Logical Reasoning for Approximate and Uncertain Computation

Michael Carbin
Microsoft Research and MIT CSAIL

Microsoft Research
Faculty Summit
2015
July 8-9, 2015

Logical Reasoning for Approximate and Uncertain Computation

Michael Carbin

Microsoft Research and MIT CSAIL

Thought Experiment.

Loop Perforation

```
for (uint i = 0; i < n; ++i) {...}
```

```
for (uint i = 0; i < n/2; ++i) {...}
```

What will happen to your program?

Faster and consumes less energy!

May give the wrong result.

Faster and consumes less energy!

May give ~~the wrong~~ result.

a ***different***

Let’s try it and see how it works!

Original

Perforated
(2x performance)

Loop Perforation Results

(ICSE '10, ASPLOS '11, FSE '11, PEPM '13)

Applications

Media Processing

Computer Vision

Machine Learning

Search

Finance

Framework

- Developer specifies maximum acceptable error using error metric
- Automatically identifies loops perforations with acceptable error

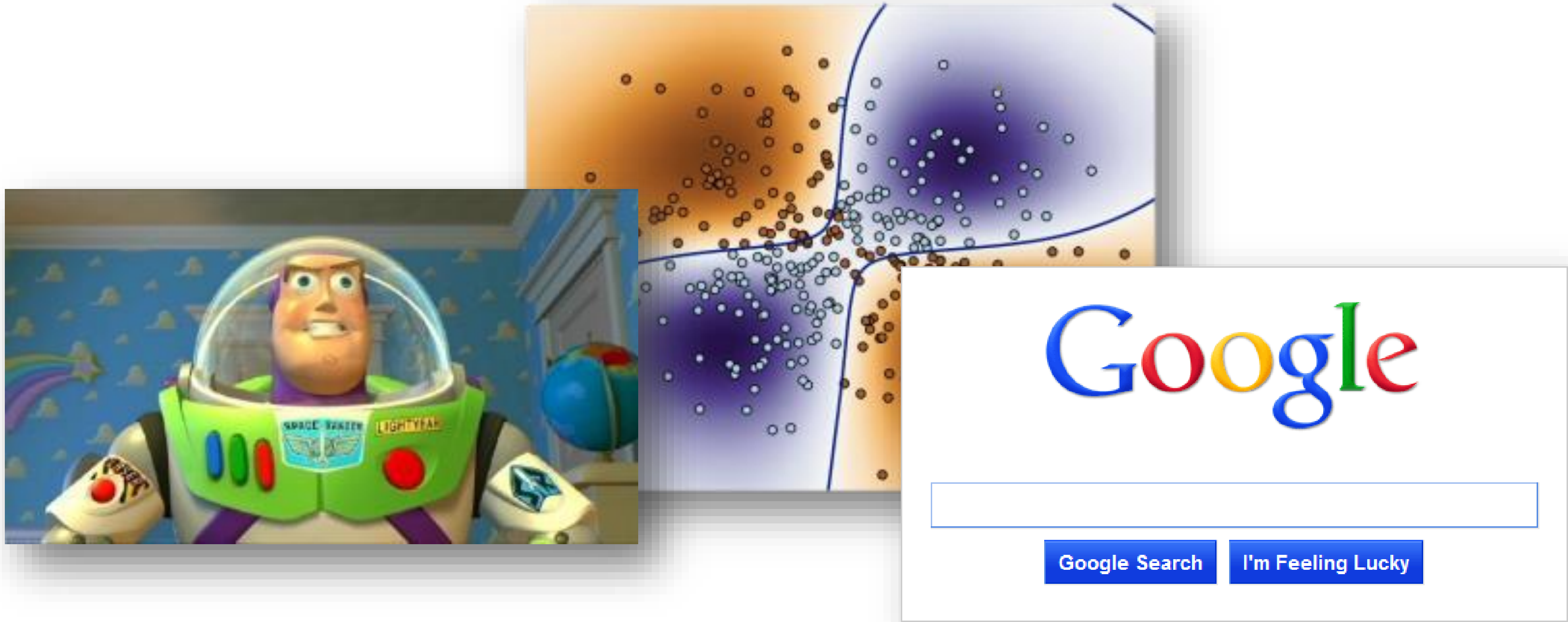
Performance improvement

- Typically over a factor of two
- Up to a factor of seven

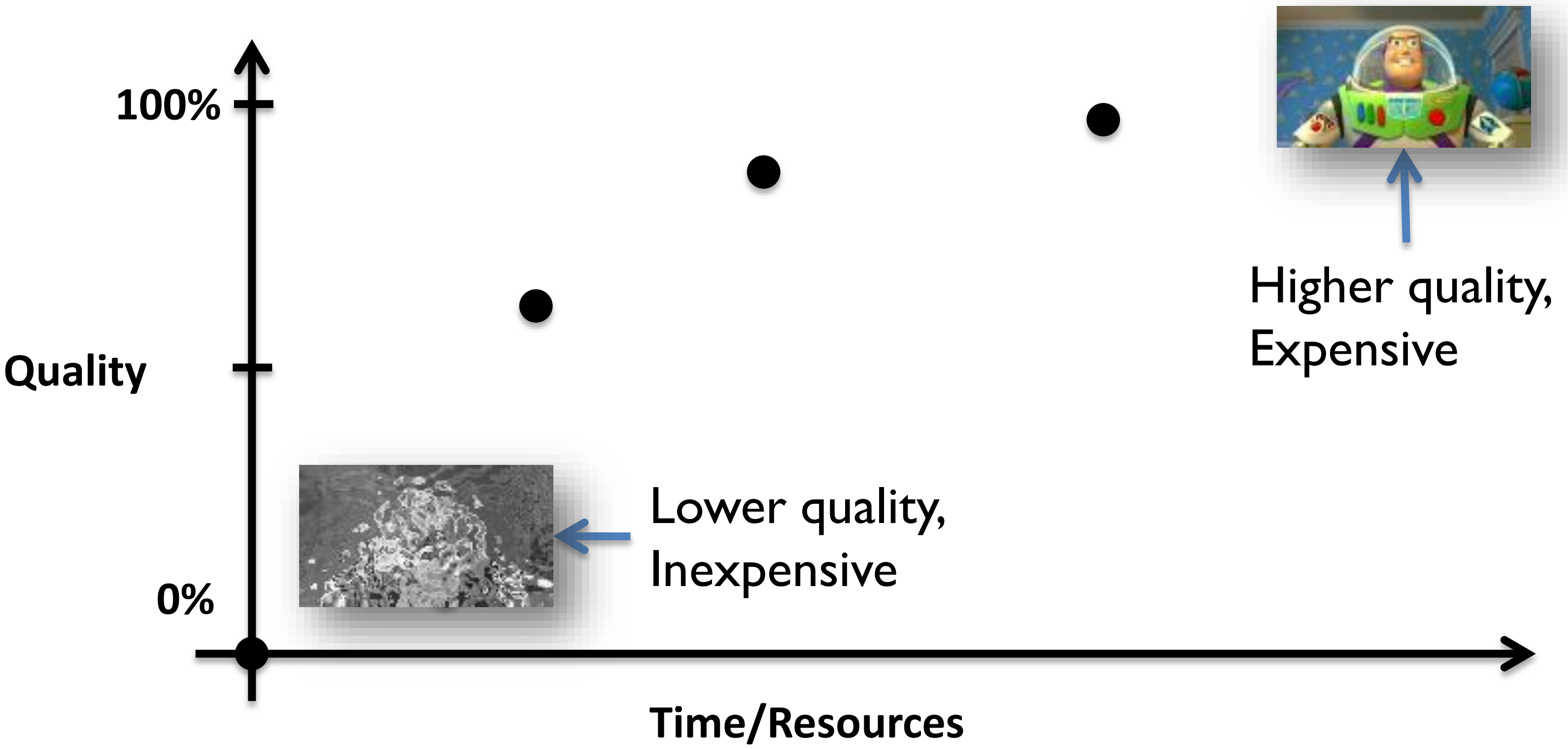
Quality Impact

- < 10% change in output

Approximate Computations



Approximate Computations



New opportunity to trade quality for increased performance

Approximation Techniques

- Code Perforation

  Rinard, ICS '06; Baek et al., PLDI 10; Misailovic et al., ICSE '10; Sidiroglou et al., FSE '11; Misailovic et al., SAS '11; Zhu et al., POPL '12; Carbin et al. PEPM '13; Samadi et al. ASPLOS '14

- Function Substitution

  Hoffman et al., APLOS '11; Ansel et al., CGO '11; Zhu et al., POPL '12

- Approximate Memoization

  Alvarez et al., IEEE TOC '05; Chaudhuri et al., FSE '12; Samadi et al., ASPLOS '14

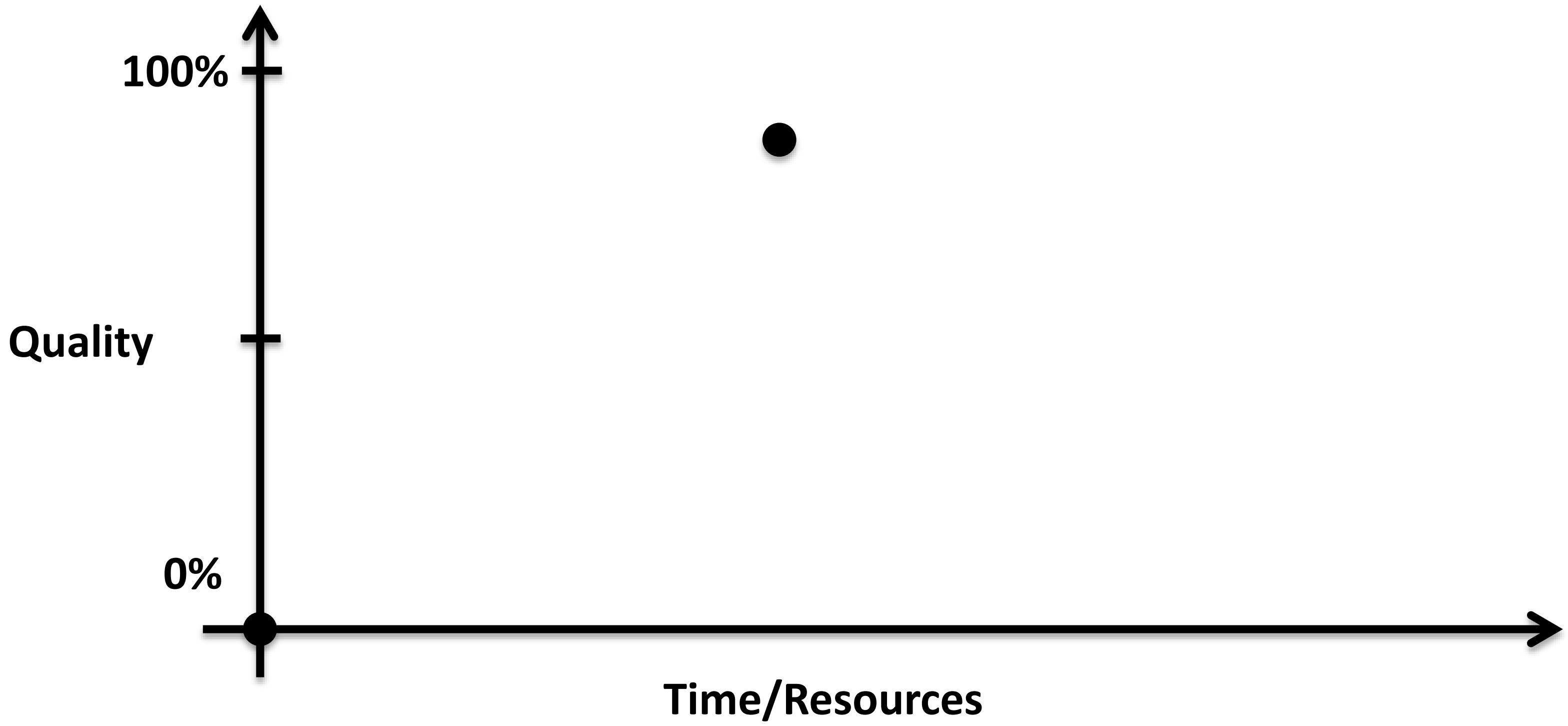
- Relaxed Synchronization (Lock Elision)

  Renganarayana et al., RACES '12; Rinard, HotPar '13; Misailovic, et al., RACES '12

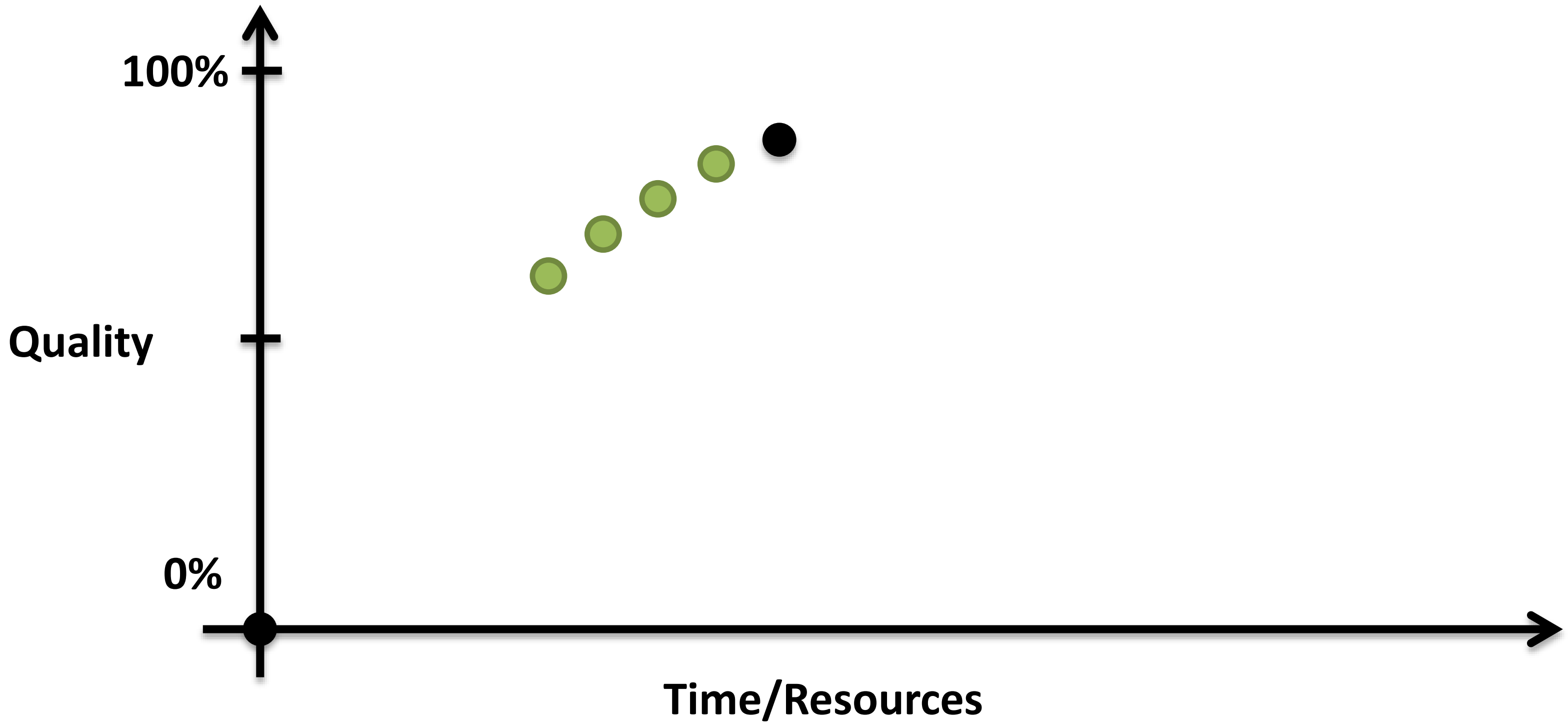
Approximate Hardware

Ernst et al, MICRO 2003; Samson et al., PLDI '11; PCMOS, Palem et al. 2005; Narayanan et al., DATE '10; Liu et al. ASPLOS '11; Venkataramani et al., MICRO '13

Original Application



Approximate Computing



Benefit: create new operating points in trade-off space

How do we develop and reason about approximate programs?

The Problem

- Produce an inaccurate result

  5 + 5 = 8

- Produce correct results too infrequently

  Pr(5 + 5 = 10) too low

- Produce an invalid result

  5 + 5 = “hello”

- Crash or do something nefarious

  5 + 5 = exec “/bin/launch_missiles”

Challenges for Developing Approximate Programs

- How to express important program properties?
- How to approximate and capture resulting program behaviors?
- How to reason about program to ensure that properties hold?

Solution: design a programming methodology and supporting programming languages to address these challenges.

Proving Acceptability Properties of Relaxed Approximate Programs

Michael Carbin, Deokhawn Kim, Sasa Misailovic, and Martin Rinard

PLDI '12: Programming Language Design and Implementation

Verifying Quantitative Reliability for Programs that Execute on Unreliable Hardware

Michael Carbin, Sasa Misailovic, and Martin Rinard

OOPSLA '13 (Best Paper Award): Object-Oriented Programming, Systems, Languages & Applications

Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels

Sasa Misailovic, Michael Carbin , Sara Achour, Zichao Qi, Martin Rinard

OOPSLA '14 (Best Paper Award): Object-Oriented Programming, Systems, Languages & Applications

Step #1: Develop a Program

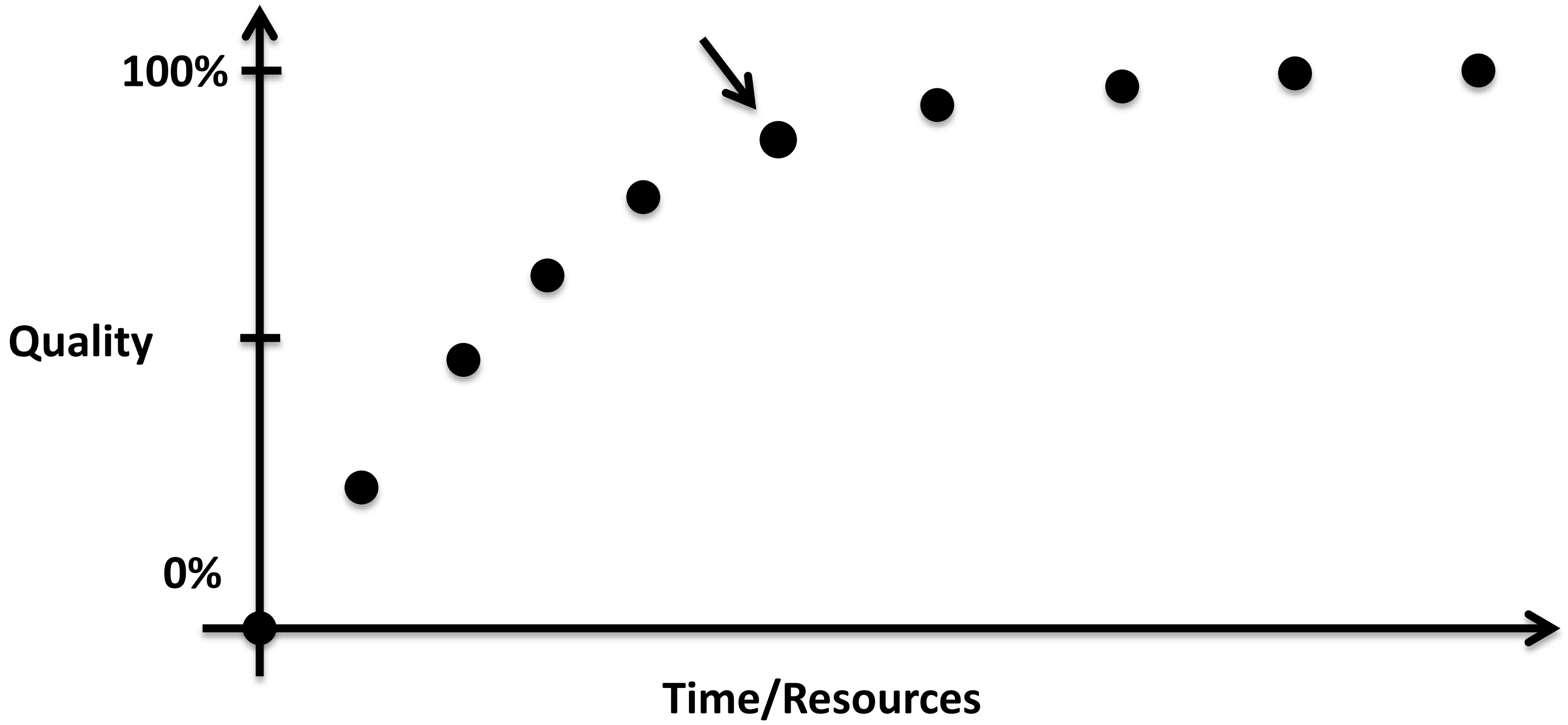


Image Scaling

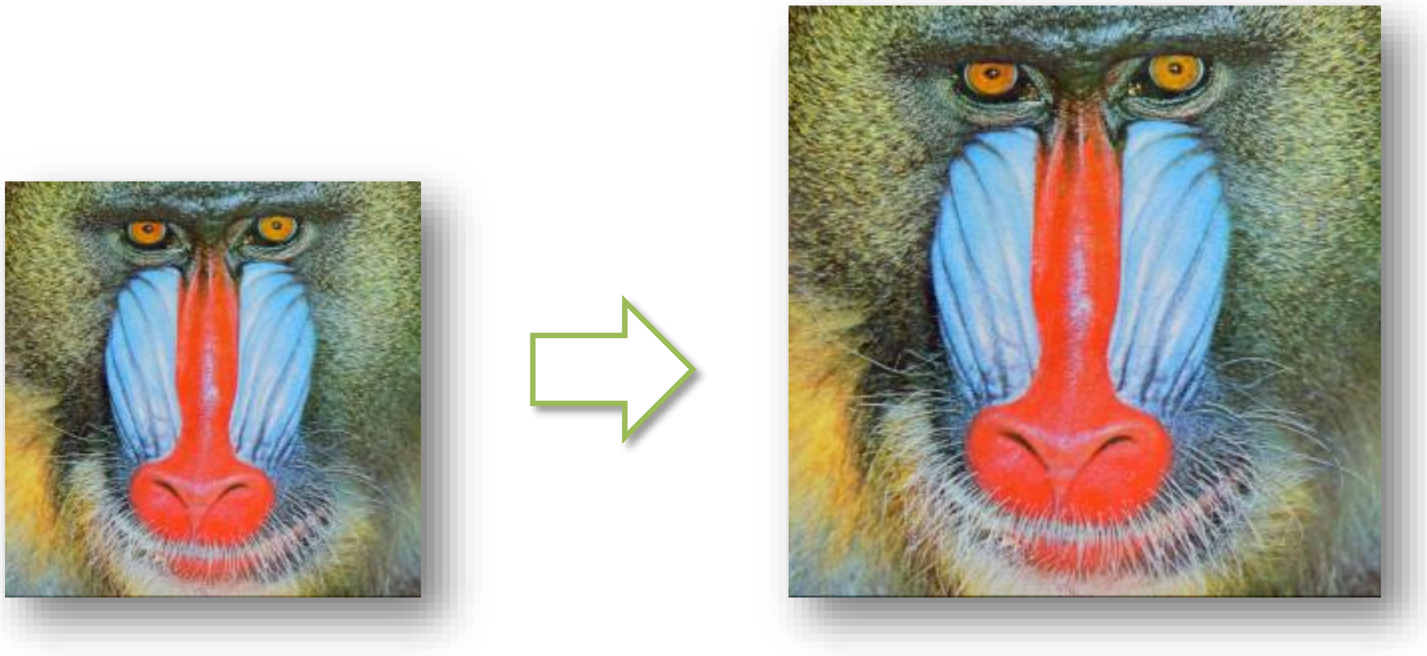
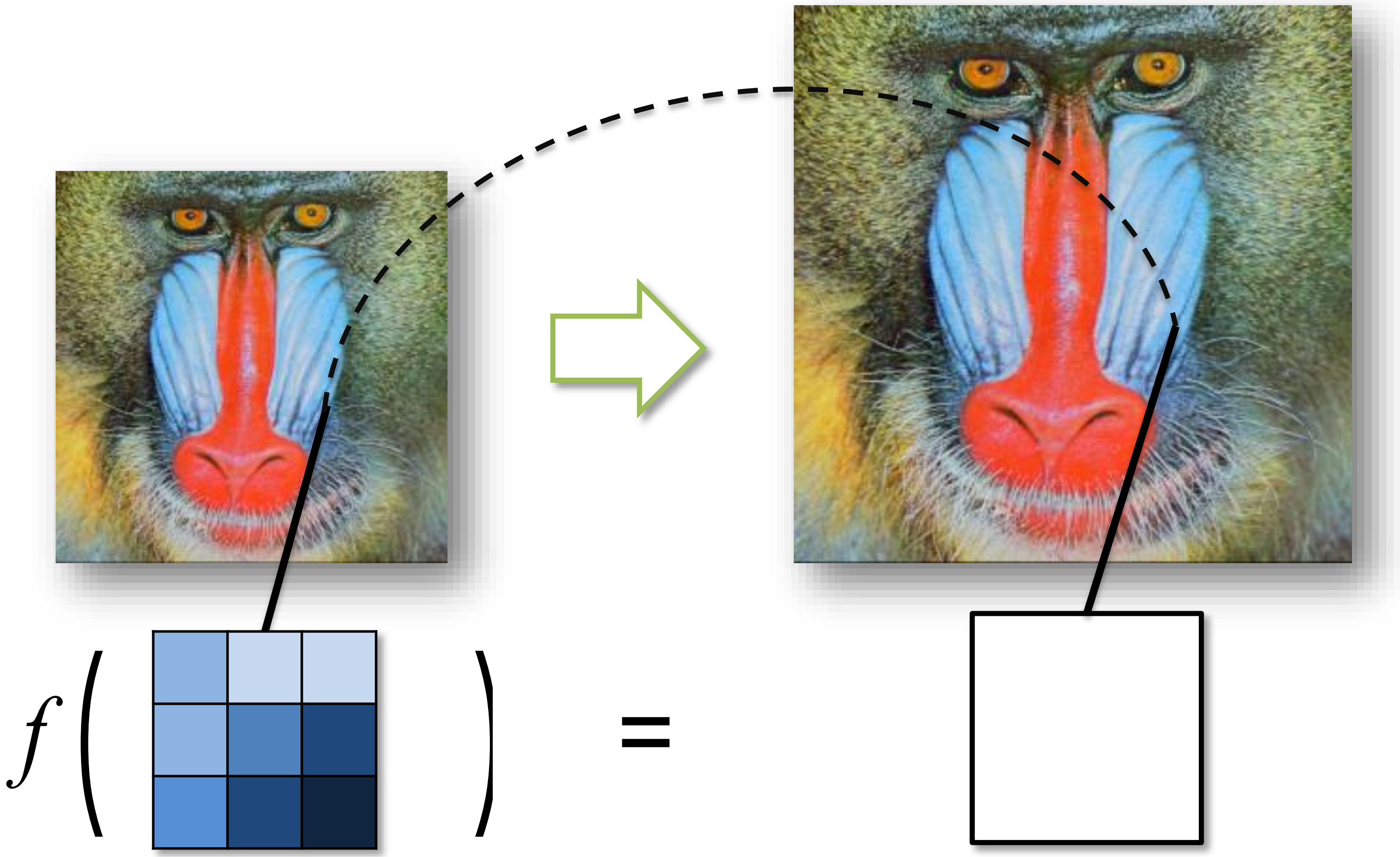


Image Scaling Kernel: Interpolation



Interpolation

```
uint interpolation(int x, int y, int src[][], int dest[][])
{









}
```

Interpolation

```
uint interpolation(int x, int y, int src[][], int dest[][])
{
    int x_src = map_x(x, src, dest),
        y_src = map_y(y, src, dest);









}
```

Interpolation

```
uint interpolation(int x, int y, int src[][], int dest[][])
{
    int x_src = map_x(x, src, dest),
        y_src = map_y(y, src, dest);

    int xs[MAX_N], ys[MAX_N];
    uint n = get_neighbors(x_src, y_src, src, xs, ys);









}
```

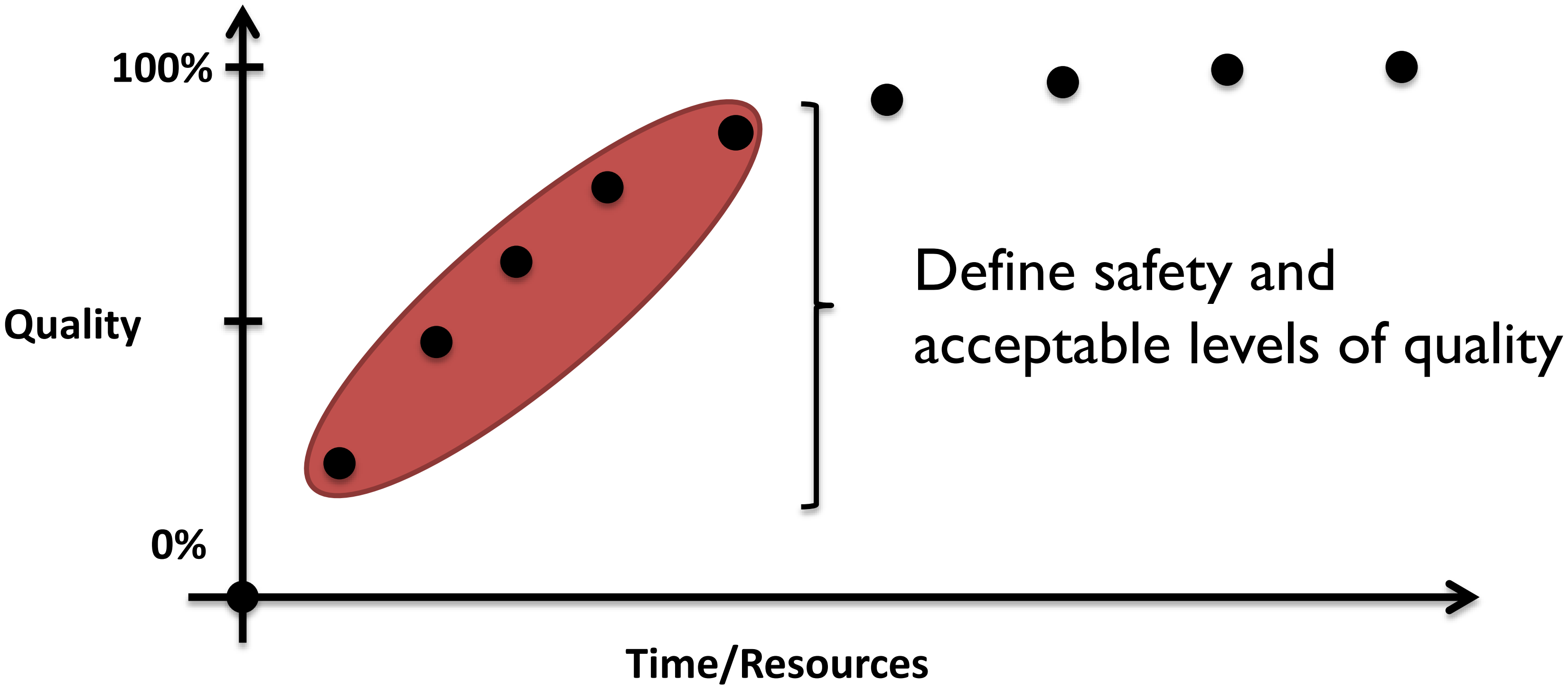
Interpolation

```
uint interpolation(int x, int y, int src[][], int dest[][])
{
    int x_src = map_x(x, src, dest),
        y_src = map_y(y, src, dest);

    int xs[MAX_N], ys[MAX_N];
    uint n = get_neighbors(x_src, y_src, src, xs, ys);

    uint val = 0;


    for (uint i = 0; i < n; ++i) {

        val += src[ys[i]][xs[i]];
    }

}
```

Interpolation

```
uint interpolation(int x, int y, int src[][], int dest[][])
{
    int x_src = map_x(x, src, dest),
        y_src = map_y(y, src, dest);

    int xs[MAX_N], ys[MAX_N];
    uint n = get_neighbors(x_src, y_src, src, xs, ys);

    uint val = 0;


    for (uint i = 0; i < n; ++i) {

        val += src[ys[i]][xs[i]];
    }
    return 1.0/n * val;
}
```

Step #2: Define and Verify/Validate Acceptability



Acceptability Properties

1. Safety – properties required to produce a valid result

2. Reliability – probability program produces correct result

3. Accuracy – worst-case difference in program result

Acceptability Properties

1. Safety – properties required to produce a valid result

2. Reliability – probability program produces correct result

3. Accuracy – worst-case difference in program result

Safety

```
uint interpolation(int x, int y, int src[][], int dest[][])
{
    int x_src = map_x(x, src, dest),
        y_src = map_y(y, src, dest);

    int xs[MAX_N], ys[MAX_N];
    uint n = get_neighbors(x_src, y_src, src, xs, ys);

    uint val = 0;
    for (uint i = 0; i < n; ++i)
    {

        val += src[ys[i]][xs[i]];
    }
    return 1.0/n * val;
}
```

Array accesses of (xs, ys, src) must be within bounds

Other Safety Properties

```
uint interpolation(int x, int y, int src[][], int dest[][])
{
    int x_src = map_x(x, src, dest),
```

- Memory Safety (pointers are valid)
- Result Validity (results in range)
- Sufficiency (forward progress)
- Sanity Checks (well-formed data structures)

```
    }
    return 1.0/n * val;
}
```

Acceptability Properties

1. Safety – properties required to produce a valid result

```
assert (x != null)
```

2. Reliability – probability program produces correct result

3. Accuracy – worst-case difference in program result

Acceptability Properties

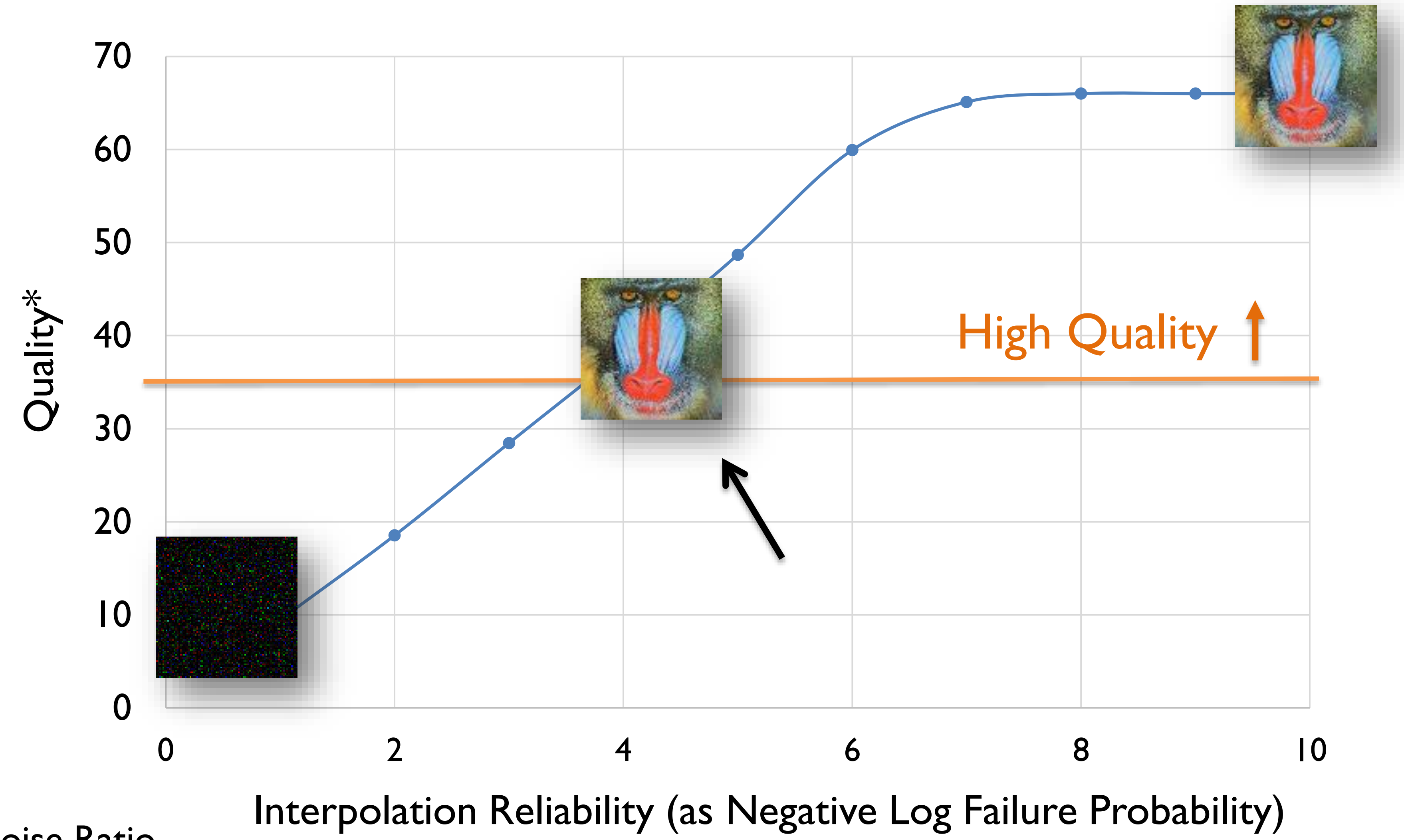
1. Safety – properties required to produce a valid result

```
assert (x != null)
```

2. Reliability – probability program produces correct result

3. Accuracy – worst-case difference in program result

Quality versus Reliability



*Peak-Signal-to-Noise Ratio

Acceptability Properties

1. Safety – properties required to produce a valid result

```
assert (x != null)
```

2. Reliability – probability program produces correct result

```
Pr(res == res') >= .99
```

3. Accuracy – worst-case difference in program result

Acceptability Properties

1. Safety – properties required to produce a valid result

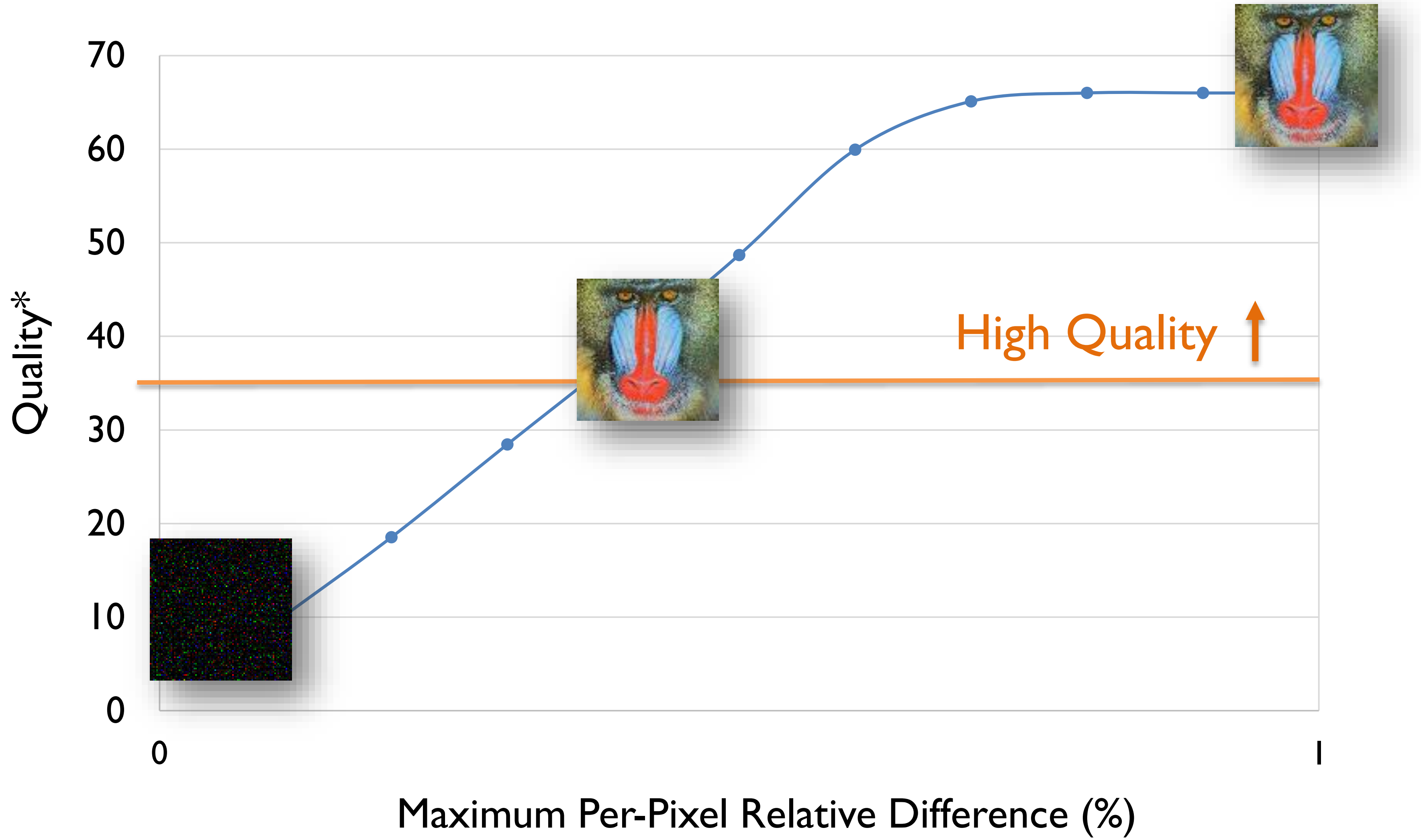
```
assert (x != null)
```

2. Reliability – probability program produces correct result

```
Pr(res == res') >= .99
```

3. Accuracy – worst-case difference in program result

Quality vs Local Accuracy



*Peak-Signal-to-Noise Ratio

Acceptability Properties

1. Safety – properties required to produce a valid result

```
assert (x != null)
```

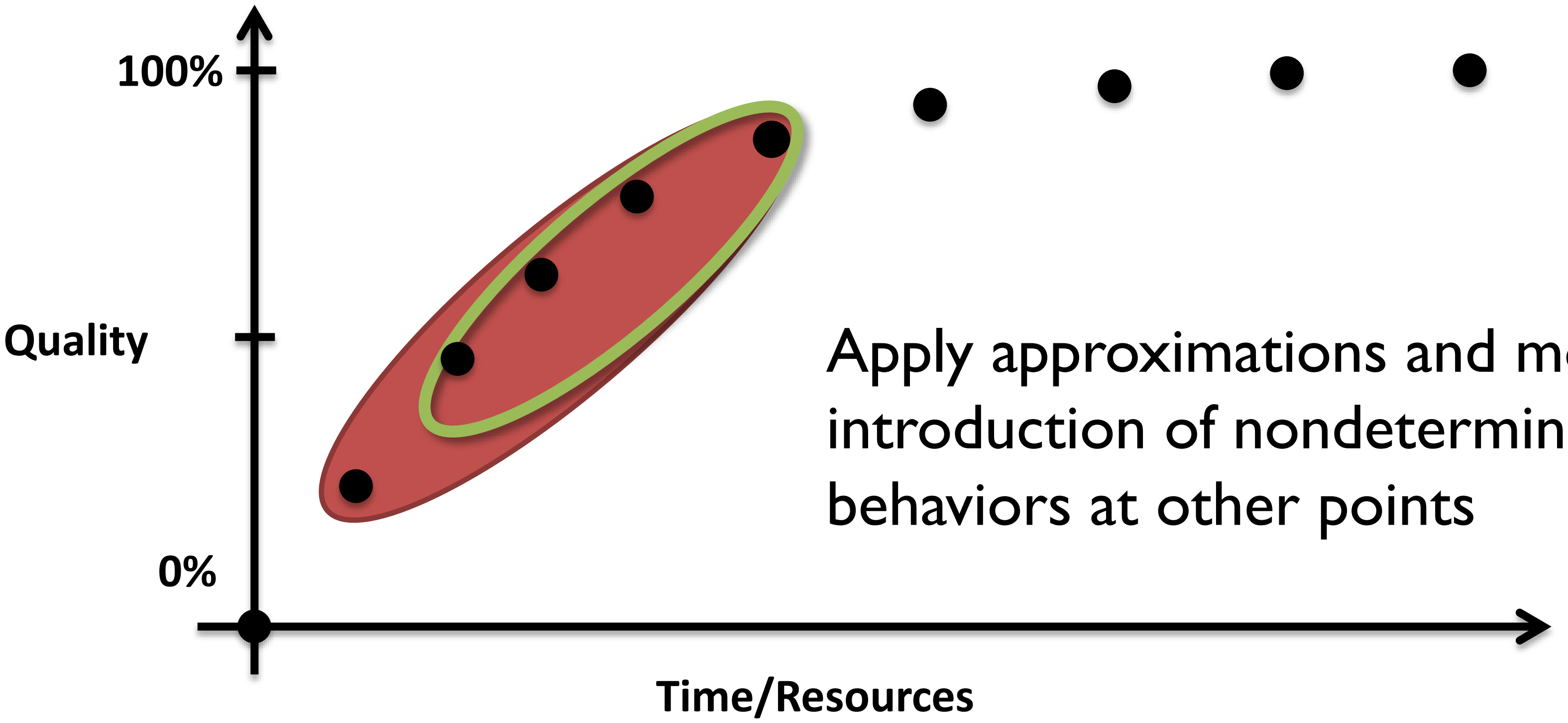
2. Reliability – probability program produces correct result

```
Pr(res == res') >= .99
```

3. Accuracy – worst-case difference in program result

```
assert_r |res - res'| <= .02 * res
```

Step #3: Approximate Programs



Apply approximations and model as introduction of nondeterministic behaviors at other points

Approximation Techniques

- Code Perforation

  Rinard, ICS '06; Baek et al., PLDI 10; Misailovic et al., ICSE '10; Sidiroglou et al., FSE '11; Misailovic et al., SAS '11; Zhu et al., POPL '12; Carbin et al. PEPM '13; Samadi et al. ASPLOS '14

- Function Substitution

  Hoffman et al., APLOS '11; Ansel et al., CGO '11; Zhu et al., POPL '12

- Approximate Memoization

  Alvarez et al., IEEE TOC '05; Chaudhuri et al., FSE '12; Samadi et al., ASPLOS '14

- Relaxed Synchronization (Lock Elision)

  Renganarayana et al., RACES '12; Rinard, HotPar '13; Misailovic, et al., RACES '12

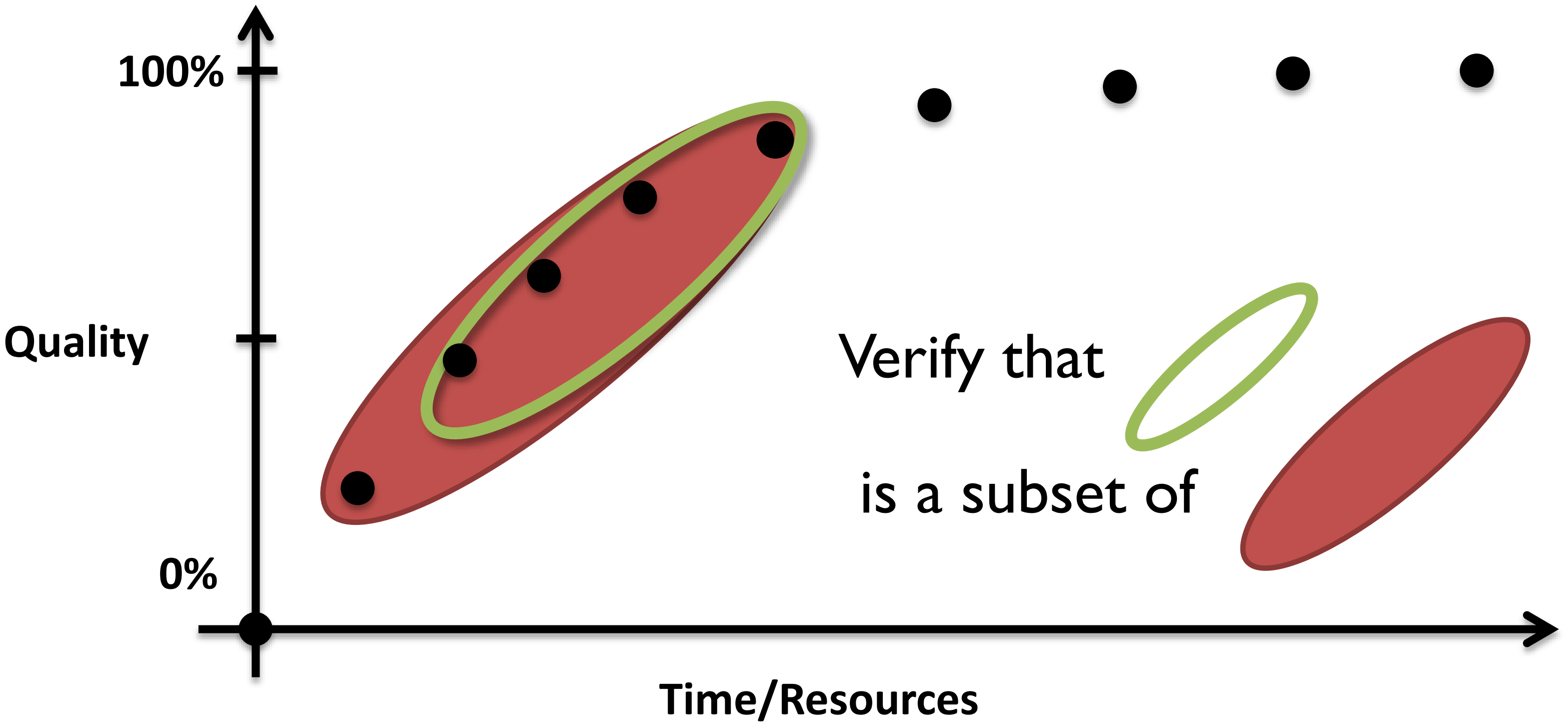
Approximate Hardware

Ernst et al, MICRO 2003; Samson et al., PLDI '11; PCMOS, Palem et al. 2005; Narayanan et al., DATE '10; Liu et al. ASPLOS '11; Venkataramani et al., MICRO '13

Key observation

original and approximate program share much of the same structure

Step #4: Verify that Approximation Preserves Acceptability



Standard Hoare Logic

"If *precondition P* is true before execution of *s*, then *postcondition Q* is true after"

$$\vdash \{0 < x\}\ y = x + 1\ \{0 < y\}$$

Standard Hoare Logic doesn't fully capture what we want

New Logics for Verifying Acceptability Properties

1. Safety – properties required to produce a valid result

```
assert (x != null) ∧ x == x’  ⊨ x’ != null
```

Relational Program Logic

2. Reliability – probability program produces correct result

```
Pr(res == res’) >= .99
```

Probabilistic Relational Program Logic

3. Accuracy – worst-case difference in program result

```
assert_r |res - res’| <= .02 * res
```

Relational Program Logic

Conclusion

- Many opportunities to approximate programs
  - Machine learning, Vision, Media Processing, Simulations
  - Both software and hardware techniques
  - Performance/Energy Usage improvements up to 7x
- Possible reason about approximate programs' behaviors
  - Step #1: Write standard program
  - Step #2: Specify acceptability properties (Safety, Reliability, Accuracy)
  - Step #3: Relax program's existing semantics
  - Step #4: Verify using novel program logics

Conclusion

- Many opportunities to approximate programs
  - Machine learning, Vision, Media Processing, Simulations
  - Both software and hardware techniques
  - Performance/Energy Usage improvements up to 7x
- Possible reason about approximate programs' behaviors
  - **Step #1: Write standard program**
  - Step #2: Specify acceptability properties (Safety, Reliability, Accuracy)
  - Step #3: Relax program's existing semantics
  - Step #4: Verify using novel program logics

Takeaway: Methodology for Programming General Uncertain Computations