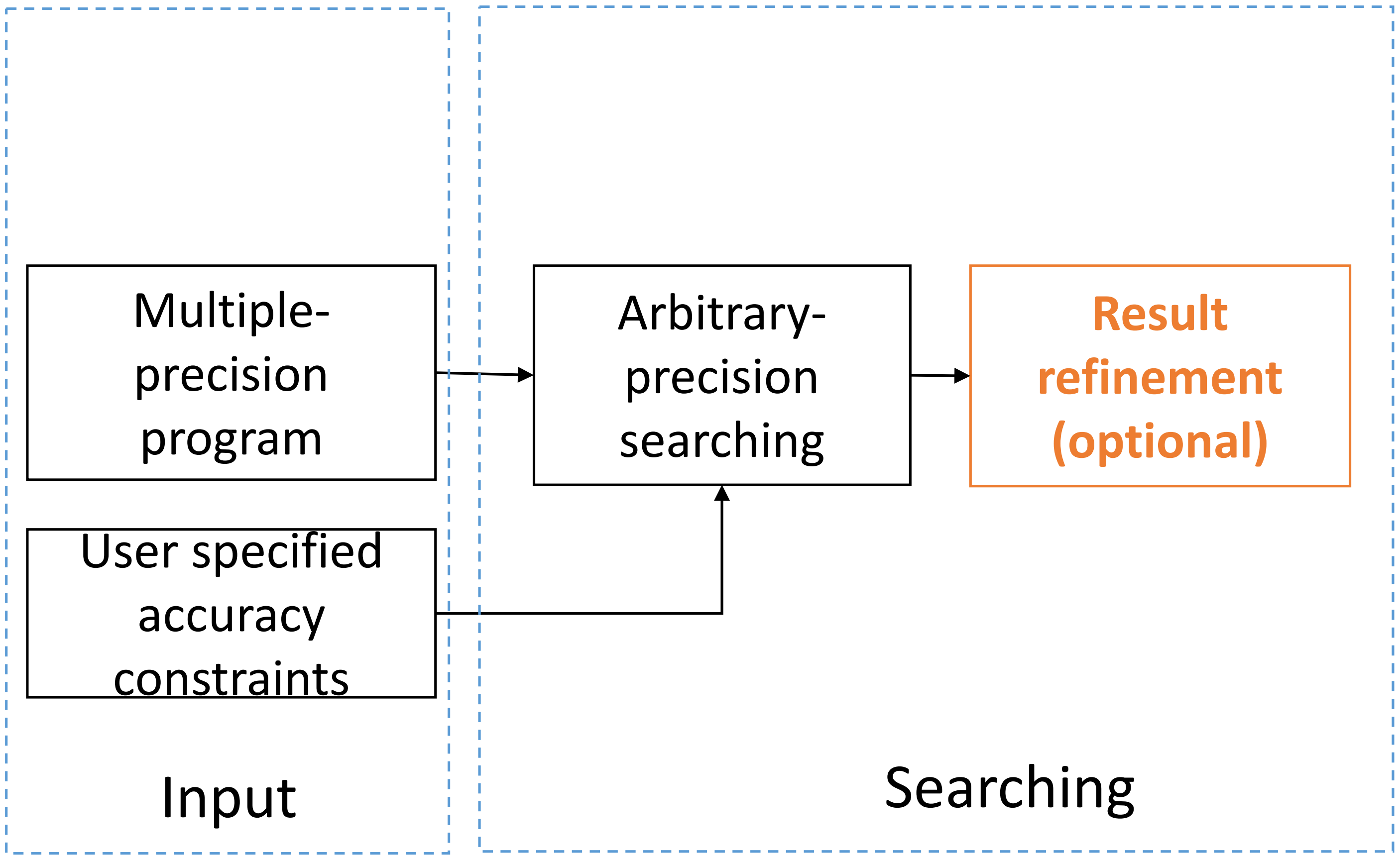


2,666 floating-point variables across 4 error thresholds (ε)

% variable can be in half-precision (11 bits) :

- ≈66% for 10^{-4}
- ≈52% for 10^{-6}
- ≈38% for 10^{-8}
- ≈31% for 10^{-10}

Result refinement



Input variation problem

```
function(float_32 input){
    float_32 output = input * input;
}
```

Input = 1.2, $\epsilon = 10^{-5}$

```
function(float_16 input){
    float_25 output = input * input;
}
```

Input = 1.2, Err $\leq 10^{-5}$

Input = 1000.0, Err $=$?

Input = 0.01, Err $=$?

Statistically guided refinement for input ∈ [0.01;1000]

Training set of M seeds for random number generator in range [0.01;1000]

1 seed number = 1 representative input
Training set of 100 seeds

Statistically guided refinement

```
function(float_32 input){
      float_32 output = input * input;
}
```

worst_seed = 1, $\epsilon = 10^{-5}$

```
function(float_20 input){
      float_18 output = input * input;
}
```

Average Err over 100 seeds = ?
worst_seed = ?

Average Err = 5.0×10^{-3}
worst_seed = 43 causes Err = 10^{-2}

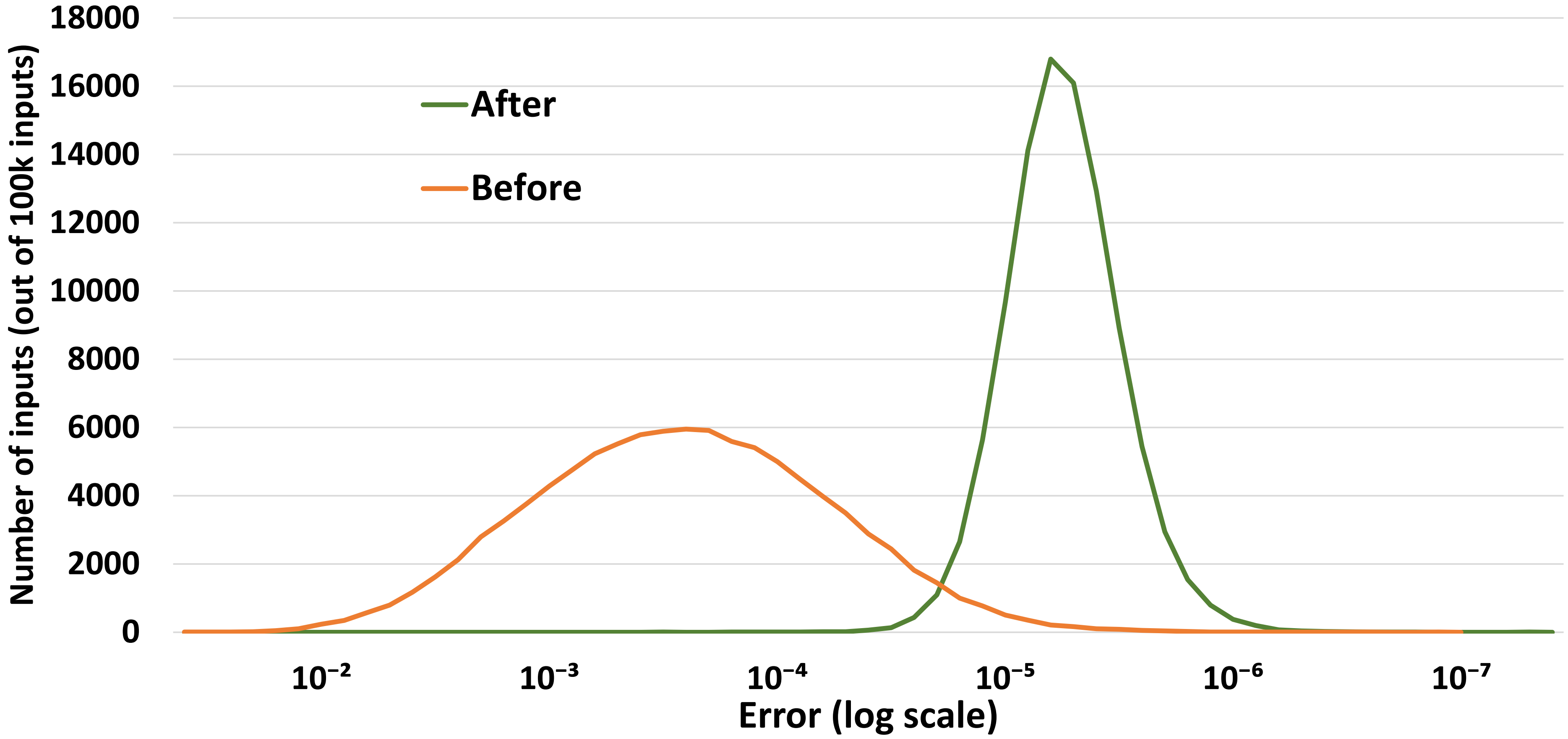
worst_seed = 43, $\epsilon = 10^{-5}$

```
function(float_16 input){
      float_22 output = input * input;
}
```

```
function(float_20 input){
      float_22 output = input * input;
}
```

Average Err over 100 seeds = ?
worst_seed = ?

Result on DSP programs, target $\epsilon = 10^{-5}$

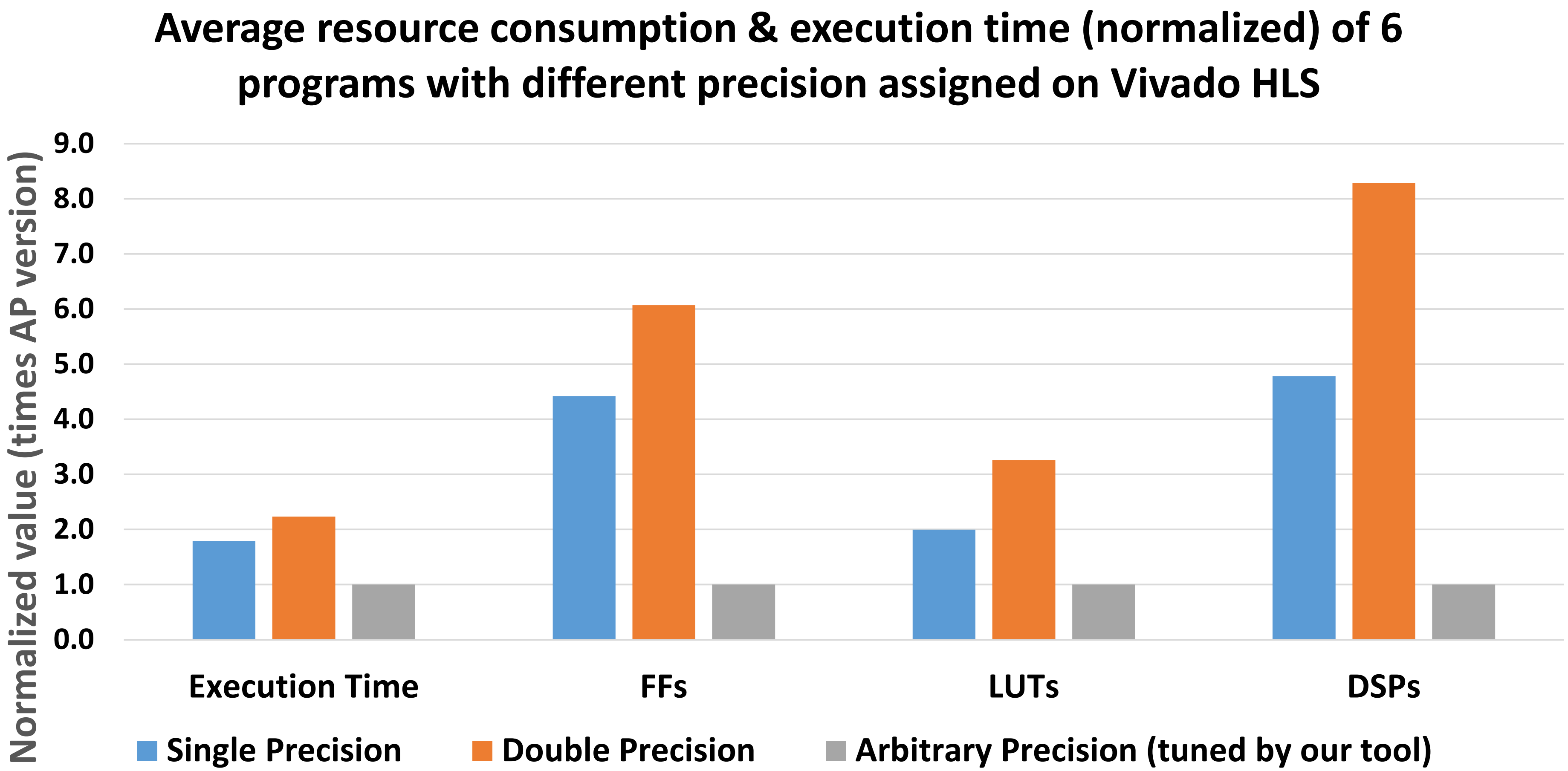


Before, average = 2.3×10^{-4}, max = 3.4×10^{-2}

After, average = 4.9×10^{-6}, max = 2.5×10^{-4}

Arbitrary precision version on Vivado HLS

Accuracy constraint: 50-60dB



Conclusion

- Our algorithm can scale to large and long running programs:
  - E.g. T_{mpfr} = 20 mins, number of variables = number of MPI threads ≤ 45
    => Expected searching = 26 x 20 ≈ 520 mins.
- We use program’s high-level dependence information to guide the distributed search process.
- Input variation problem can be mitigated with our statistics guided refinement process.
- This tool paves the way for using HLS with arbitrary precision on large programs.

Thanks for listening

Q&A

Link to github repository

https://github.com/minhhn2910/fpPrecisionTuning