CORVETTE: Program Correctness, Verification, and Testing for Exascale

PI: **Koushik Sen**, UC Berkeley
coPI: James W. Demmel, UC Berkeley
coPI: Costin Iancu, LBNL

Post-doc and students
Cindy Rubio Gonzalez, Chang-Seo Park, Ahn Cuong Nguyen

Correctness Tools for HPC

- **Dire lack of theoretical and engineering know-how**
- **Overall as a community, we are not very sophisticated when using testing and correctness tools**
  - How many of you have a "Test Engineer" or a "QA Engineer" position posted?
  - How many of you know of Purify, Coverity, or SilkTest?
- **There are very good reasons for the status quo**
  - Sociological – we like hero programmers
  - Practical – hero programmers can find bugs
    - Serial code between two MPI_... calls
- **Things are changing**

Motivation

- High performance scientific computing
  - Exascale: $O(10^6)$ nodes, $O(10^3)$ cores per node
  - Side-effects through global address spaces
  - Unstructured parallelism and dynamic tasking
  - Non-blocking, highly asynchronous behavior
- Correctness challenges
  - Hard to diagnose correctness and performance bugs
    - Data races, atomicity violations, deadlocks …
  - Scientific applications use floating-points: non-determinism leads to non-reproducible results
  - Numerical exceptions can cause rare but critical bugs
    - hard for non-experts to detect and fix
    - existing compilers and analyses are not good at floating-point

Goals

- Correctness tools for parallel programs written using hybrid parallelism: OpenMP+MPI, UPC+MPI, OpenMP+UPC
- Testing and Verification
  – Identify sources of non-determinism in executions
  – Concurrency bugs include data races, atomicity violations, non-reproducible floating point results
  – Develop precise and scalable tools with < 2x run-time overhead at large scale
- Debugging
  – Use minimal amount of concurrency to reproduce bug
  – Support two-level debugging of high-level abstractions
  – Detect causes of floating-point anomalies and determine the minimum precision needed to fix them

I. Testing and Debugging Large-Scale Parallel Programs

def/use data race

In Knapsack (dynamic programming)

```
int build_table (int nitems, int cap,  shared int *T, shared int *w, shared int *v) {
    int wj, vj;
    wj = w[0];
    vj = v[0];
    upc_forall(int i = 0; i < wj; i++; &T[i])
        T[i] = 0;
    upc_forall(int i = wj; i <= cap; i++; &T[i])
        T[i] = vj;
    upc_barrier;
 }

int main( int argc, char** argv ) {
upc_forall(i = 0; i < nitems; i++; i) {
        weight[i] = 1 + (lrand48()%max_weight);
        value[i]  = 1 + (lrand48()%max_value);
    }
    best_value = build_table(nitems, capacity, total, weight, value );
}
```

Scalable Testing of Parallel Programs

- Hybrid Parallel Programming is hard
  - Bugs happen non-deterministically
  - Data races, deadlocks, atomicity violations, etc.
- Goals: build a tool to test and debug concurrent and parallel programs
  - Efficient: reduce overhead from 10x-100x to 2x
  - **Precise**
  - Reproducible
  - **Scalable**
- **Active random testing**

Active Testing

- Phase 1: Static or dynamic analysis to find potential concurrency bug patterns
  - such as data races, deadlocks, atomicity violations
- Phase 2: "Direct" testing (or model checking) based on the bug patterns obtained from phase 1
  - Confirm bugs

Active Testing:
Predict and Confirm Potential Bugs

- Phase I: Predict potential bug patterns:
  - Data races: Eraser or lockset based [PLDI'08]
  - Atomicity violations: cycle in transactions and happens-before relation [FSE'08]
  - Deadlocks: cycle in resource acquisition graph [PLDI'09]
  - Publicly available tool for Java/Pthreads/UPC [CAV'09]
  - Memory model bugs: cycle in happens-before graph [ISSTA'11]
  - For UPC programs running on thousands of cores [SC'11]
- Phase II: Direct testing using those patterns to confirm real bugs

Challenges for Exascale

- Java and pthreads programs
  - Synchronization with locks and condition variables
  - Single node
- Exascale has different programming models
  - Large scale
  - Bulk communication
  - Collective operations with data movement
  - Memory consistency
  - Distributed shared memory
- Cannot use centralized dynamic analyses
- Cannot instrument and track every statement

Summary of Challenges

- **Challenge 1: Scalability with LOCs**
- **Challenge 2: Scalability with input size**
- **Challenge 3: Scalability with cores**

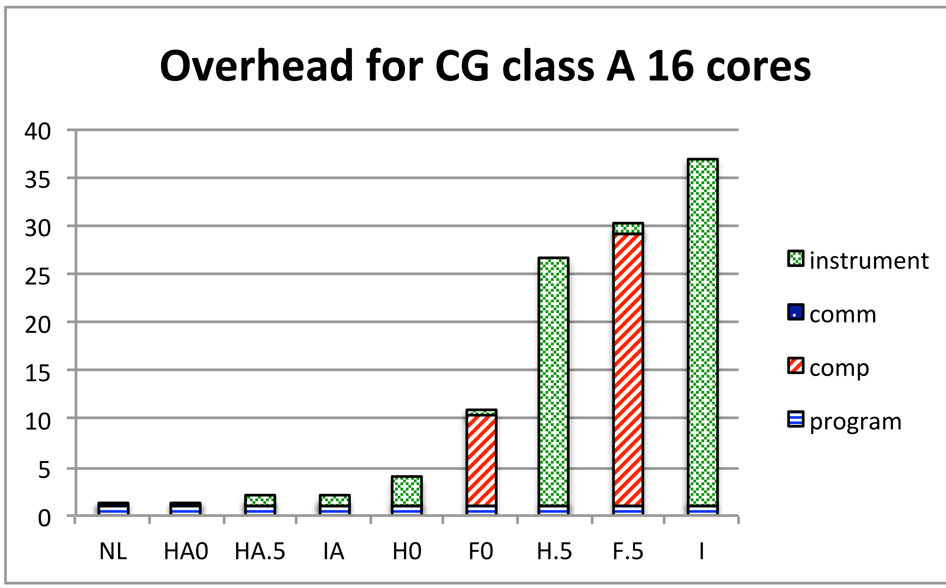
Finding Data Races in UPC

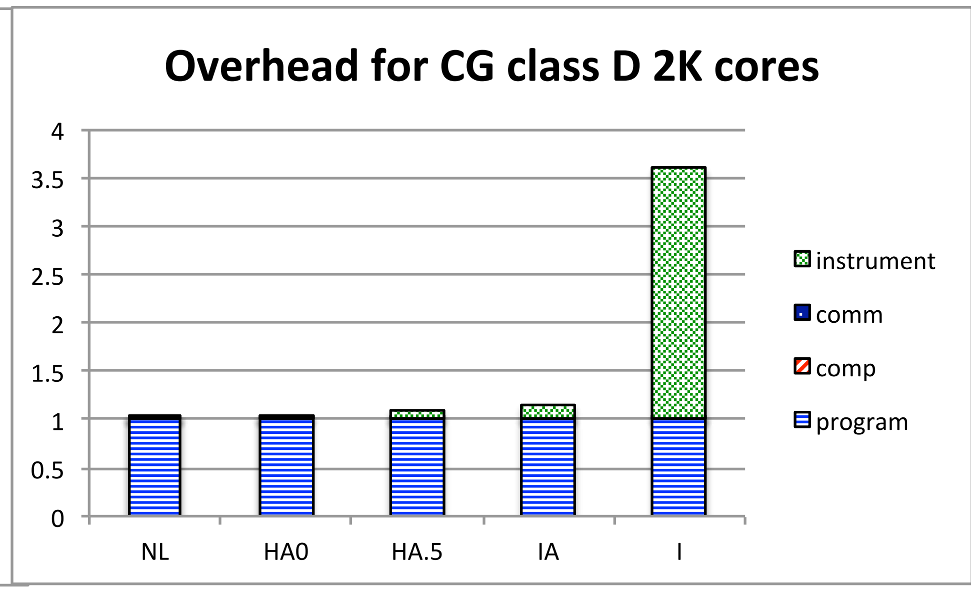
- **THRead Interposition Library and Lightweight Extensions (THRILLE): Active Testing framework for UPC**
- **Download available at http://upc.lbl.gov/thrille.shtml**

- **Implementation of race detector and tester for programs written in PGAS style**
  - Instrument load/stores to local heap
  - Instrument load/stores to global heap
  - Instrument bulk transfers (`upc_memcpy`)
  - Track fine-grained synchronization (locks) and bulk synchronization (single- and split-phase barriers)

Challenge: Scalability with Input

- **Sources of overhead**
  - Tracking memory references **(Instrumentation)**
  - Reasoning on collected data **(Data Management)**



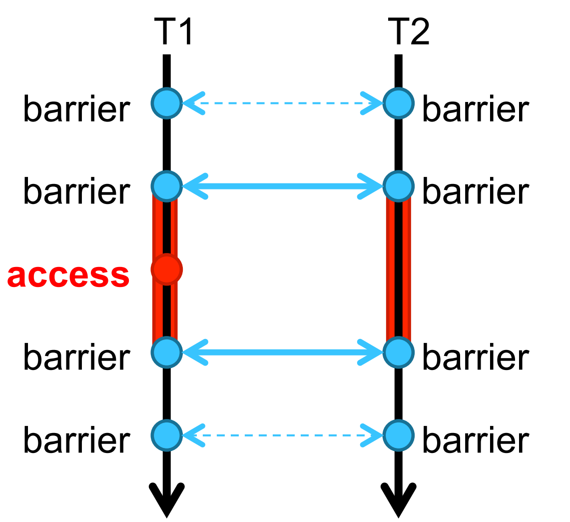


Solution: Scalability with Inputs

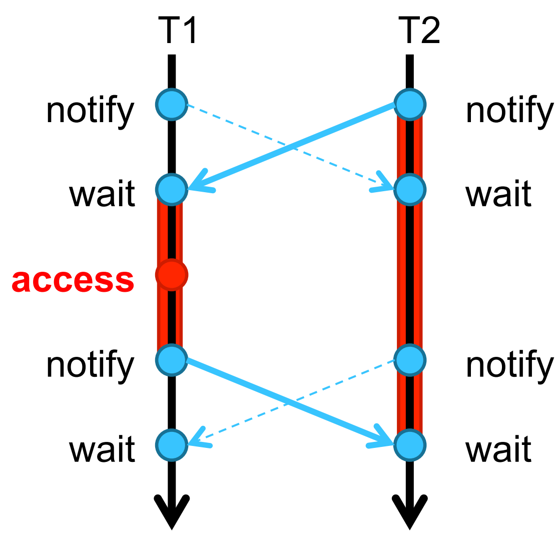
- **Reducing instrumentation overhead through sampling**
  - State-of-the-art function level sampling does NOT work
  - Instruction level sampling is slow
  - Novel hierarchical sampling approach provides best performance
  - Alias based pruning

Solution: Scalability with Cores

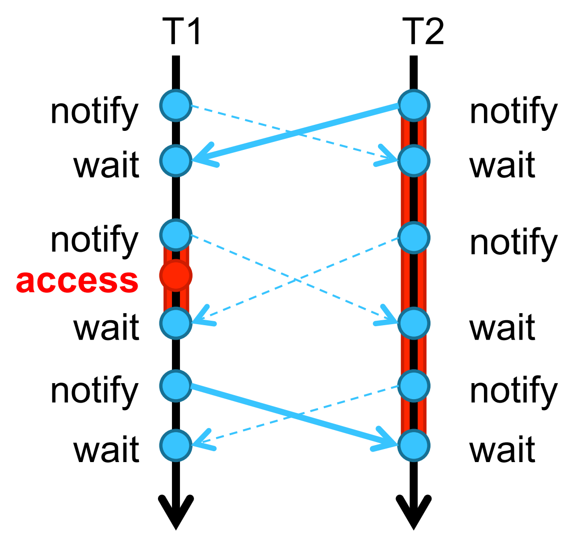
- **Per task memory access traces are collected and exchanged during execution** (alltoallv)
  - Novel distributed algorithm using *barrier aware may-happen in parallel analysis*
  - Novel use of efficient data structures - *Interval skip lists*
  - Analysis is carefully overlapped with communication of memory traces



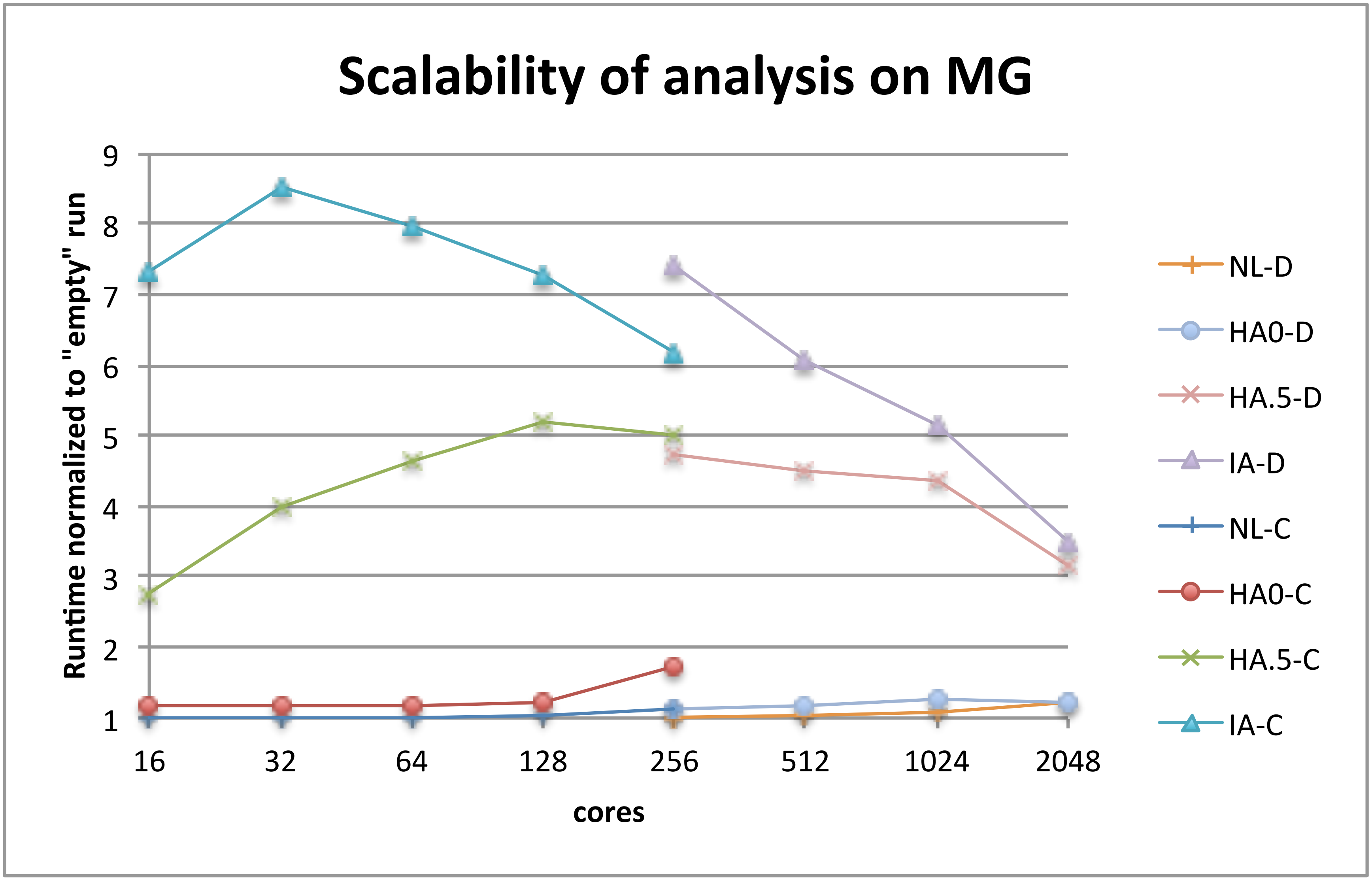
Shared access between barriers



Shared access after wait



Shared access after notify



Scalability of analysis on MG
Runtime normalized to "empty" run
9
8
7
6
5
4
3
2
1
16
32
64
128
256
512
1024
2048
cores
NL-D
HA0-D
HA.5-D
IA-D
NL-C
HA0-C
HA.5-C
IA-C

Results

Bench	LoC	Run time (s)	Races	Overhead (%)				
				NL	HA.5	IA	FA0	I
guppie	271	19.070	2(2)+0(0)	54.9	54.2	53.7	DNF	74.9
psearch	803	0.697	3(1)+2(2)	2.48	10.8	666	8.01	6490
BT 3.3	9698	189.48	7(0)+3(1)	0.574	1.16	77.6	DNF	-
CG 2.4	1654	39.573	0(0)+1(1)	1.09	27.6	57.6	DNF	2579
EP 2.4	678	54.453	0(0)+0(0)	-0.618	0.805	2.09	4.74	111
FT 2.4	2289	62.663	2(2)+0(0)	0.601	30.1	121	DNF	2744
IS 2.4	1426	5.130	0(0)+0(0)	0.376	119	159	DNF	1201
LU 3.3	6348	155.997	0(0)+24(2)	-0.425	-	75.7	DNF	-
MG 2.4	2229	18.687	2(2)+4(0)	0.336	176	632	DNF	2020
SP 3.3	5740	247.937	10(0)+3(1)	0.160	0.861	29.1	DNF	-

Races: A(B) + C(D), where A represents the number of races detected by the original UPC-Thrille tool (NL) with B of them confirmed, and C represents the additional number of races detected with our extensions (HA.5) with D of them confirmed through phase 2

KEY FOR VARIANTS
NL: no instrumentation on local accesses (SC'11) / H: hierarchical sampling / I: instruction-level sampling only / F: function-level sampling only
A: indicates the use of the persistent alias heuristic
 # (0 or .5): Back-off factor for function-level sampling (0 means only first invocation of functions sampled)

< 50% slowdown up to 2K cores with opt.

II. Debugging and Tuning Floating-point Programs

Example (D.H. Bailey)

- Calculate the arc length of the function g defined as

$$g(x) = x + \sum_{0 \le k \le 5} 2^{-k} \sin(2^k \cdot x), \quad \text{over } (0, \pi).$$

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

$$\sqrt{h^2 + (g(x_k + h) - g(h))^2},$$

with $h = \pi/n$ and $x_k = k \cdot h$. If $n = 1000000$, we have

$$\begin{aligned} \text{result} &= 5.795776322412856 \quad (\text{ all double-double}) \longrightarrow \text{slower} \\ &= 5.795776322413031 \quad (\text{all double}) \\ &= 5.795776322412856 \quad (\text{only the summand is in double-double}) \end{aligned}$$

Example (D.H. Bailey)

- Calculate the arc length of the function g defined as

$$g(x) = x + \sum_{0 \le k \le 5} 2^{-k} \sin(2^k \cdot x), \quad \text{over } (0, \pi).$$

How can we find a minimal set of code fragments whose precision must be high?

$$\sqrt{h^2 + (g(x_k + h) - g(h))^2},$$

with $h = \pi / n$ and $x_k = k \cdot h$. If $n = 1000000$, we have

$$\begin{aligned} \text{result} &= 5.795776322412856 \quad (\text{ all double-double}) \longrightarrow \text{slower} \\ &= 5.795776322413031 \quad (\text{all double}) \\ &= 5.795776322412856 \quad (\text{only the summand is in double-double}) \end{aligned}$$

Why do we care?

- Usage of floating point programs has been growing rapidly
  - HPC
  - Cloud, games, graphics, finance, speech, signal processing
- Most programmers are not expert in floating-point!
  - Why not use highest precision everywhere
- High precision wastes
  - Energy
  - Time
  - Storage

What we can do?

- We can reduce precision "safely"
  - reduce power, improve performance, get better answer
- Automated testing and debugging techniques
  - To recommend "precision reduction"
  - Formal proof of "safety" can be replaced by concolic testing
- Approach: automate previously hand-made debugging
  - Concolic testing
  - Delta debugging [Zeller et al.]

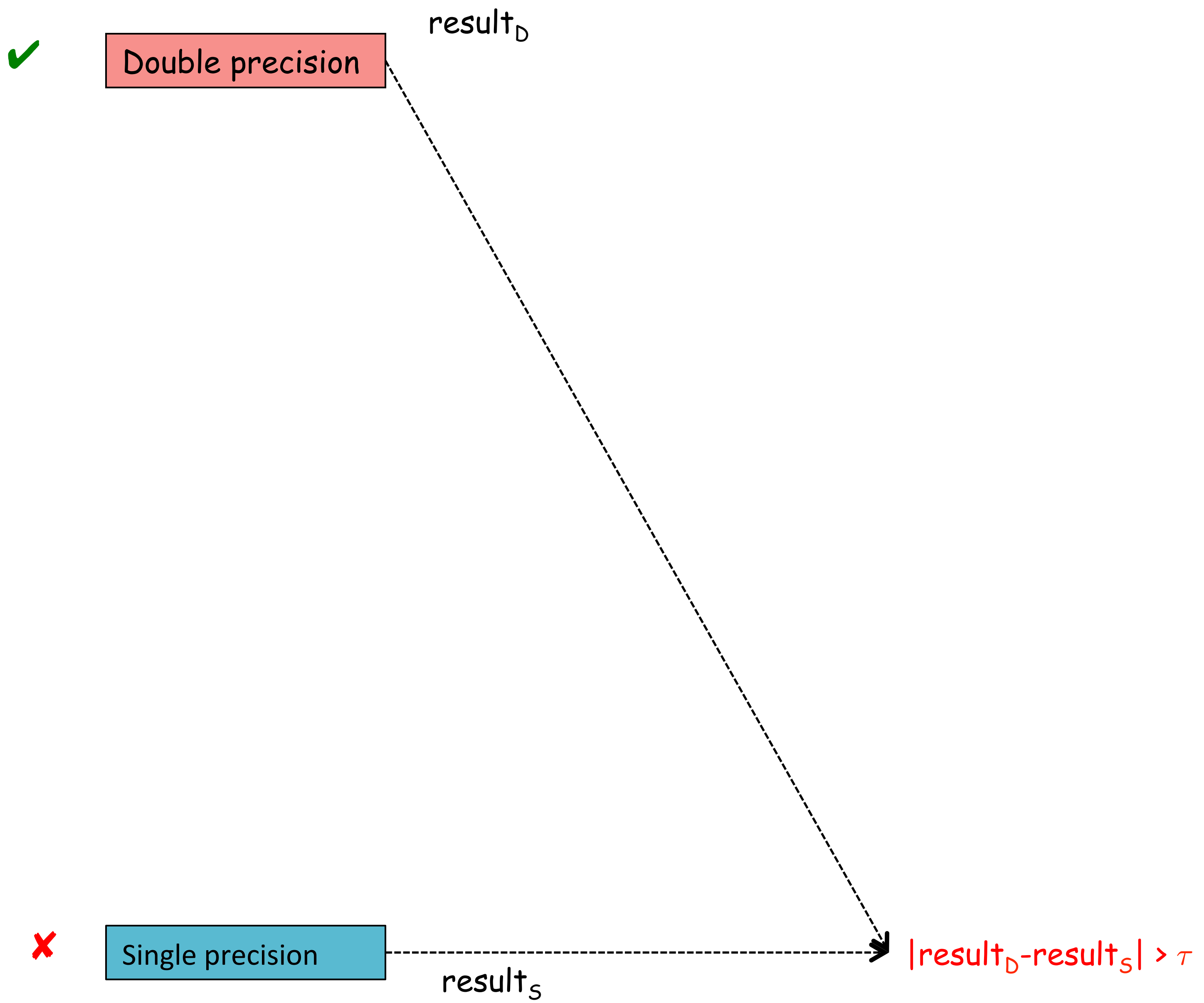
Non-expert developer usage scenario

- Developer writes code in highest precision
- Developer specifies accuracy requirements
  - In the absence of such requirements, consider inaccuracies that could lead to exceptions
  - Exceptions due to the use of low precision
- Our tool
  - Proposes “safe” precision reduction
  - Uses concolic testing to gain safety confidence
  - Expect to run on 10K LOC, but modular

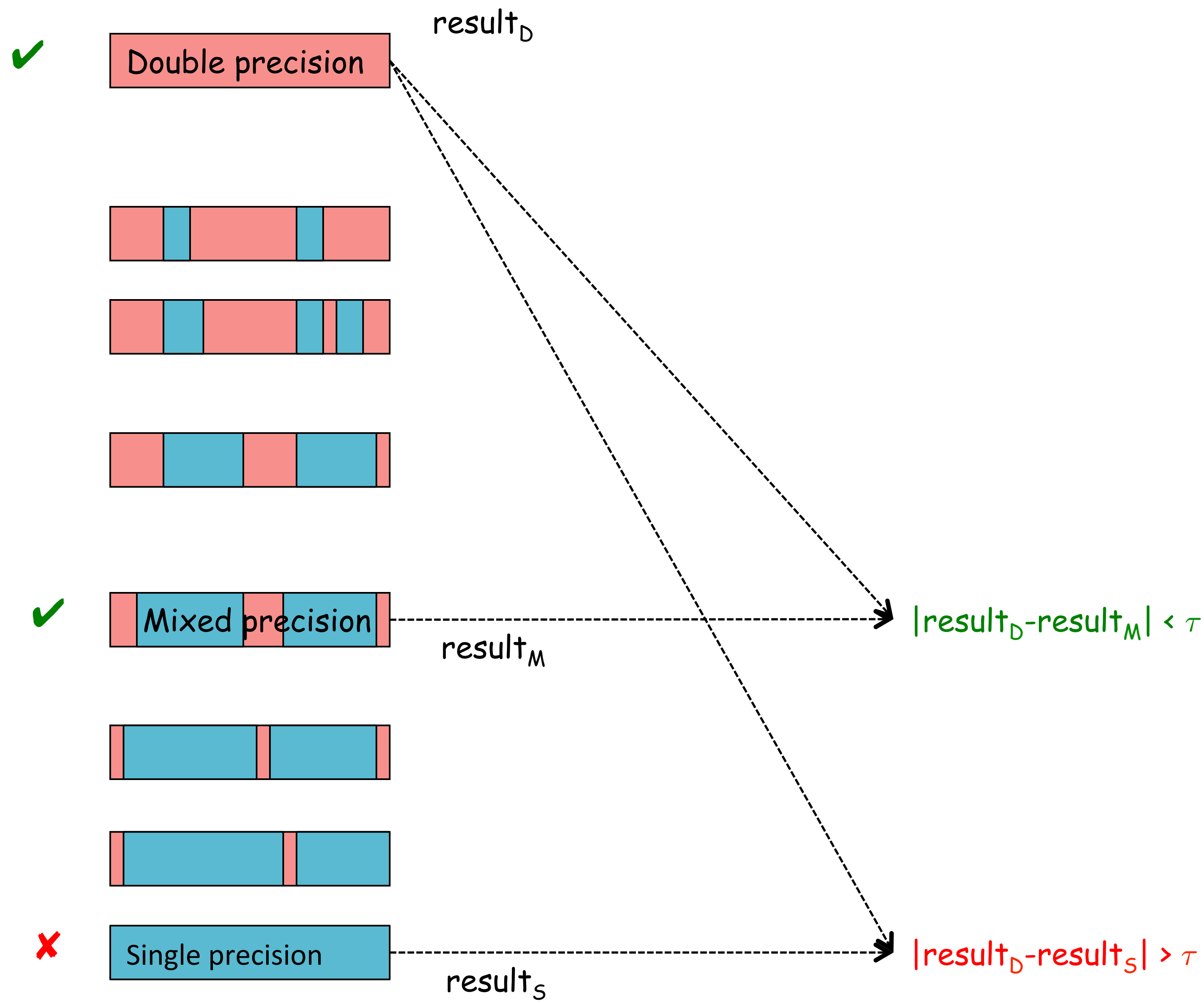
Delta Debugging to Propose Precision Reduction

✔ Double precision $result_D$

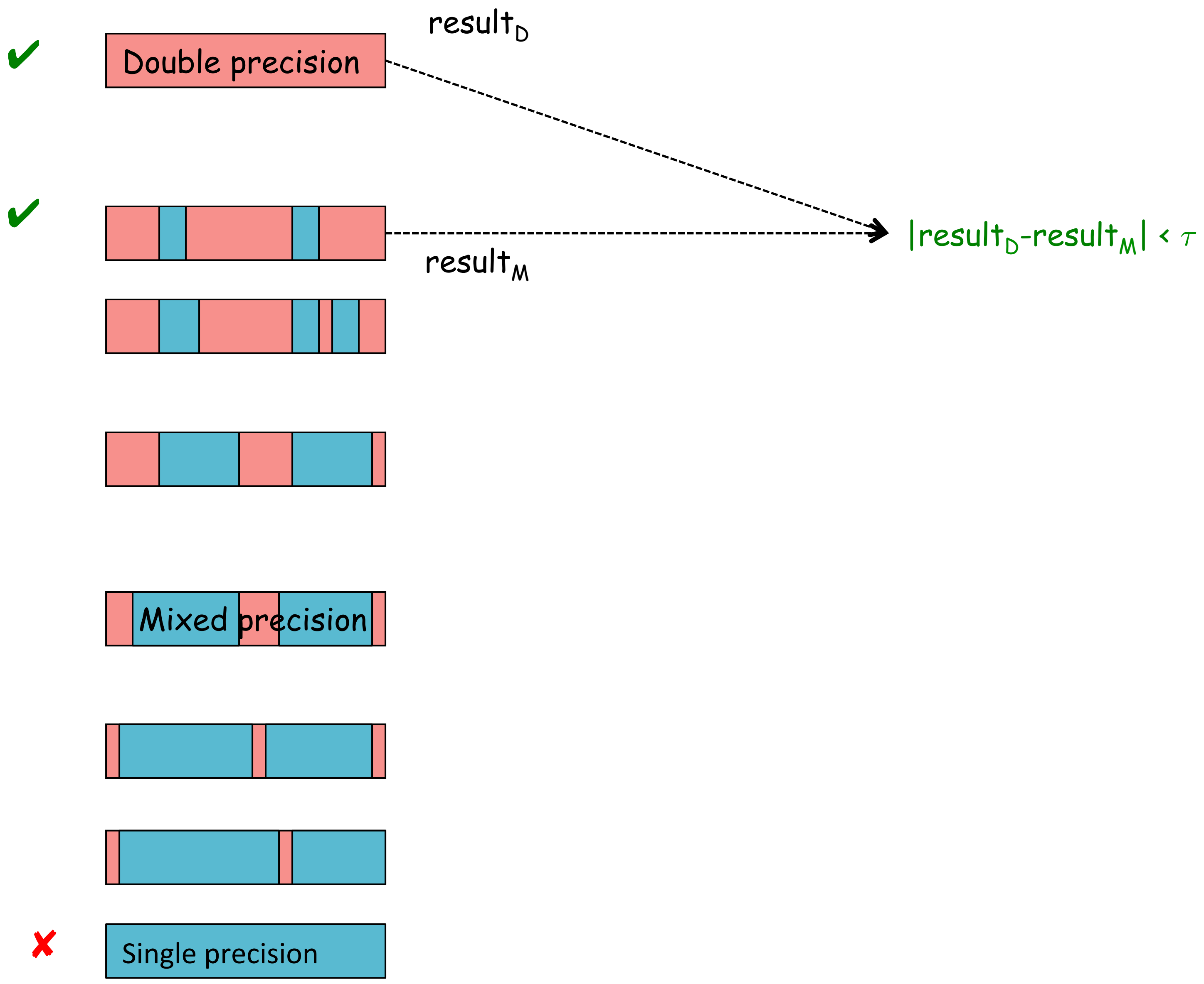
Delta Debugging to Propose Precision Reduction



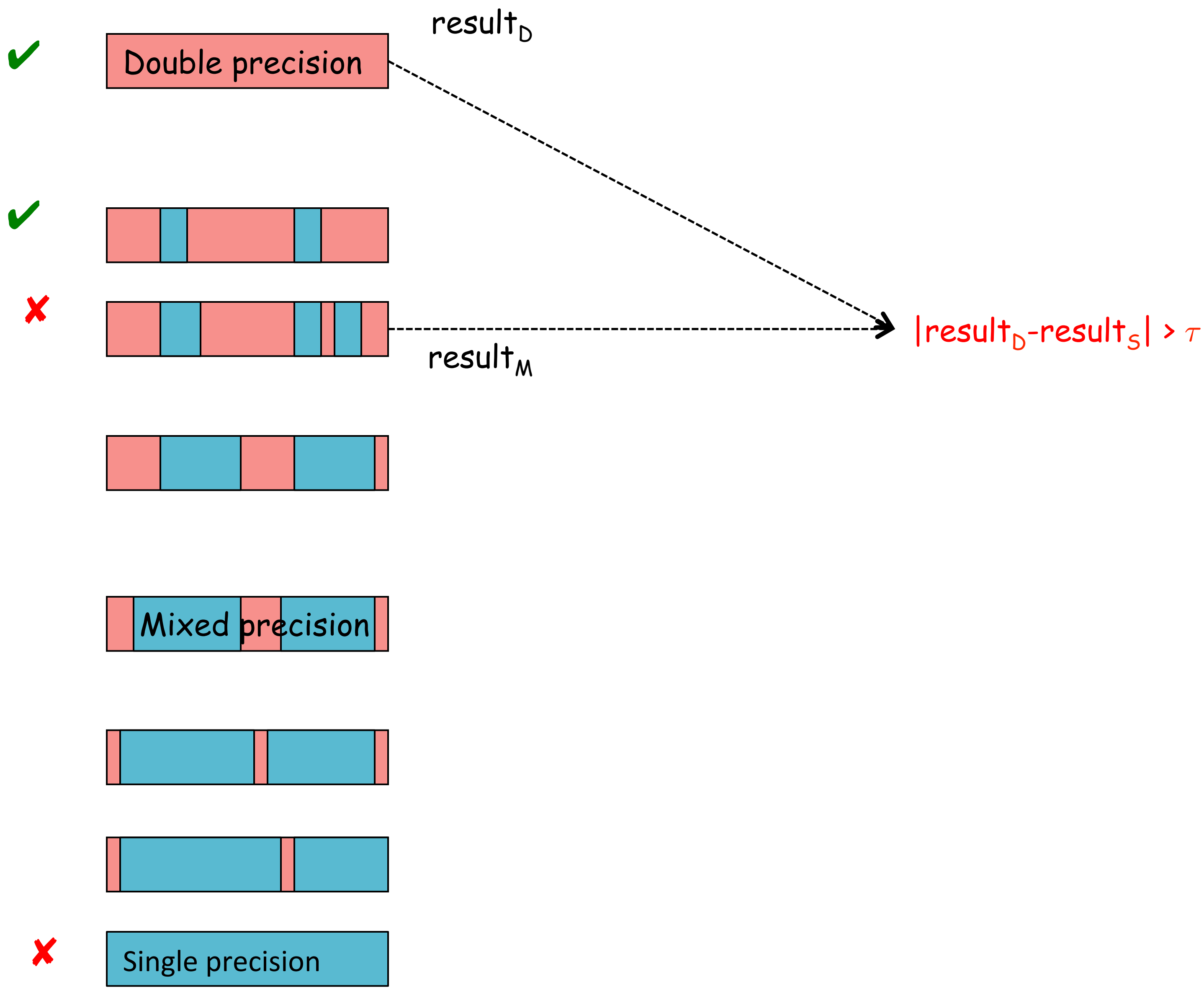
Delta Debugging to Propose Precision Reduction



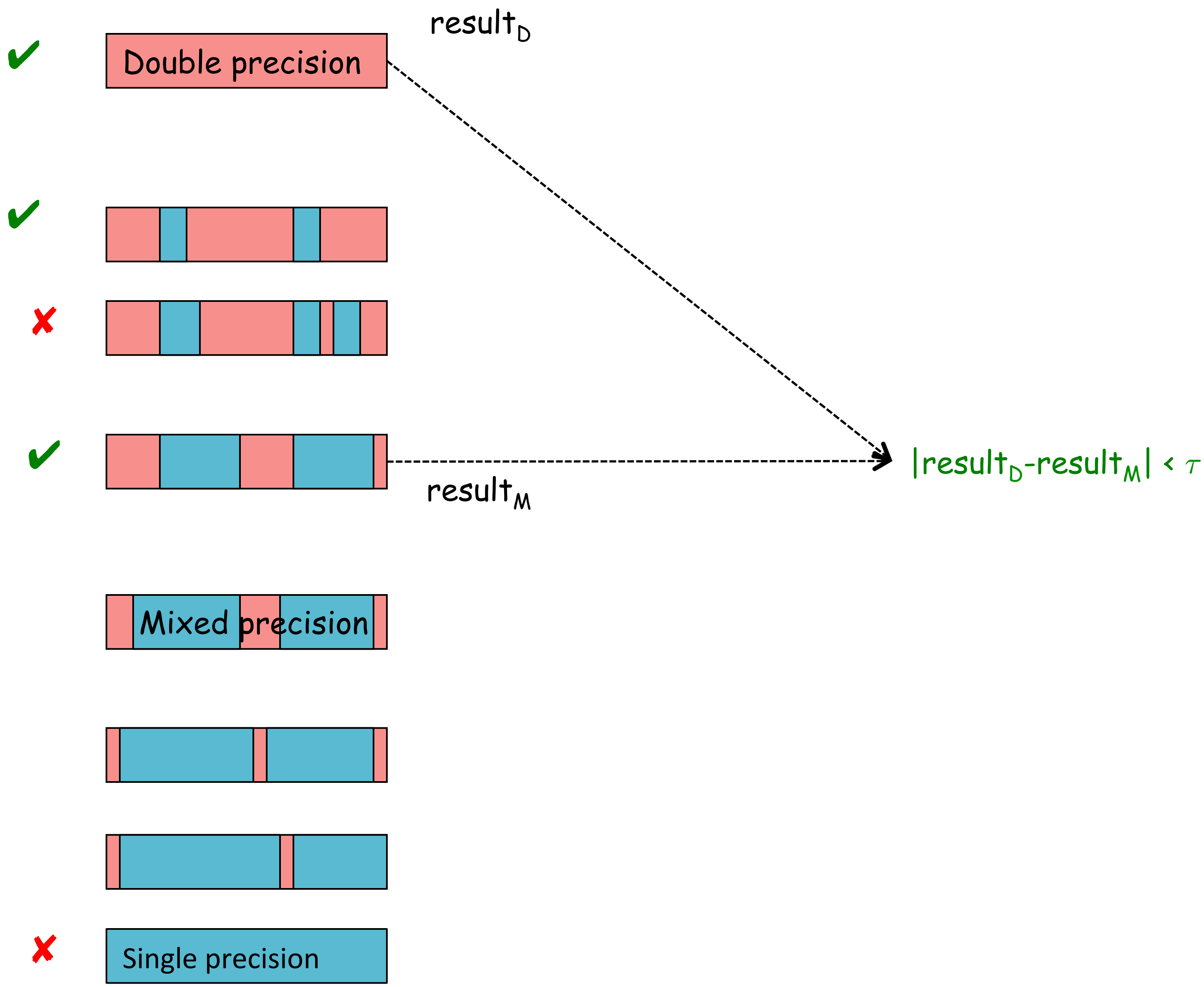
Delta Debugging to Propose Precision Reduction



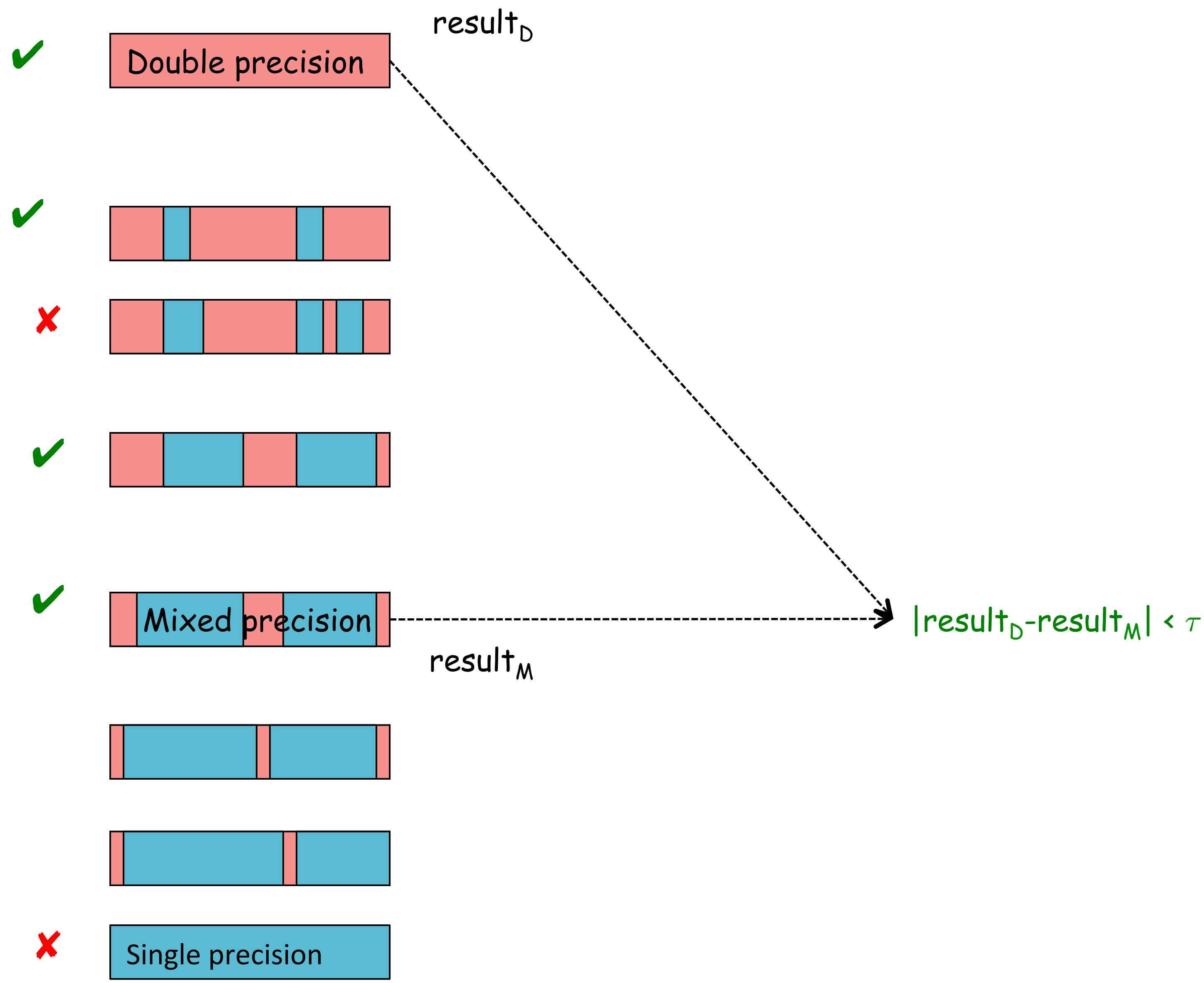
Delta Debugging to Propose Precision Reduction



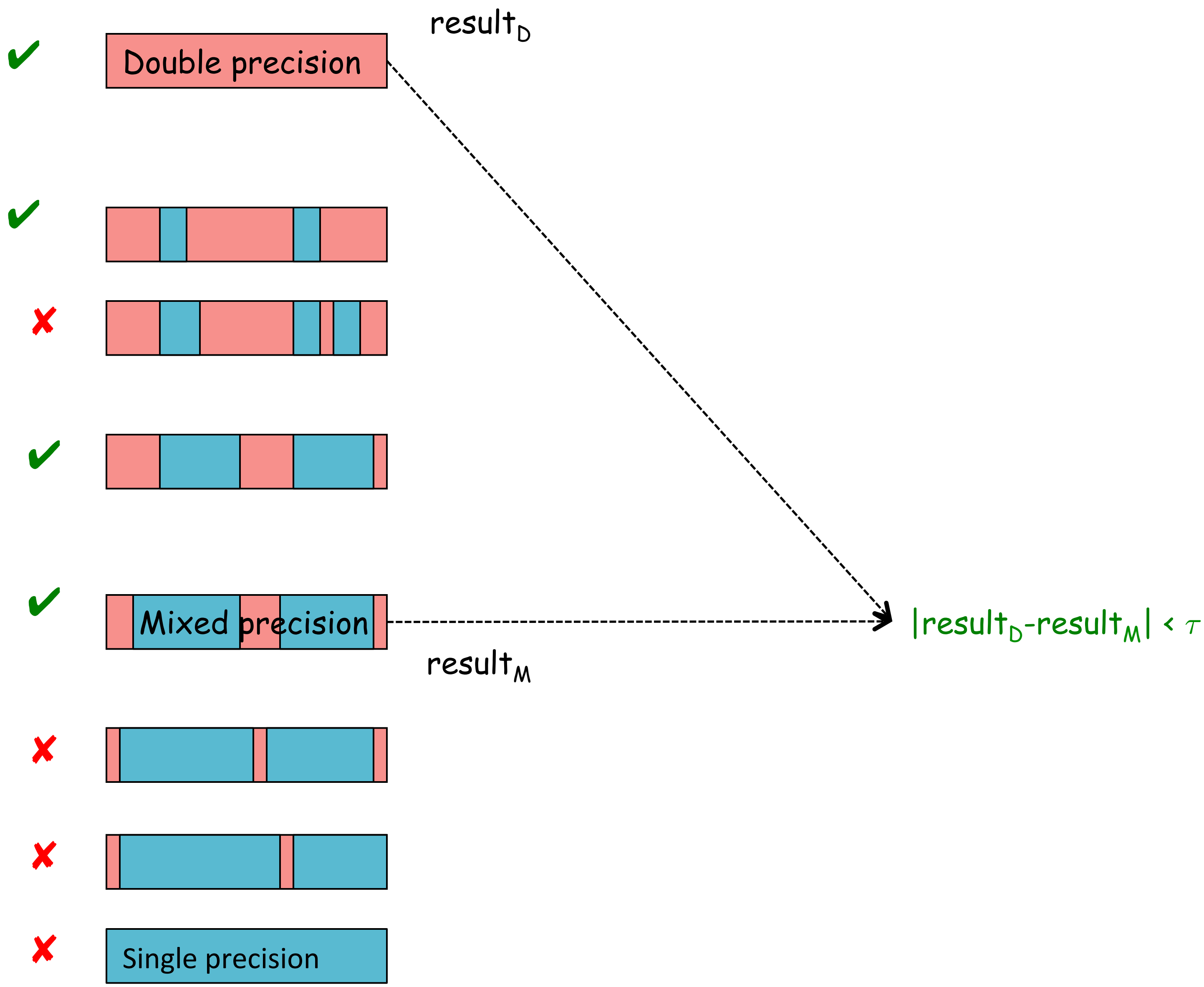
Delta Debugging to Propose Precision Reduction



Delta Debugging to Propose Precision Reduction



Delta Debugging to Propose Precision Reduction



Code Transformation: Create Variants

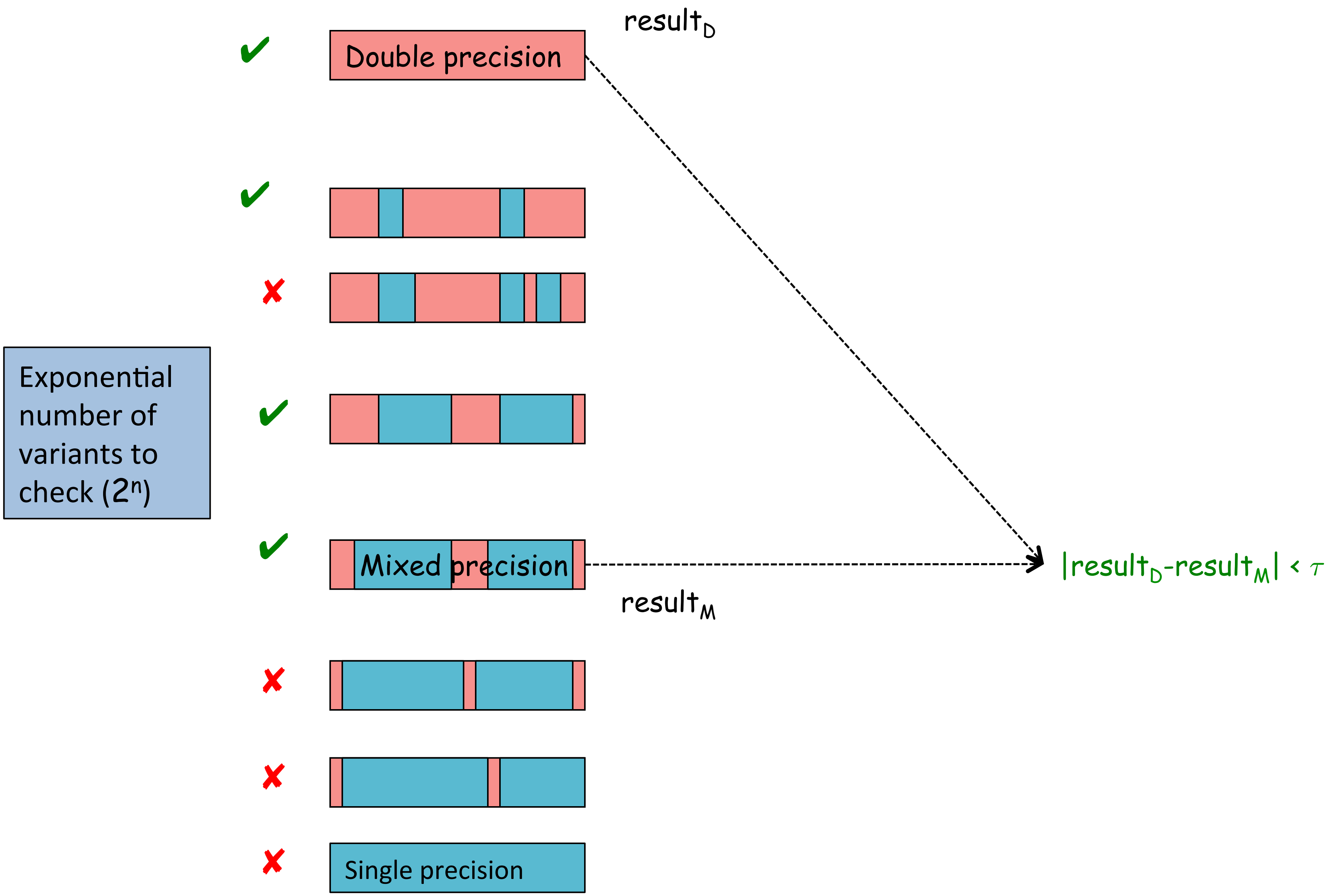
Use a compile framework (LLVM or CIL) or binary instrumentation

```
main() {
    float a;
    float b;
    float c;

    ...
    a = b + c;
    ...
}
```

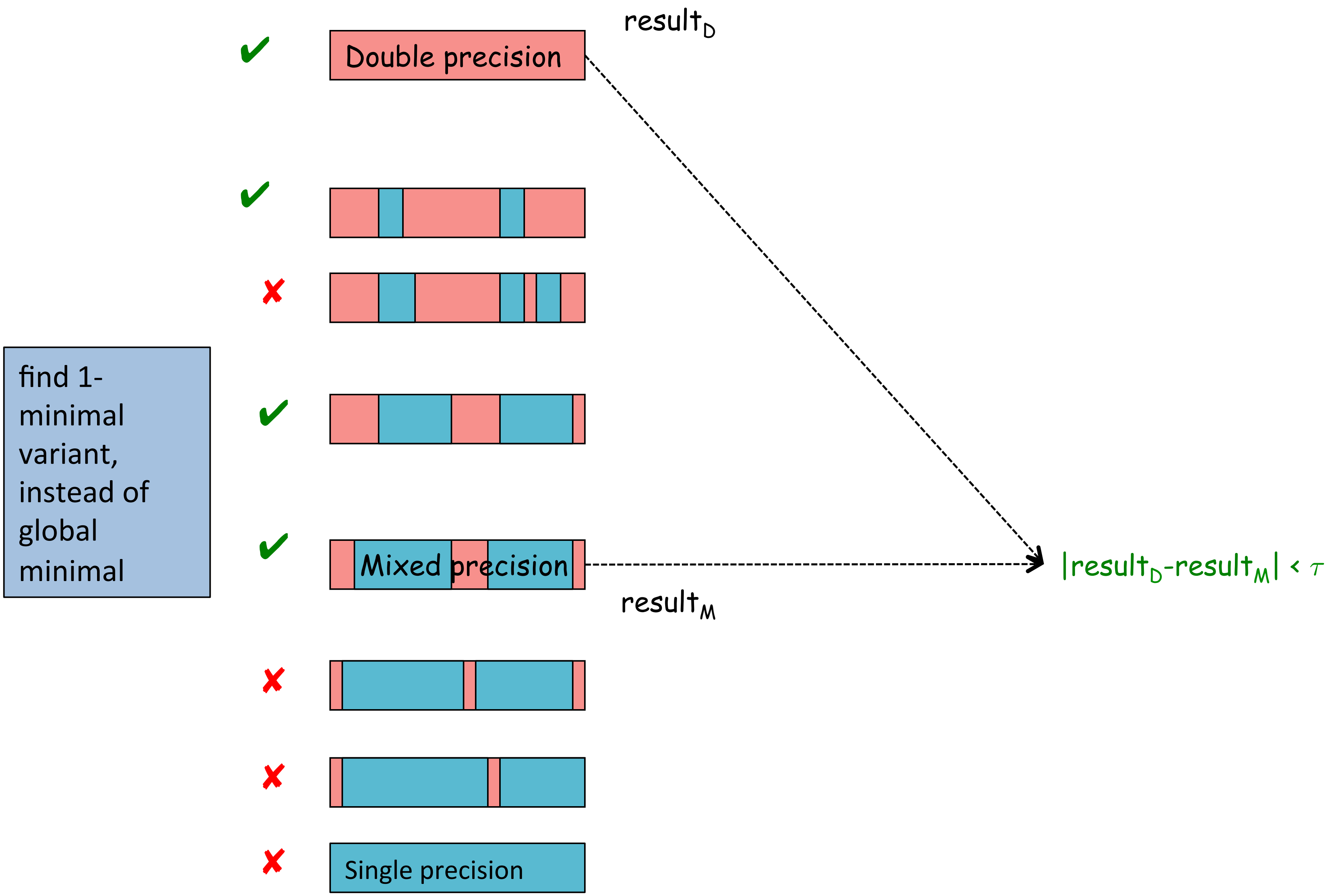
→

```
main() {
    double a;
    float b;
    double c;

    ...
    a = b + c;
    ...
}
```

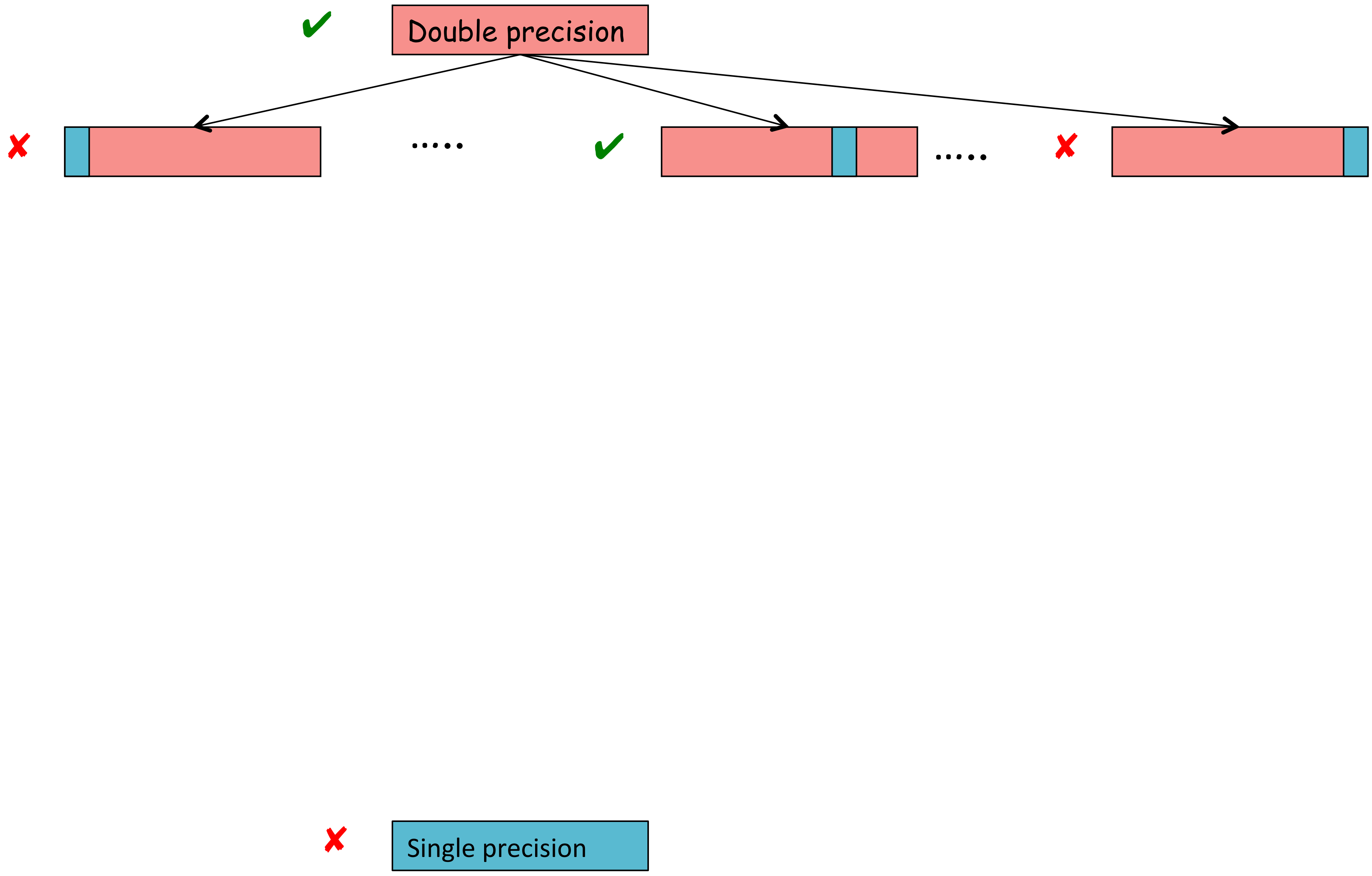
Delta Debugging to Propose Precision Reduction



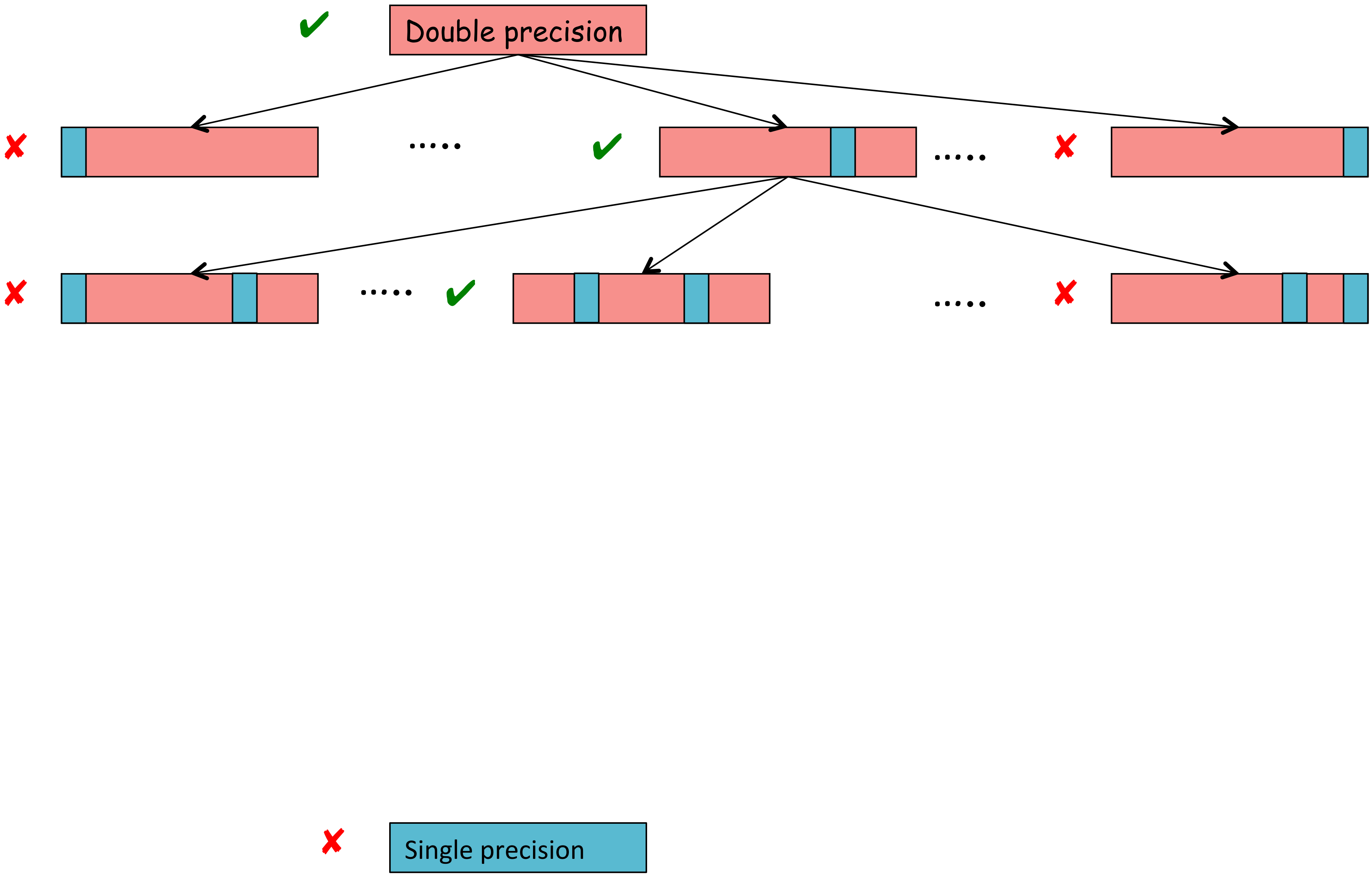
Delta Debugging to Propose Precision Reduction



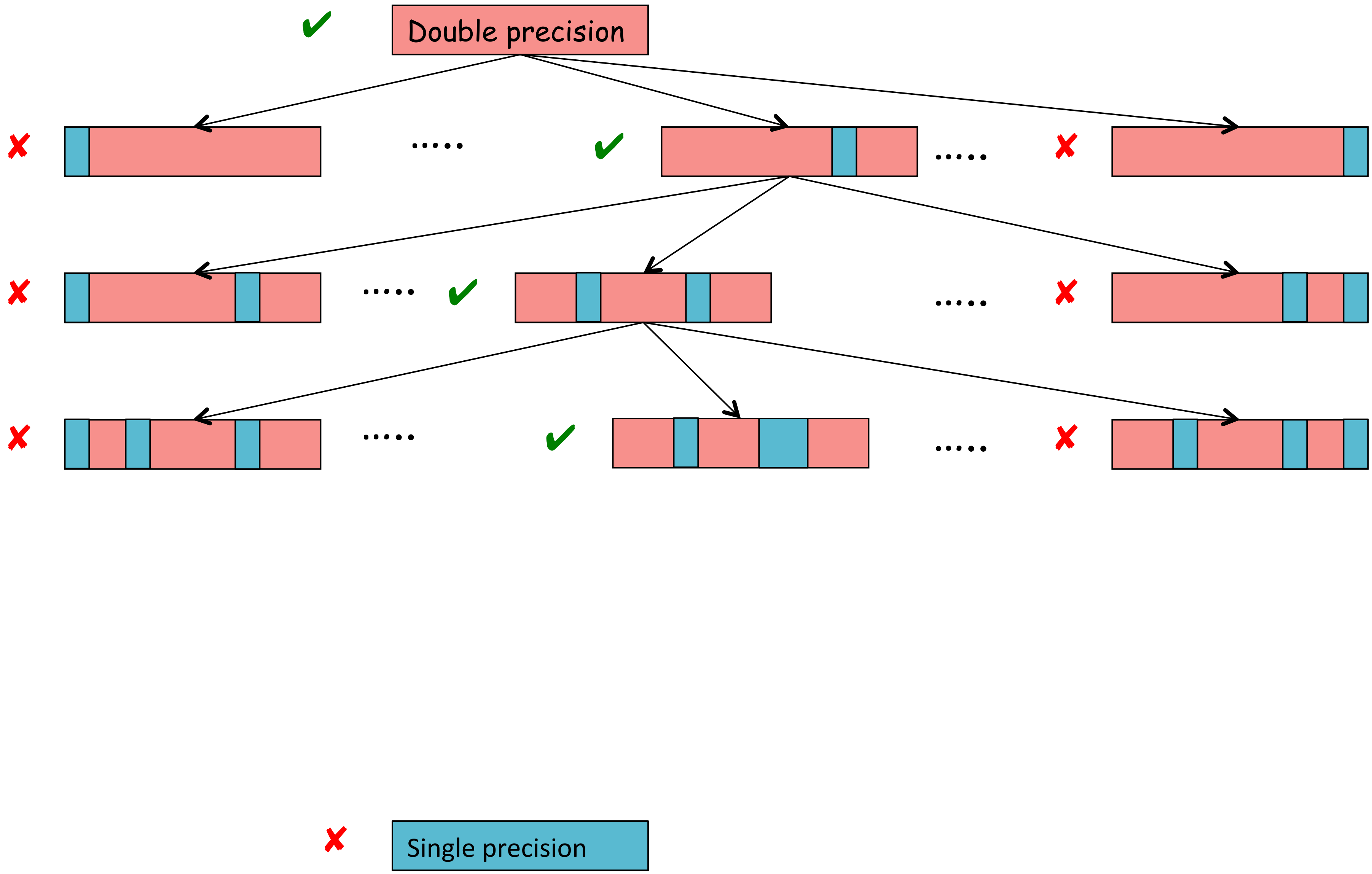
Delta Debugging to Propose Precision Reduction



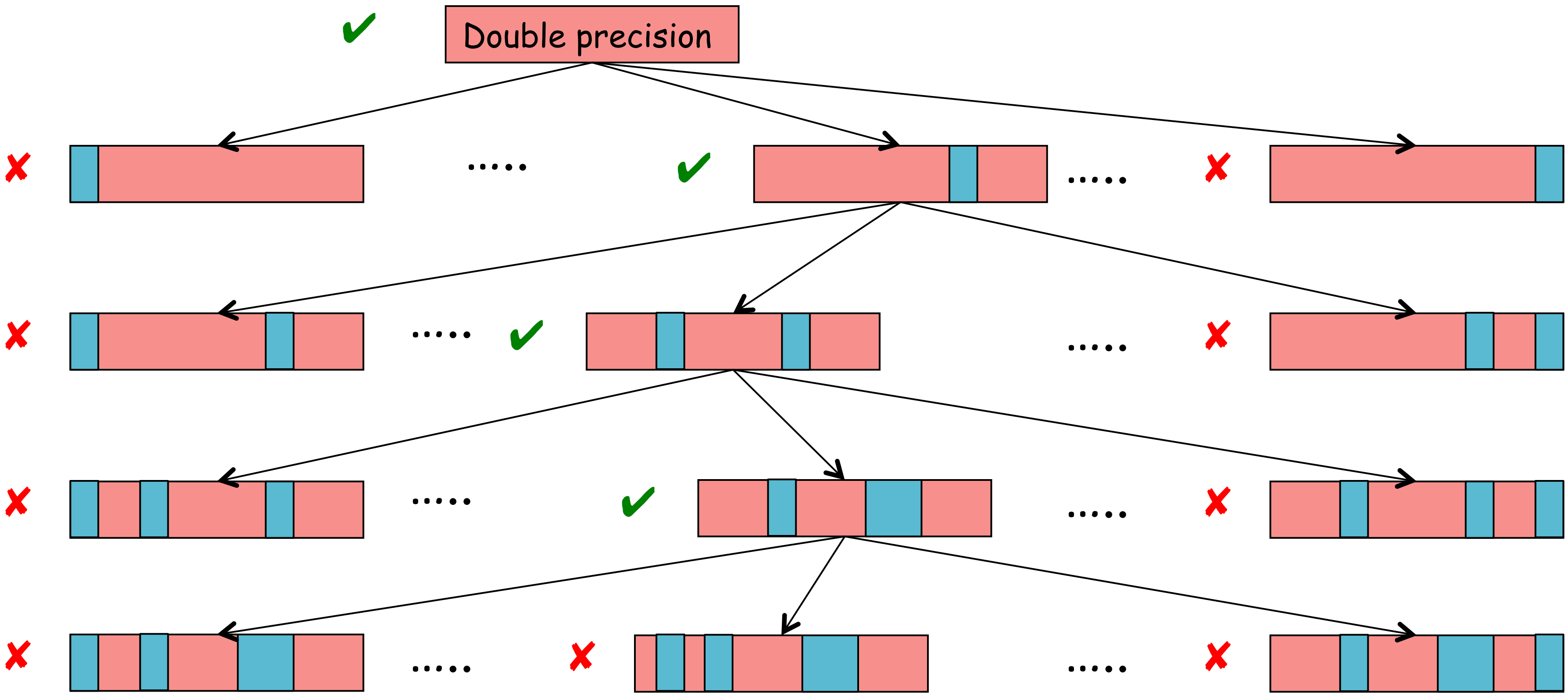
Delta Debugging to Propose Precision Reduction



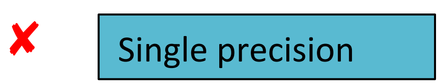
Delta Debugging to Propose Precision Reduction



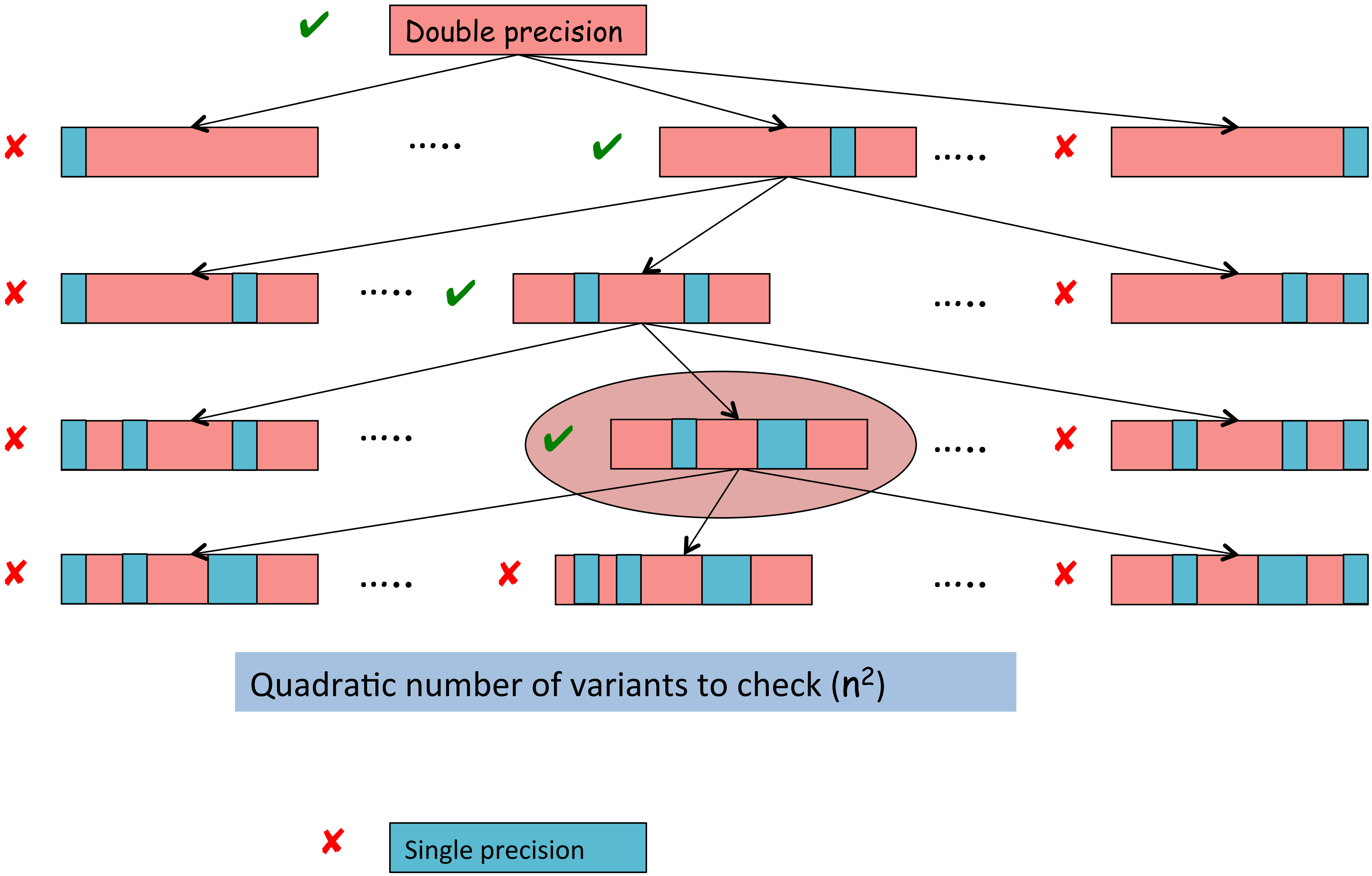
Delta Debugging to Propose Precision Reduction



Cannot change further without getting wrong result



Delta Debugging to Propose Precision Reduction



Quadratic number of variants to check (n^2)

Delta Debugging: Work Smarter, Not Harder

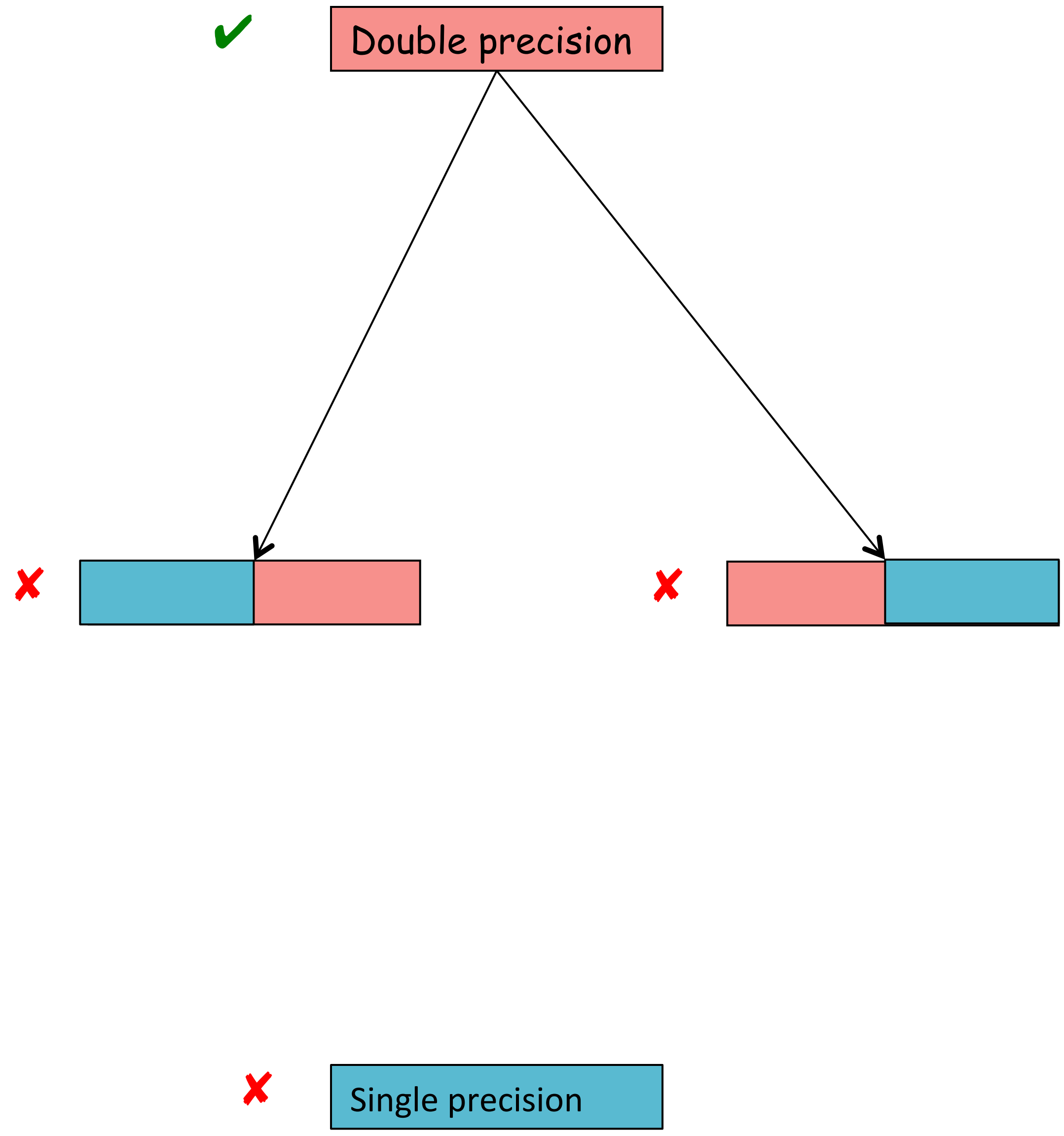
- [Zeller et al.]
- We can often do better
- Silly to modify 1 variable at a time
  - Try modifying half of the variables initially
  - Decrease the number of variables to modify if we can't make progress
  - If we get lucky, search will converge quickly

Delta Debugging to Propose Precision Reduction

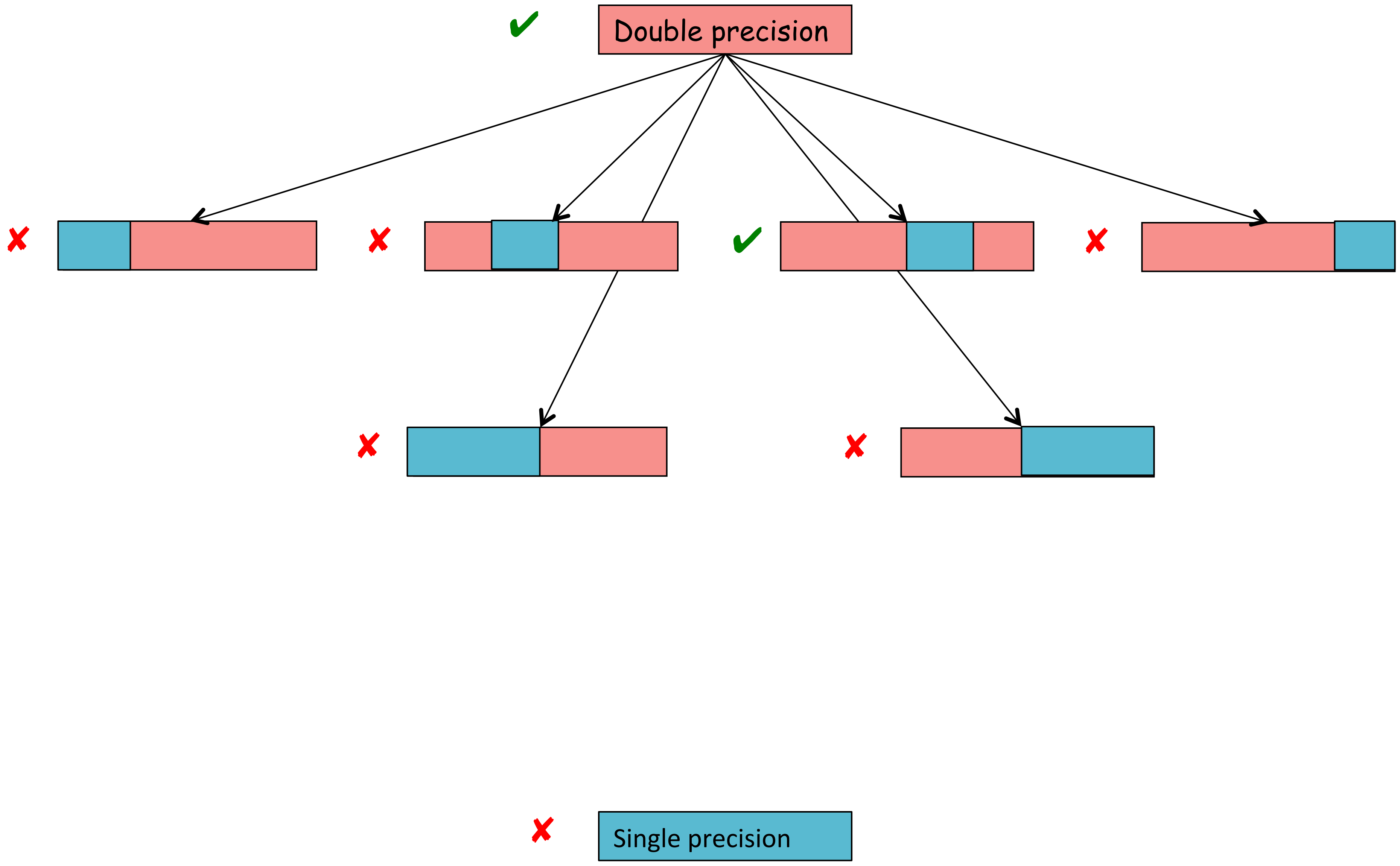
✔ Double precision

✘ Single precision

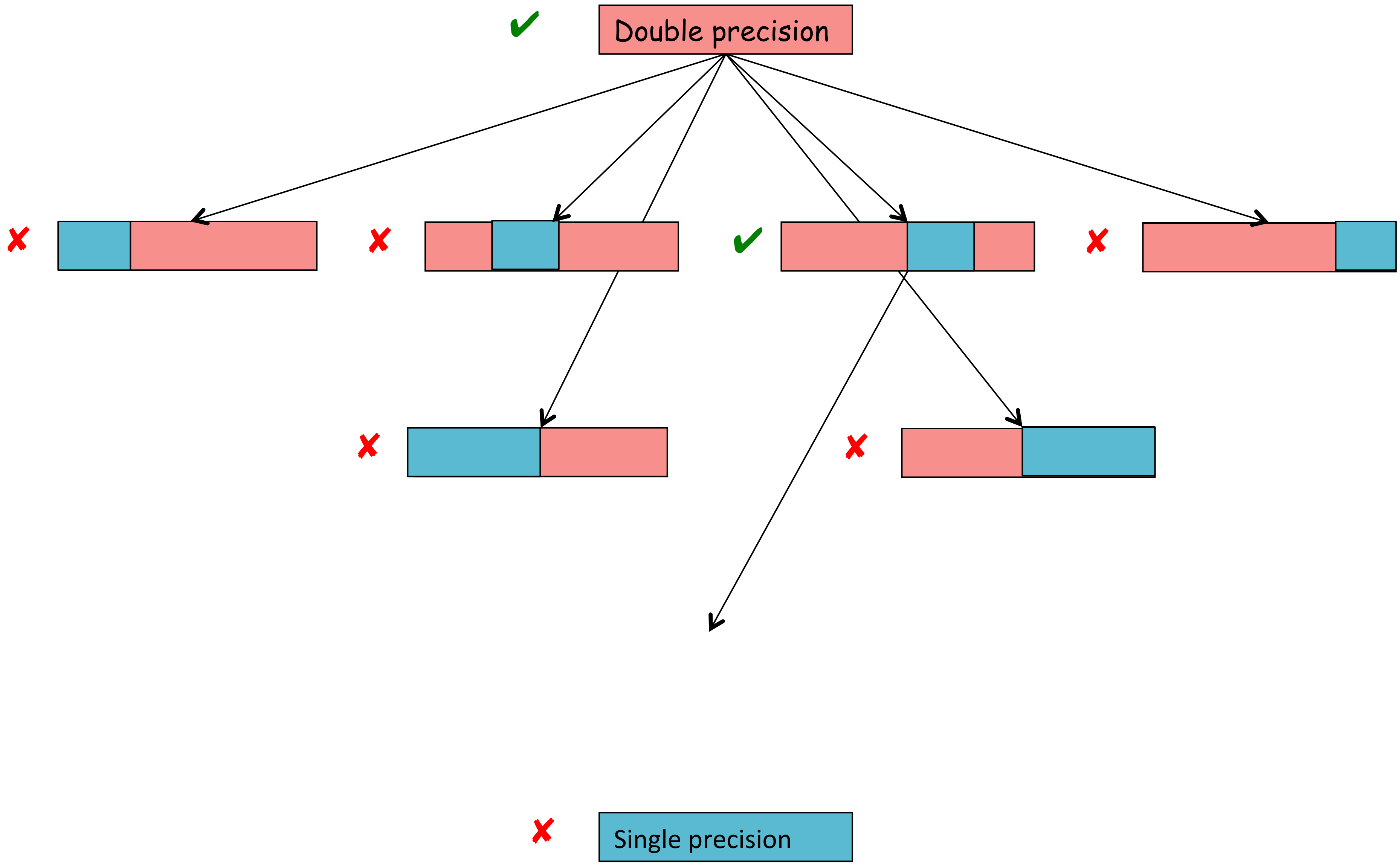
Delta Debugging to Propose Precision Reduction



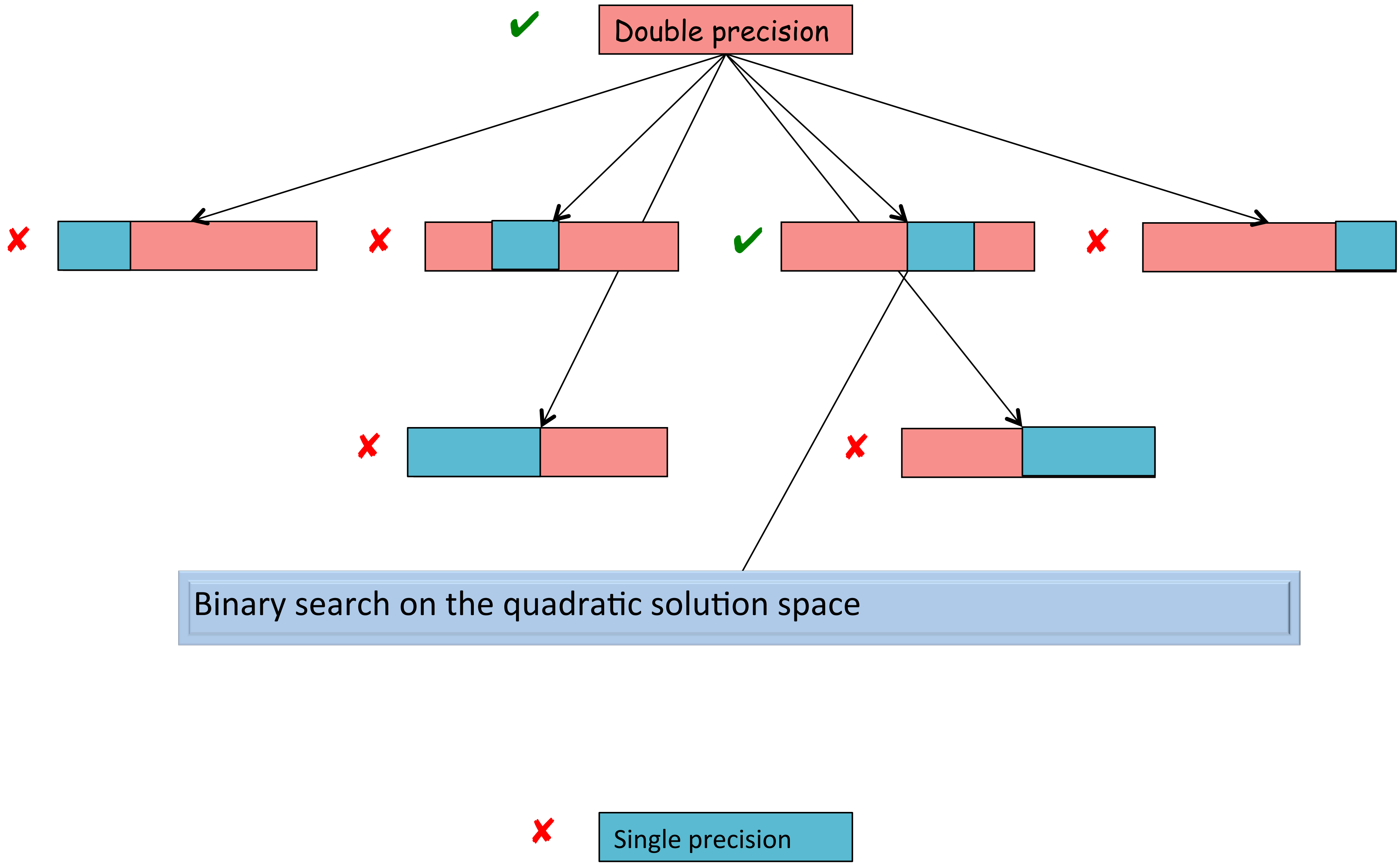
Delta Debugging to Propose Precision Reduction



Delta Debugging to Propose Precision Reduction



Delta Debugging to Propose Precision Reduction



Example (D.H. Bailey)

```
#include <math.h>
#include <stdio.h>

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


  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;
    t1 = t1 + sin (
  }
  return t1;
}


int main( int argc
  int i, j, k, n = 1
  long double h
  long double s



  t1 = −1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 − t1)*(t2 − t1));
    t1 = t2;
  }

  // final answer is stored in variable s1
  return 0;
}
```

Original Program

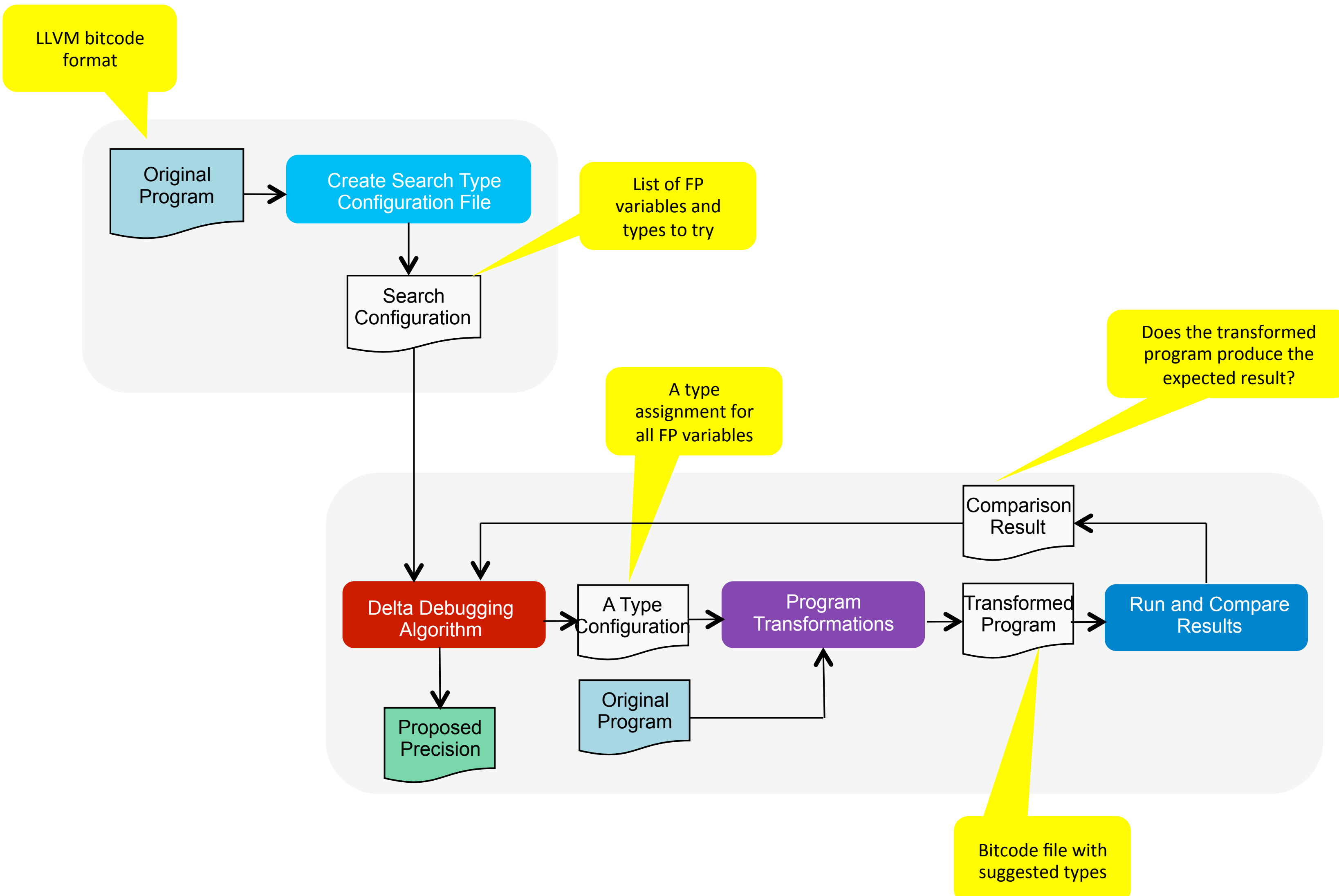
```
#include <math.h>
#include <stdio.h>

double fun( double x ) {
  int k, n = 5;
  double t1;
  float d1 = 1.0f; // double before

  t1 = x;
  for( k = 1; k <= n; k++ ) {
    d1 = 2.0 * d1;

                                              efore

  float threshold = 1e−14f; // long double before

  t1 = −1.0;
  dppi = acos(t1);
  s1 = 0.0;
  t1 = 0.0;
  h = dppi / n;

  for( i = 1; i <= n; i++ ) {
    t2 = fun (i * h);
    s1 = s1 + sqrt (h*h + (t2 − t1)*(t2 − t1));
    t1 = t2;
  }

  // final answer is stored in variable s1
  return 0;
}
```

Modified Program

4 seconds to find type configuration

7.6% speedup

Framework Components



GNU Scientific Library (GSL)

- Applying analysis to programs using GSL library
- Preliminary results on three programs:

GSL Program		Variables		Loads		Stores		Arith Ops		
		F	D	F	D	F	D	F	D	Speedup %
bessel	original	0	18	0	557	0	217	0	359	-
	tuned	14	4	14	543	5	212	1	358	5.34
gaussian	original	0	56	0	271	0	129	0	152	-
	tuned	37	19	83	188	30	99	6	146	84.49
roots	original	0	15	0	678	0	352	0	178	-
	tuned	12	3	122	556	62	290	19	159	8.47

Progress to date

- Testing and Debugging of Distributed Parallel Programs
  - First complete analysis for hybrid programming models: handles both communication and load/store
  - THRILLE released under BSD license
  - PPoPP'13 poster and submitted paper
- Floating-point Debugging
  - LLVM-based prototype
  - Works on some programs in GNU Scientific Library
  - Preliminary results are encouraging!

Current and Future Work

- Analyze other programs that use the GSL library
  - Computing thresholds => Can we automate it?
  - Single inputs => Will the results be general enough?
  - Impact on real-world program clients
- Support pointers and structures
- Analyze other code bases
  - CLAPACK
  - Gyrokinetic Toroidal Code (GTC) from LBNL

Conclusions

- Build testing tools
  - Close to what programmers use
  - Hide program analysis under testing
- Automated testing and debugging tools
  - Can help to find nondeterministic bugs and floating point anomalies
  - Can propose precision reduction in FP programs
  - Will help to reduce power, improve performance, get desired accuracy
- If you are not obsessed with formal correctness
  - Testing and debugging can help you solve these problems with high confidence