

Approximate Computing on Approximate Data

Martin Rinard

MIT CSAIL

Loop Perforation

Problem

Program Takes Too Long To Run
(Or Consumes Too Much Energy)

Solution

[Misailovic et. al. ICSE 2010, Sidiroglou et. al. FSE 2011]

Profile program

Find loops that take most time

Perforate the loops

- Don't execute all loop iterations
- Instead, skip some iterations

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

Solution

[Misailovic et. al. ICSE 2010, Sidiroglou et. al. FSE 2011]

Profile program

Find loops that take most time

Perforate the loops

- Don't execute all loop iterations
- Instead, skip some iterations

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



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

Solution

[Misailovic et. al. ICSE 2010, Sidiroglou et. al. FSE 2011]

Profile program

Find loops that take most time

Perforate the loops

- Don't execute all loop iterations
- Instead, skip some iterations

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



```
for (i = 0; i < n; i += 4) { ... }
```

Common Reaction

OK, I agree program should run faster

But you can't do this

Because you will get wrong result!

Our Response

OK, I agree program should run faster

~~But you can't do this~~

~~Because you will get wrong result!~~

We absolutely can do this

You won't get the wrong result

You may get a **different** result





Key Points

Need to find right loops to perforate

Testing can identify viable loops

Converging loops

Distance metrics for heuristic searches

When you find right loops to perforate, can get

Significant time/energy savings (6X)

Acceptable inaccuracy

What This Talk Is About

Approximation to Reduce Energy Consumption

Approximate
Data



Phillip Stanley-Marbell

Approximate
Computation



Sasa Misailovic

OLED Displays Are Increasingly Popular

Smart Watches with OLED Displays:



LG



Samsung



Apple

Tablets with OLED Displays:



OLED Displays Consume Lots of Power

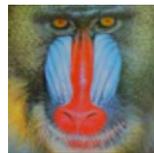
ARM Cortex-M0 Processor:	0.3-18 mWatts
Intel Atom 21000	6 Watts
120x120 OLED Display:	8.5 Watts



(39.63% ↓)



(18.70% ↓)



(22.75% ↓)



(36.30% ↓)



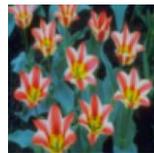
(54.03% ↓)



(38.53% ↓)



(18.38% ↓)



(22.20% ↓)



(35.24% ↓)



(52.59% ↓)



(42.64% ↓)



(20.05% ↓)



(24.08% ↓)



(39.12% ↓)



(57.54% ↓)



Original 7328.02
mW



Shape transform 1
5927.78 mW
(19.1% savings)



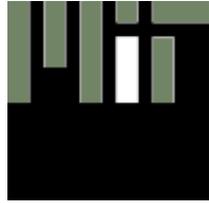
Shape transform 2
8514.26 mW
(-16.2% savings)



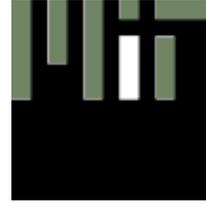
Shape transform 3
6753.95 mW
(7.8% savings)



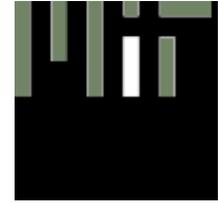
Original
1647.3 mW
(0.0% savings)



Color Transform
1319.48 mW
(19.9% savings)



Color + 0.88x area
1123.46 mW
(31.8% savings)



Color + 0.72x area
930.72 mW
(43.5% savings)



Original
7716.36 mW
(0.0% savings)



Color Transform
7616.04 mW
(1.3% savings)

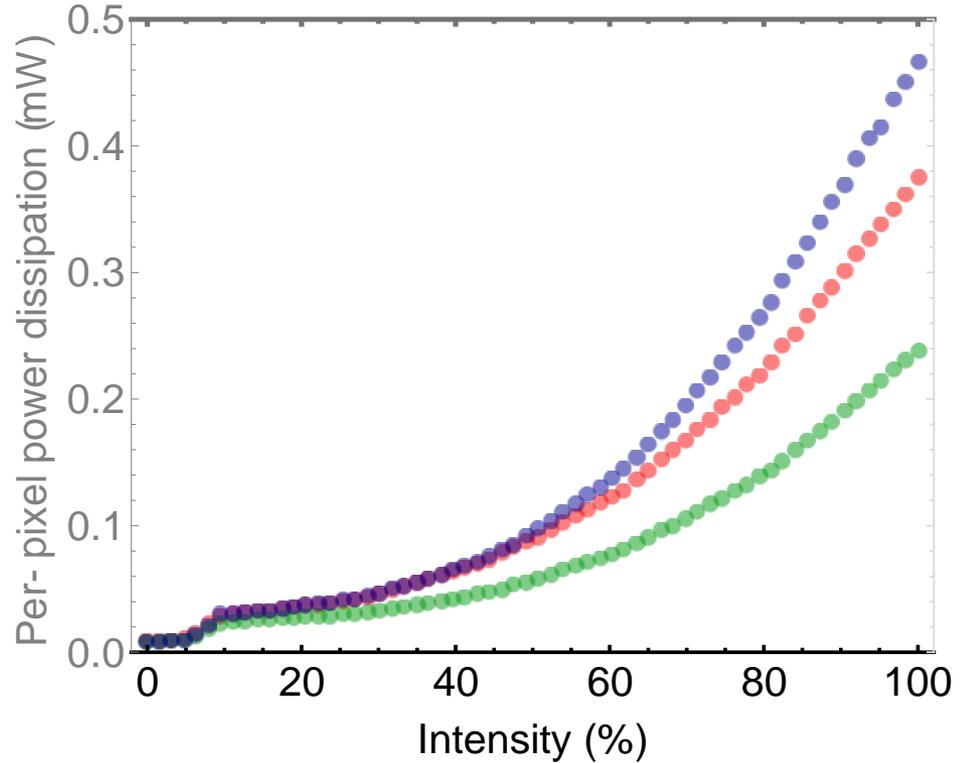


Color + 0.88x area
7924.70 mW
(-2.7% savings)



Color + 1.4x area
6358.28 mW
(17.6% savings)

On OLED displays, **blue** uses almost twice power of **green**



Numbers From On-Board Current Measurements

Wrote device drivers to measure current driving display
Characterized power for all colors

Color Transforms

Characterize Tradeoff Curve:

Start with an image and a distance in an underlying color space
Find a new image within that distance that minimizes power

Power model used in minimization formulation

(calibrated with measurement data)

$$p(\mathbf{v}) = \sum_{c \in \{r, g, b\}} \sum_{i=1}^N \frac{1}{2} \alpha_c \mathbf{v}_c[i]^2 + \beta_c \mathbf{v}_c[i]$$

Closed-form formula for color transformation

(power minimization under a distance constraint)

$$u_c[i] = \frac{\lambda \mathbf{v}_c[i] - \beta_c}{\lambda + \alpha_c}$$

Where

- $u_c[i]$ Transformed pixel value for i th pixel on c th channel
- $v_c[i]$ Original value of i th pixel on c th channel
- α Parameter from power model
- β Parameter from power model
- λ Power-vs-distance tradeoff parameter

Key Question

Relationship between λ and human perception

Use Amazon Mechanical Turk

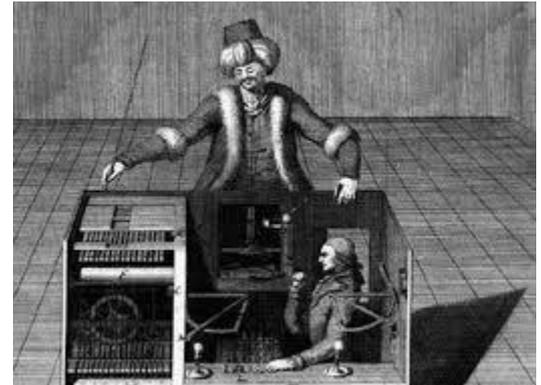
Discretize λ

Generate λ variants of pictures

Mechanical Turk workers

rate pictures (0-3)

370 People, 2636 picture ratings

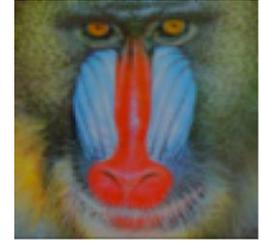
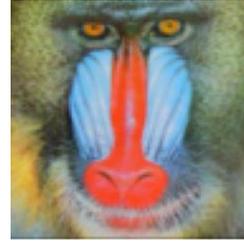


Result: f

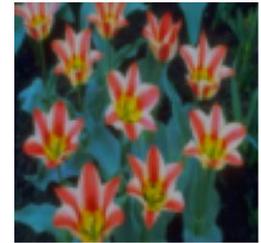
perceived goodness = $f(\lambda)$

choose λ such that $f(\lambda) = 2$, use that λ

For $\lambda = 3.28$,
CIELAB space,
get 33%-50%
power savings



(39.63% ↓)



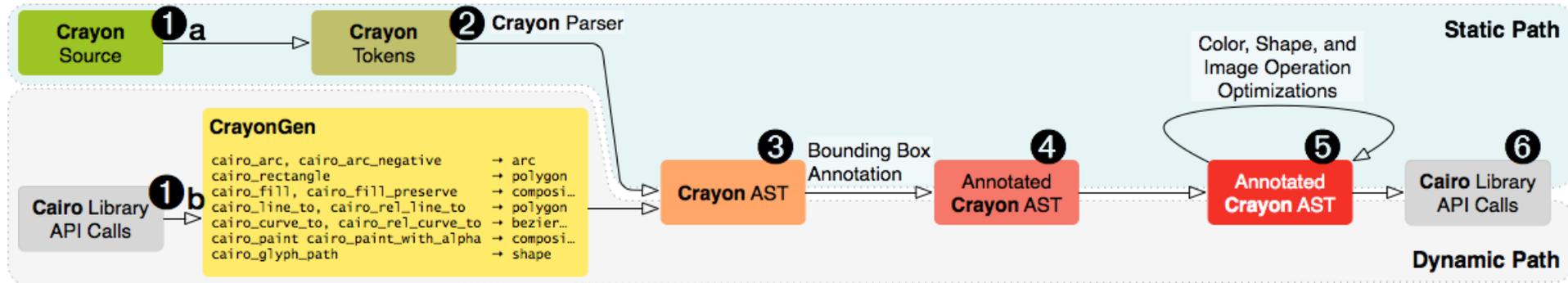
(38.53% ↓)



(42.64% ↓)

Implemented These Ideas in Crayon System

Crayon Software Architecture



Cairo is a standard, widely-used graphics API

Firefox, Graphviz, Poppler, ...

CrayonGen intercepts Cairo calls, generates Crayon IR

Capture layering information

Shape optimization for drawing operations

Closed-form color transformation

Crayon implementation is about 12K LOC

Cairo-specific code is about 232 LOC

Crayon via Cairo Example

```
int
main (int argc, char * argv[])
{
    cairo_surface_t * surface;
    cairo_t * cr;

    surface = cairo_image_surface_create(CAIRO_FORMAT_RGB24,
        96, 96);
    cr = cairo_create(surface);

    cairo_set_line_width(cr, DRAWING_LINE_WIDTH);
    cairo_scale(cr, 96, 96);

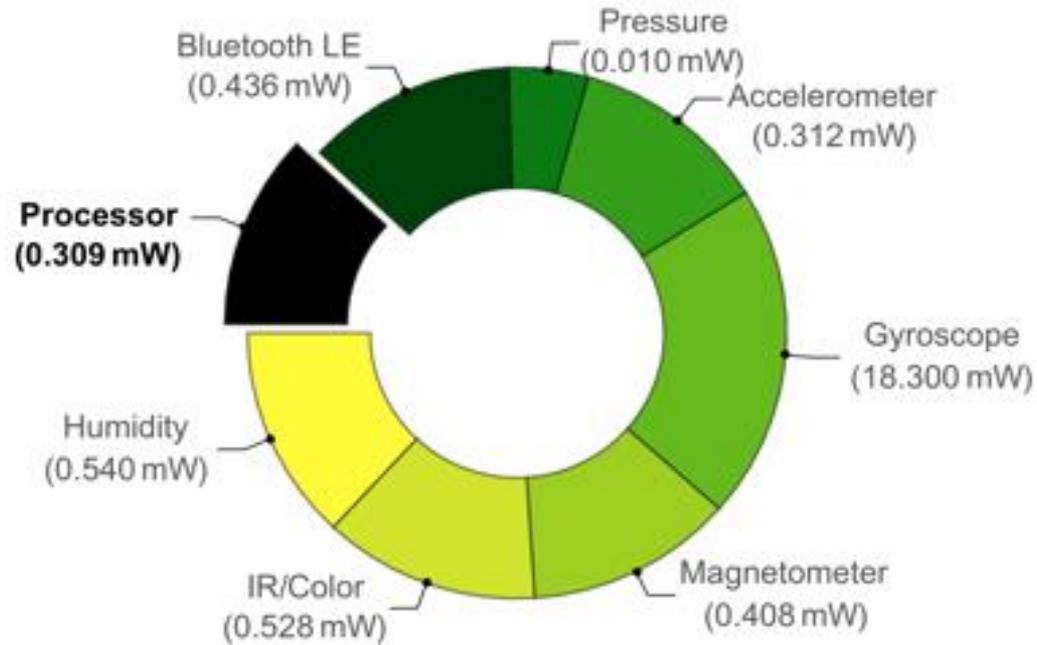
    draw_mit_logo(cr,
        /* mtidot_red */ MIT_LOGO_MITLIGHTGRAY_R,
        /* mtidot_green */ MIT_LOGO_MITLIGHTGRAY_G,
        /* mtidot_blue */ MIT_LOGO_MITLIGHTGRAY_B,
        /* istem_red */ 1.0,
        /* istem_green */ 1.0,
        /* istem_blue */ 1.0);

    cairo_surface_flush(surface);
    cairo_surface_write_to_png (surface, "mitlogo.png");
    cairo_destroy(cr);
    cairo_surface_destroy(surface);

    return 0;
}
```

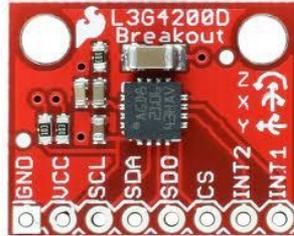
```
void
draw_mit_logo(cairo_t *cr, double mtidot_red, double
    mtidot_green, double mtidot_blue, double
    istem_red, double istem_green, double
    istem_blue)
{
    /* M: */
    cairo_set_source_rgb(cr, mtidot_red, mtidot_green,
        mtidot_blue);
    cairo_rectangle(cr,
        /* x */ 0.0,
        /* y */ 0.0,
        /* width */ MIT_LOGO_BAR_WIDTH,
        /* height */ MIT_LOGO_LETTER_HEIGHT);
    cairo_fill(cr);
    cairo_rectangle(cr,
        /* x */ MIT_LOGO_BAR_WIDTH+MIT_LOGO_BAR_SPACING,
        /* y */ 0.0,
        /* width */ MIT_LOGO_BAR_WIDTH,
        /* height */ 3*MIT_LOGO_BAR_WIDTH);
    cairo_fill(cr);
    cairo_rectangle(cr,
        /* x */ 2*(MIT_LOGO_BAR_WIDTH+MIT_LOGO_BAR_SPACING),
        /* y */ 0.0,
        /* width */ MIT_LOGO_BAR_WIDTH,
        /* height */ MIT_LOGO_LETTER_HEIGHT);
    cairo_fill(cr);
    ...
}
```

Sensor Power Consumption



Reduced Power Sensor Operation

L3G4200D
Gyro Sensor



L3G4200D Manual: Run sensor at 1.8V-3.6V

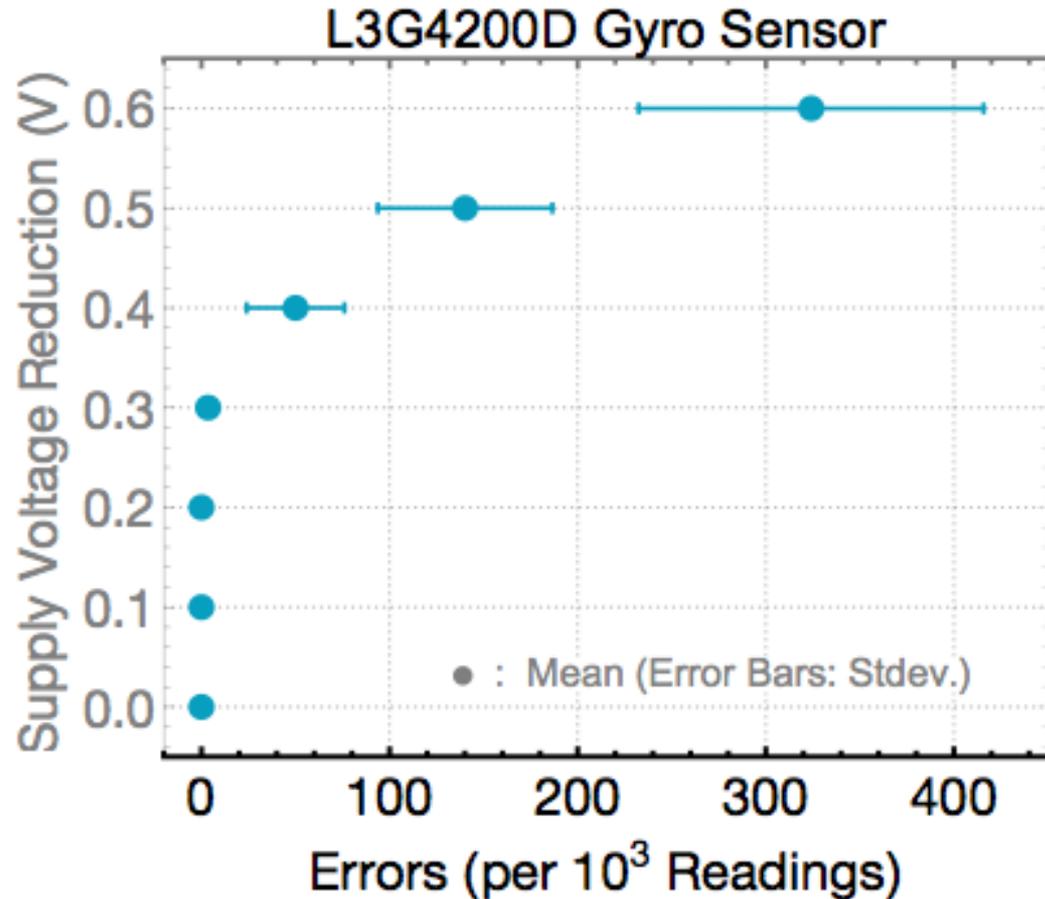
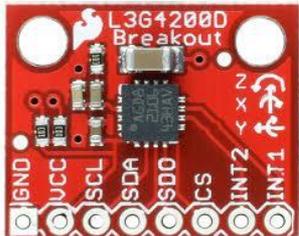
Us: Why would we do this?

We can save lots of power if reduce V!

$$\text{Power} = C_{L3G4200D} V^2$$

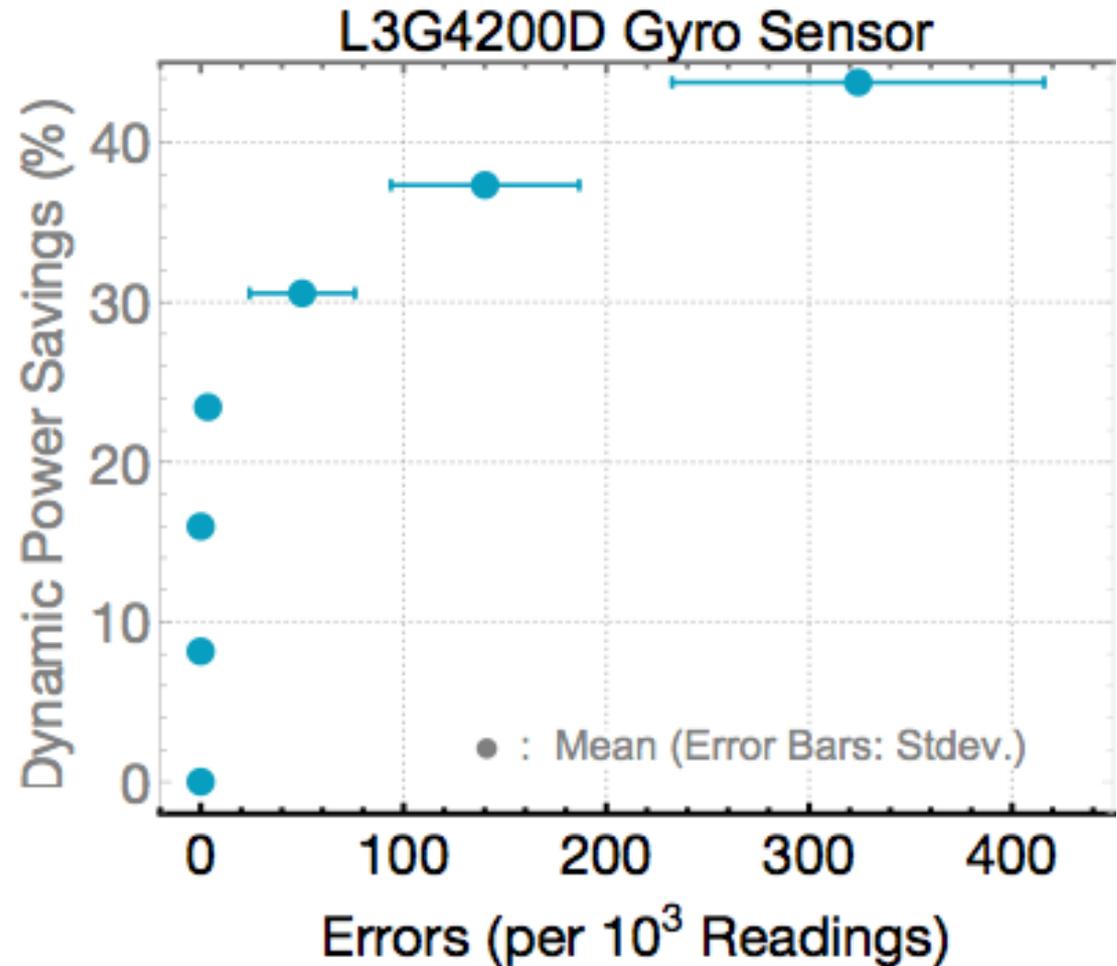
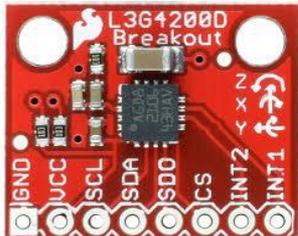
Sensor Still Works (Mostly)!

L3G4200D
Gyro Sensor

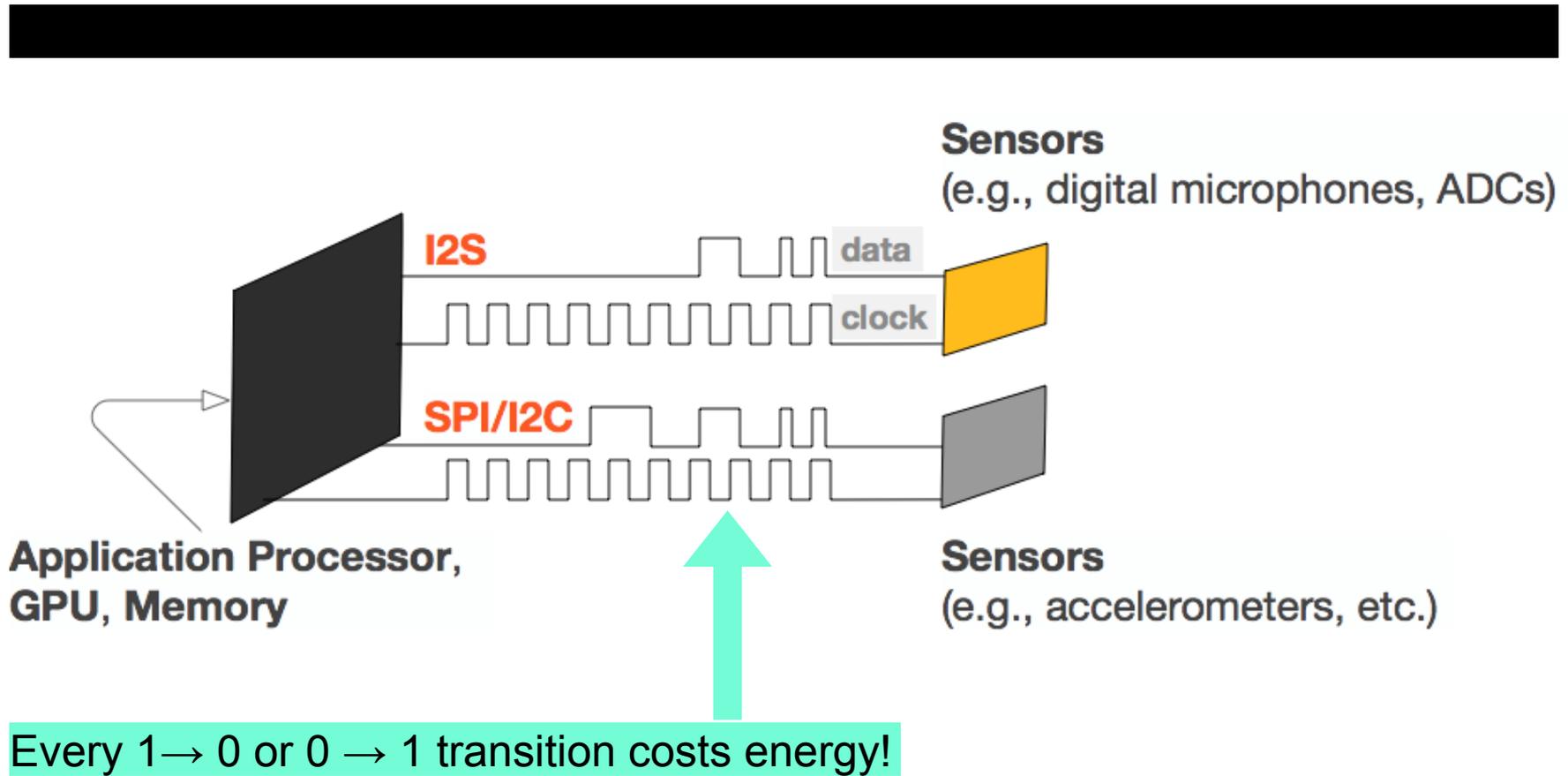


And We Save Power

L3G4200D
Gyro Sensor

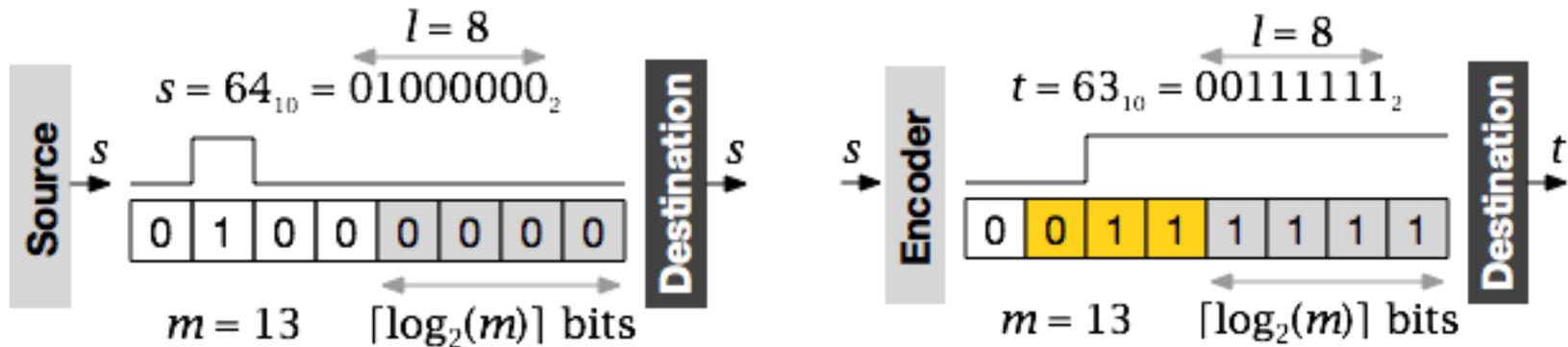


Moving Sensor Data To Processor Costs Energy



Reducing Transitions

Let's say we decide to tolerate m error
Why not just make $\log_2(m)$ bits all 0 or all 1?



Encoding Algorithm Overview

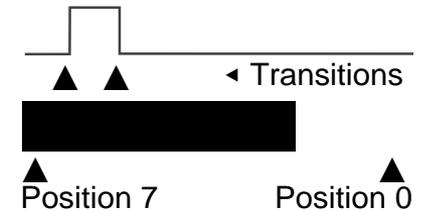
Problem: Given input $s=64$, find an encoded value t , such that $|s-t| \leq m$, for $m = 16$, and for which $\#_0(t) < \#_0(s)$

Phase 1: Identify transition positions and cumulative run counts

Moving left from **LSB**, transition position set is $\{5, 6\}$

Run of **0s** at bit position 5 will have contribution 63 if flipped

Run of **1s** at bit position 6 will have contribution 64 if flipped



Phase 2: Find runs at that can be flipped in opposite directions

① Start from first transition seen, **moving right from MSB**

② Can complement removing transition at position 6 (with contribution -64) with lower-order run of 0s (contribution +63)

– Resulting deviation is 1

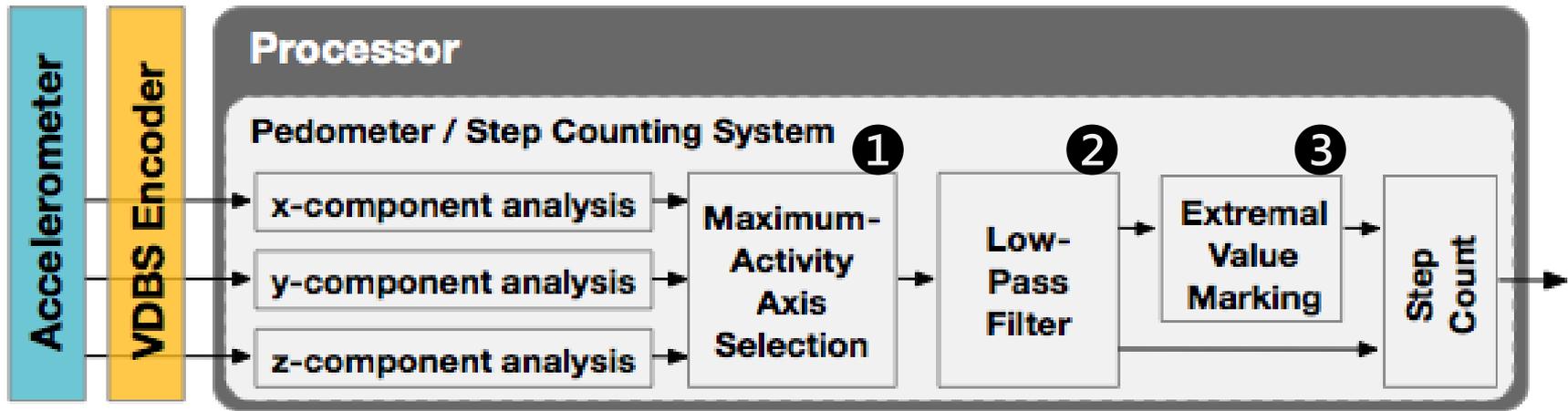
– Number of transitions is reduced from 2 to 1



Great, But How Does it Affect Real End-to-End Applications?

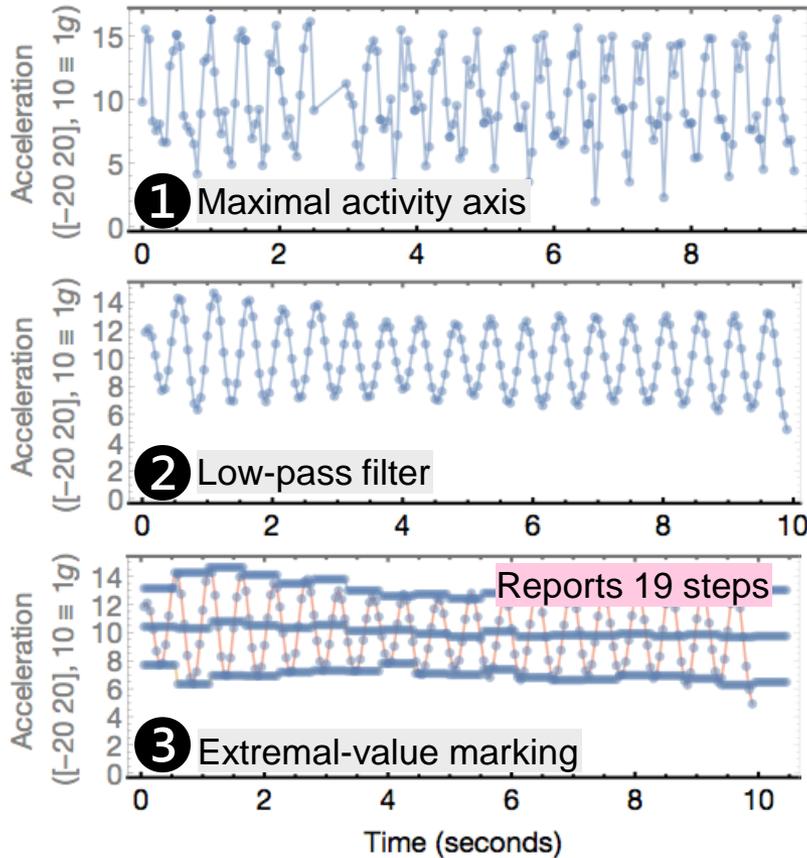
Tolerable Deviation	Image A	OCR Text	Transition Reduction	Image B	OCR Text	Transition Reduction
0%		"EXIT"	0%↓		"EXIT"	0%↓
4%		"EXIT"	58%↓		"EXIT"	49%↓
10%		"EXIT"	72%↓		"LTXIT"	61%↓
20%		"EXIT"	75%↓		""	73%↓

Great, But How Does it Affect Real End-to-End Applications?

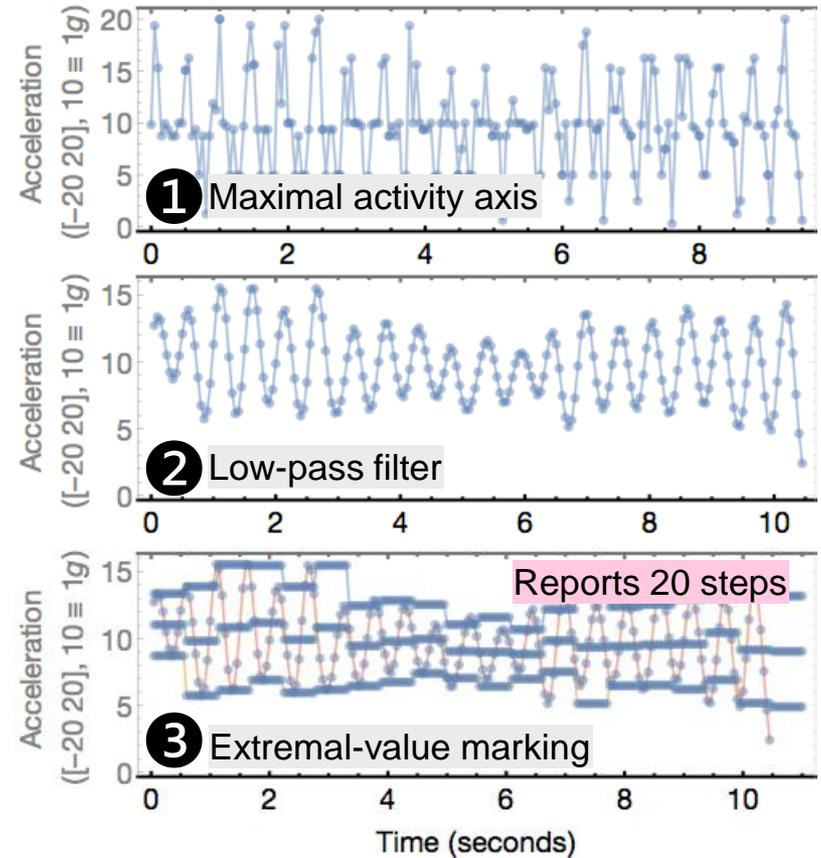


Great, But How Does it Affect Real End-to-End Applications?

Without Transition Reduction



With Transitions Reduced By 54%



Approximate Computation

Chisel: Reliability- and Accuracy-Aware Optimization of
Approximate Computational Kernels

Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, Martin Rinard
(OOPSLA 2014 Best Paper Award)

Image Scaling

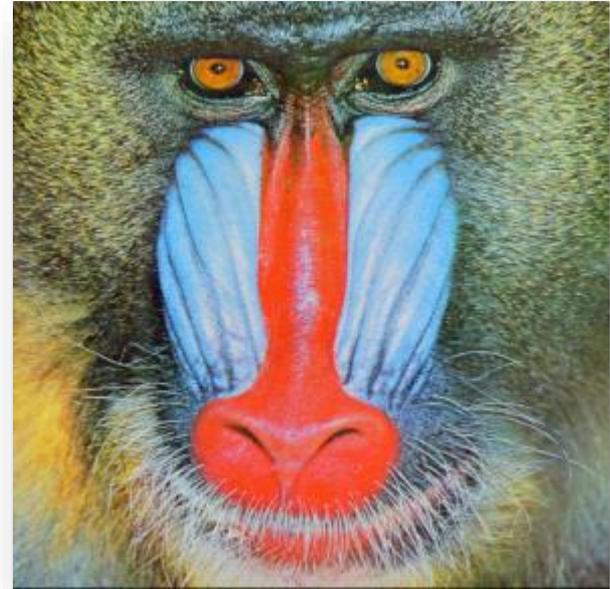
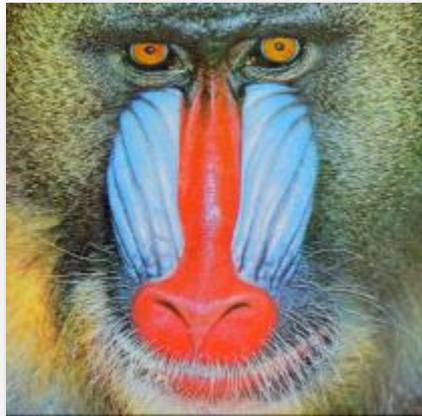
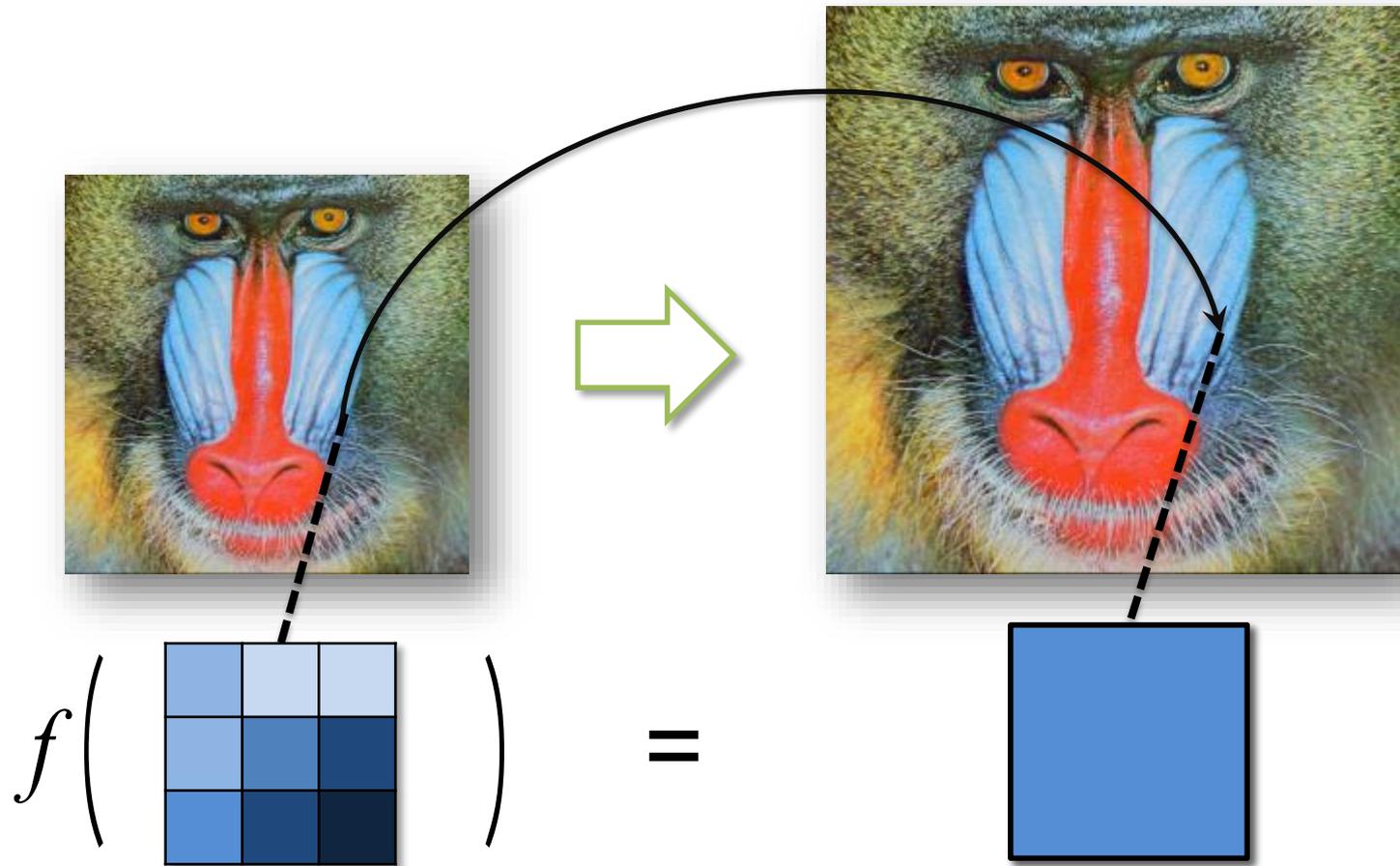


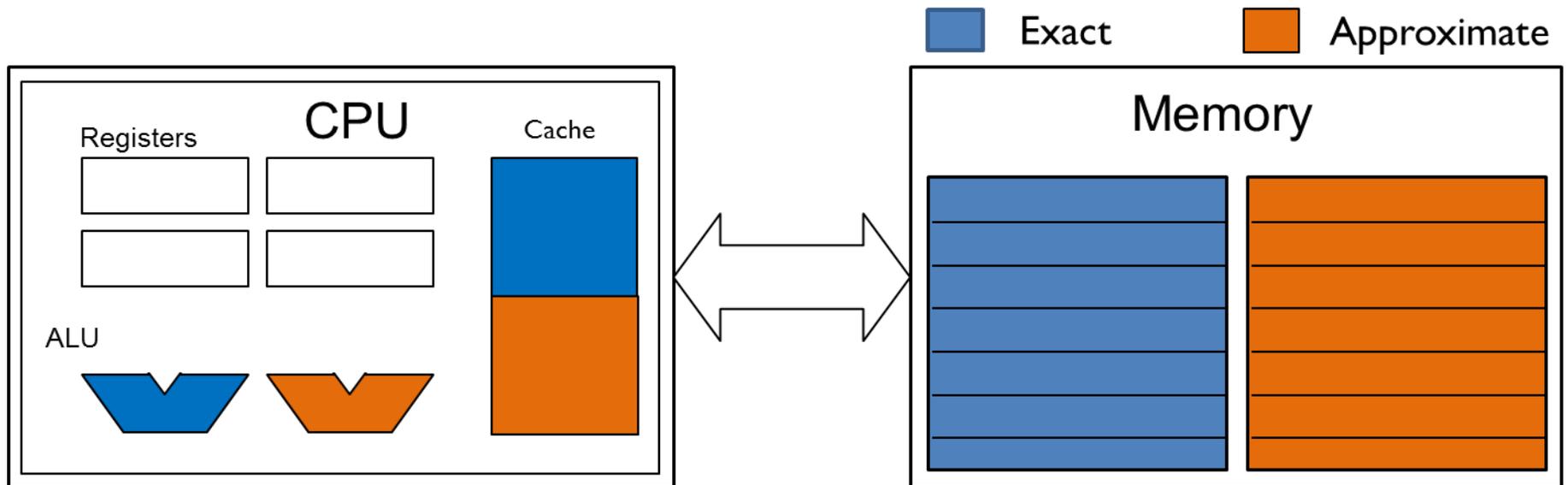
Image Scaling: Interpolation Function



Interpolation Function

```
int interpolation(int dst_x, int dst_y, int src[][])  
{  
    int x = src_location_x(dst_x, src),  
        y = src_location_y(dst_y, src);  
  
    int up    = src[y - 1][x],  
        down  = src[y + 1][x],  
        left  = src[y][x - 1],  
        right = src[y][x + 1];  
  
    int val = up + down + left + right;  
  
    return 0.25 * val;  
}
```

Approximate Hardware Model

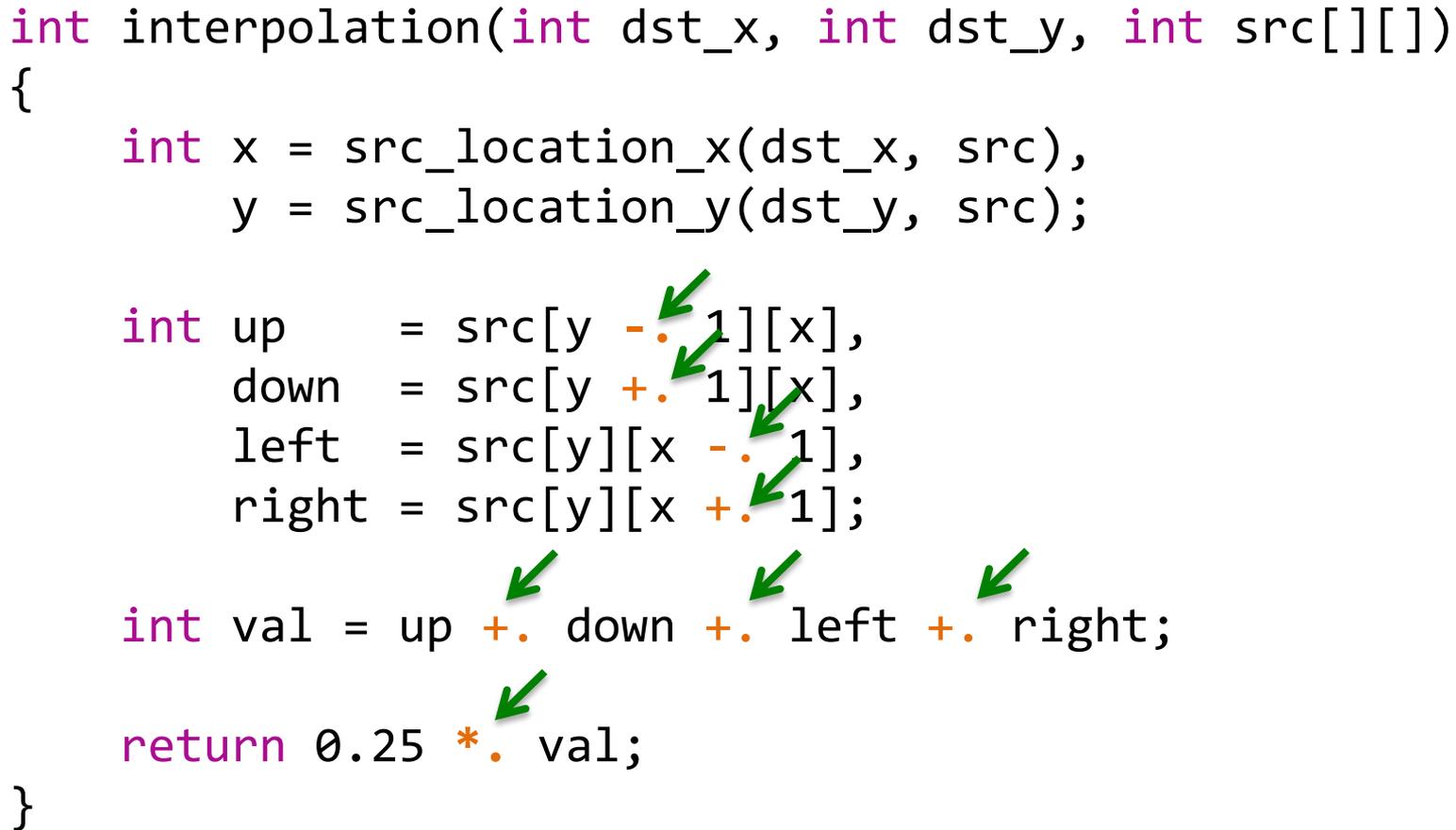


Approximate Units (ALUs and Main/Cache Memories)

- May produce incorrect results
- Hardware specification contains savings and reliability

Run Function on Approximate Hardware

```
int interpolation(int dst_x, int dst_y, int src[][])  
{  
    int x = src_location_x(dst_x, src),  
        y = src_location_y(dst_y, src);  
  
    int up    = src[y - 1][x],  
        down  = src[y + 1][x],  
        left  = src[y][x - 1],  
        right = src[y][x + 1];  
  
    int val = up + down + left + right;  
  
    return 0.25 * val;  
}
```

The diagram consists of several green arrows pointing to specific arithmetic operators in the code. One arrow points to the minus sign in 'y - 1', another to the plus sign in 'y + 1', a third to the minus sign in 'x - 1', and a fourth to the plus sign in 'x + 1'. In the next line, three arrows point to the plus signs between 'up + down', 'down + left', and 'left + right'. Finally, an arrow points to the asterisk in '0.25 * val'.

Run Function on Approximate Hardware

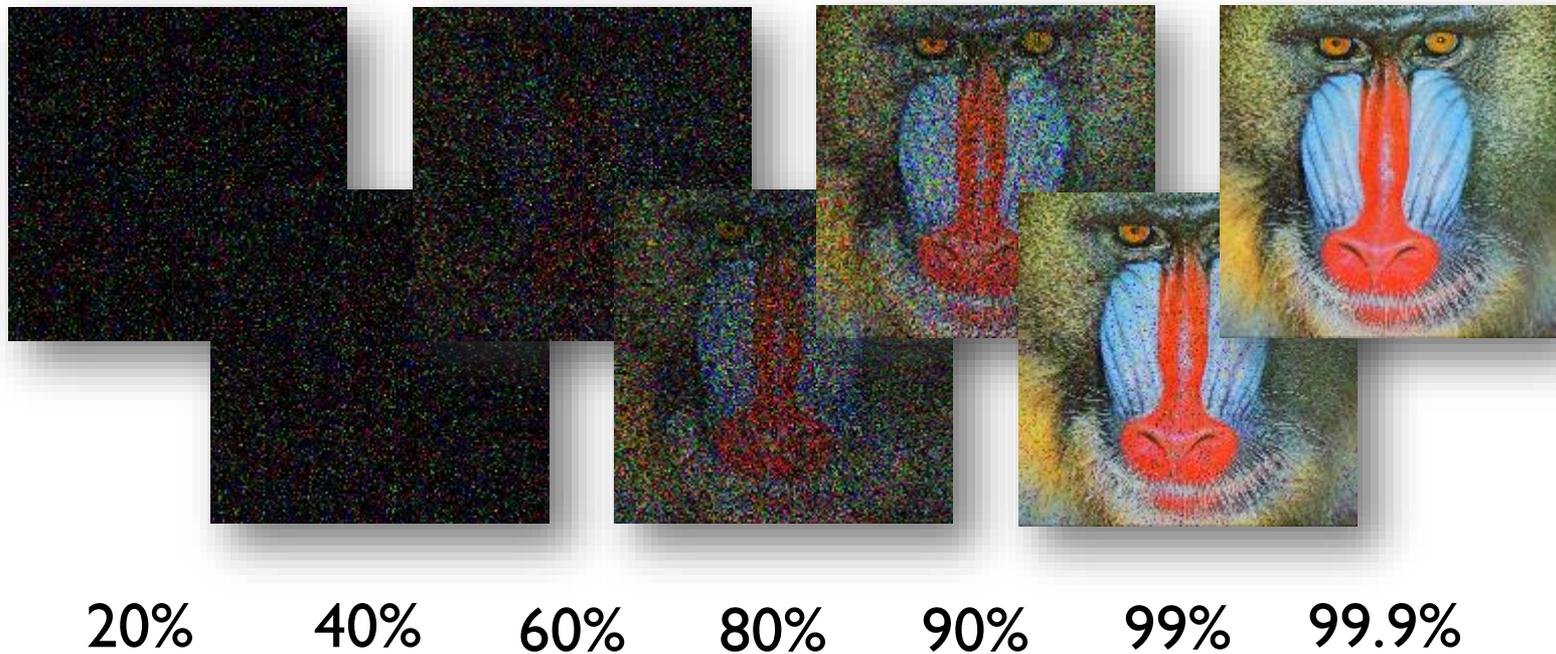
```
int interpolation(int@ dst_x, int@ dst_y, int@ src[][])
{
    int@ x = src_location_x(dst_x, src),
        y = src_location_y(dst_y, src);

    int@ up    = src[y -. 1][x],
        down  = src[y +. 1][x],
        left   = src[y][x -. 1],
        right  = src[y][x +. 1];

    int@ val = up +. down +. left +. right;

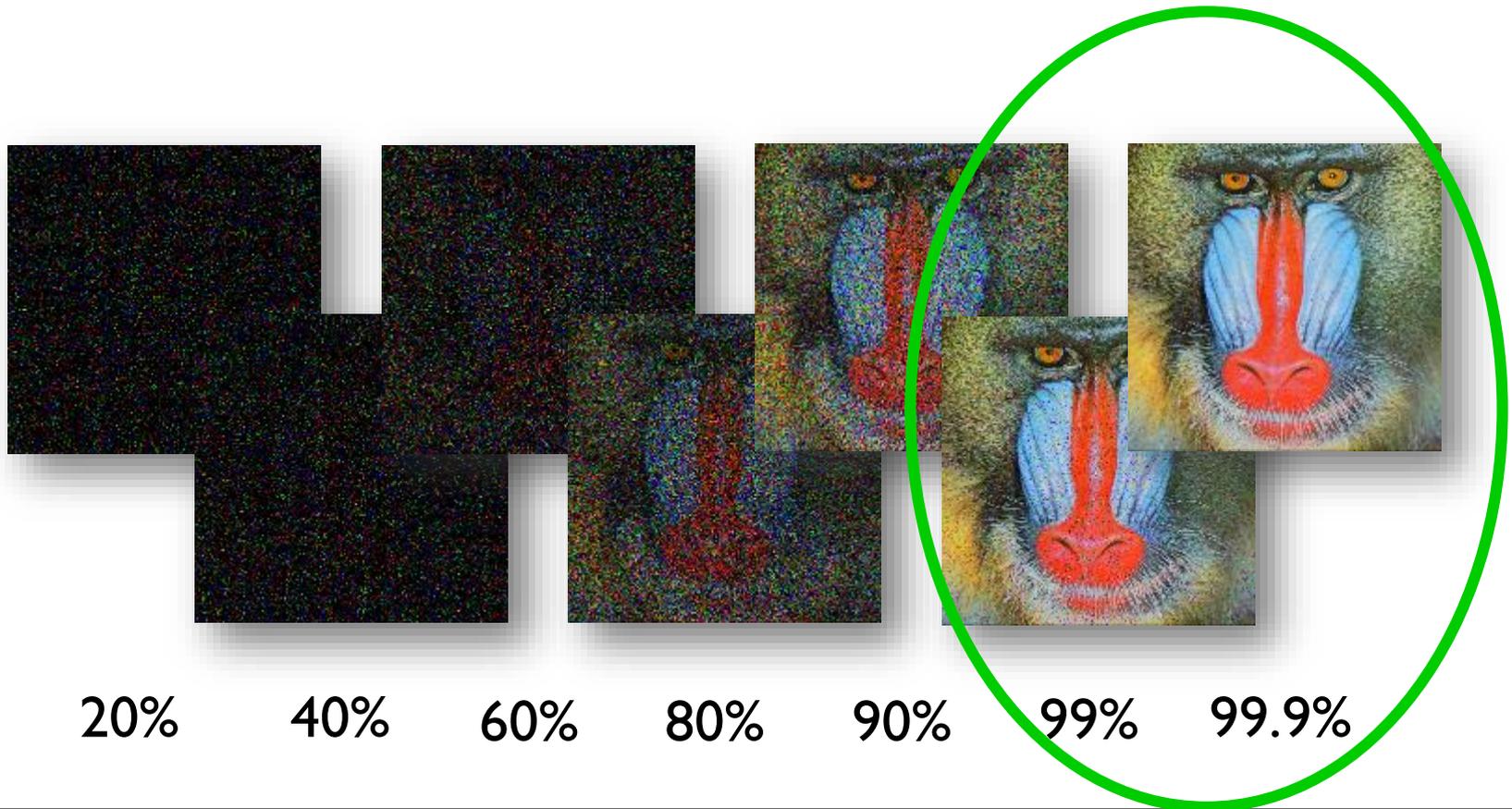
    return 0.25 *. val;
}
```

Function and Program Accuracy



Probability p with which interpolation kernel produces a correct pixel

Function's and Program's Accuracy



Produce a correct pixel with probability > 0.99

Produce a correct pixel with probability at least **0.99**

```
int interpolation(int dst_x, int dst_y, int src[][])  
{  
    int x = src_location_x(dst_x, src),  
        y = src_location_y(dst_y, src);  
  
    int up    = src[y - 1][x],  
        down  = src[y + 1][x],  
        left  = src[y][x - 1],  
        right = src[y][x + 1];  
  
    int val = up +. down +. left +. right;  
  
    return 0.25 *. val;  
}
```

Produce a correct pixel with probability at least **0.99**

```
int interpolation(int dst_x, int dst_y, int@ src[][])  
{  
    int x = src_location_x(dst_x, src),  
        y = src_location_y(dst_y, src);  
  
    int up    = src[y - 1][x],  
        down  = src[y + 1][x],  
        left  = src[y][x - 1],  
        right = src[y][x + 1];  
  
    int@ val = up + down + left + right;  
  
    return 0.25 * val;  
}
```

Produce a correct pixel with probability at least **0.99**

```
int interpolation(int dst_x, int dst_y, int src[][][])  
{
```

How to find approximate function
with maximum energy savings?

```
int@ val = up + down + left + right;
```

```
return 0.25 *. val;
```

```
}
```

Software Developer

Program

Accuracy
Specification

Hardware Architect

Approximate
Hardware
Specification

Chisel

Automates placement of

- approximate arithmetic operations
- variables in approximate memory

Approximate function:

Maximizes energy savings
Satisfies accuracy specifications

Accuracy Specification

Reliability

Function computes result correctly with probability > 0.99

Absolute Error

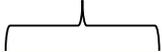
Absolute error of function's result < 2.0

Reliability and Absolute Error

Absolute error of function's result < 2.0 with probability > 0.99

Reliability Specification

Reliability
degradation


`int < 0.99 * R($\Delta x = 0, \Delta y = 0, \Delta src = 0$) >`
`interpolation(int x, int y, int src[][]);`

The function computes result correctly
with probability at least **0.99**

Reliability Specification

Reliability
degradation

Parameter
Reliability

`int < 0.99 * R($\Delta x = 0, \Delta y = 0, \Delta src = 0$) >`
`interpolation(int x, int y, int src[][]);`

The diagram shows a code snippet with two brackets above it. The first bracket is under the number 0.99 and is labeled 'Reliability degradation'. The second bracket is under the function call `R($\Delta x = 0, \Delta y = 0, \Delta src = 0$)` and is labeled 'Parameter Reliability'. The function call itself is highlighted in yellow.

Probability that the parameters have
correct values before function starts executing
(facilitates function composition)

Reliability Specification

Reliability degradation Parameter Reliability

`int < 0.99 * R($\Delta x = 0, \Delta y = 0, \Delta src = 0$) >`
`interpolation(int x, int y, int src[][]);`

- **Reliability factor:** $R(\Delta v_1 \leq d_1, \dots, \Delta v_n \leq d_n)$

$$\Delta v \equiv |v_{exact} - v_{approx}|$$

Numerical bound

Function Optimization Problem

Find Function Configuration q :

max EnergySavings (q)

s. t. Reliability (q) \geq ReliabilityBound

AbsoluteError (q) \leq ErrorBound



Analysis of the Function



Specifications

Function Configuration

Binary vector $\mathbf{q} = (q_1, q_2, \dots, q_n)$

Variable Declarations:

- q_i - if 1, variable is stored in approximate memory
if 0, variable is stored in exact memory

Arithmetic Operations:

- q_i - if 1, the operation is approximate,
if 0, the operation is exact

Function Configuration

```
int interpolation(int dst_x, int dst_y, int src[][])  
{  
    int x = src_location_x(dst_x, src);  
    int y = src_location_y(dst_y, src);  
  
    int up    = src[y - 1][x];  
    int down  = src[y + 1][x];  
    int left  = src[y][x - 1];  
    int right = src[y][x + 1];  
  
    int val = up + down + left + right;  
  
    return 0.25 * val;  
}
```

Function Configuration

```
int interpolation(intqdstx dst_x, intqdsty dst_y, intqsrc src[][])  
{  
    intqx x = src_location_x(dst_x, src);  
    intqy y = src_location_y(dst_y, src);  
  
    intqup up = src[y - 1][x];  
    intqdown down = src[y + 1][x];  
    intqleft left = src[y][x - 1];  
    intqright right = src[y][x + 1];  
  
    intqval val = up + down + left + right;  
    return 0.25 * val;  
}
```

Each assignment of vector q denotes
a different approximate function

```
int interpolation(int $q_{dstx}$  dst_x, int $q_{dsty}$  dst_y, int $q_{src}$  src[][])  
{  
    int $q_x$  x = src_location_x(dst_x, src);  
    int $q_y$  y = src_location_y(dst_y, src);  
  
    int $q_{up}$  up = src[y -  $q_7$  1][x];  
    int $q_{down}$  down = src[y +  $q_6$  1][x];  
    int $q_{left}$  left = src[y][x -  $q_5$  1];  
    int $q_{right}$  right = src[y][x +  $q_4$  1];  
  
    int $q_{val}$  val = up +  $q_1$  down +  $q_2$  left +  $q_3$  right;  
  
    return 0.25 *  $q_0$  val;  
}
```

Reliability Analysis

- Efficiently represent reliability of **all approximate** versions of the function
- Construct constraints that describe those approximate functions **that satisfy specification**

Reliability Analysis

Approximate hardware specification:

- Reliability of arithmetic operations: $r_{op} \in (0, 1]$
- Reliability of memory reads and writes: $r_{rd}, r_{wr} \in (0, 1]$

```
operator (*) = 0.9999;  
memory approx {rd = 0.99998, wr = 0.99999};
```

Analysis:

- Sound **static analysis**, operates backward
- Constructs **symbolic expressions** that characterize reliability of traces

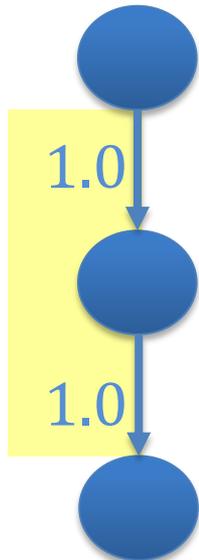
Reliability Analysis

Statement

`return val * 0.25;`

Exact Statement

*val and * exact*



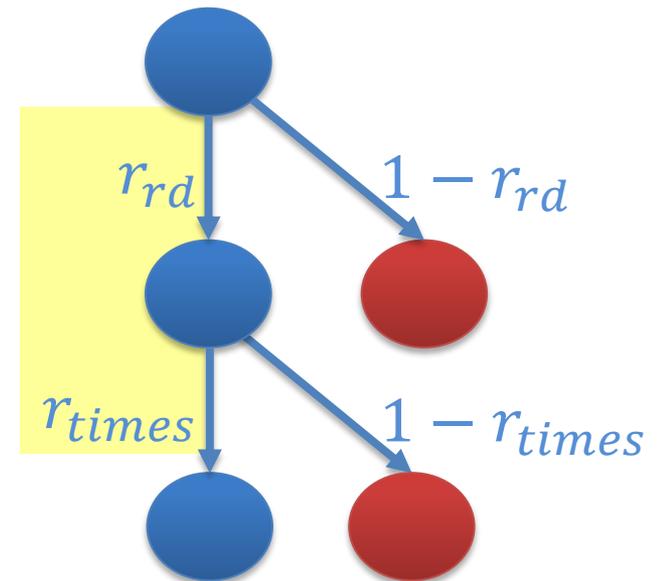
Read val

Multiply

Return result

Approximate Statement

*val and * approximate*



Reliability Analysis

Statement `return val * 0.25;`

Exact Statement

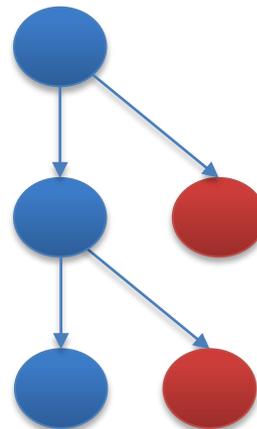
Approximate Statements

*val and *
exact*



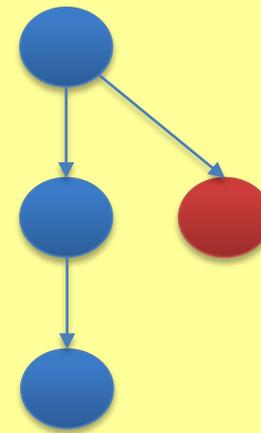
1.0

*val and *
approximate*



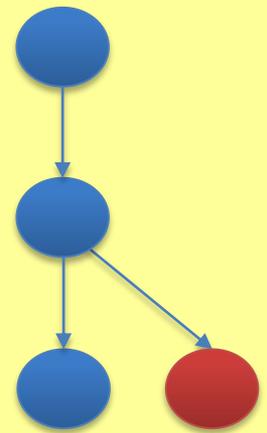
$r_{rd} \cdot r_{times}$

*val
approximate*



r_{rd}

**
approximate*



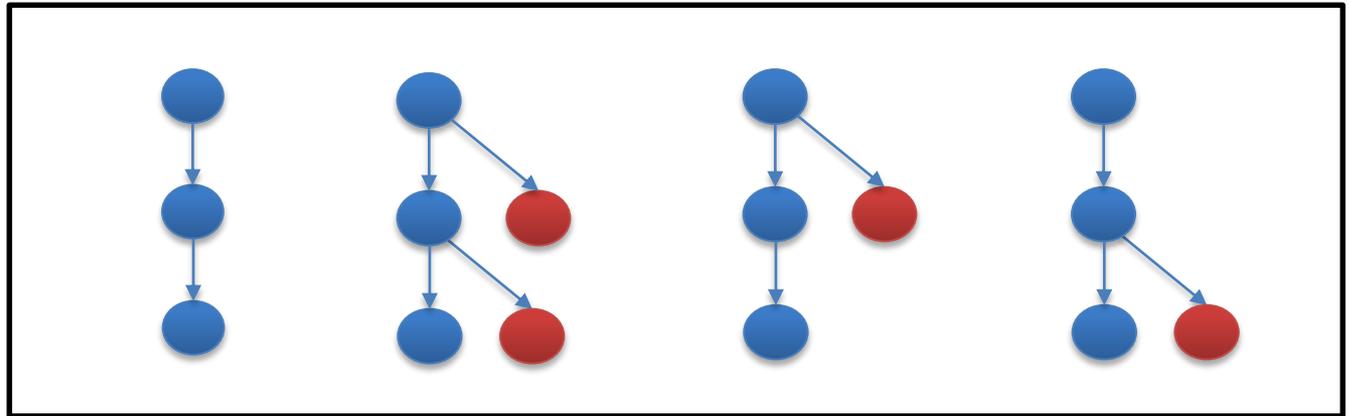
r_{times}

**Statement
reliability**

Reliability Analysis

Statement

```
return val * 0.25;
```



Encode approximation choice:

- Variable declaration: `int q_{val} val;`
- Multiplication: `val * q_* 0.25;`

Reliability Analysis

Statement `return val * 0.25;`

**Reliability
Expression**

$$(r_{rd})^{q_{val}} \cdot (r_{times})^{q_*} \cdot R(\Delta val = 0)$$

Encode approximation choice:

- Variable declaration: `int q_{val} val;`
- Multiplication: `val * q_* 0.25;`

Reliability Analysis

Statement

```
return val * 0.25;
```

**Reliability
Expression**

$$(r_{rd})^{q_{val}} \cdot (r_{times})^{q_*} \cdot R(\Delta val = 0)$$

Reliability of reading `val` from
either exact or approximate memory:

$$(r_{rd})^0 = 1.0$$

$$(r_{rd})^1 = r_{rd}$$

Reliability Analysis

Statement

```
return val * 0.25;
```

**Reliability
Expression**

$$(r_{rd})^{q_{val}} \cdot (r_{times})^{q_*} \cdot R(\Delta val = 0)$$

Reliability of either exact or
approximate multiplication

Reliability Analysis

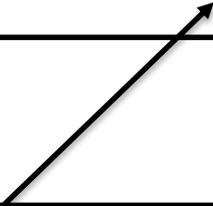
Statement

```
return val * 0.25;
```

**Reliability
Expression**

$$(r_{rd})^{q_{val}} \cdot (r_{times})^{q_*} \cdot \mathbf{R}(\Delta\mathbf{val} = \mathbf{0})$$

Probability that previous statements
computed **val** correctly



Interpolation Function

```
int interpolation(intqdstx dst_x, intqdsty dst_y, intqsrc src [][])  
{
```

$$(r_{rd})^{q_{val}} \cdot (r_{times})^{q_*} \cdot \mathbf{R}(\Delta val = 0)$$

```
return val * q* 0.25;
```

```
}
```

Interpolation Function

```
int interpolation(intqdstx dst_x, intqdsty dst_y, intqsrc src [][])  
{
```

$$(r_{rd})^{q_{val}} \cdot (r_{times})^{q_*} \cdot (r_{plus})^{q_1+q_2+q_3} \cdot (r_{rd})^{q_{up}+q_{down}+q_{left}+q_{right}} \cdot \mathbf{R(\Delta_{up} = 0, \Delta_{down} = 0, \Delta_{left} = 0, \Delta_{right} = 0)}$$

```
intqval val = up +q1 down +q2 left +q3 right;
```

$$(r_{rd})^{q_{val}} \cdot (r_{times})^{q_*} \cdot \mathbf{R(\Delta_{val} = 0)}$$

```
return val *q* 0.25;
```

```
}
```

Reliability Expression

Function's Reliability Expression:

$$\underbrace{r_1^{q_1} \cdot r_2^{q_2} \cdot \dots \cdot r_n^{q_n}}_{\substack{\text{Probability operations} \\ \text{executed reliably} \\ \text{(for all approximate versions} \\ \text{of the function)}}} \cdot \underbrace{R \left(P_{param} \right)}_{\substack{\text{Probability parameters} \\ \text{have correct values} \\ \text{at function start}}}$$

Reliability Constraint

Relate developer's specification and analysis result:

$$r_{spec} \cdot R(P_{spec}) \leq r_1^{q_1} \cdot r_2^{q_2} \cdot \dots \cdot r_n^{q_n} \cdot R(P_{param})$$

Reliability Constraint

$$r_{spec} \leq r_1^{q_1} \cdot r_2^{q_2} \cdot \dots \cdot r_n^{q_n}$$

and

$$\underbrace{R(P_{spec}) \leq R(P_{param})}$$

Can Immediately Solve

Reliability Constraint

$$r_{spec} \leq r_1^{q_1} \cdot r_2^{q_2} \cdot \dots \cdot r_n^{q_n}$$

and

$$\underbrace{R(P_{spec}) \leq R(P_{param})}$$

$$P_{spec} \Rightarrow P_{param}$$

Reliability Constraint

$$r_{spec} \leq r_1^{q_1} \cdot r_2^{q_2} \cdot \dots \cdot r_n^{q_n}$$

Denotes approximate function versions that satisfy the developer's specification

Reliability Constraint

for the optimization problem

$$\log(r_{spec}) \leq q_1 \cdot \log(r_1) + q_2 \cdot \log(r_2) + \dots + q_n \cdot \log(r_n)$$

**Denotes approximate function versions that
satisfy the developer's specification**

Reliability and Control Flow

Conditionals

Constraints for each program path

Analysis removes redundant constraints

(most constraints can be removed - OOPSLA '13)

Bounded

Loops

Statically known loop bound

Analysis unrolls loop

Optimization

Granularity

Optimize blocks of code instead of individual instructions

Reliability and Function Calls

Choose between alternative implementations:

```
int <1.00*R( $\Delta x = 0$ )> f(float x)
```

```
int <0.99*R( $\Delta x = 0$ )> f'(float x)
```

- Reliability degradation: **0.99^{q_f}**
- Enables composition of approximate components

Function Optimization Problem

Find Function Configuration q : ✓

max EnergySavings (q)

Reliability (q) \geq ReliabilityBound ✓

AbsoluteError (q) \leq ErrorBound

Function Optimization Problem

Find Function Configuration q : ✓

max EnergySavings (q)

Reliability (q) \geq ReliabilityBound ✓

AbsoluteError (q) \leq ErrorBound ✓

Energy Savings Analysis

Profile information:

- Collects traces from running representative inputs

Analysis:

- Estimates savings for instructions and variables from traces

instruction

$$q_\ell \cdot Count_\ell \cdot Saving_{ALU}$$

variable

$$q_m \cdot Size_m \cdot Saving_{MEM}$$

Energy Savings Analysis

Profile information:

- Collects traces from running representative inputs

Analysis:

- Estimates savings for instructions and variables from traces

$$c_{ALU} \sum_{\ell \in Instr} \text{instruction } q_{\ell} \cdot Count_{\ell} \cdot Saving_{ALU} + c_{MEM} \sum_{m \in Var} \text{variable } q_m \cdot Size_m \cdot Saving_{MEM}$$

Approximate hardware specification:

- Relative savings for operations and memories
- Percentage of system energy that ALU and memory consume

Function Optimization Problem

Find Function Configuration q : ✓

max EnergySavings (q) ✓

Reliability (q) \geq ReliabilityBound ✓

AbsoluteError (q) \leq ErrorBound ✓

Reduces to Integer Programming

Find Function Configuration q : ✓

max EnergySavings (q) ✓

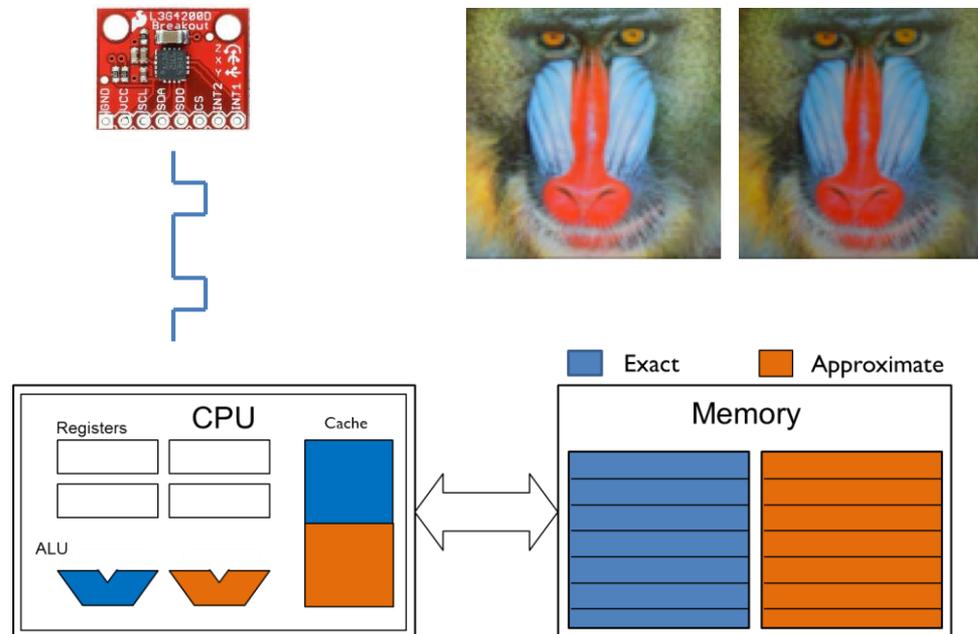
Reliability (q) \geq ReliabilityBound ✓

AbsoluteError (q) \leq ErrorBound ✓

Solve using off-the-shelf solvers (we use Gurobi)

Putting the Pieces Together for a Unified Optimization Framework

- Approximate Sensors, Approximate Data
- Approximate Communication
- Approximate Memory, Computation
- Approximate Output



More Research

Approximation for Time/Energy Savings

- **Outlier Detection/Correction** [MIT TR 2014]
- Skipping Tasks [ICS 2006]
- Early Barrier Termination [OOPSLA 2007]
- Loop Perforation [ICSE 2010, FSE 2011, PLDI 2012]
- Dynamic Knobs [ASPLOS 2011]
- Synchronization Elimination [SPLASH 2012, HOTPAR 2013]
- Approximate Parallelization [ACM TOCS 2013]
- Accuracy/Reliability Guarantees [SAS 2011, PLDI 2012]
- Optimal Approximate Map/Fold Programs [POPL 2012]

Even More Research

Resilience Techniques to Obtain Immortal Programs

- Identification/hardening critical data/computation [ISSTA 2010]
- Surviving out of bounds accesses [ACSAC 2004, OSDI 2004]
- Repairing corrupt data structures [OOPSLA 2003, ICSE 2005]
- Eliminating memory leaks [ISMM 2007]
- Escaping infinite loops [ECOOP 2011, OOPSLA 2012]
- Recovering from null pointer/divide by zero errors [PLDI 2014]

Finding Security Vulnerabilities

- At API boundaries [ICSE 2009]
- Integer, buffer overflows [ASPLOS 2015]

Eliminating Security Vulnerabilities

- Input Rectification/Filtering [ICSE 2012, POPL 2014]
- Automatic Patch Generation [SOSP 2009, MIT TR 2015]
- Automatic Code Transfer [PLDI 2015]