An SMT Based Method for Optimizing Arithmetic Computations in Embedded Software Code

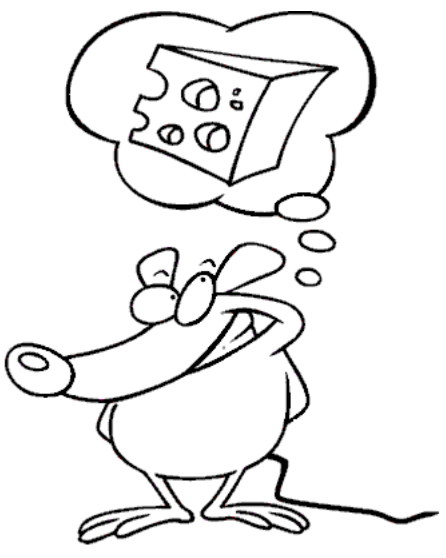
Hassan Eldib and Chao Wang

VirginiaTech
Invent the Future

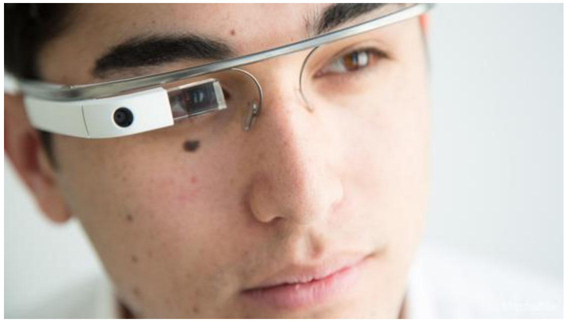
FMCAD, October 22, 2013

The Dream

- Having a tool that automatically synthesizes the optimum version of a software program.

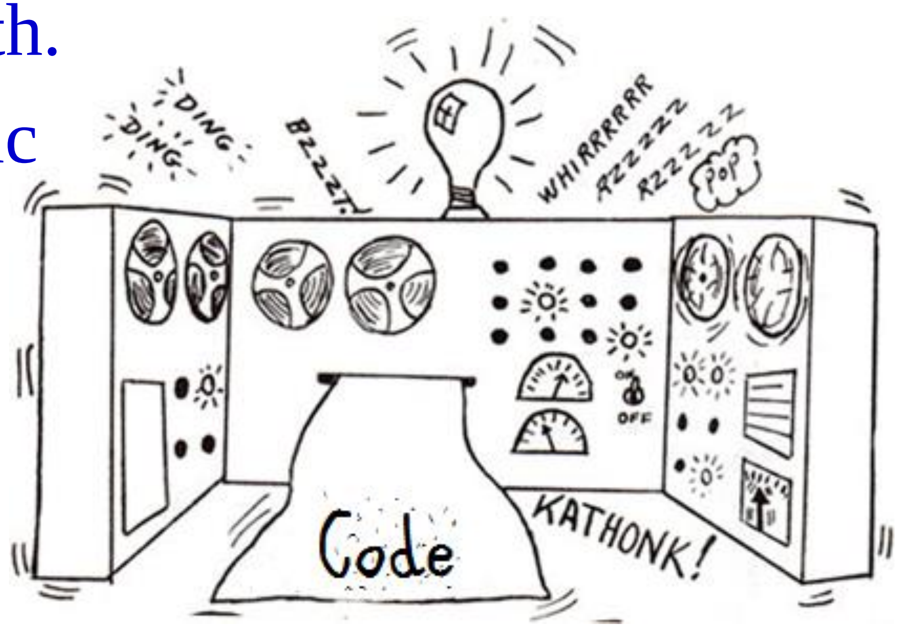


Embedded Software



Objective

- Synthesizing an optimal version of the C code with fixed-point linear arithmetic computation for embedded devices.
  - Minimizing the bit-width.
  - Maximizing the dynamic range.



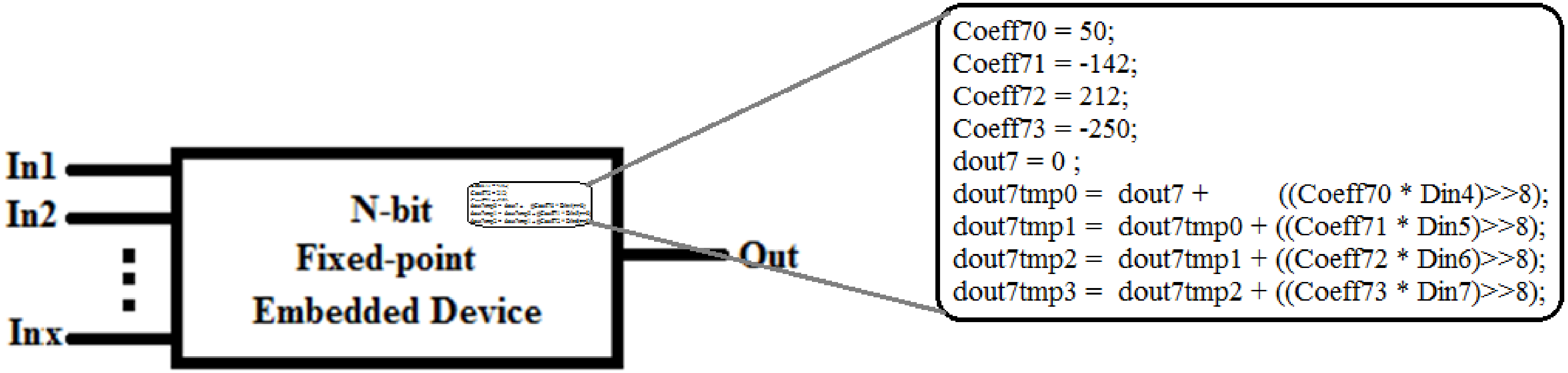
Motivating Example

- Compute average of A and B on a microcontroller with signed 8-bit fixed-point

- Given: A, B ∈ [-20, 80].

  - $\frac{A+B}{2}$ *may* have overflow errors.
  - $\frac{A}{2}+\frac{B}{2}$ *may* have truncation errors.
  - $B+\frac{A-B}{2}$ has neither overflow nor truncation errors.

Bit-width versus Range



- Larger range requires a larger bit-width.
- Decreasing the bit-width, will reduce the range.

Fixed-point Representation

Representations for 8-bit fixed-point numbers

0	1	1	0	1	0	0	1
-128	64	32	16	8	4	2	1

- Range: -128 ↔ 127
- Resolution = 1

0	1	1	0	1	0	0	1
-16	8	4	2	1 .	½	¼	⅛

- Range : -16 ↔ 15.875
- Resolution = 1/8

Range ∝ Bit-width

Resolution ∝ Bit-width

Problem Statement

Program:

```
int comp(int A,int B,int H,int E,int D,int F,int K) {
 int t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12;
 t12 = 3 * A;
 t10 = t12 + B;
 t11 = H << 2;
 t9  = t10 + t11;
 t6  = t9 >> 3;
 t8  = 3 * E;
 t7  = t8 + D;
 t5  = t7 - 16469;
 t3  = t5 + t6;
 t4  = 12 * F;
 t2  = t3 - t4;
 t1  = t2 >> 2;
 t0  = t1 + K;
 return t0;
}
```

Range & resolution of the input variables:

A -1000 3000

res. 1/4

B -1000 3000

res. 1/4

…

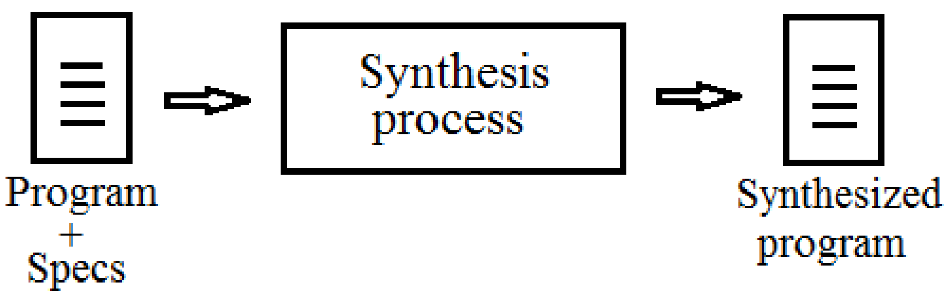
Optimized program:

```
int comp(int A,int B,int H,int E,int D,int F,int K) {
 int t0,t1,t3,t4,t5,t6,t8,t12;
 int N1,N2,N3,N4,N5,N6,N7,N9,N10;
 t12 = 3 * A;
 N6  = H;
 N10 = t12 - B;
 N9  = N10 >> 1;
 N7  = B + N9;
 N5  = N7 >> 1;
 N4  = N5 + N6;
 t6  = N4 >> 1;
 t8  = 3 * E;
 N3  = t8 - 16469;
 t5  = N3 + D;
 t3  = t5 + t6;
 t4  = 12 * F;
 N2  = t4 >> 2;
 N1  = t3 >> 2;
 t1  = N1 - N2;
 t0  = t1 + K;
 return t0;
}
```

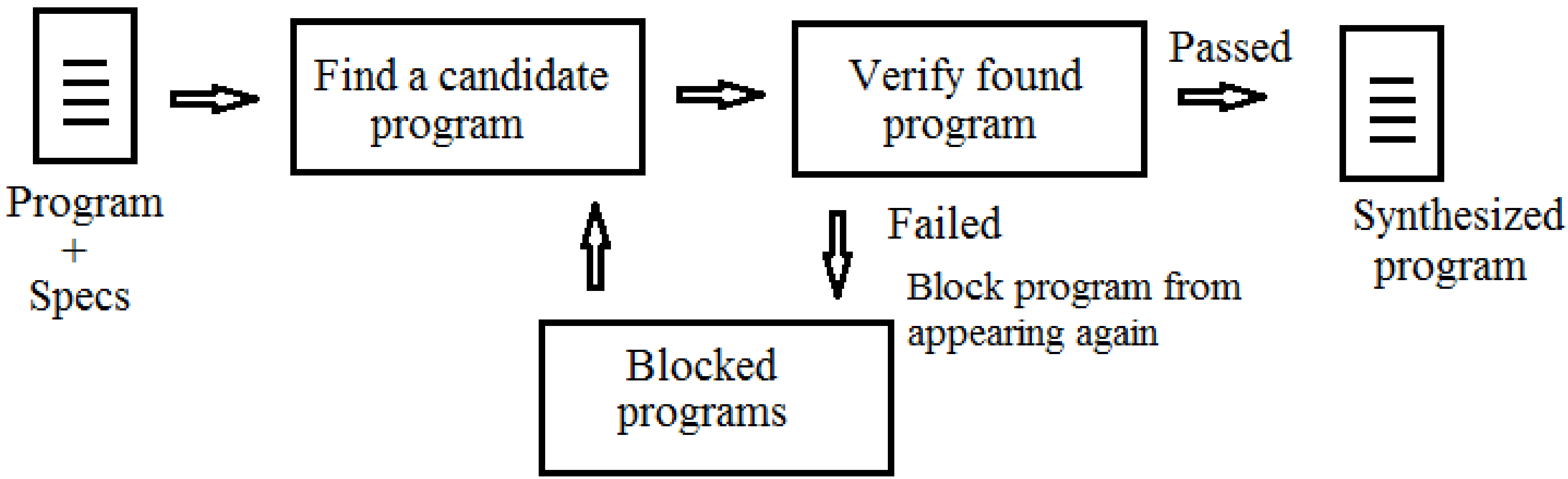
Problem Statement

- Given
  - The C code with fixed-point linear arithmetic computation
  - The range and resolution of all input variables

- Synthesize the optimized C code with
  - Reduced bit-width with same input range, or
  - Larger input range with the same bit-width



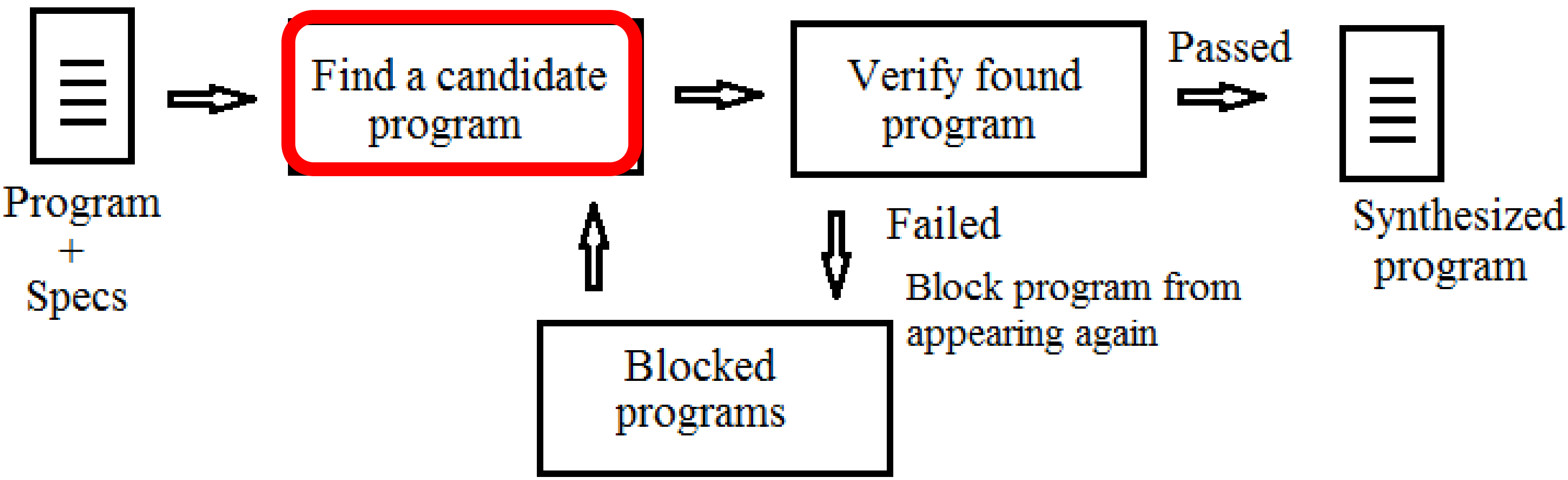
SMT-based Inductive Program Synthesis



Some Related Work

- Jha, 2011
  - Use an SMT solver to choose the best fixed-point representation in order to reduce error. No new programs are synthesized.
- Majumdar, Saha, and Zamani, 2012
  - Use a mixed integer linear programing (MILP) solver to minimize the error bound by only changing the fixed-point representation.
- Schkufza, Sharma, and Aiken, 2013
  - Use a compiler based method for optimization, which is an exhaustive approach.

SMT-based Inductive Program Synthesis



Step 1: Finding a Candidate Program

- Create the most general AST that can represent any arithmetic equation, with reduced bit-width.

- Use SMT solver to find a solution such that
  – For some test inputs (samples),
  – output of the AST is the same as the desired computation

SMT-based Solution

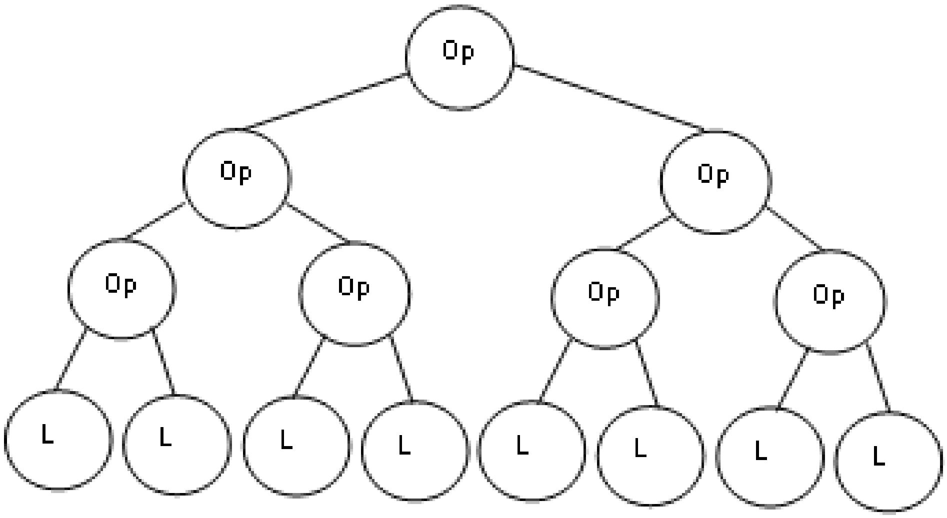


Fig. General Equation AST.

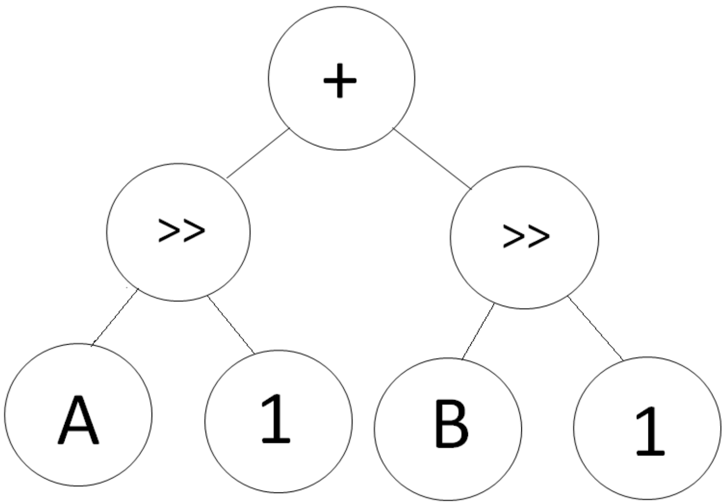
- SMT encoding for <u>the general equation</u> AST structure
  - Each *Op* node can any operation from *, +, -, >> or <<.
  - Each *L* node can be an input variable or a constant value.
- SMT Solver finds a solution by equating the AST output to that of the desired program

SMT Encoding

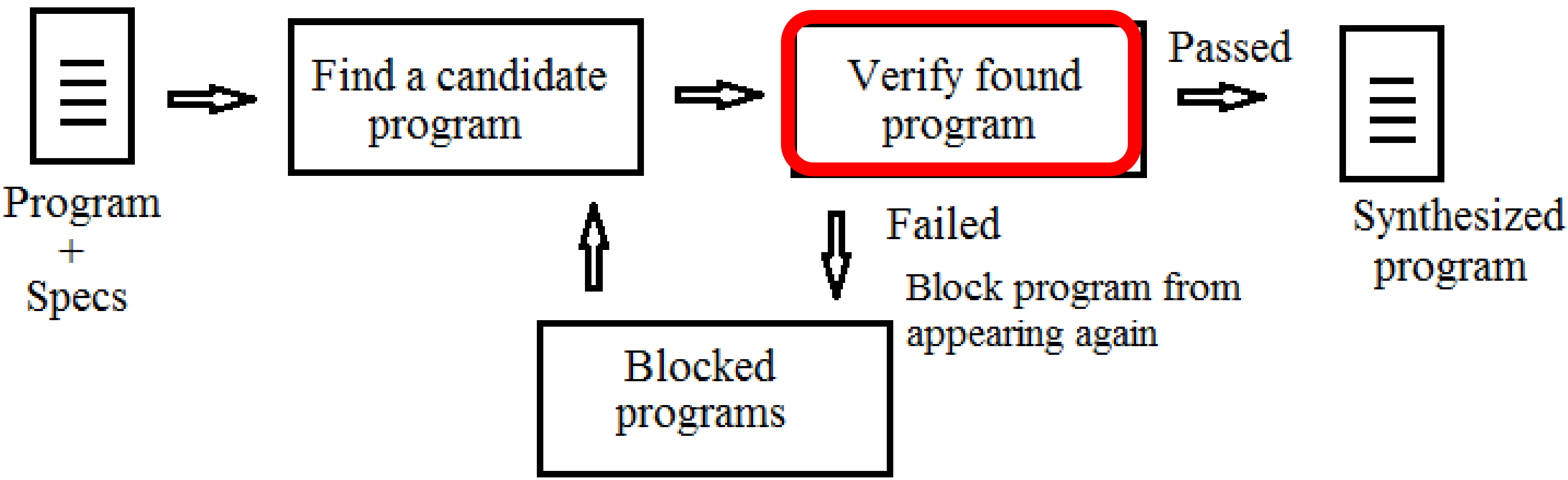
- $\Psi \ = \Phi_{prog} \land \Phi_{AST} \land \Phi_{sameI} \land \Phi_{sameO} \land \Phi_{in} \land \Phi_{block}$
  - Φ_{prog} : Desired input program to be optimized.
  - Φ_{AST} : General AST with reduced bit-width.
  - Φ_{sameI} : Same input values.
  - Φ_{sameO} Same output value.
  - Φ_{in} : Test cases (inputs).
  - Φ_{block} : Blocked solutions.

SMT-based Solution (an example)

$$\frac{A}{2} + \frac{B}{2} \equiv$$



SMT-based Inductive Program Synthesis



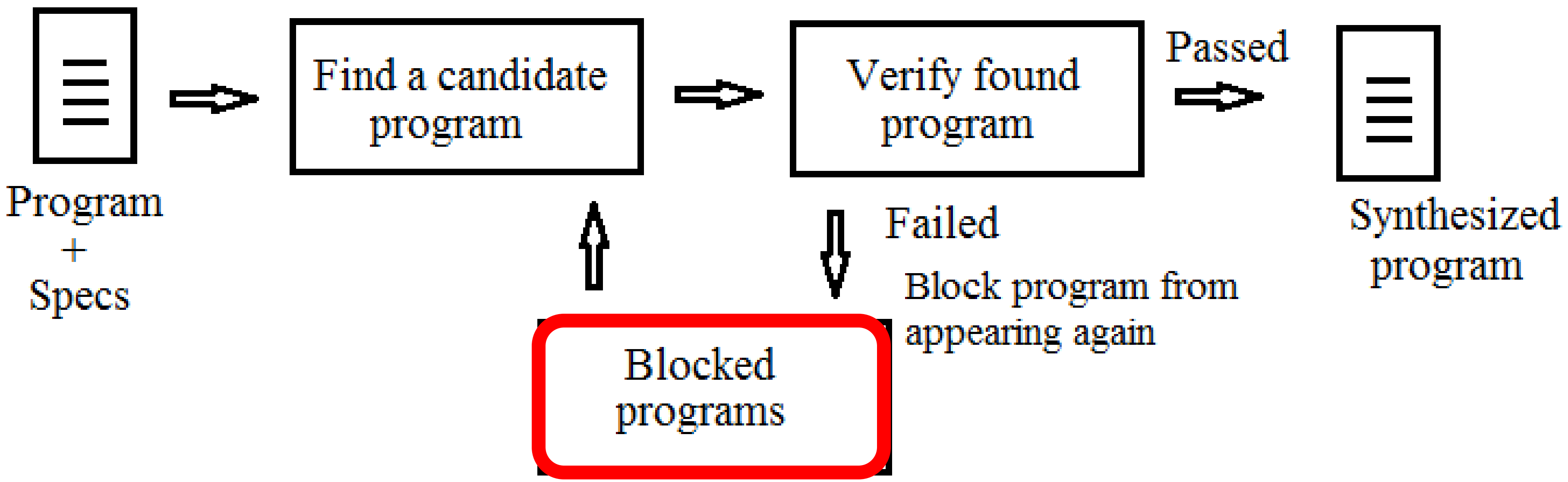
Step 2: Verifying the Solution

- Is the program good for all possible inputs?
  - Yes, we found an optimized program
  - No, block this (bad) solution, and try again

SMT Encoding

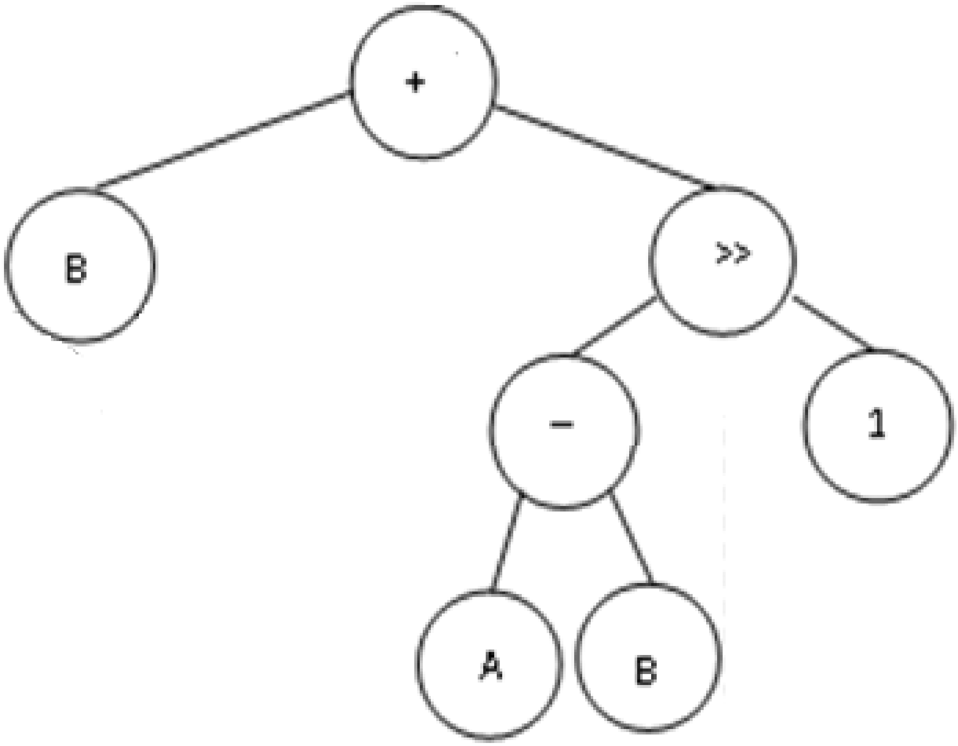
- $\Phi = \Phi_{prog} \wedge \Phi_{sol} \wedge \Phi_{sameI} \wedge \Phi_{diffO} \wedge \Phi_{ranges} \wedge \Phi_{res}$
  - Φ_{prog} : Desired input program to be optimized.
  - **Φ_{sol} : Found candidate solution.**
  - Φ_{sameI} : Same input values.
  - **Φ_{diffO} : Different output value.**
  - Φ_{ranges} : Ranges of the input variables.
  - Φ_{res} : Resolution of the input variables.

SMT-based Inductive Program Synthesis

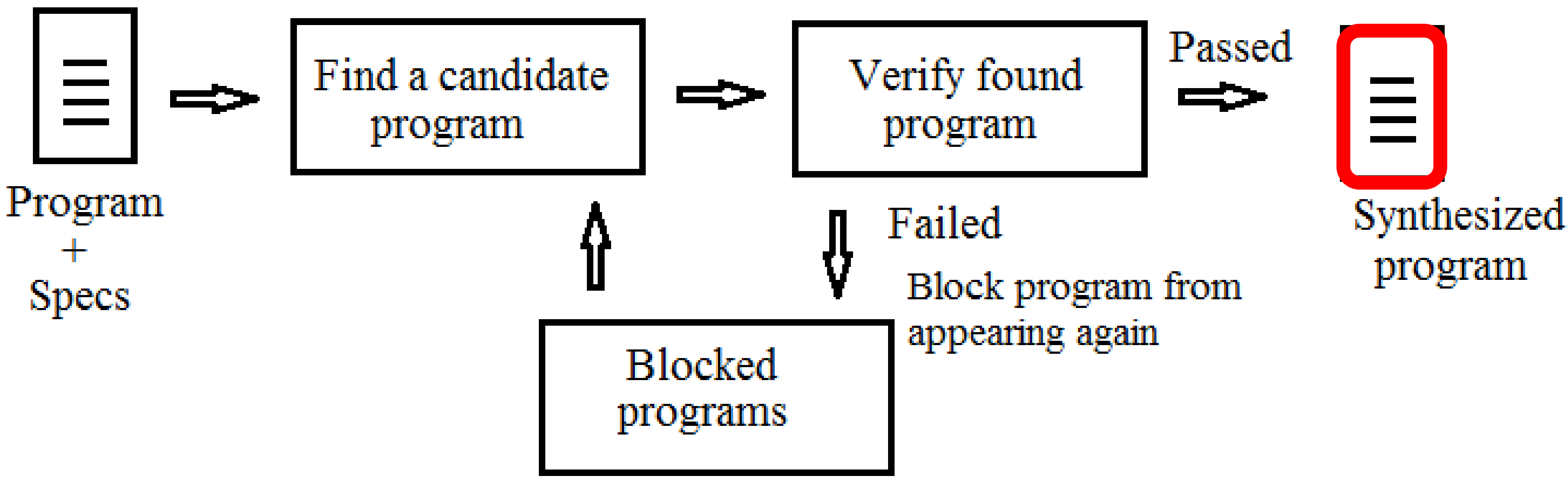


The Next Solution

$$B + \frac{A-B}{2} \equiv$$



SMT-based Inductive Program Synthesis



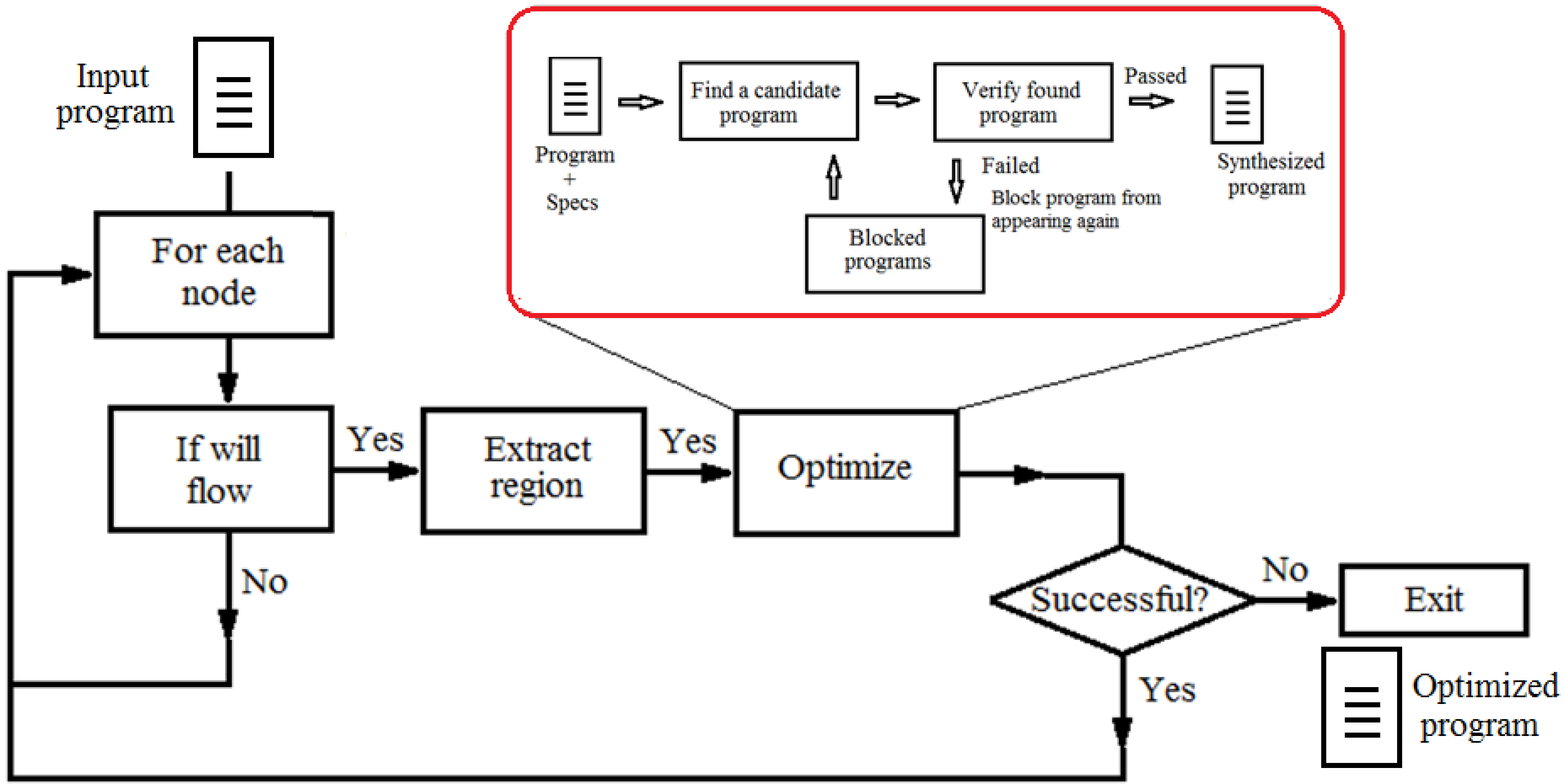
Scalability Problem

- Advantage of the SMT-based approach
  - Find optimal solution within an AST depth bound
- Disadvantage
  - Cannot scale up to larger programs
    - Sketch tool by Solar-Lezama & Bodik (5 nodes)
    - Our own tool based on YICES (9 nodes)

Incremental Optimization

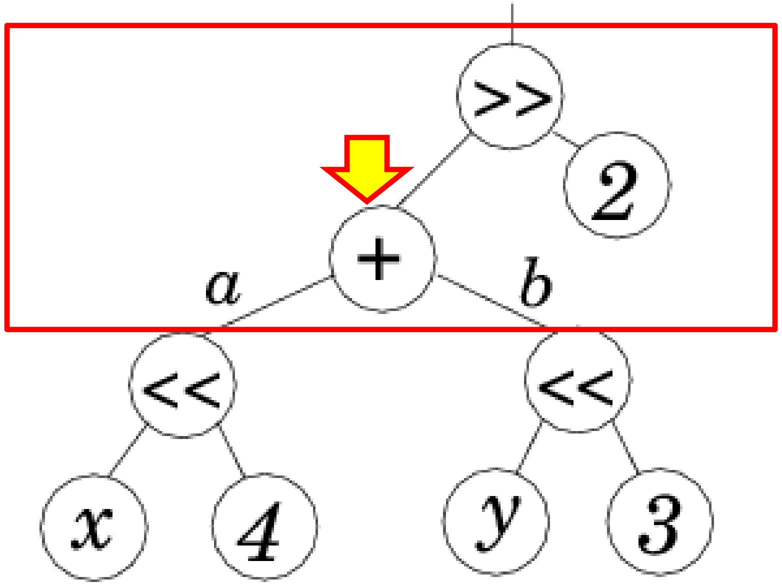
- Combine static analysis and SMT-based inductive synthesis.
- Apply SMT solver only to small code regions
  - Identify an instruction that causes overflow/underflow.
  - Extract a small code region for optimization.
  - Compute redundant LSBs (allowable truncation error).
  - Optimize the code region.
  - Iterate until no more further optimization is possible.

Our Incremental Approach



Example

<u>Detecting Overflow Errors</u>

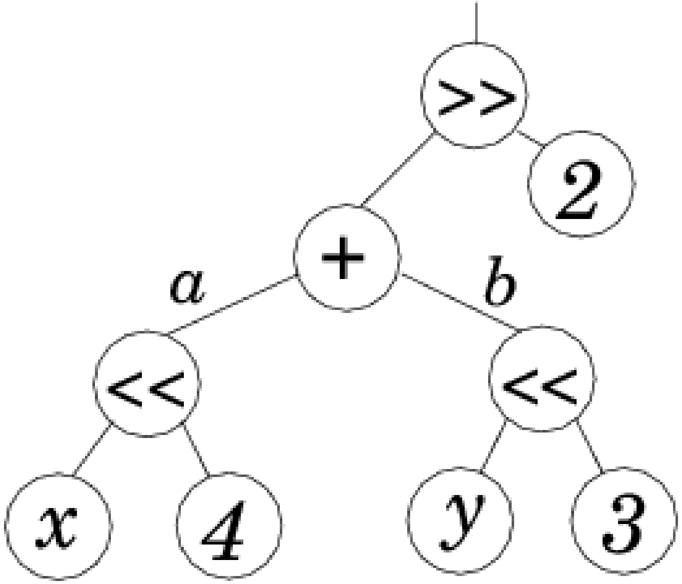


The parent nodes
Some sibling nodes
Some child nodes

- The addition of *a* and *b* may overflow

Example

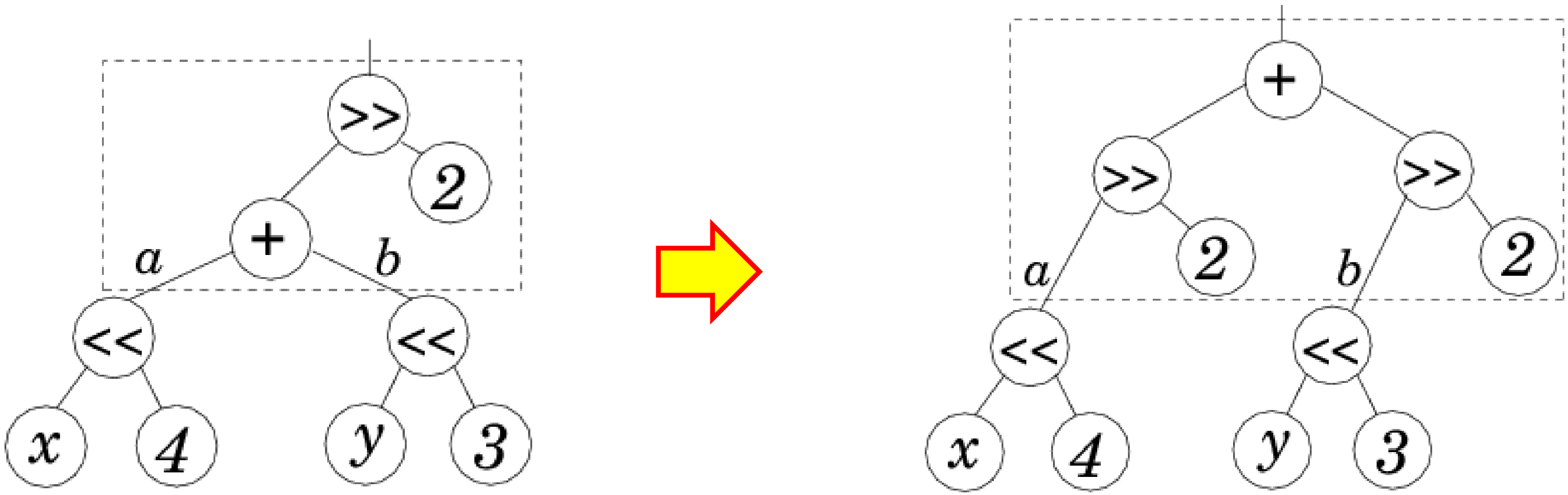
Computing Redundant LSBs



- The redundant LSBs of *a* are computed as 4 bits
- The redundant LSBs of *b* are computed as 3 bits.

Example

<u>Extracting Code Region</u>



- Extract the code surrounding the overflow operation.
- The new code requires a smaller bit-width.

Implementation

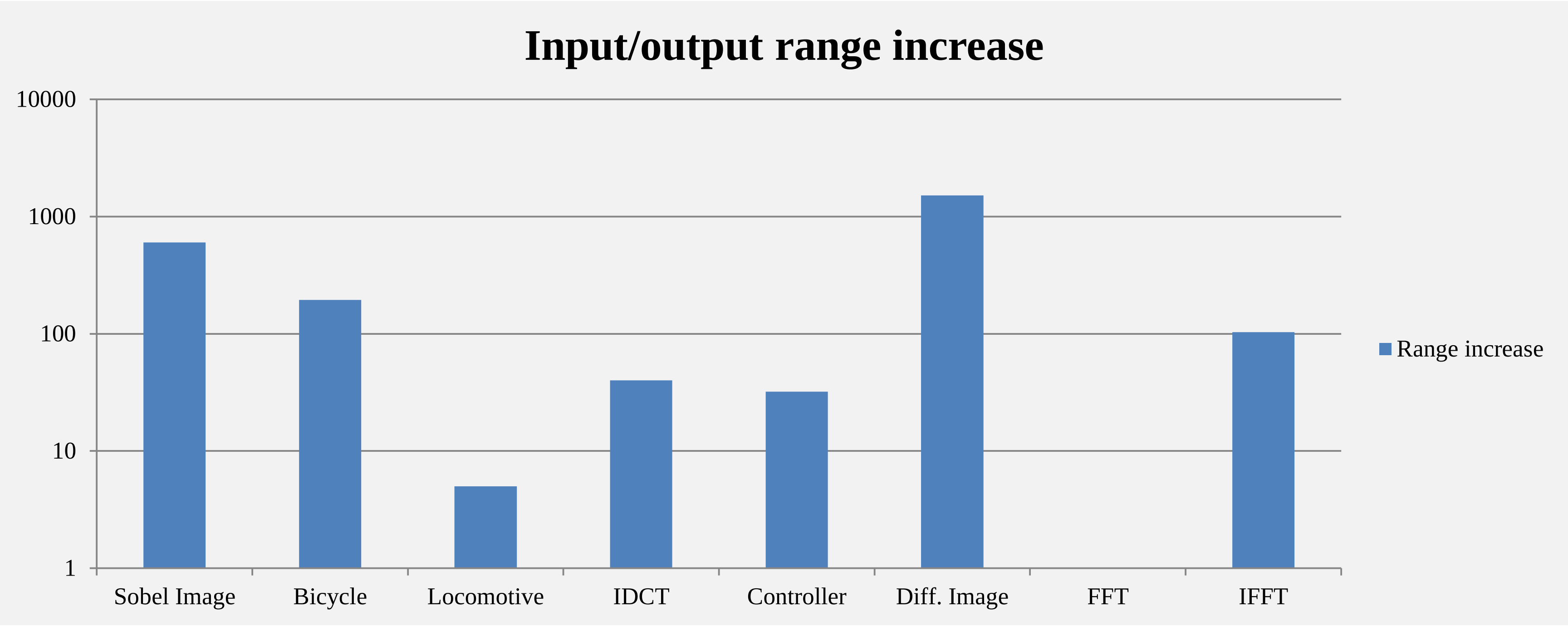
- Clang/LLVM + Yices SMT solver
- Bit-vector arithmetic theory
- Evaluated on a set of public benchmarks for embedded control and DSP applications

Benchmarks (*embedded control software*)

Benchmark	Bits	LoC	Arithmetic Operations	Citation
Sobel Image filter	32	42	28	Qureshi, 2005
Bicycle controller	32	37	27	Rupak, Saha & Zamani, 2012
Locomotive controller	64	42	38	Martinez, Majumdar, Saha & Tabuada, 2010
IDCT (N=8)	32	131	114	Kim, Kum, & Sung, 1998
Controller impl.	32	21	8	Martinez, Majumdar, Saha & Tabuada, 2010
Differ. image filter	32	131	77	Burger, & Burge, 2008
FFT (N=8)	32	112	82	Xiong, Johnson, & Padua,2001
IFFT (N=8)	32	112	90	Xiong, Johnson, & Padua,2001

All benchmark examples are public-domain examples

Experiment *(increase in range)*



- Average increase in range is 307%

  (602%, 194%, 5%, 40%, 32%, 1515%, 0% , 103%)

Experiment *(decrease in bit-width)*

Name of Benchmark	Original (bit-width)		Optimized (bit-width)	
	Minimum	Average	Minimum	Average
Sobel image filter (3x3)	17	10.26	15	6.67
Bicycle controller	18	14.47	16	14.16
Locomotive controller	33	29.41	32	29.32
IDCT (N=8)	20	16.29	19	16.38
Control. Impl.	17	15	16	14.67
Diff. image filter (5x5)	17	11.11	13	8.09
FFT (N=8)	18	7.32	16	6.95
IFFT (N=8)	17	7.11	16	7.26

- Required bit-width: ***32-bit → 16-bit***

  64-bit → 32-bit

Experiment *(scaling error)*

Benchmark	Scaling	Error original (Original program)	Error optimized (New program)
Sobel Image filter (3x3)	32-b $\rightarrow$ 16-b	$3.1 * 10^{-2}$	0.0
Bicycle controller	32-b $\rightarrow$ 16-b	$3.5 * 10^{-4}$	$2.0 * 10^{-4}$
Locomotive controller	64-b $\rightarrow$ 32-b	$2.9 * 10^{-8}$	$1.5 * 10^{-9}$
IDCT (N=8)	32-b $\rightarrow$ 16-b	$9.2 * 10^{-3}$	$1.8 * 10^{-5}$
Control. Impl.	32-b $\rightarrow$ 16-b	$5.2 * 10^{-4}$	$2.9 * 10^{-4}$
Diff. image filter (5x5)	32-b $\rightarrow$ 16-b	$1.2 * 10^{-2}$	$2.5 * 10^{-3}$
FFT (N=8)	32-b $\rightarrow$ 16-b	$8.1 * 10^{-2}$	$4.4 * 10^{-3}$
IFFT (N=8)	32-b $\rightarrow$ 16-b	$8.4 * 10^{-2}$	$3.2 * 10^{-}2$

If we reduce microcontroller's bit-width, how much error will be introduced?

Experiment *(runtime statistics)*

Benchmark	Optimized Code Regions	Time
Sobel image filter	22	2s
Bicycle controller	2	5s
Locomotive controller	1	5m 41s
IDCT (N=8)	3	2.7s
Controller impl.	1	46s
Differ. image filter	23	10s
FFT (N=8)	14	1m 9s
IFFT (N=8)	1	4s

64 bit

Conclusions

- We presented a new SMT-based method for optimizing fixed-point linear arithmetic computations in embedded software code
  - Effective in reducing the required bit-width
  - Scalable for practice use

- Future work
  - Other aspects of the performance optimization, such as execution time, power consumption, etc.

More on Related Work

- Solar-Lezama *et al*. **Programming by sketching for bit-streaming programs**, *ACM SIGPLAN'05*.
  - General program synthesis. Does not scale beyond 3-4 LoC for our application.
- Gulwani *et al*. **Synthesis of loop-free programs**, *ACM SIGPLAN'11*.
  - Synthesizing bit-vector programs. Largest synthesized program has 16 LoC, taking >45mins. Do not have incremental optimization.
- Jha. **Towards automated system synthesis using sciduction**, Ph.D. dissertation, UC Berkeley, 2011.
  - Computing the minimal required bit-width for fixed-point representation. Do not change the code structure.
- Rupak *et al*. **Synthesis of minimal-error control software**, EMSOFT'12.
  - Synthesizing fixed-point computation from floating-point computation. Again, only compute minimal required bit-widths, without changing code structure.