UC DAVIS
UNIVERSITY OF CALIFORNIA

Scalable Precision Tuning of Numerical Software

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

Best Practices for HPC Software Developers Webinar, October 14th, 2020

Floating-Point Precision Tuning

- Reasoning about floating-point programs is difficult
  - Large variety of numerical problems
  - Most programmers not expert in floating point

- Common practice: use highest available precision
  - Disadvantage: more expensive!

- Automated techniques for tuning precision

  Given : *Accuracy Requirement*

  Action: *Reduce precision*

  Goal : *Accuracy and/or Performance*

Precision Tuning Example

```
long double fun(long double p) {
long double pi = acos(-1.0);
long double q = sin(pi * p);
     return q;
}

void simpsons() {
long double a, b;
long double h, s, x;
const long double fuzz = 1e-26;
const int n = 2000000;
…
L100:
     x = x + h;
     s = s + 4.0 * fun(x);
     x = x + h;
    if (x + fuzz >= b) goto L110;
     s = s + 2.0 * fun(x);
     goto L100;
L110:
     s = s + fun(x);
     …
}
```

Original Program

Tuned Program

Error threshold 10^{-8}

Precision Tuning Example

```
long double fun(long double p) {
long double pi = acos(-1.0);
long double q = sin(pi * p);
     return q;
}

void simpsons() {
long double a, b;
long double h, s, x;
const long double fuzz = 1e-26;
const int n = 2000000;
…
L100:
     x = x + h
     s = s + 4
     x = x + h;
    if (x + fuzz >= b) goto L110;
     s = s + 2.0 * fun(x);
     goto L100;
L110:
     s = s + fun(x);
     …
}
```

Original Program

```
long double fun(double p) {
double pi = acos(-1.0);
long double q = sinf(pi * p);
     return q;
}

void simpsons() {
float a, b;
double s, x; float h;
const long float fuzz = 1e-26;
const int n = 2000000;
…
     x = x + h;
    if (x + fuzz >= b) goto L110;
     s = s + 2.0 * fun(x);
     goto L100;
L110:
     s = s + fun(x);
     …
}
```

Tuned Program

Tuned program runs 78.7% faster!

Challenges in Precision Tuning

- Searching efficiently over variable types and function implementations
  - Naïve approach → exponential time
    - 2^n or 3^n where n is the number of variables
  - 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?

Precision Tuning Approaches

- Reducing precision vs. improving performance
  - Different objectives
- Dynamic vs. static approaches
  - *Dynamic*: Performed at runtime, requires program inputs, handles larger and more complex code, no guarantees for untested inputs
  - *Static*: Analyzes program without running it, limitations with certain program structures (e.g., loops), formal guarantees for analyzed code
- Instructions vs. variables vs. function calls
  - Various granularities of program transformation
  - Different scopes
- Binary vs. IR vs. source code
  - Tradeoff between granularity of transformation and tool usability

Dynamic Tools for Precision Tuning

Precimonious

- Dynamic Analysis for Precision Tuning
  - Black-box approach to systematically search over variable types and functions

HiFPTuner

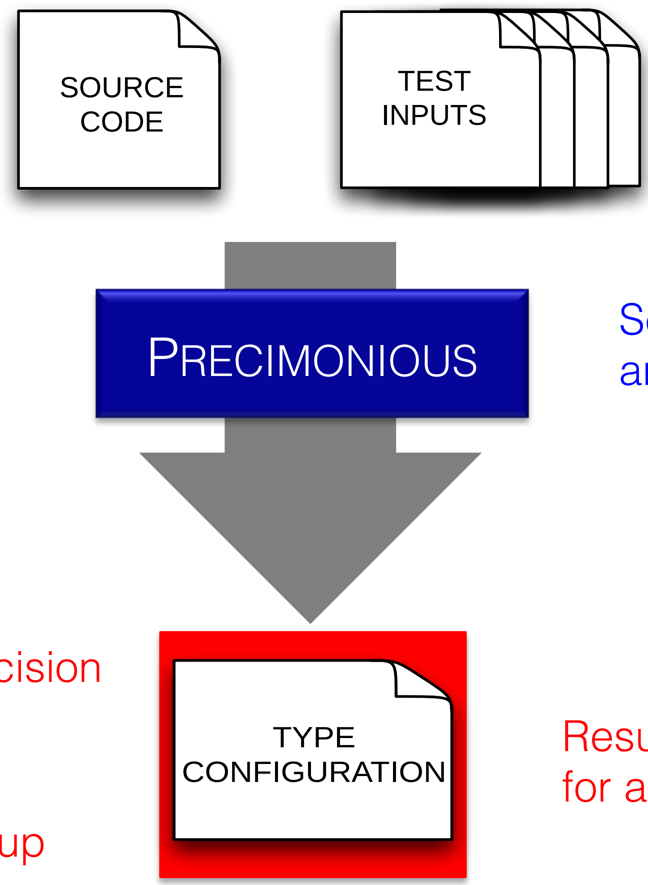
- Hierarchical Precision Tuner
  - Leverages relationship among variables to reduce search space and number of runs

PRECIMONIOUS

Dynamic Analysis for Floating-Point Precision Tuning

https://github.com/ucd-plse/precimonious

Annotated with error threshold



Search over types of variables and function implementations

Less Precision

Speedup

Result within error threshold for all test inputs

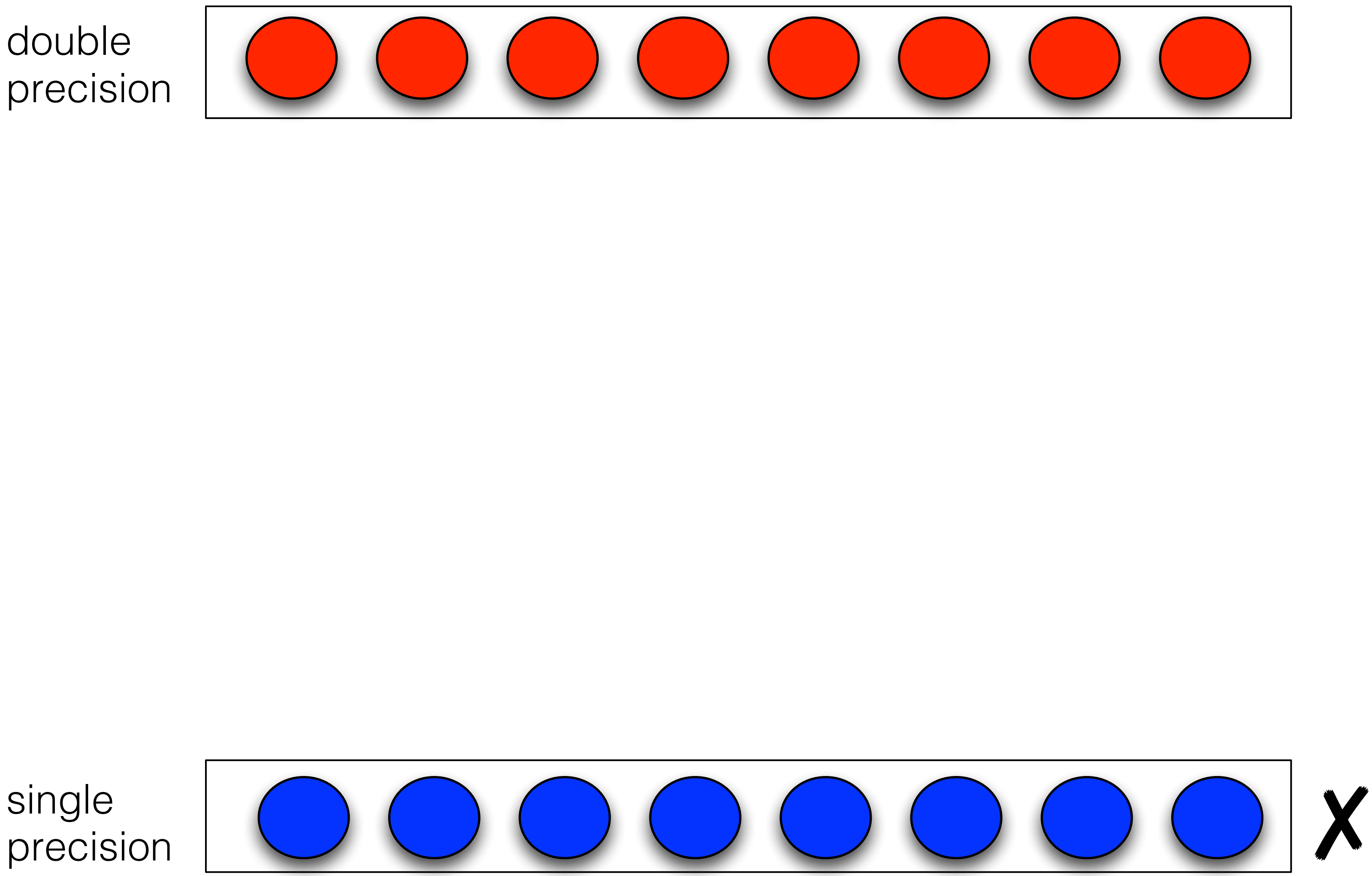
C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D.H. Bailey, C. Iancu, and D. Hough. "Precimonious: Tuning Assistant for Floating-Point Precision", SC 2013.

Search Algorithm

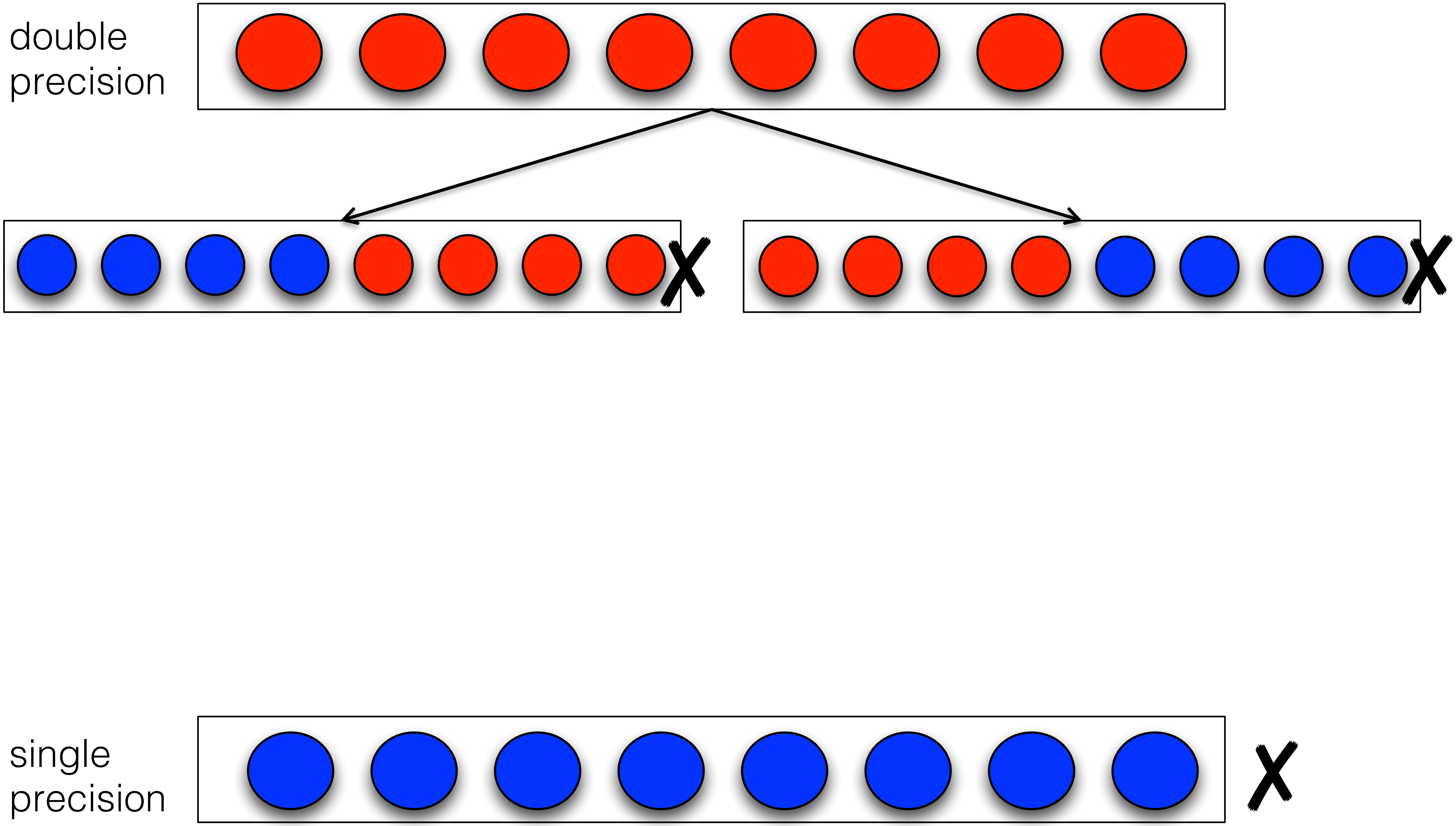
- Based on the Delta-Debugging Search Algorithm [1]
- Change the types of variables and function calls
  - Examples: double x → float x, sin → sinf
- Our success criteria
  - Resulting program produces an "accurate enough" answer
  - Resulting program is **faster** than the original program
- Main idea
  - 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

[1] A. Zeller and R. Hildebrandt. "Simplifying and Isolating Failure-Inducing Input", TSE 2002.

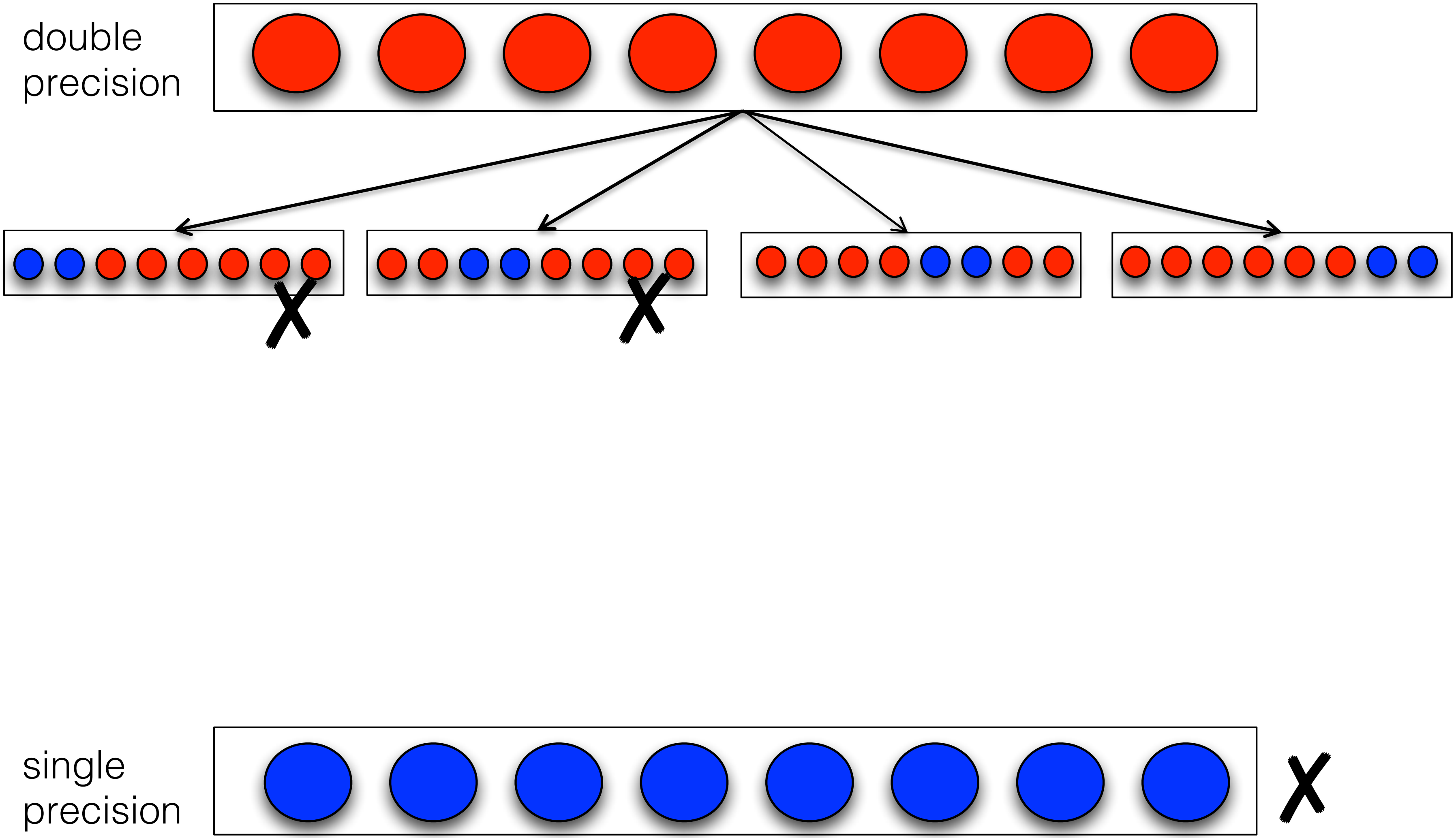
Searching for Type Configuration



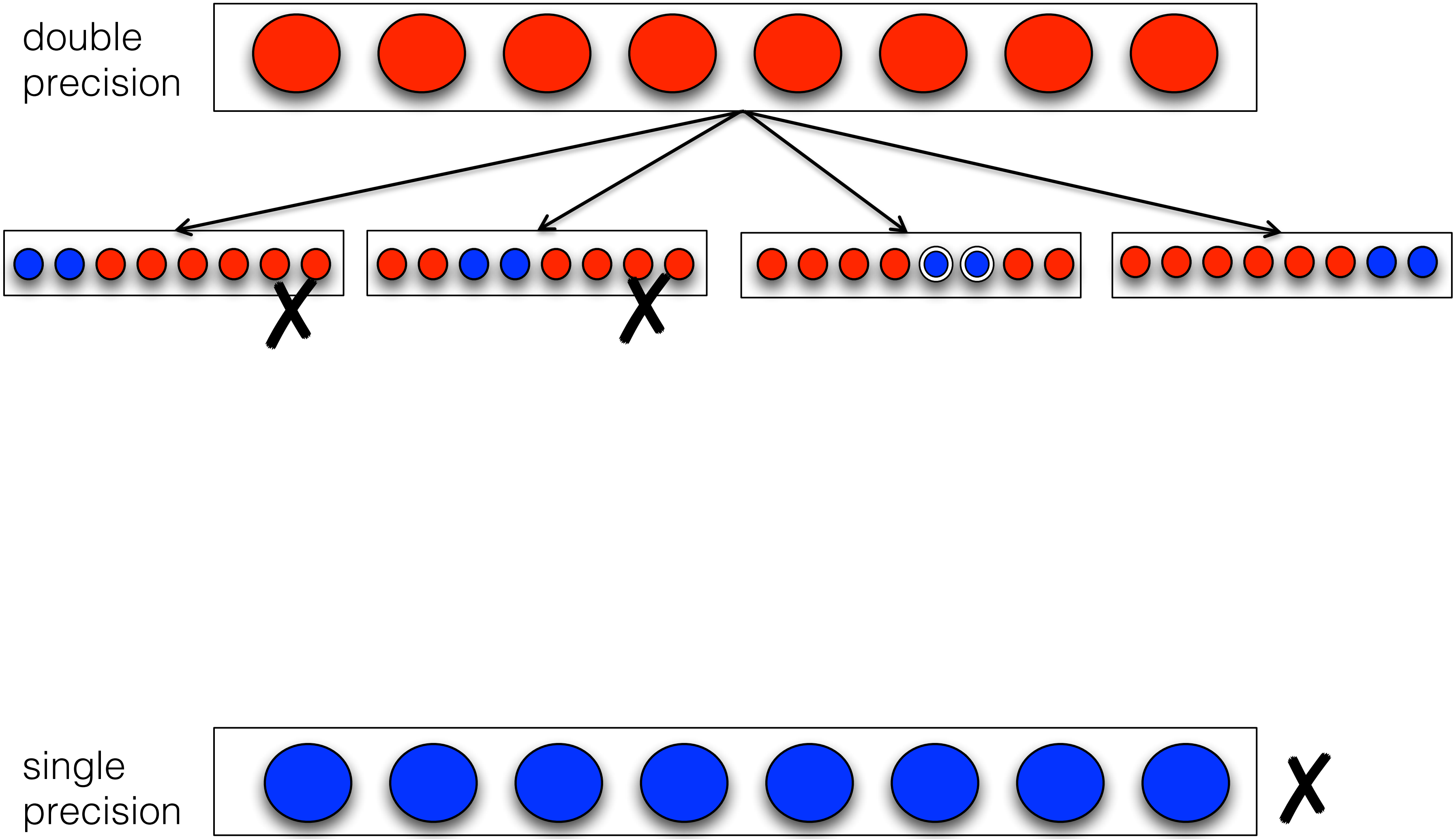
Searching for Type Configuration



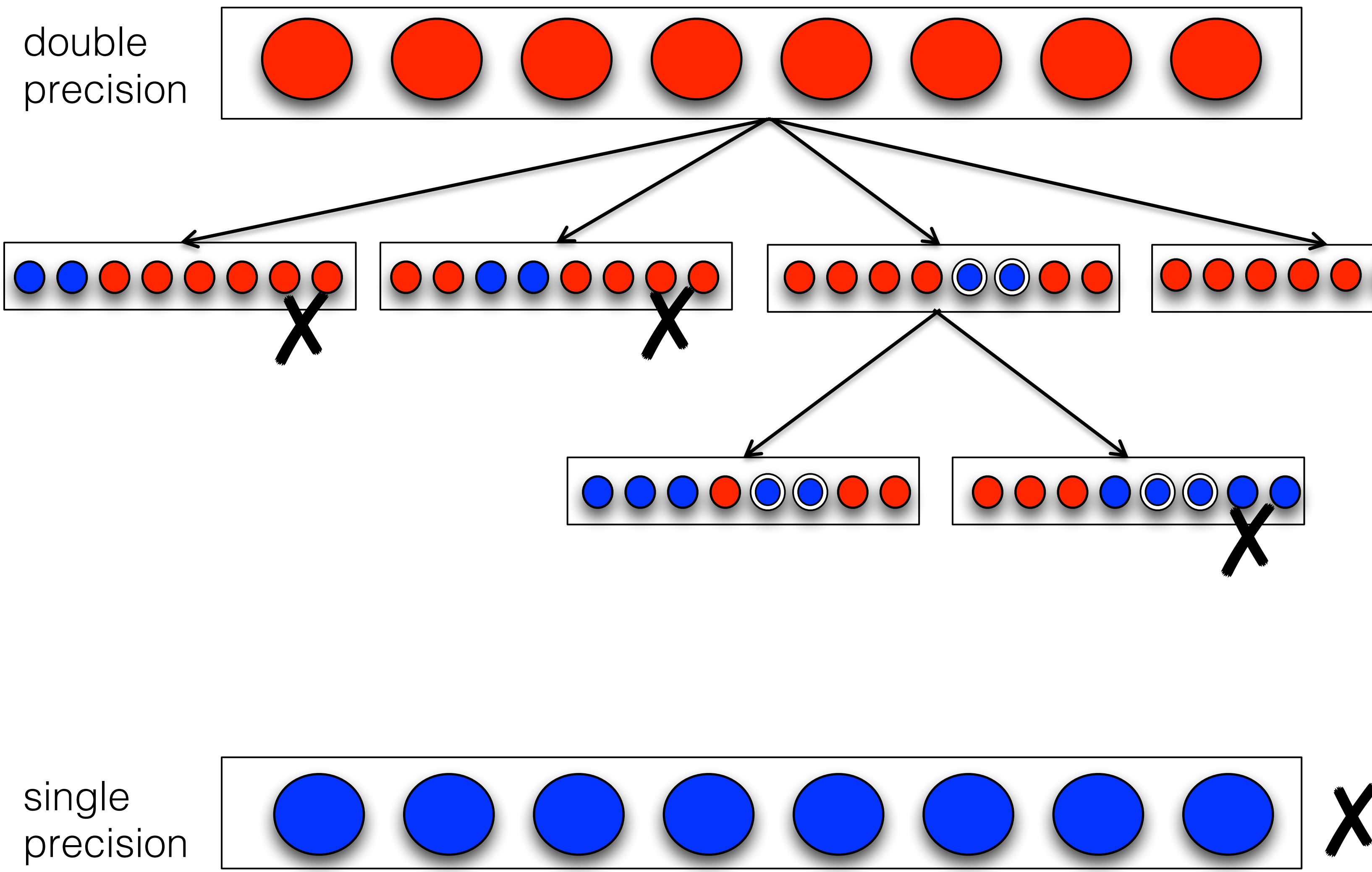
Searching for Type Configuration



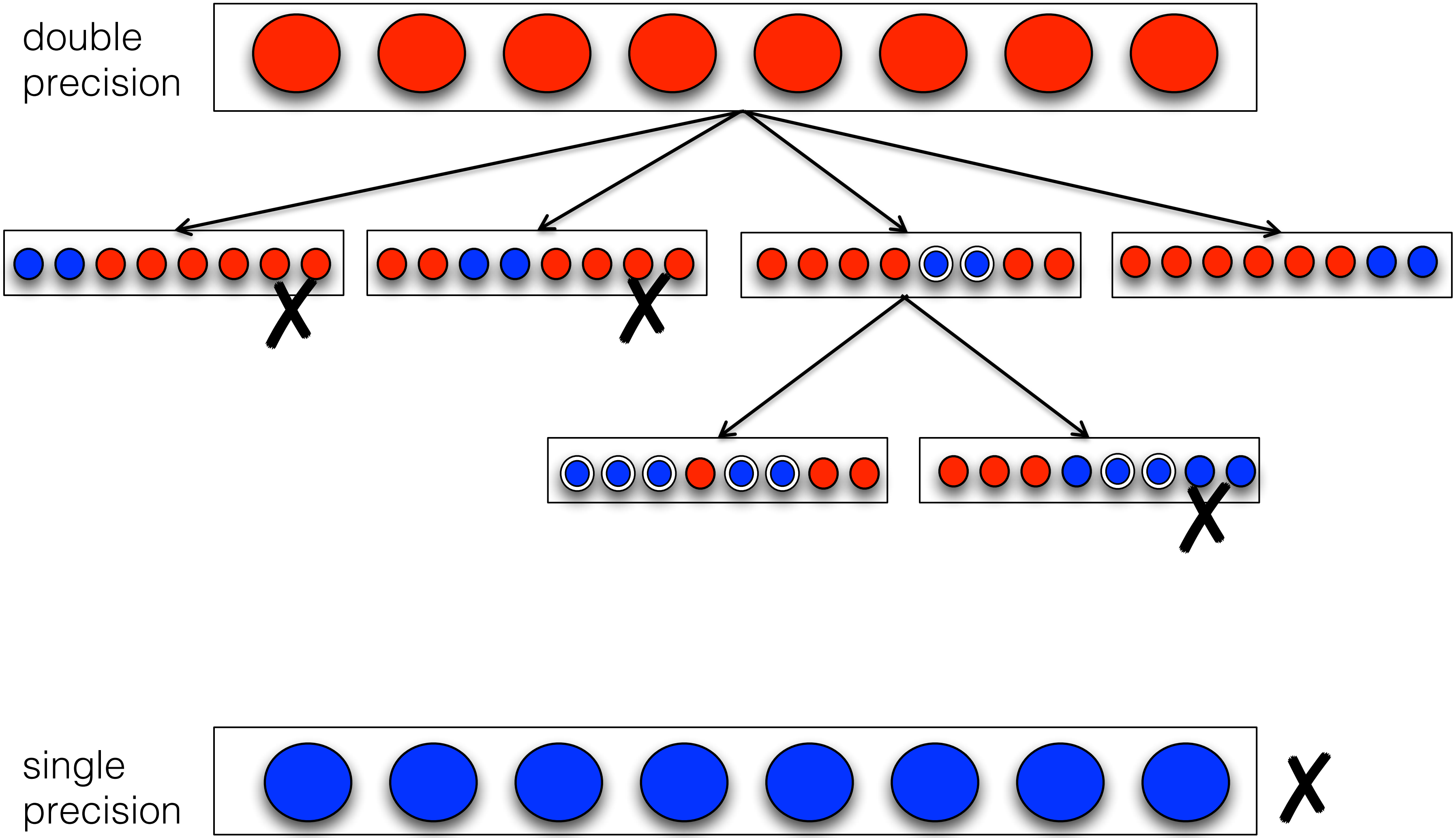
Searching for Type Configuration



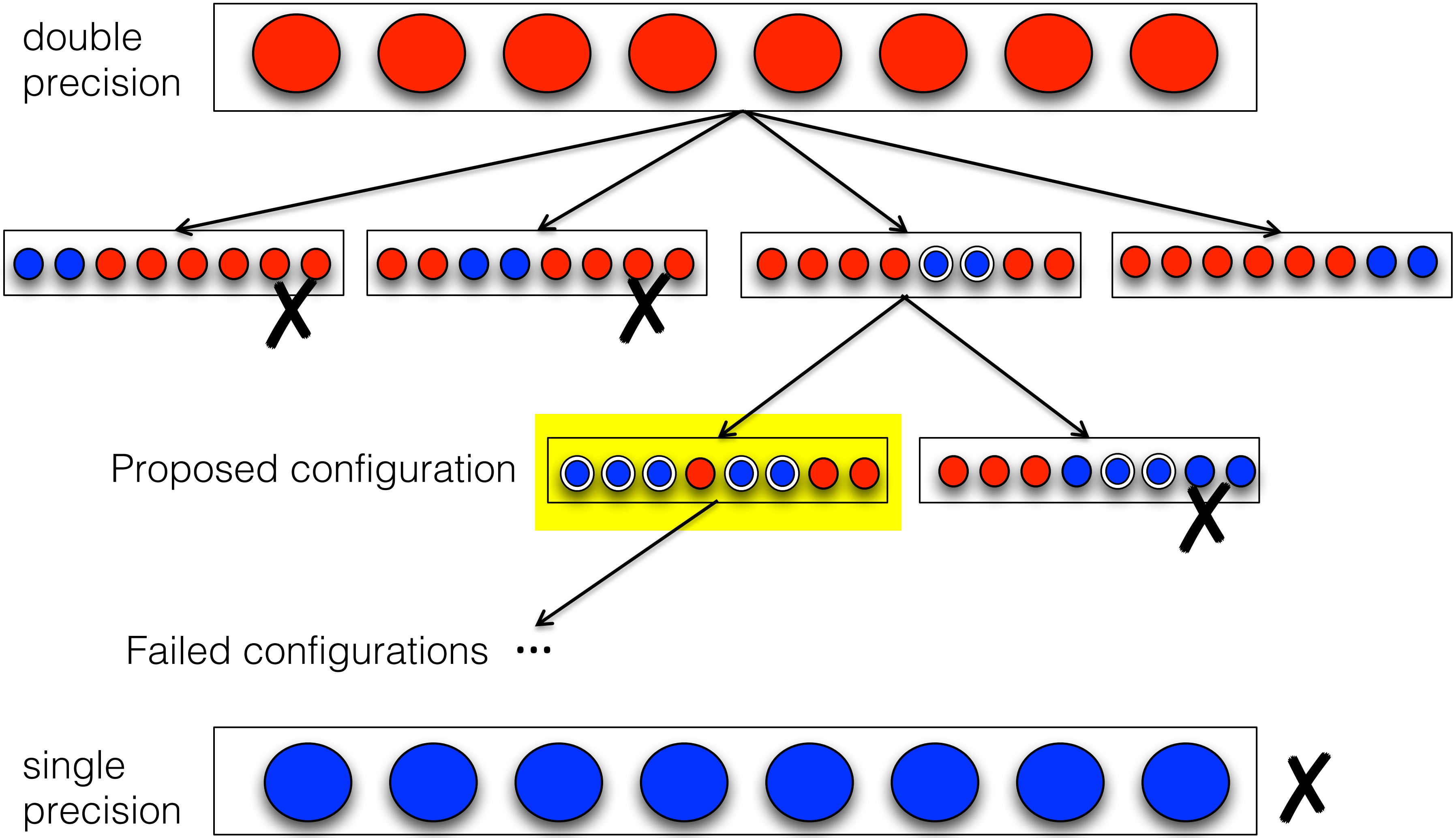
Searching for Type Configuration



Searching for Type Configuration



Searching for Type Configuration



Applying Type Configuration

- Automatically generate program variants
  - Reflect type configurations produced by the algorithm
- Intermediate representation
  - LLVM IR
- Transformation rules for each LLVM instruction
  - alloca, load, store, fadd, fsub, fpext, fptrunc, etc.
  - Changes equivalent to modifying the program at the source level
  - Clang plugin to provide modified source code
- Able to run resulting modified program
  - Evaluate type configuration: accuracy & performance

Where to Find Precimonious

- Precimonious is open source
  - Most recent version can be found at https://github.com/ucd-plse/precimonious
- Dockerfile and examples
  - Tutorial on Floating-Point Analysis Tools at SC'19 and PEARC'19 http://fpanalysistools.org
  - Dockerfile and examples can be found at https://github.com/ucd-plse/tutorial-precision-tuning

How to Use Precimonious

- Initial requirements
  - Does your program compile with clang?
  - Where does your program store the result?
  - How much error are you willing to tolerate?
    - Examples: 10^{-4},10^{-6}, 10^{-8}, and 10^{-10}
  - Do you have representative inputs to use during tuning?
- Optional information
  - Are there specific functions/variables to focus on, or to ignore during tuning?
- What you get
  - Listing of variables (and function) and their proposed types
  - Useful start point to identify areas of interest

Limitations and Recommendations

- Type configurations rely on program inputs tested
  - No guarantees if worse conditioned input
  - Use representative inputs whenever possible
  - Consider input generation tools, e.g., S3FP [1], FPGen [2], etc.
- Analysis scalability
  - Scalability limitations when tuning long-running applications
  - Need to reduce search space, and reduce number of runs
  - Consider starting with a specific area of the program
  - Consider synthesizing smaller workloads
- Analysis effectiveness
  - Black-box approach does not exploit relationship among variables

[1] W. Chiang, G. Gopalakrishnan, Z. Rakamaric and A. Solovyev. "Efficient Search for Inputs Causing High Floating-point Errors", PPoPP 2014.
[2] H. Guo and C. Rubio-González. "Efficient Generation of Error-Inducing Floating-Point Inputs via Symbolic Execution", ICSE 2020.

Dynamic Tools for Precision Tuning

Precimonious

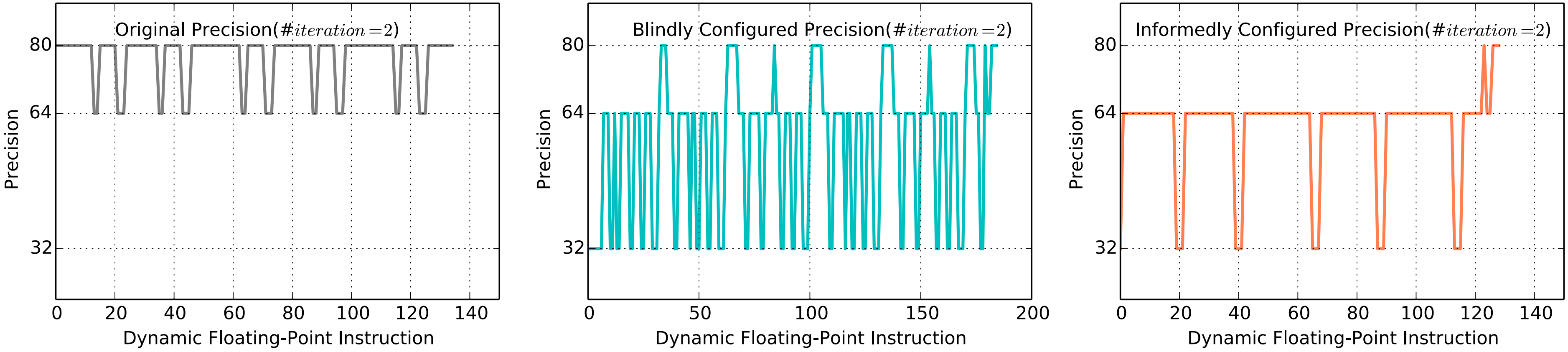
- Dynamic Analysis for Precision Tuning
  - Black-box approach to systematically search over variable types and functions

HiFPTuner

- Hierarchical Precision Tuner
  - Leverages relationship among variables to reduce search space and number of runs

Impact of Precision Shifting

- Precimonious follows a black-box approach
  - Related variables assigned types independently
  - Large number of variables → Slow search
  - More type casts → Less speedup



Original

Local minimum

Uses lower precision

Speedup: 78.7%

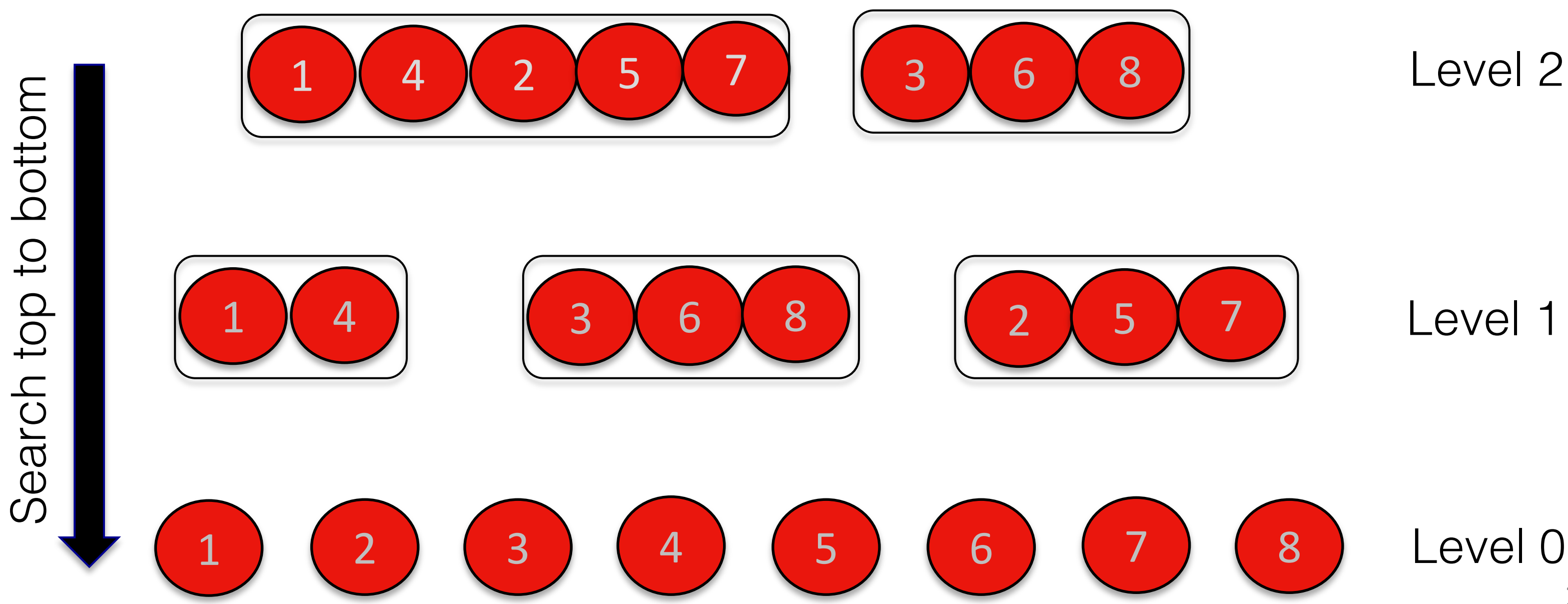
Global minimum

Shifts precision less often

Speedup: 90%

Exploiting Community Structure

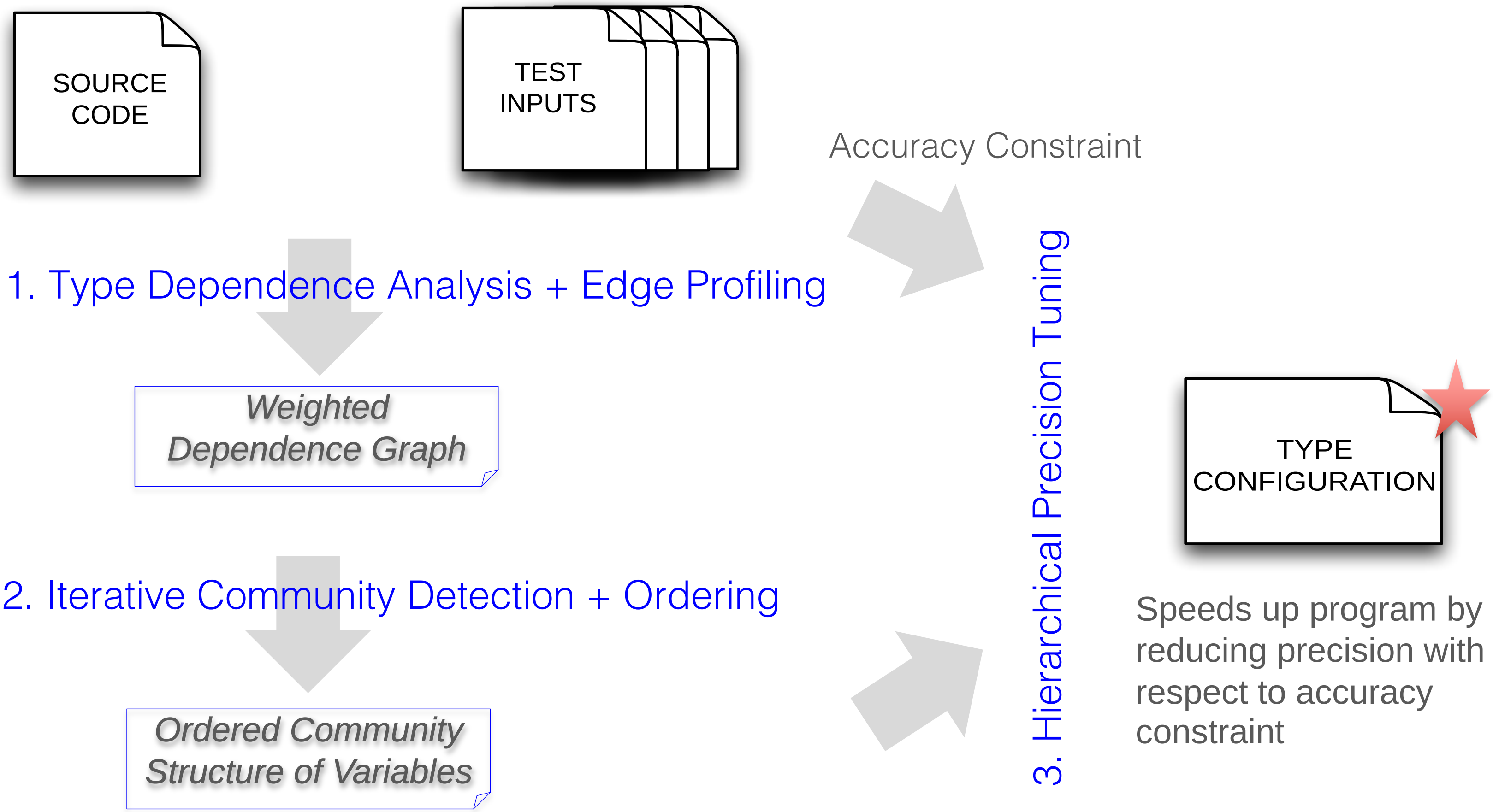
- Can we leverage the program to perform a more informed precision tuning?
- White box nature
  - Related variables pre-grouped into hierarchy → Same type
  - Fewer groups in search space → Faster search
  - Fewer type casts → Larger speedups



HiFPTuner Approach

Hierarchical Floating-Point Precision Tuning

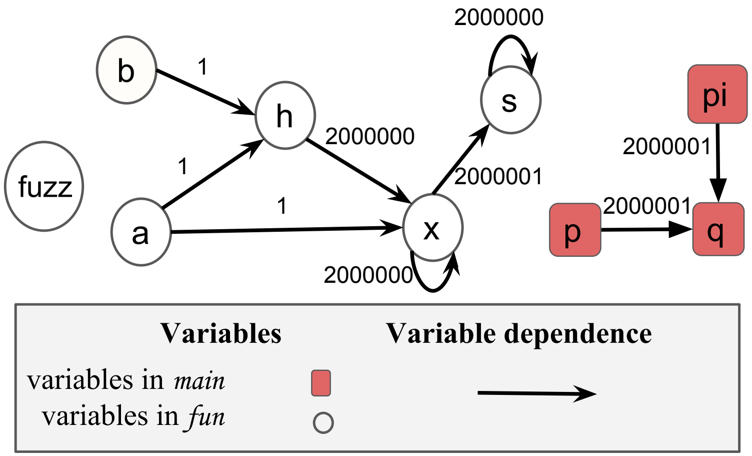
https://github.com/ucd-plse/HiFPTuner



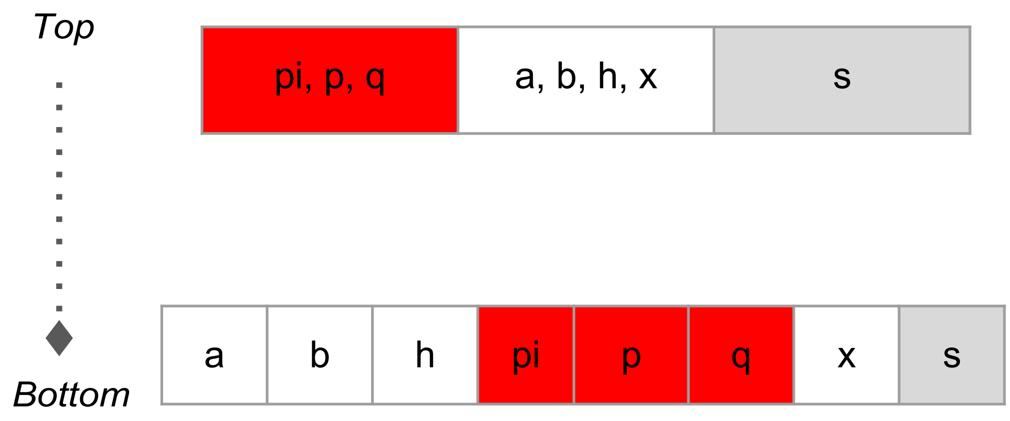
H. Guo and C. Rubio-González. “Exploiting Community Structure for Floating-Point Precision Tuning”, ISSTA 2018.
M. Girvan and M.E. Newman. “Community Structure in Social and Biological Networks”, NAS 2002.
F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi. “Defining and Identifying Communities in Networks”, NAS 2004.

Simpsons Example

Weighted dependence graph



Ordered community structure



Found global minimum configuration that leads to 90% speedup!

HiFPTuner explores 24 configurations, almost 5x fewer configurations

Better Scalability & Speedup

- Items at top level of hierarchy reduced by 53% on average in comparison to Precimonious
- Higher search efficiency over Precimonious for 75% of the programs in our study
  - Explored 45% fewer configurations
- HiFPTuner finds better configurations for half of the programs, with up to 90% speedup

Where to Find HiFPTuner

- HiFPTuner is open source
  - https://github.com/ucd-plse/HiFPTuner
- Dockerfile and examples
  - Tutorial on Floating-Point Analysis Tools at SC'19 and PEARC'19 http://fpanalysistools.org
  - Dockerfile and examples can be found at https://github.com/ucd-plse/tutorial-precision-tuning
- Same requirements as Precimonious

Comparison of Precision Tuners

	PROS	CONS
Precimonious	+ Considers both accuracy and performance + Works for medium size non-trivial programs + Easily configurable	- Requires a run for each type configurations - Ordering of variables may give different results
HiFPTuner	+ White-box *hierarchical* approach, groups variables based on their usage + Over twice as fast as Precimonious + Finds configurations that lead to higher speedups	- Requires program profiling - Still requires a run for each type configuration
Blame Analysis [1]	+ Performs shadow execution, requires a single run of the program + Identifies variables that can be single precision + Combined with Precimonious leads to 9x faster analysis	- Focuses on accuracy, not performance - 50x overhead by shadow execution engine - Still black box approach

[1] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D.H. Bailey and D. Hough. "Floating-Point Precision Tuning Using Blame Analysis", ICSE 2016.

Current Challenges for HPC Applications

1. Type configurations rely on program inputs tested
   - How problematic is this for HPC applications?
   - Can we leverage application-dependent correctness metrics?
2. Analysis scalability
   - How can we further reduce the search space?
   - How can we reduce the number of program runs?
3. Analysis effectiveness
   - How far are we from the best configuration(s)?
   - Are there other program transformations to explore?
   - Can we incorporate domain knowledge to guide search?
4. Benchmarks
   - Difficult to find programs to test precision tuners at scale
   - Need for collaboration between application and tool developers

Some Useful Resources

- Other recent precision tuners

I. Laguna, P.C. Wood, R. Singh and S. Bagchi. “GPUMixer: Performance-Driven Floating-Point Tuning for GPU Scientific Applications”, ISC 2019.

M. Lam, T. Vanderbruggen, H. Menon and M. Schordan. “Tool Integration for Source-Level Mixed Precision”. CORRECTNESS@SC 2019.

S. Cherubin, D. Cattaneo, M. Chiari and G. Agosta. “Dynamic Precision Autotuning with TAFFO”. ACM Trans. Archit. Code Optim. 2019.

P.V. Kotipalli, R. Singh, P. Wood, I. Laguna and S. Bagchi. “AMPT-GA: Automatic Mixed Precision Floating Point Tuning for GPU Applications”. ICS 2019.

H. Menon, M. Lam, D. Osei-Kuffuor, M. Schordan, S. Lloyd, K. Mohror and J. Hittinger. “ADAPT: Algorithmic Differentiation Applied to Floating-Point Precision Tuning”, SC 2018.

E. Darulova, E. Horn and S. Sharma. “Sound Mixed-Precision Optimization with Rewriting”. ICCPS 2018.

W. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan and Z. Rakamaric. “Rigorous Floating-Point Mixed-Precision Tuning”. POPL 2017.

- Check out recent survey on reduced precision

S. Cherubin and G. Agosta. “Tools for Reduced Precision Computation: A Survey. ACM Computing Surveys 2020.

- An exhaustive list of tools: https://fpbench.org/community.html

SC Workshop on Software Correctness

Co-Organized with Ignacio Laguna from Lawrence Livermore National Lab
November 11th, 2020 (half day, 2:30pm to 6:30pm EDT)

Summary

- Precision tuning can have an important impact on the performance of HPC applications
- Many techniques for precision tuning
  – Different approaches: dynamic vs. static
- We discussed two of our tools for precision tuning
  – Precimonious and HiFPTuner
- A lot of progress, but there are still challenges and opportunities to apply precision tuning at scale
- **Application and tool developers** must work together to improve scalability and effectiveness of precision tuning

Collaborators

UC Berkeley

Cuong Nguyen

Diep Nguyen

Ben Mehne

James Demmel

William Kahan

Koushik Sen

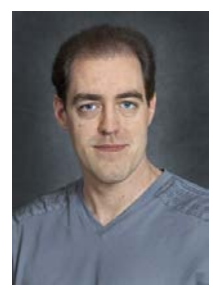
UC Davis

Hui Guo

David Bailey

LBNL

Costin Iancu



Wim Lavrijsen

Oracle

David Hough

Acknowledgements/Sponsors

NSF

U.S. DEPARTMENT OF ENERGY
Office of Science

UC DAVIS
UNIVERSITY OF CALIFORNIA

Hellman Fellows Fund

better scientific software