# Dynamic Analyses for Floating-Point Precision Tuning

Cindy Rubio-González
Department of Computer Science
*University of California, Davis*

Ecole Thématique Maths-Info-HPC – Lyon, France, May 2016

# Floating-Point Precision Tuning

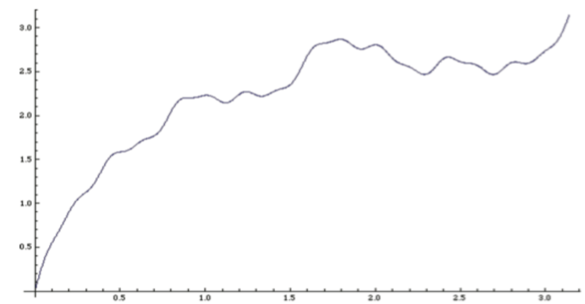
- Floating-point arithmetic used in variety of domains
- Reasoning about FP programs is difficult
  - Large variety of numerical problems
  - Most programmers are not experts in FP

- Common practice: use highest available precision
  - Disadvantage: more expensive!
- Goal: develop automated techniques to assist in tuning floating-point precision

# Example: Mixed Precision

- Consider the problem of finding the arc length of the function

$$g(x) = x + \sum_{0 \leq k \leq 5} 2^{-k} \sin(2^k x)$$




- Summing for $x_k \in (0, \pi)$ into n subintervals

$$\sum_{k=0}^{n-1} \sqrt{h^2 + (g(x_{k+1}) - g(x_k))^2} \quad \text{with} \quad h = \pi/n \quad \text{and} \quad x_k = kh$$

| | Precision | Slowdown | Result | |
|---|---|---|---|---|
| 1 | double-double | 20X | 5.79577632241285**6** | ✔ |
| 2 | double | 1X | 5.795776322413031 | ✖ |
| 3 | mixed precision | < 2X | 5.79577632241285**6** | ✔ |

# Example: Mixed Precision

```
long double g(long double x) {
  int k, n = 5;
  long double t1 = x;
  long double d1 = 1.0L;

  for(k = 1; k <= n; k++) {
     ...
  }
  return t1;
}

int main() {
  int i, n = 1000000;
  long double h, t1, t2, dppi;
  long double s1;
  ...
  for(i = 1; i <= n; i++) {
    t2 = g(i * h);
    s1 = s1 + sqrt(h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer stored in variable s1
  return 0;
}
```

Original Program

Tuned Program

# Example: Mixed Precision

```
long double g(long double x) {
  int k, n = 5;
  long double t1 = x;
  long double d1 = 1.0L;

  for(k = 1; k <= n; k++) {
     ...
  }
  return t1;
}

int main() {
  int i, n = 1000000;
  long double h, t1, t2, dppi;
  long double s1;
  ...
  for(i = 1; i <= n; i++) {
    t2 = g(i * h);
    s1 = s1 + sqrt(h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer stored in variable s1
  return 0;
}
```

Original Program

```
double g(double x) {
  int k, n = 5;
  double t1 = x;
  float d1 = 1.0f;

  for(k = 1; k <= n; k++) {
     ...
  }
  return t1;
}

int main() {
  int i, n = 1000000;
  double h, t1, t2, dppi;
  long double s1;
  ...
  for(i = 1; i <= n; i++) {
    t2 = g(i * h);
    s1 = s1 + sqrtf(h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer stored in variable s1
  return 0;
}
```

Tuned Program

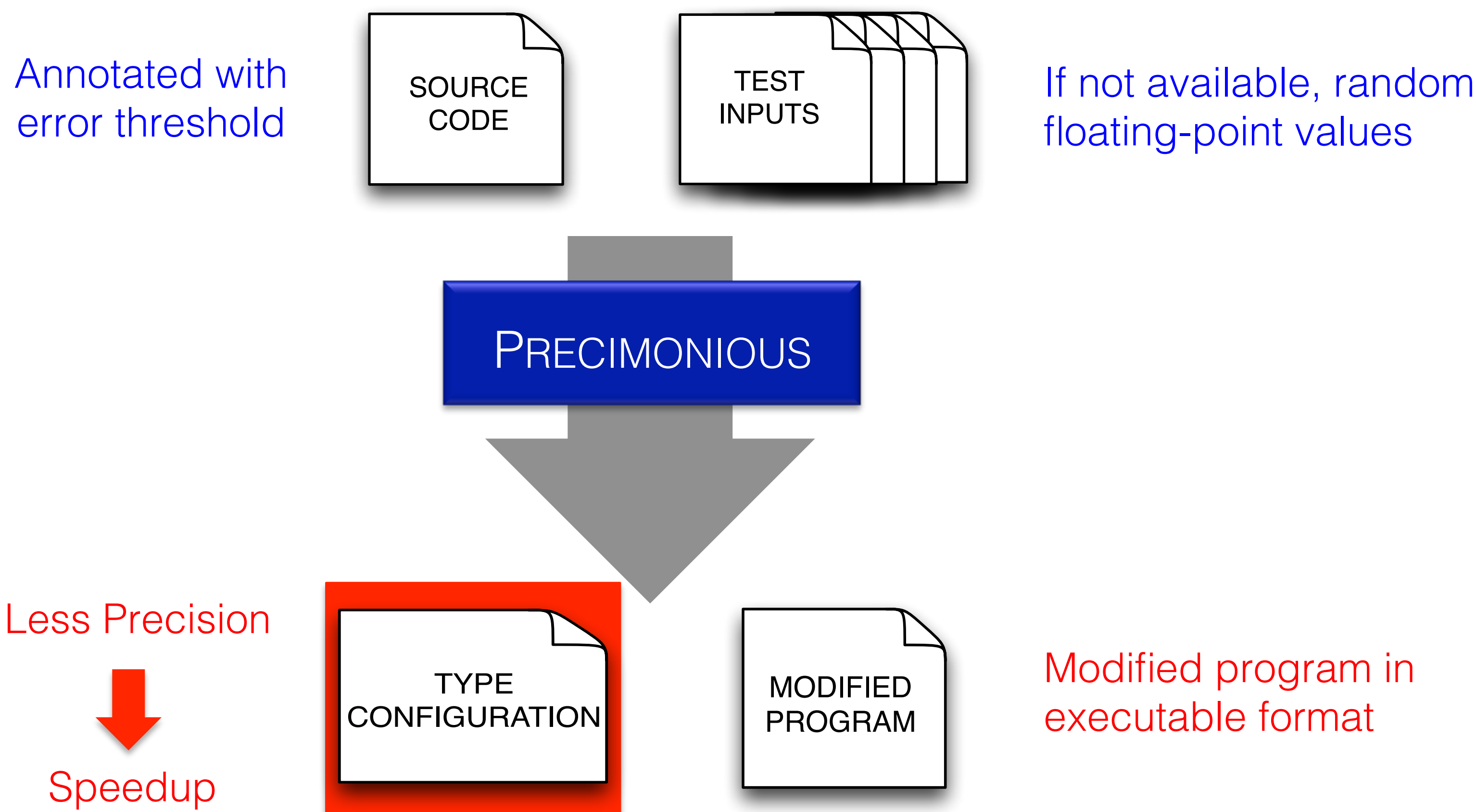
# PRECIMONIOUS

[SC'13]

# PRECIMONIOUS

*"Parsimonious or Frugal with Precision"*

Dynamic Analysis for Floating-Point Precision Tuning



# Challenges for Precision Tuning

- Searching efficiently over variable types and function implementations
  - Naïve approach → exponential time
    - 19,683 configurations for arc length program ($3^9$)
    - 11 hours 5 minutes
  - Global minimum vs. a local minimum
- Evaluating type configurations
  - Less precision → not necessarily faster
  - Based on run time, energy consumption, etc.
- Determining accuracy constraints
  - How accurate must the final result be?
  - What error threshold to use?

Automated

Specified by the user

# Searching: Delta Debugging

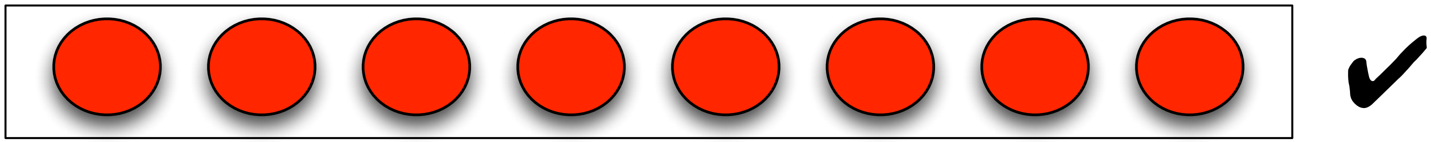
- Delta Debugging Search Algorithm [Zeller et. al]
  - An approach to debugging
  - Isolates failures systematically
    - Failing test → Isolate the change(s) that introduced failure
- Main idea:
  - We can do better than making a change at the time
  - Start by dividing the change set in two equally sized subsets
  - Narrow the search to the subset that still causes the failure
  - Otherwise, increase the number of subsets
- Efficient search algorithm
  - Average time complexity: $O(n \log n)$
  - Worst case: $O(n^2)$

# LCCSearch Algorithm

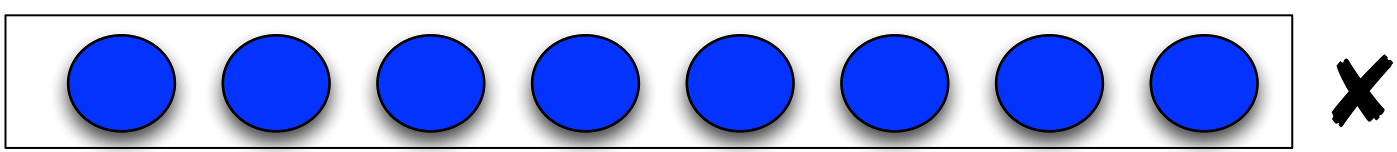
- Based on the Delta-Debugging Search Algorithm [Zeller et. Al]
- Our definition of a change
  - Lowering the precision of a floating-point variable in the program
    - Example: double x → float x
- Our success criteria
  - Resulting program produces an "accurate enough" answer
  - Resulting program is faster than the original program
- Main idea:
  - Start by associating each variable with set of types
    - Example: x → {long double, double, float}
  - Refine set until it contains only one type
- Find a local minimum
  - Lowering the precision of one more variable violates success criteria

# Searching for Type Configuration

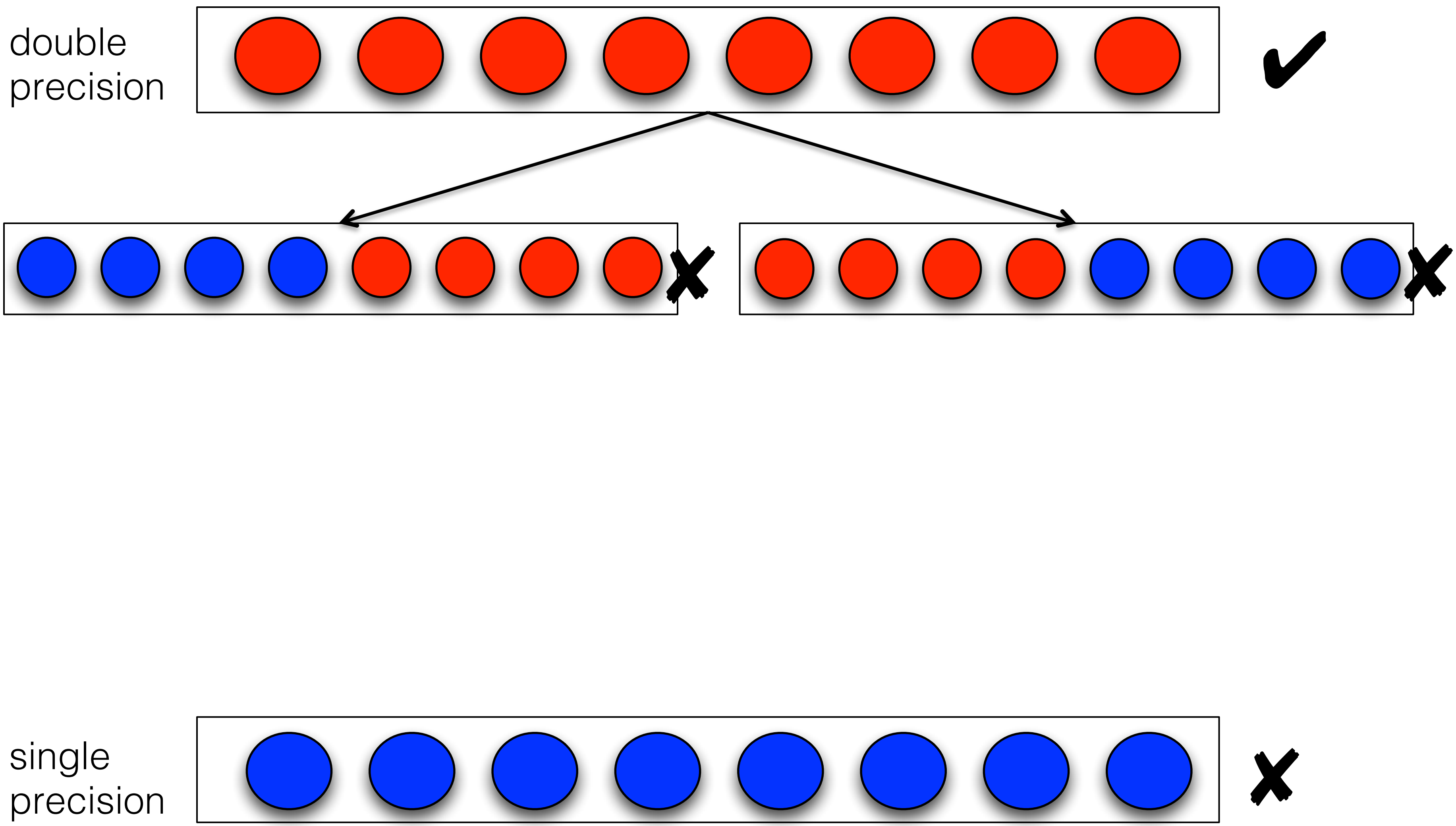
double
precision



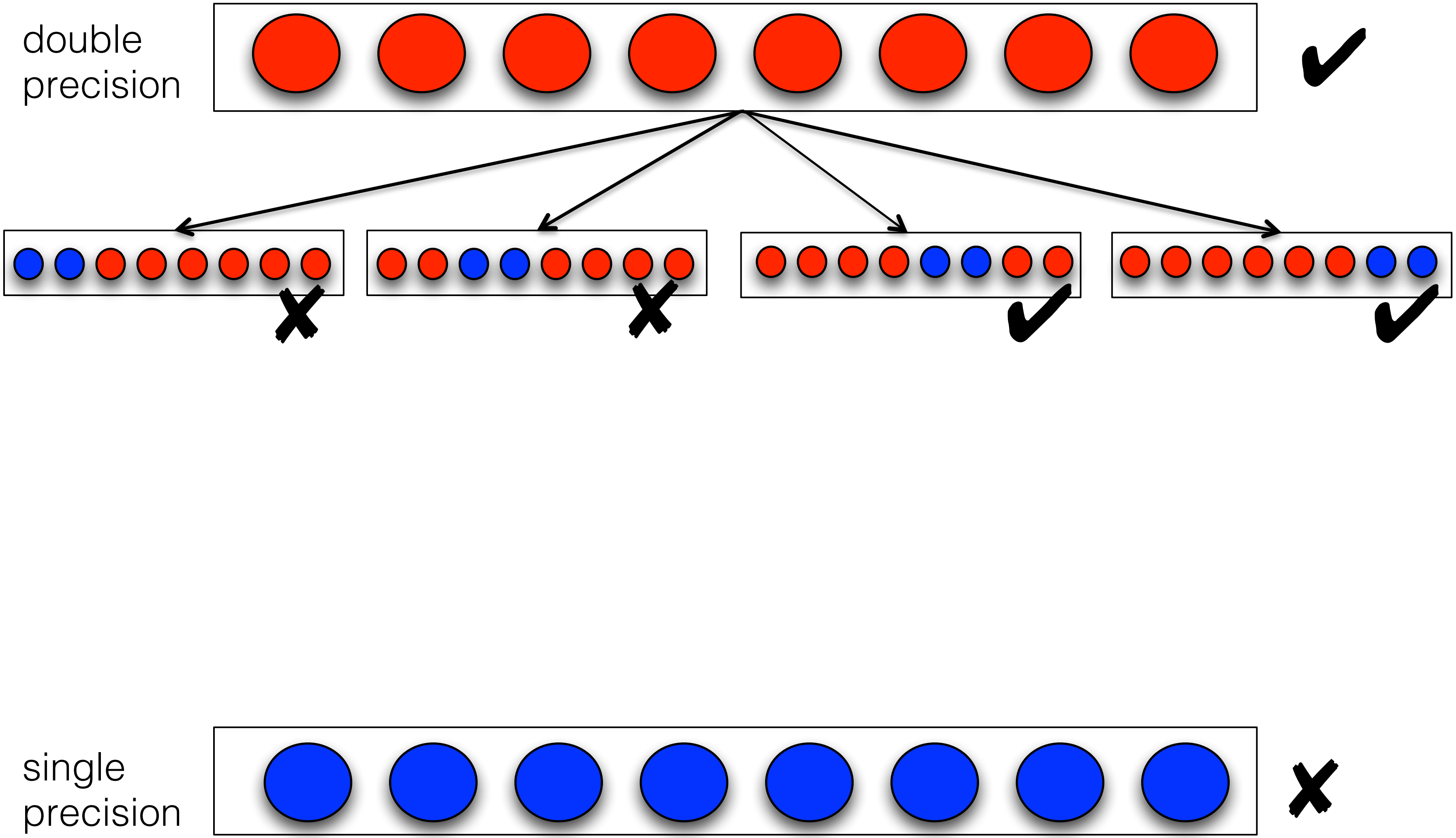
single
precision



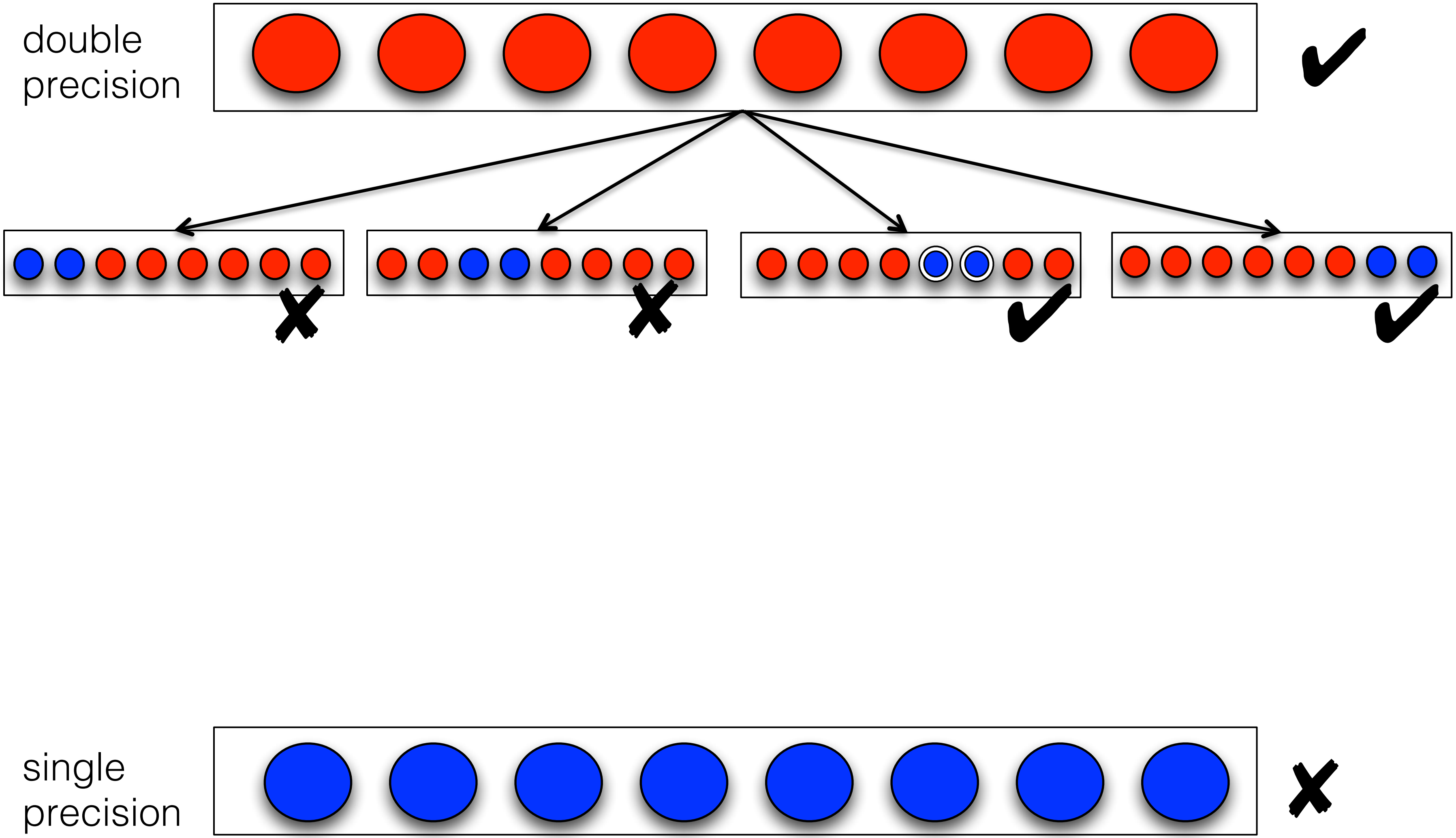
# Searching for Type Configuration



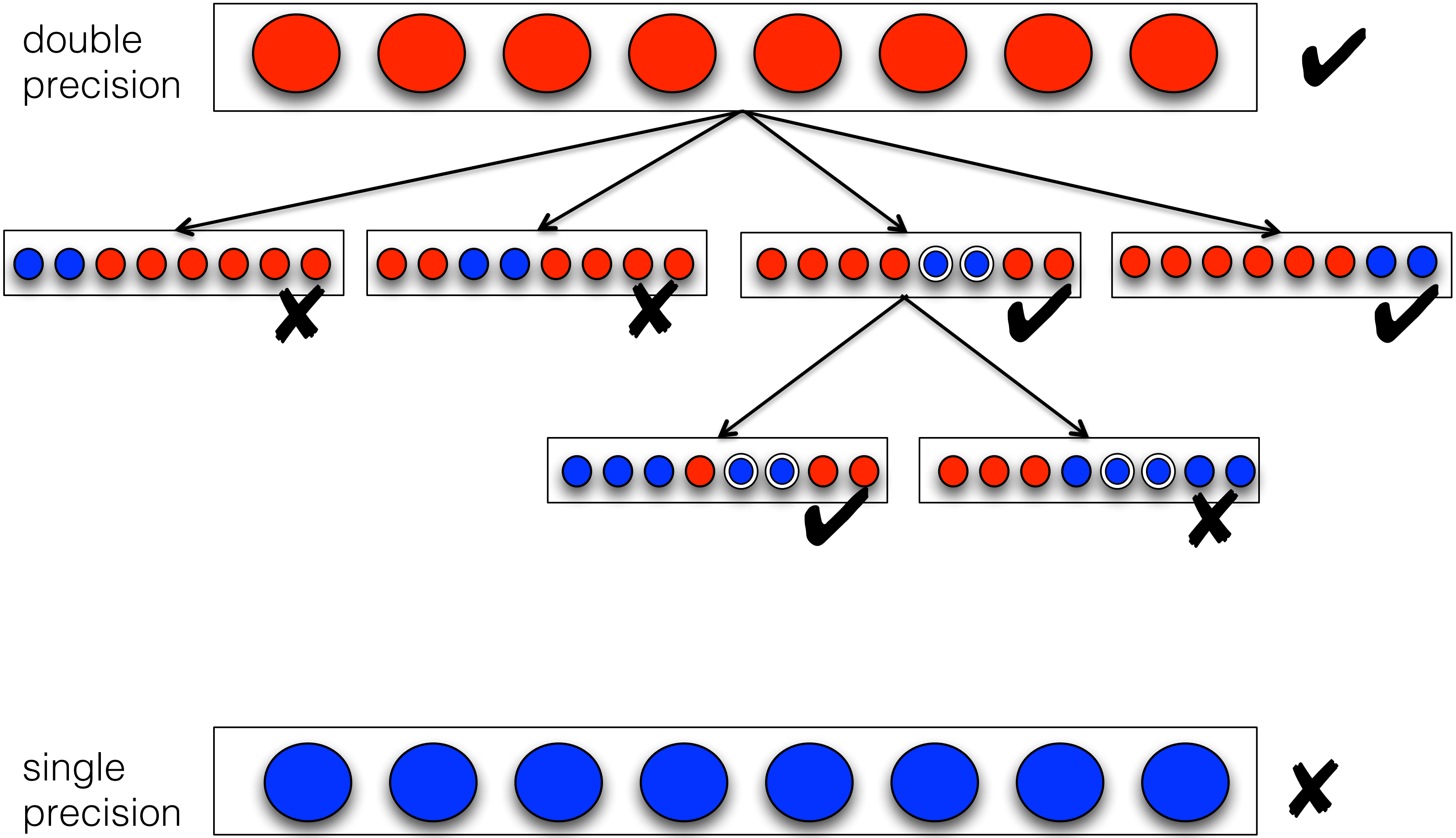
# Searching for Type Configuration



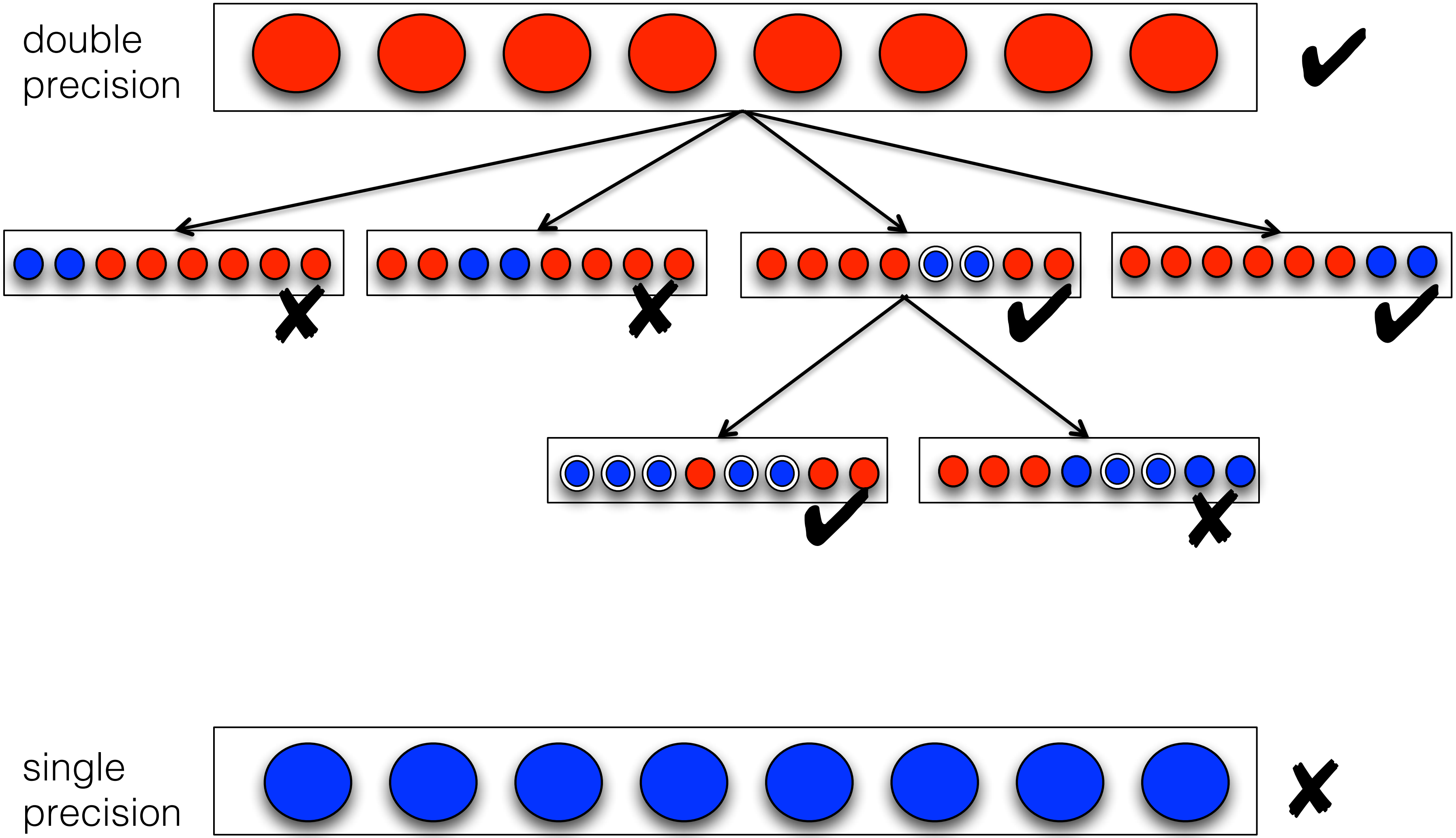
# Searching for Type Configuration



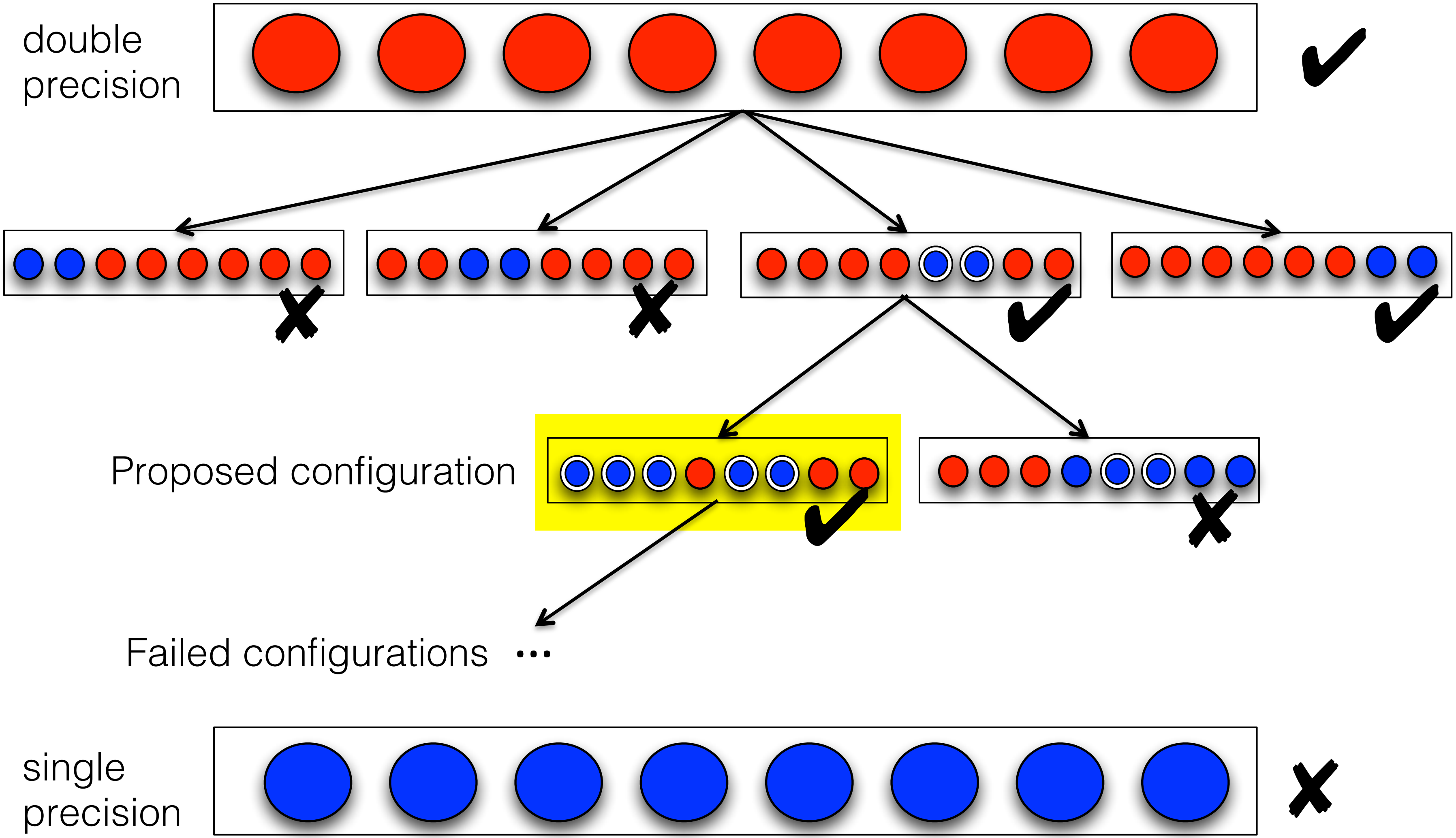
# Searching for Type Configuration



# Searching for Type Configuration



# Searching for Type Configuration



# Applying Type Configurations

- Automatically generate program variants
  - Reflect type configurations produced by search algorithm
- Intermediate representation
  - LLVM IR
- Transformation rules for each LLVM instruction
  - `alloca`, `load`, `store`, `fpext`, `fptrunc`, `fadd`, `fsub`, etc.
  - Changes equivalent to modifying the program at the source level
- Able to run resulting modified program

# Experimental Setup

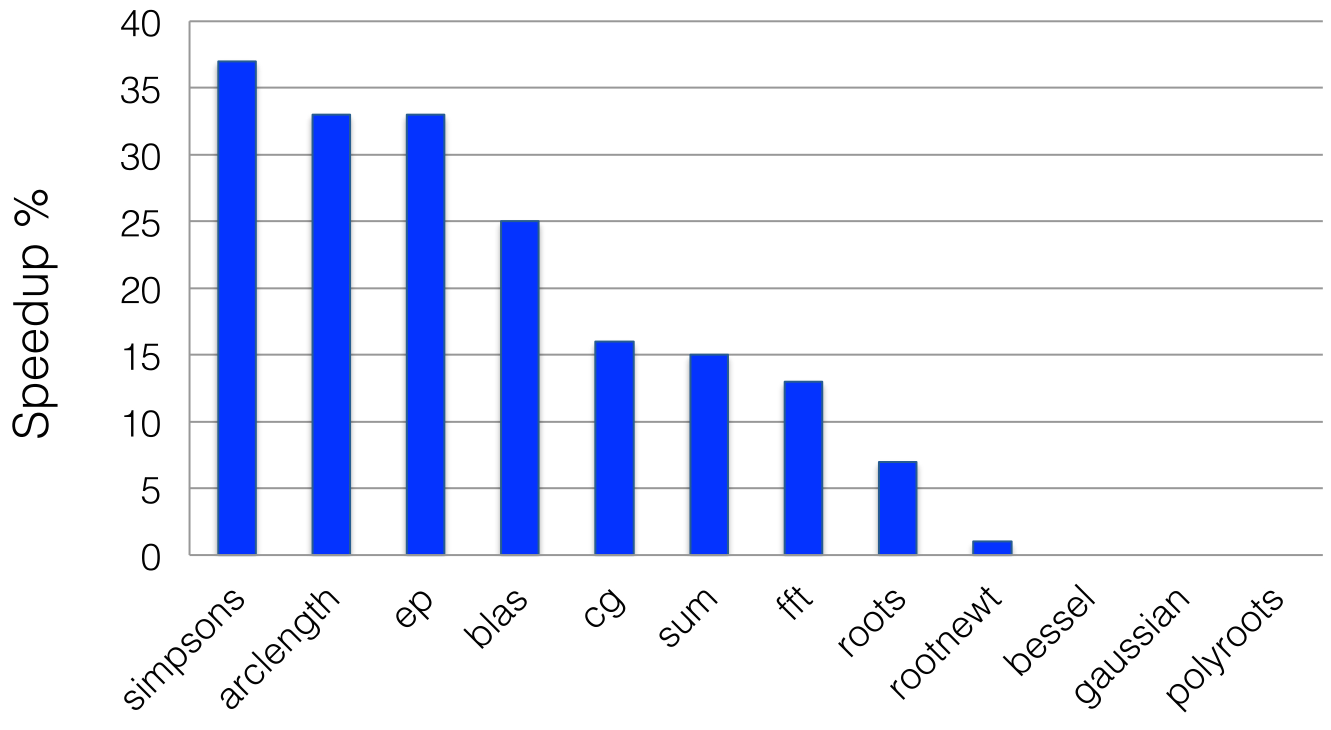
- Benchmarks
  - 8 GSL programs
  - 2 NAS Parallel Benchmarks: *ep* and *cg*
  - 2 other numerical programs
- Test inputs
  - Inputs Class A for *ep* and *cg* programs
  - 1000 random floating-point inputs for the rest
- Error thresholds
  - Multiple error thresholds: $10^{-4}$,$10^{-6}$, $10^{-8}$, and $10^{-10}$
  - User can evaluate trade-off between accuracy and speedup

# Experimental Results

| Group | Program | Original L | Original D | Original F | Original Calls | Proposed L | Proposed D | Proposed F | Proposed Calls | # Config | mm:ss |
|---|---|---|---|---|---|---|---|---|---|---|---|
| GSL | bessel | 0 | 18 | 0 | 0 | 0 | 18 | 0 | 0 | 130 | 37:11 |
| GSL | gaussian | 0 | 52 | 0 | 0 | 0 | 52 | 0 | 0 | 201 | 16:12 |
| GSL | roots | 0 | 19 | 0 | 0 | 0 | 0 | 19 | 0 | 3 | 1:03 |
| GSL | polyroots | 0 | 28 | 0 | 0 | 0 | 28 | 0 | 0 | 336 | 43:17 |
| GSL | rootnewt | 0 | 12 | 0 | 0 | 0 | 4 | 8 | 0 | 61 | 16:56 |
| GSL | sum | 0 | 31 | 0 | 0 | 0 | 9 | 22 | 0 | 325 | 28:14 |
| GSL | fft | 0 | 22 | 0 | 0 | 0 | 0 | 22 | 0 | 3 | 1:16 |
| GSL | blas | 0 | 17 | 0 | 0 | 0 | 0 | 17 | 0 | 3 | 1:06 |
| NAS | EP | 0 | 13 | 0 | 4 | 0 | 5 | 8 | 4 | 111 | 23:53 |
| NAS | CG | 0 | 32 | 0 | 3 | 0 | 2 | 30 | 3 | 44 | 0:57 |
| | arclength | 9 | 0 | 0 | 3 | 0 | 2 | 7 | 3 | 33 | 0:40 |
| | simpsons | 9 | 0 | 0 | 2 | 0 | 0 | 9 | 2 | 4 | 0:07 |

Original Type Configuration; Proposed Type Configuration, Error threshold: $10^{-4}$

# Speedup for Error Threshold $10^{-4}$




Maximum speedup observed across all error thresholds: 41.7%

DEMO

# BLAME ANALYSIS

[ICSE'16]

# BLAME ANALYSIS

- Goal: alleviate scalability limitations of existing search-based FP precision tuning approaches
  - Reduce number of executions/transformations
  - Perform local, fine-grained isolated transformations
- Executes the program only *once* while performing shadow execution
- Focuses on accuracy, not performance
- Best results when used to prune the search space of PRECIMONIOUS

# High-Level Approach

Execution

```c
int main() {
 double a = 1.84089642;
 double res, t1, t2, t3, t4;
 double r1, r2, r3;

 t1 = 4*a;
 t2 = mpow(a, 6, 2);
 t3 = mpow(a, 4, 3);
 t4 = mpow(a, 1, 4);

 // res = a^4 - 4a^3 + 6a^2 - 4a + 1
 r1 = t4 - t3;
 r2 = r1 + t2;
 r3 = r2 - t1;
 res = r3 + 1;

 printf("res = %.10f\n", res);
 return 0;
}
```

Target instruction & precision

Concrete + Shadow

Two main components:

1. Shadow execution runs the program both in single and double precision
2. Blame analysis determines precision requirements for each program instruction

# Shadow Execution

- Floating-point value associated with shadow value
- Shadow value defined as

| Shadow Value | double | float |
| --- | --- | --- |

- Shadow execution computes on shadow values
- Maintains shadow memory and label map

SHADOW MEMORY

$$M : A \rightarrow S$$

LABEL MAP

$$LM : A \rightarrow L$$

$A$: set of all memory addresses

$S$: set of all shadow values

$L$: set of all instruction labels

# Shadow Execution in Action

```
z = x - y; // label l1
FSubShadow(x, y, z, l1); // instrumentation
```

1. SHADOW MEMORY → Retrieve shadow values for operands → x_shadow: $x_d$ | $x_s$; y_shadow: $y_d$ | $y_s$

2. z_shadow: $x_d - y_d$ | $x_s - y_s$ → Update shadow value for **&z** → SHADOW MEMORY

3. Instruction `l1` last to compute value `z` → Update label for **&z** → LABEL MAP

# BLAME ANALYSIS - Local Precision

- Determines for each instruction *i* and each precision *p* the precision requirements for the operands so that *i* has at least precision *p*
- We consider various precisions *p*
  - `fl`✔, $\mathtt{db}_4$, $\mathtt{db}_6$, $\mathtt{db}_8$, $\mathtt{db}_{10}$, `db`✔
  - Example: computing $\mathbf{db}_8$ from **db** value

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | . | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 3 | db |
| 0 | . | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | ~~1~~ | ~~3~~ | $\mathtt{db}_8$ |

8 significant digits

# Example – Local Precision

Instruction: `z = x - y`

Precision: $db_8$

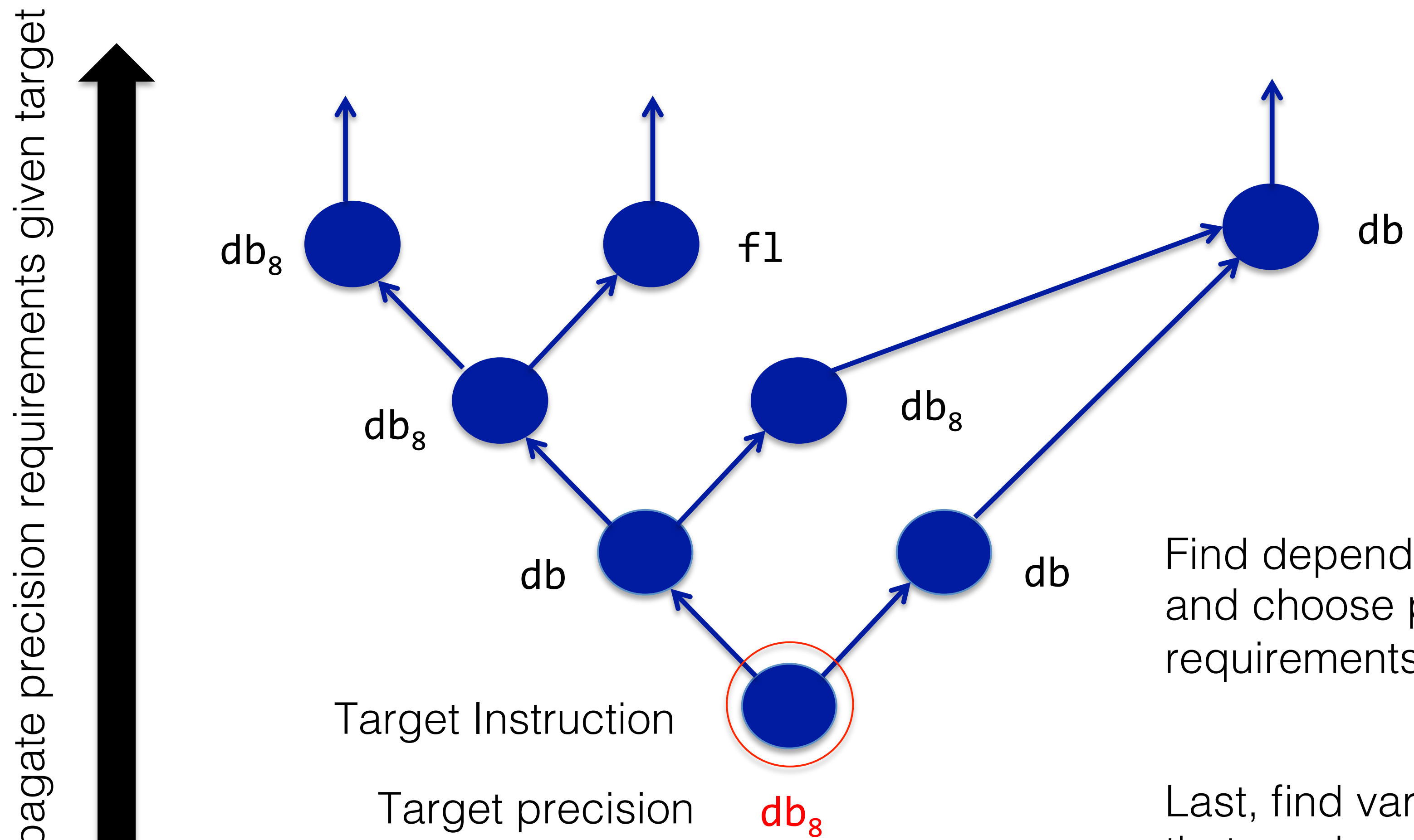
z's db value: -0.4999999887
z's $db_8$ target value: -0.499999988

Assume: P = {fl, $db_8$, db}

| Precision | x | y | z |
|---|---|---|---|
| (fl, fl) | 6.8635854721 | 7.3635854721 | -0.5000000000 |
| (fl, $db_8$) | 6.8635854721 | 7.3635856000 | -0.5000001279 |
| (fl, db) | 6.8635854721 | 7.3635856800 | -0.5000002079 |
| ($db_8$, fl) | 6.8635856000 | 7.3635854721 | -0.4999998721 |
| ($db_8$, $db_8$) | 6.8635856000 | 7.3635856000 | -0.5000000000 |
| … | … | … | … |
| (db, db) | 6.8635856913 | 7.3635856800 | -0.4999999887 |

Operands require precision (db, db) for result to be at least $db_8$

# BLAME ANALYSIS - Global Precision




Find dependencies, and choose precision requirements

Last, find variables that can be allocated in single precision

# Experimental Evaluation

- Evaluation in different settings
  - BLAME ANALYSIS by itself
  - BLAME ANALYSIS + PRECIMONIOUS (B+P)
  - Compared to PRECIMONIOUS (P)
- Benchmarks
  - 2 NAS Parallel Benchmarks (ep and cg)
  - 8 GSL programs
- Test inputs
  - Inputs Class A for ep and cg programs
  - 1000 random floating-point inputs for the rest
- Error thresholds
  - Multiple error thresholds: $10^{-4}$, $10^{-6}$, $10^{-8}$, and $10^{-10}$
  - User can evaluate trade-off between accuracy and speedup

# Analysis Performance (I)

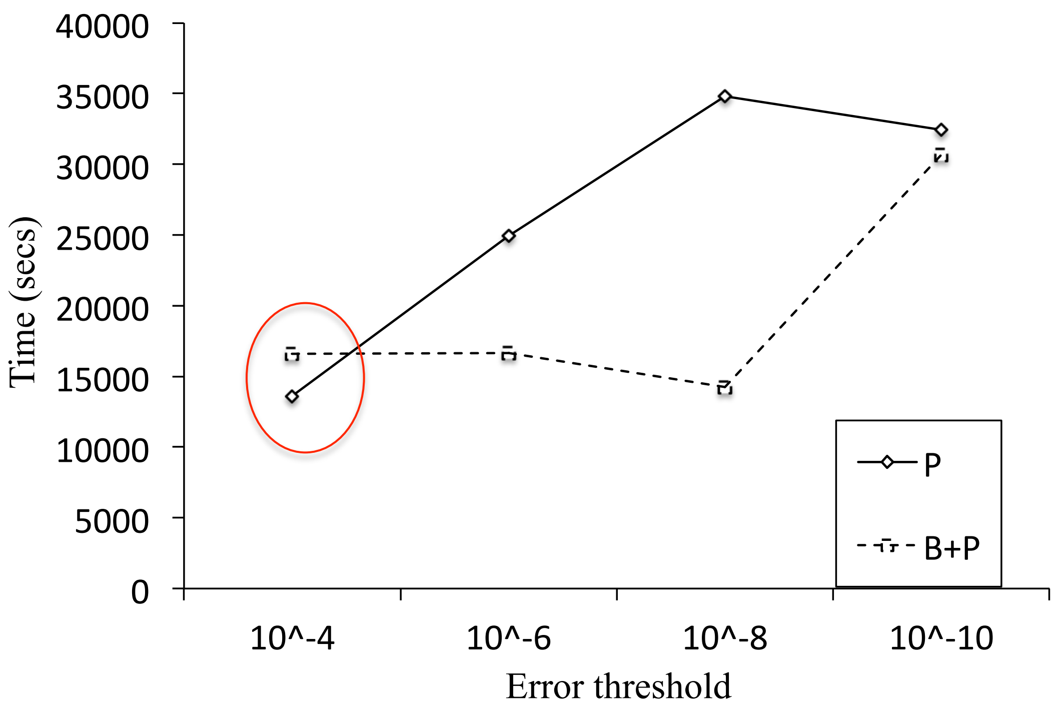
- BLAME ANALYSIS introduces 50x slowdown
- B+P is faster than P in 31 out of 39 experiments

| Program | Speedup | Program | Speedup |
|---|---|---|---|
| bessel | 22.48x | sum | 1.85x |
| gaussian | 1.45x | fft | 1.54x |
| roots | 18.32x | blas | 2.11x |
| polyroots | 1.54x | ep | 1.23x |
| rootnewt | 38.42x | cg | 0.99x |

Combined analysis time is 9x faster on average, and up to 38x in comparison with PRECIMONIOUS alone

# Analysis Performance (II)

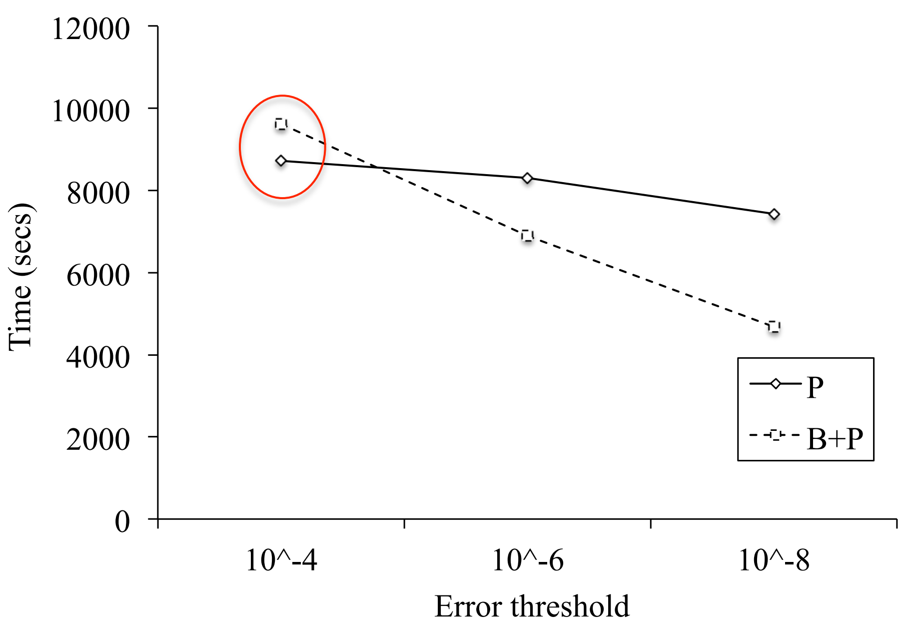
- B+P is slower in 8 out of 39 experiments
- Example: different search path makes P more expensive

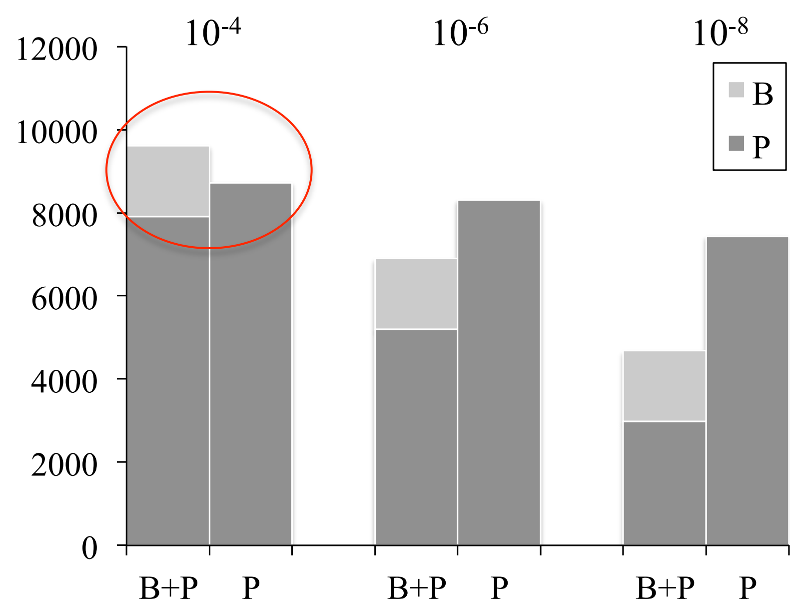



gaussian

# Analysis Performance (III)

- B+P is slower in 8 out of 39 experiments
- Example: combined analysis more expensive than P




ep

# Analysis Results (I)

- BLAME ANALYSIS identifies at least 1 float variable in all 39 experiments
- Overall, BLAME ANALYSIS removes 40% of the variables from the search space (117 out of 293 variables), median 28%
- B+P and P agree on 28 out of 39 experiments
- B+P is slightly better in remaining 11 experiments

# Analysis Results (II)

Original Type Configuration

| | Program | D | F |
|---|---|---|---|
| GSL | bessel | 26 | 0 |
| | gaussian | 56 | 0 |
| | roots | 16 | 0 |
| | polyroots | 31 | 0 |
| | rootnewt | 14 | 0 |
| | sum | 34 | 0 |
| | fft | 22 | 0 |
| | blas | 17 | 0 |
| NAS | ep | 45 | 0 |
| | cg | 32 | 0 |

Proposed Type Configurations

Error threshold: $10^{-4}$

| B | | B+P | |
|---|---|---|---|
| D | F | D | F |
| 1 | 25 | x | x |
| 54 | 2 | x | x |
| 1 | 15 | x | x |
| 10 | 21 | 10 | 21 |
| 1 | 13 | x | x |
| 24 | 10 | 11 | 23 |
| 16 | 6 | 0 | 22 |
| 1 | 16 | 0 | 17 |
| 42 | 3 | 42 | 3 |
| 26 | 6 | 2 | 30 |

No configuration speeds up the program

BLAME ANALYSIS finds good configuration

B+P finds a better configuration

Many variables lowered to single precision

# Analysis Results (II)

Original Type Configuration

Proposed Type Configurations
Error threshold: $10^{-4}$

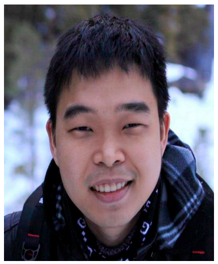
| | Program | D | F | B D | B F | B+P D | B+P F | P D | P F |
|---|---|---|---|---|---|---|---|---|---|
| GSL | bessel | 26 | 0 | 1 | 25 | x | x | x | x |
| GSL | ga[illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | x | x |
| GSL | ro[illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | [illegible] | x | x |
| GSL | polyroots | 31 | 0 | 10 | 21 | 10 | 21 | x | x |
| GSL | rootnewt | 14 | 0 | 1 | 13 | x | x | x | x |
| GSL | sum | 34 | 0 | 24 | 10 | 11 | 23 | 11 | 23 |
| GSL | fft | 22 | 0 | 16 | 6 | 0 | 22 | 0 | 22 |
| GSL | blas | 17 | 0 | 1 | 16 | 0 | 17 | 0 | 17 |
| NAS | ep | 45 | 0 | 42 | 3 | 42 | 3 | x | x |
| NAS | cg | 32 | 0 | 26 | 6 | 2 | 30 | 2 | 30 |

**Program speedup up to 40%**

P does not find a configuration

# Collaborators

## University of California, Berkeley



Cuong Nguyen

Diep Nguyen

Ben Mehne

James Demmel

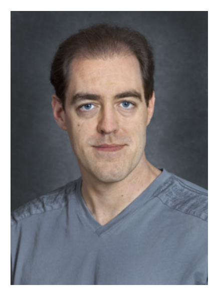
William Kahan

Koushik Sen

## Lawrence Berkeley National Lab

Costin Iancu

David Bailey



Wim Lavrijsen

## Oracle

David Hough