Approximate Programming

Wei Li

2013. 3. 20

Outline

- Background
- Why Approximate Computation
- State-of-the-art Approaches
- Problem
- Conclusion

Background

- Exact computations with discrete logical correctness requirements.
- Approximate computations aspire only to produce an acceptably accurate approximation to an exact output.

Potential Applications

- **Growth of data**
  – Information retrieval and analysis. Eg. Google search
  – Data mining
- **Stream processing**
  – Audio, video, image stream processing
- **Machine Learning**
  – Recommender systems

Why Approximate Computation

- **Energy efficiency**
  – Mobile devices, servers.

- Trade-off accuracy for benefits such as increased performance and reduced resource consumption.

- From a **higher level** to address the energy-efficiency problem

Why Today

- The development of energy-efficient hardware
- The prominence of the potential applications
- The step-by-step mature of the approximate computations

Three Levels of Techniques

- **Algorithmic level**
  - Algorithm and application
- **Architecture level**
  - Software / hardware interface, compiler
- **Implementation level**
  - Hardware
  - Redundancy to combat unreliability

State-of-The-Art

- **Algorithmic level**
  - Program transformation
- **Architecture level**
  - EnerJ
- **Implementation level**
  - Architecture support for disciplined approximate programming

Algorithmic level

- **Randomized accuracy-aware program transformations for efficient approximate computations**

POPL, 2012

Accuracy-Aware Transformations

- Given a computation and a probabilistic **accuracy specification**
- Transformations change the computation so that it operates more efficiently while satisfying the specification.

Two Classes of Transformations

- **Substitution transformations** replace one implementation of a function node with another implementation.

- **Sampling transformations** cause the transformed reduction node to operate on a randomly selected subset of its inputs

Example

$$I = \Delta x \cdot \sum_{i=1}^{n} f(x_i) = \frac{1}{n} \sum_{i=1}^{n} (b - a) \cdot f(x_i)$$

$$f(x) = x \cdot \sin\left(\log(x)\right)$$

Other Technologies

- Task skipping
- Loop perforation
  - Skip instructions
- Substitution of multiple alternate implementations

Architecture level

- **EnerJ: approximate data types for safe and general low-power computation**

PLDI, 2011

EnerJ

- Implement a **type system** on top of Java with **annotations** for variables and objects
- To isolate parts of the program that must be precise from those that can be approximated

```
final long N = 100000;
final long T = 100;

@Approx double m, S, pi;
```

Approximation

- Variables and objects

- **Memory**: registers, caches, main memory
- **Operation**: +, -, Math.sqrt(), etc.

Implementation level

- **Architecture support for disciplined approximate programming**

ASPLOS, 2012

A Dual-Voltage Microarchitecture

- Dual-voltage multiplexers
- **Duplicated** hardware for registers, ALU, etc. to support approximate computation in a low voltage.

Problem

- The specification of the possibility of accuracy
  - **Controlled**
- Redundancy in hardware design
- Neglect the **overhead of switching**

Can we do better?

- **Architecture level**
  - Hybrid voltage regulator
  - Fuzzycall function
  - **Static program analysis**
  - **Symbolic execution**

AgileRegulator

HPCA 2012

- Hybrid scheme of on-chip and off-chip voltage regulator.
- Off-chip: higher power delivery efficiency, but is not responsive
- On-chip: has much shorter latency, relatively lower power delivery efficiency and it dictates significant amount of chip area.

Example

- Calculate Pi with Monte Carlo method
- Call a function flip_coin() 1000 times
- Return whether the coin is in the unit circle of a square
- Specify 0.1 error rate

Example

```
public static int coin()
{
    final int R = 1;
    double x = nextDouble();
    double y = nextDouble();
    double d = sqrt(pow(x - R, 2) + pow(y - R, 2)) - R;
    if (d <= 0) return 1;
    return 0;
}
```

- **Dependency graph** to calculate the error rate of each operation node

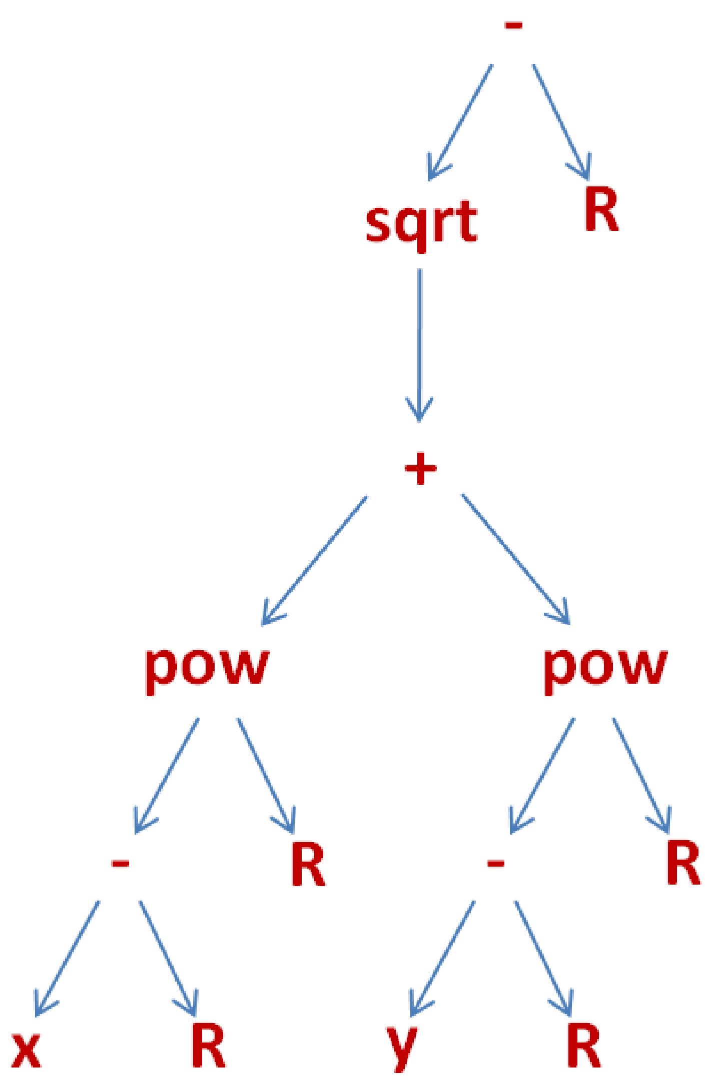
Example

The tree has an error rate of 0.1

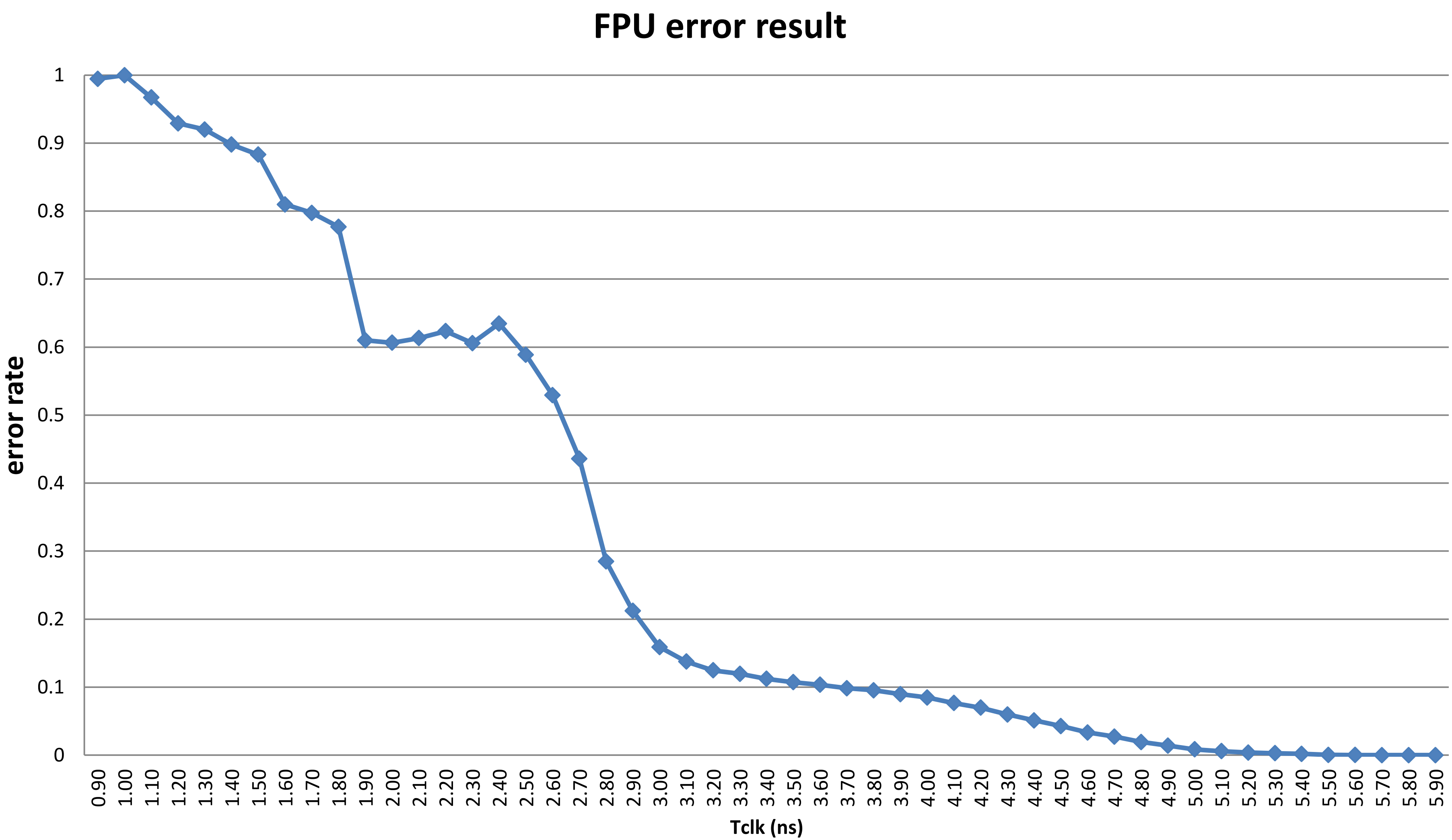
Each node has the same error rate

The error rate of each subtree depends on the error rate of its **left subtree** and **right subtree**

e = 1 – (1 - e1) * (1 - e2)



Error rate to Voltage



Functions containing loops ?

- Leverage the **symbolic execution** and **static program analysis**
- **program analysis** is the process of automatically analyzing the behavior of computer programs.

Static Analysis

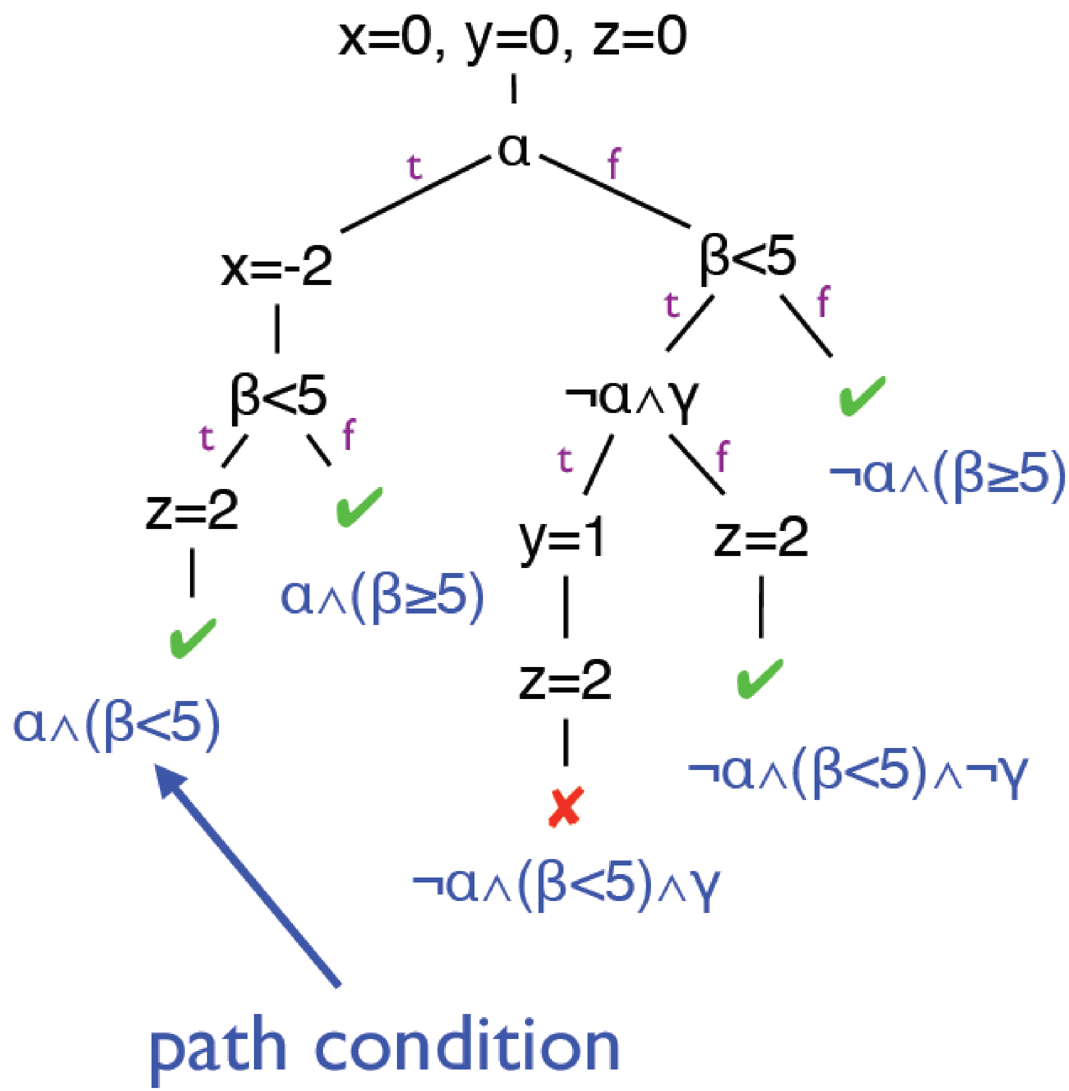
- Static analysis allows us to **reason** about all **possible executions** of a program
  - Gives assurance about any execution, prior to deployment
- **Abstract interpretation** lets us scale and model all possible runs
  - But must be conservative
  - Try to balance precision and scalability

Symbolic Execution

- generalize testing by using unknown **symbolic variables** in evaluation
- Symbolic executor executes program, tracking symbolic state.
- If execution path depends on unknown, we fork symbolic executor

Example

```
int a = α, b = β, c = γ;
                 // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Symbolic Execution

- During symbolic execution, we are trying to determine if certain formulas are **satisfiable**
  - E.g., is a particular program point reachable?

  (Figure out if the path condition is satisfiable)
  - E.g., generate concrete inputs that execute the same paths

- This is enabled by powerful SMT/SAT solvers

Conclusion

- Acceptably **trade-off** accuracy for the benefits of performance and resource consumption
- The State-of-the-art approaches
- To move on
  - Static analysis
  - Symbolic execution

Any Questions?

- Acceptably **trade-off** accuracy for the benefits of performance and resource consumption
- The State-of-the-art approaches
- To move on
  - Static analysis
  - Symbolic execution

Thanks !