EECS
Electrical Engineering and Computer Sciences

PRECIMONIOUS: Tuning Assistant for Floating-Point Precision

Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen
University of California, Berkeley

David H. Bailey, Costin Iancu
Lawrence Berkeley National Laboratory

David Hough
Oracle

ECS 289C Seminar on Program Analysis
February 24, 2015

Motivation

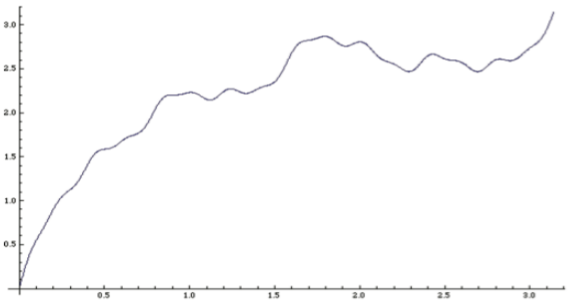
- Floating-point arithmetic used in wide variety of domains
- Reasoning about these programs is difficult
  - Large variety of numerical problems
  - Most programmers not experts FP

- Common practice: use highest available precision
  - Disadvantages: more expensive!
- Automated technique for tuning precision
  - Search over variable types → type configurations
  - Program produces “accurate enough” answer and runs faster

Example (D.H. Bailey)

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

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



- 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	5.795776322412856	✔
2	double	1X	5.795776322413031	✖
3	Summation variable is double-double	< 2X	5.795776322412856	✔

Example (D.H. Bailey)

```
long double fun(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 = fun(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 (D.H. Bailey)

```
long double fun(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 = fun(i * h);
    s1 = s1 + sqrt(h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer stored in variable s1
  return 0;
}
```

Original Program

```
double fun(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 = fun(i * h);
    s1 = s1 + sqrtf(h*h + (t2 - t1)*(t2 - t1));
    t1 = t2;
  }
  // final answer stored in variable s1
  return 0;
}
```

Tuned Program

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.

Automated

- Determining accuracy constraints
  - How accurate must the final result be?
  - What error threshold to use?

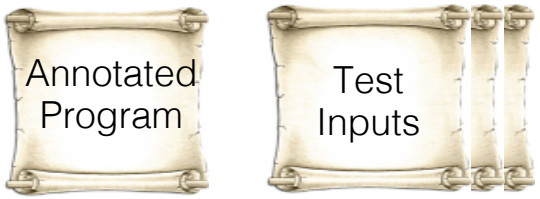
Specified by the user

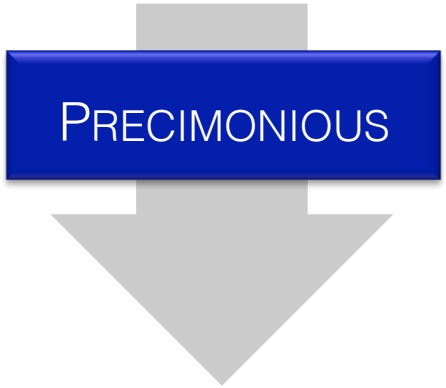
Precimonious

"Parsimonious or frugal with precision"

Dynamic Program Analysis for Floating-Point Precision Tuning

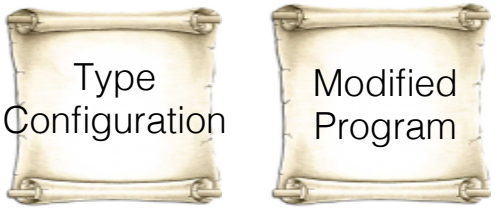
Annotated with Error Threshold



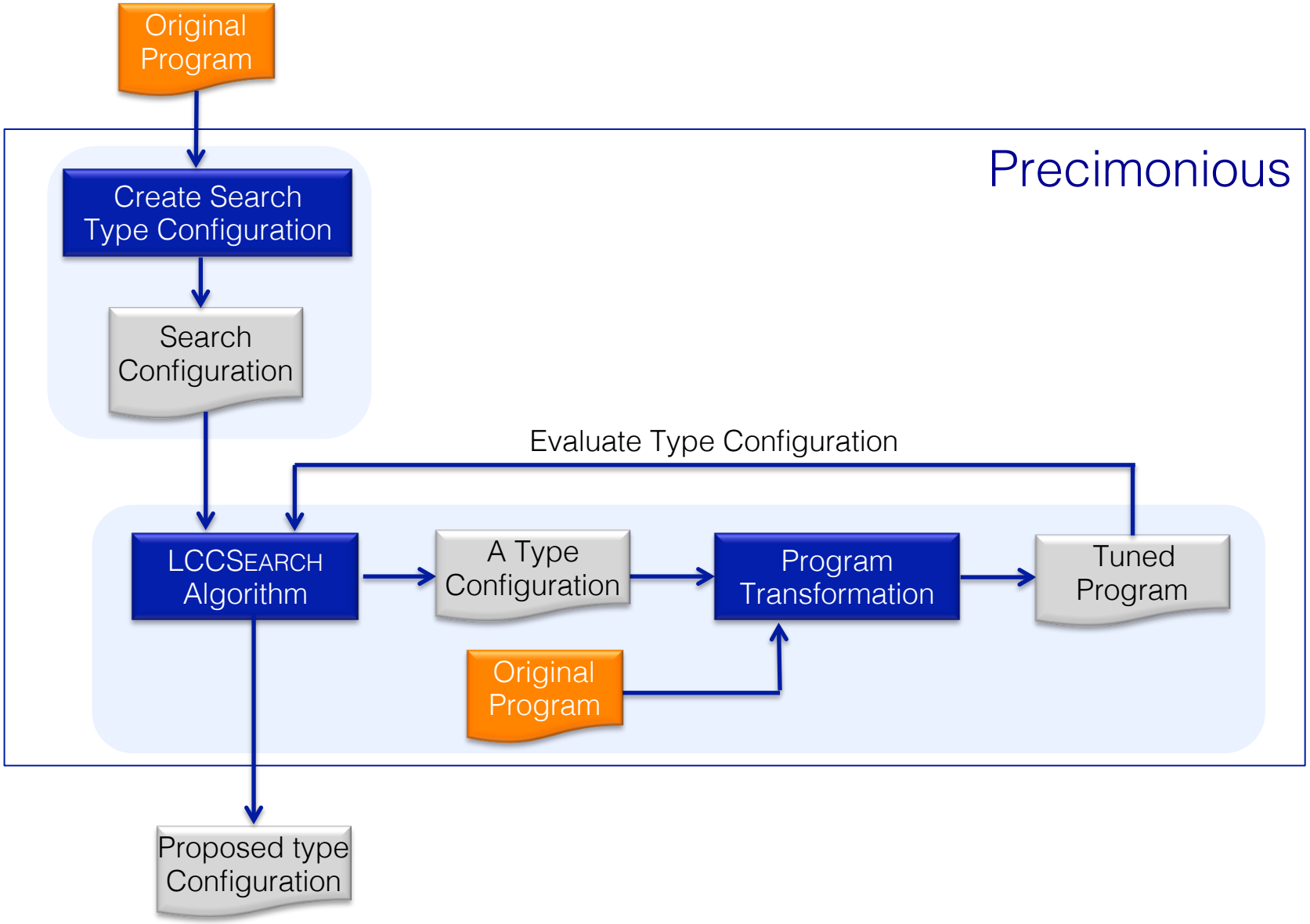


Less Precision

Speedup



Analysis Components



Searching: Delta Debugging

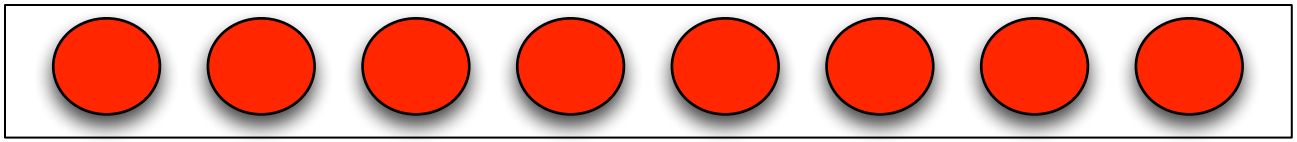
- Delta Debugging Search Algorithm [Zeller et. al]
  - An approach to debugging
  - Isolates failures systematically
    - Failing test ➔ Isolate the change(s) that introduced failure
- Main idea:
  - We can do better than making a change at the time
  - Start by dividing the change set in two equally 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

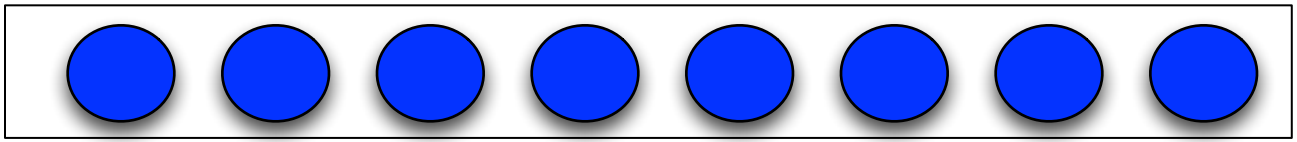
Searching for Type Configuration

double precision



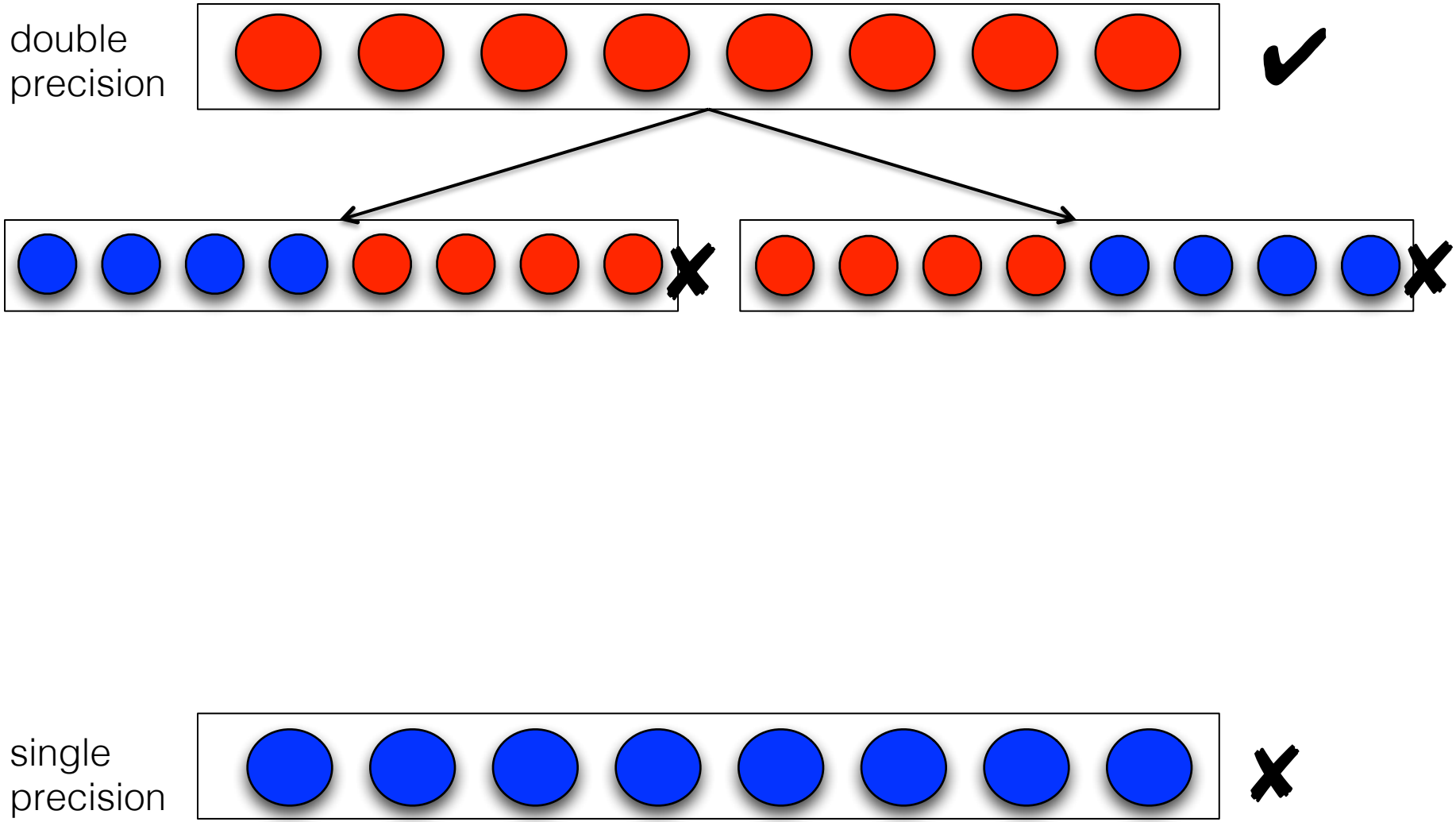
✔

single precision

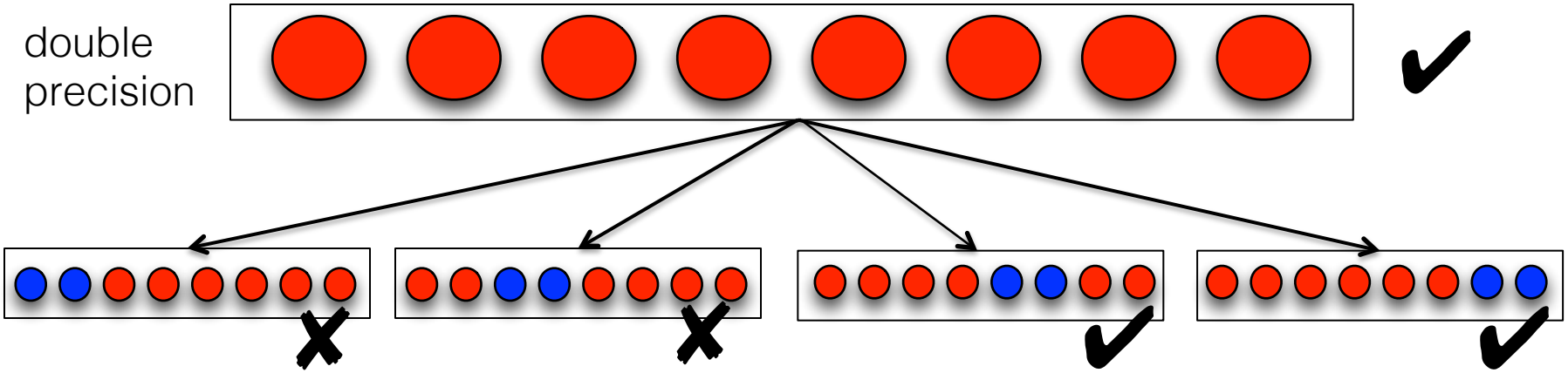


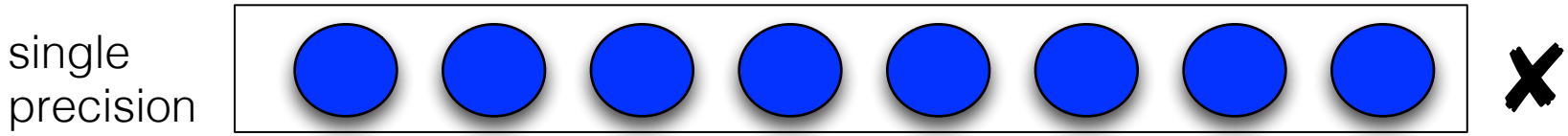
✘

Searching for Type Configuration

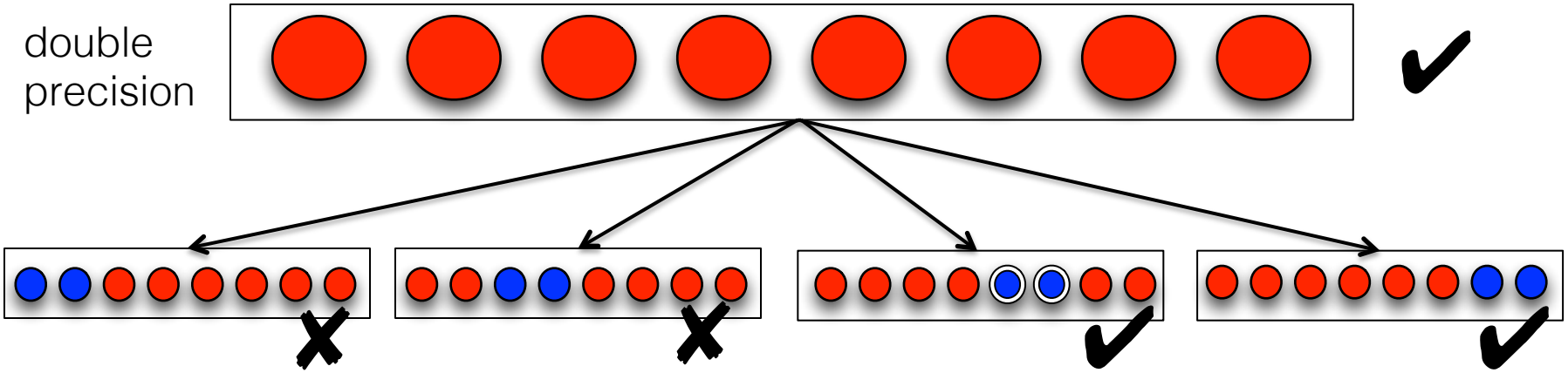


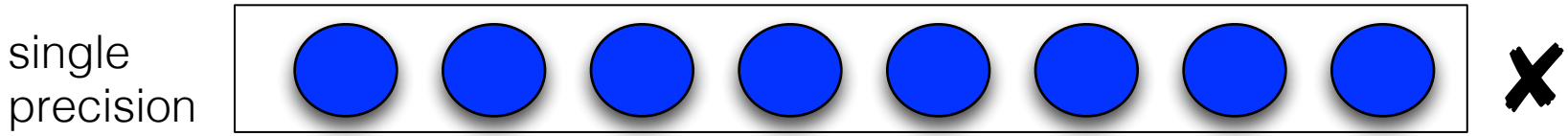
Searching for Type Configuration



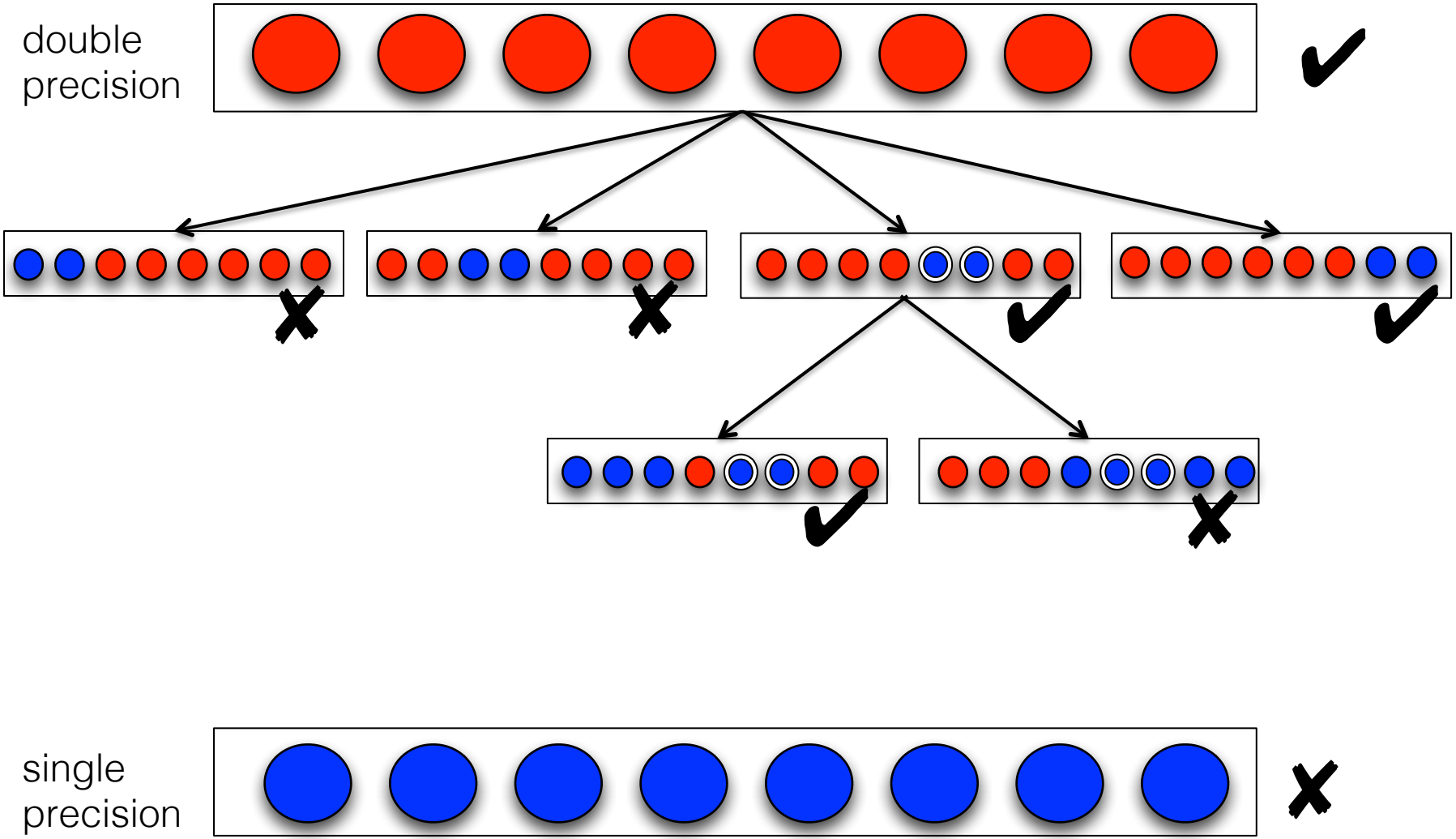


Searching for Type Configuration

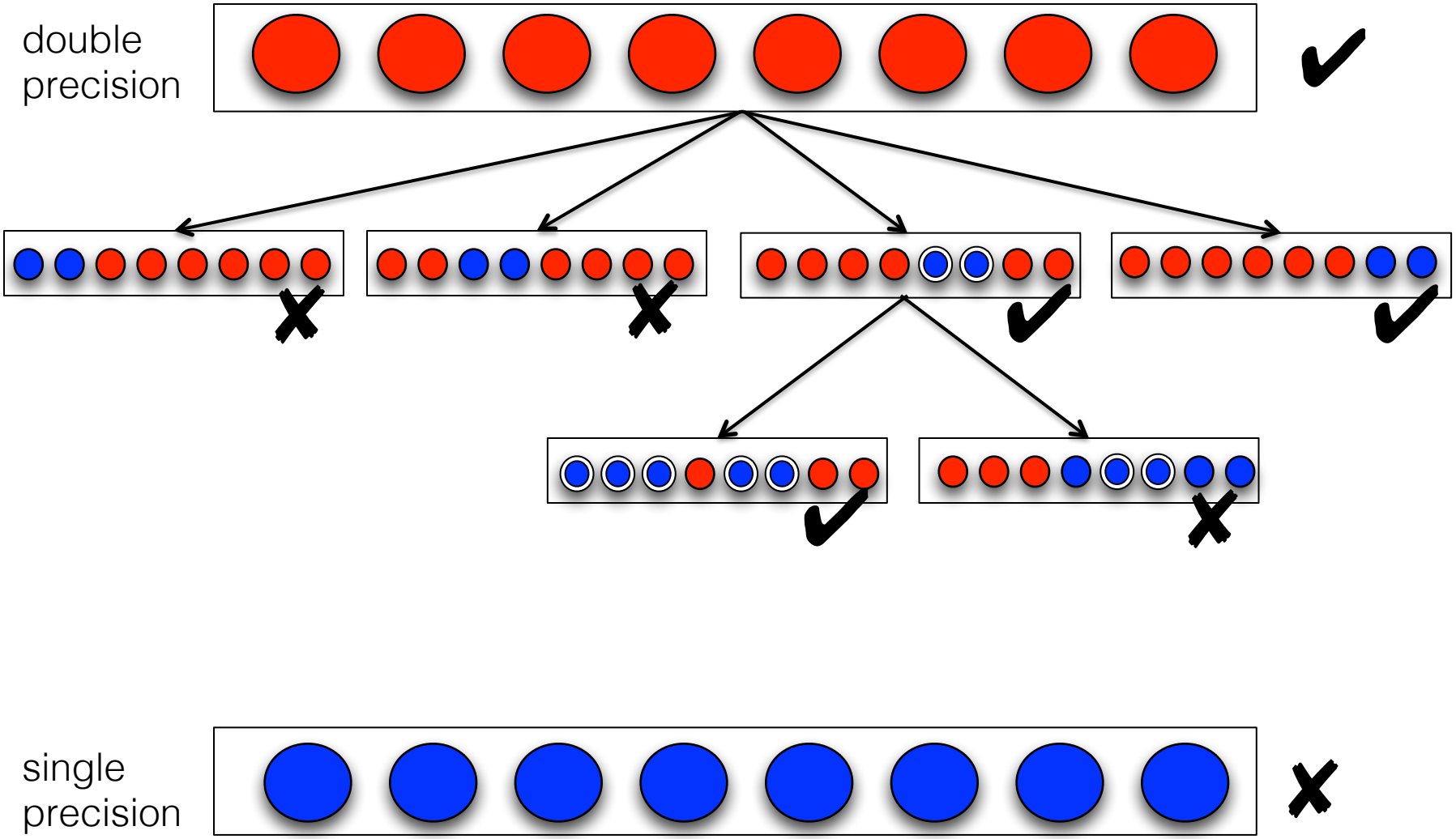




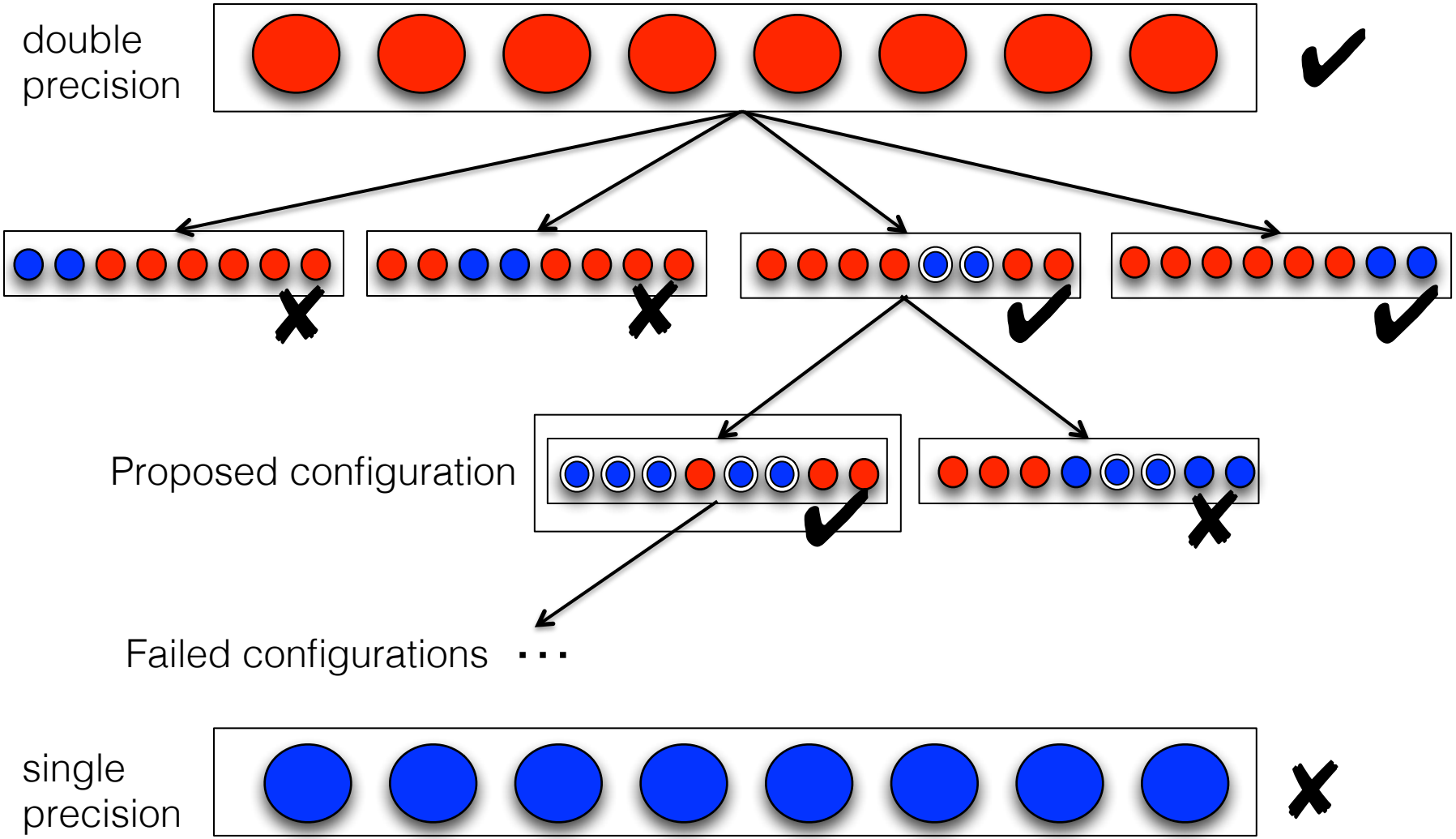
Searching for Type Configuration



Searching for Type Configuration



Searching for Type Configuration



Program Transformation

- Automatically generate program variants
  - Reflect type configurations produced by LCCSEARCH algorithm
- Intermediate representation
  - LLVM IR
- Define 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

Intermediate Representation

Source Code

```
long double fun(long double x) {
 int k, n = 5;
 long double t1;
 long double d1 = 1.0L;
 t1 = x;
 for(k = 1; k <= n; k++) {
   d1 = 2.0 * d1;
   t1 = t1 + sin(d1 * x) / d1;
 }
 return t1;
}
```

LLVM IR

```
define x86_fp80 @fun(x86_fp80) {

}
```

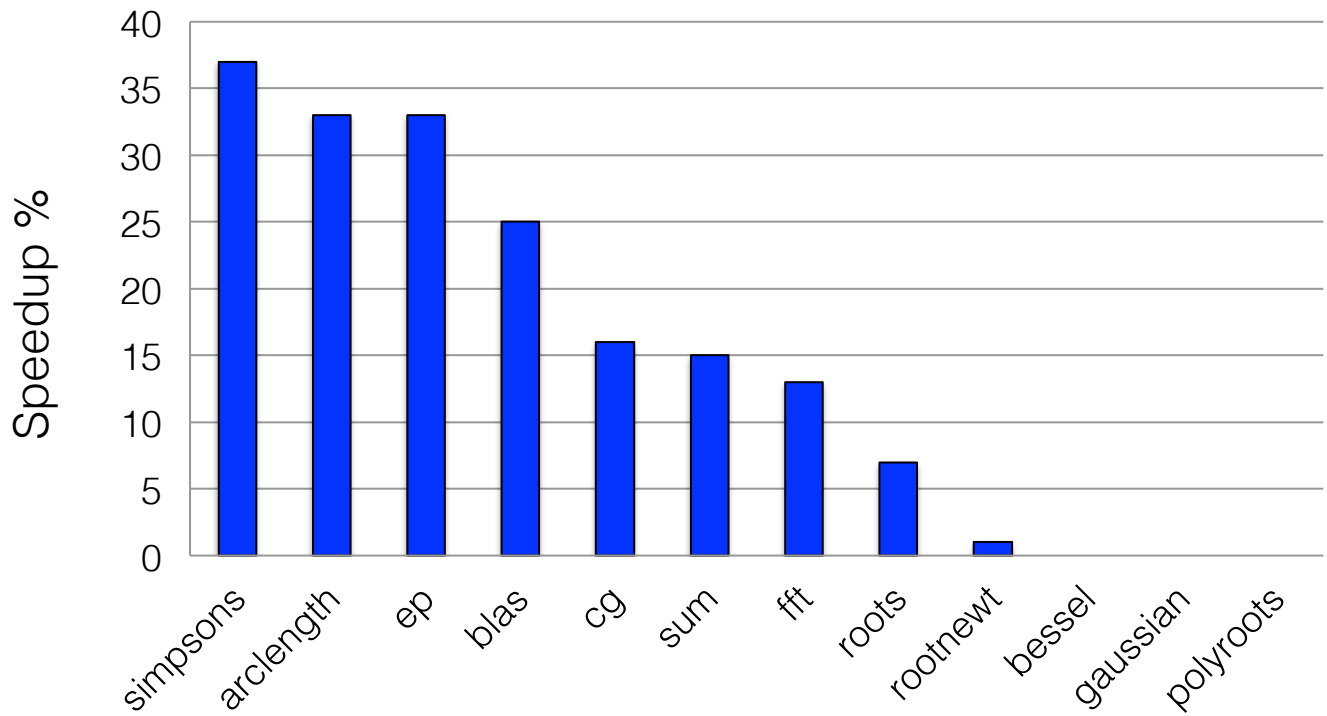
Experimental Setup

- 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^{-10}, 10^{-8}, 10^{-6}, and 10^{-4}
  - User can evaluate trade-off between accuracy and speedup

Experimental Results

		Original Type Configuration				Proposed Type Configuration (Error threshold: 10^{-4})					
	Program	L	D	F	C	L	D	F	S	# 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

Speedup for Error Threshold 10^{-4}



Maximum speedup observed across all error thresholds: 41.7%

Additional Example

Calculating infinite sum $\sum_{k>0}(3465/((k+1/2)^2-1/16)+3465/(k^2-1/16))=9240$

```
int main() {
  double sum, oldsum;
  long int k;

  sum = 0.0;
  oldsum = -1.0;
  k = 0;

  while(sum > oldsum) {
    oldsum = sum;
    k = k + 1;
    sum = sum + term(k);
  }

  // infinite sum
  sum = sum + tail(k);
  return 0;
}
```

- If single precision is sufficient, number of terms reduces from 87,290,410 to only 3,768
- Unlike other programs, the arithmetic's precision determines number of iterations
- Speedup as high as 5000x

Limitations and Future Work

- Type configurations rely on program inputs tested
  - No guarantees if worse conditioned input
  - Additional experiments to assess inputs used in evaluation
- Getting trapped in local minimum
  - Improve search by exploiting relationships among variables
- Providing support for other data types such as double-double as implemented in the QD library
  - Emulating higher precision in software is significantly more expensive

Conclusions

- Devised a dynamic analysis for tuning the precision of floating-point programs
- Implemented in an efficient and publicly available tool named PRECIMONIOUS

  https://github.com/corvette-berkeley/precimonious
- Initial evaluation on 12 programs shows encouraging speedups of up to 41%
- PRECIMONIOUS is under active development
  - Feature suggestions and programs to analyze are welcome!