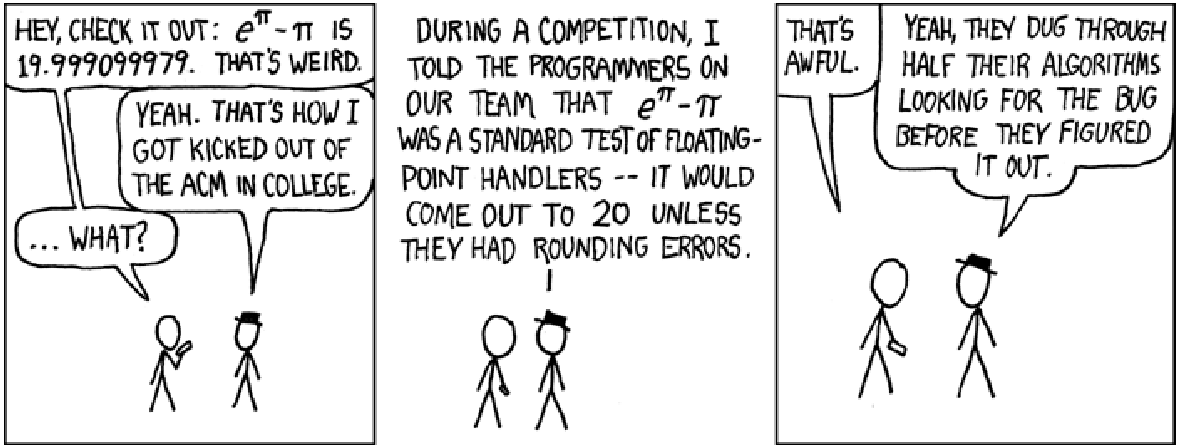
JMU CS
Undergraduate Research Lab

Mike Lam



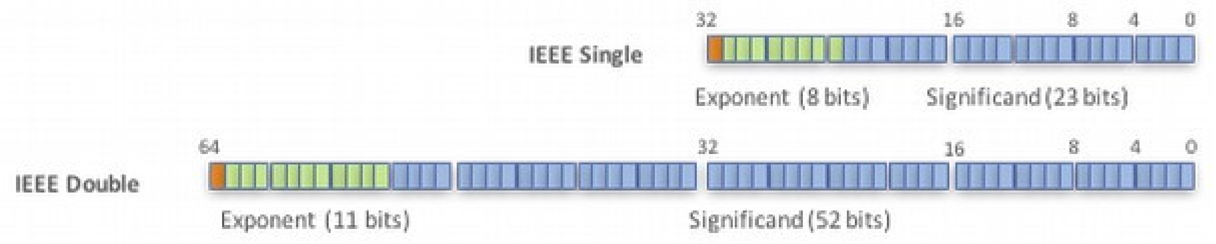
Office Space and Salami:
Automated Floating-Point Program Analysis

Remember “Office Space”?

- Plan: steal fractions of cents from financial transactions
- Idea: lots of small amounts add up eventually
  - Sometimes called “salami slicing”
  - My research involves similar effects in computer arithmetic
  - (as well as other issues)

(Accurate) Math is Hard™

- Computers are digital and memory is finite
- Real numbers must be rounded to be stored
- Modern supercomputers do millions of billions of operations per second
  - Rounding errors can add up
  - Just like Office Space...



Example

```
sum  = 0.0
incr = 0.001
for i = 1 to 1,000,000 do
    sum = sum + incr
print sum
```

```
Correct answer                 1000.0000000000000

Single precision                991.14154052734375
Double precision                999.99999998326507
Long double precision          1000.0000000000085
```

How do we deal with it?

How do we deal with it?

One option:

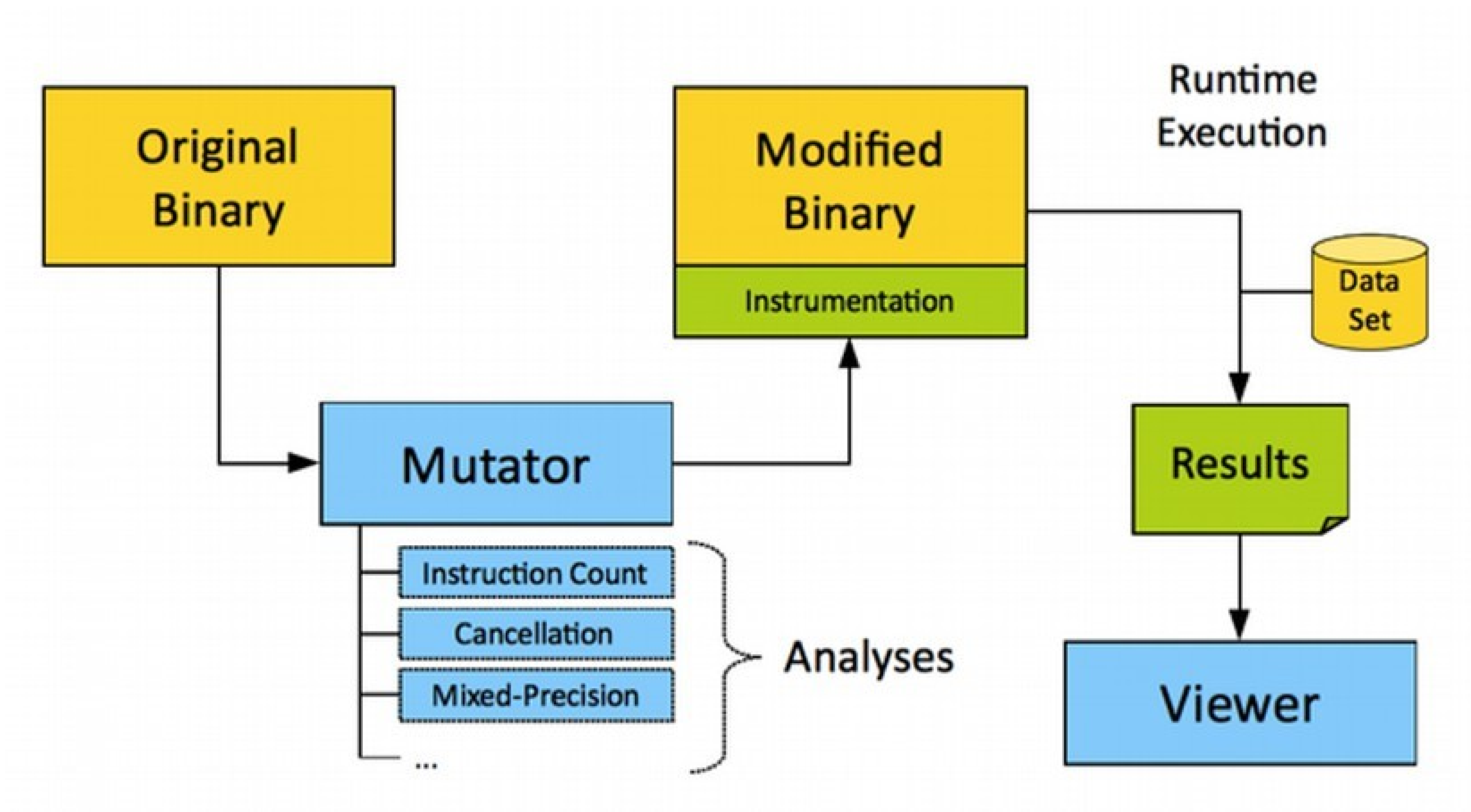
How do we deal with it?

- Use more bits (e.g., doubles instead of floats)
  - Space is valuable (especially bandwidth!)
- Sophisticated error analysis
  - Most developers aren’t numerical analysts
  - Static analyses yield overly-conservative results
- Interval / universal number arithmetic
  - Slow, and also too conservative
- Manual trial and error
  - This sucks

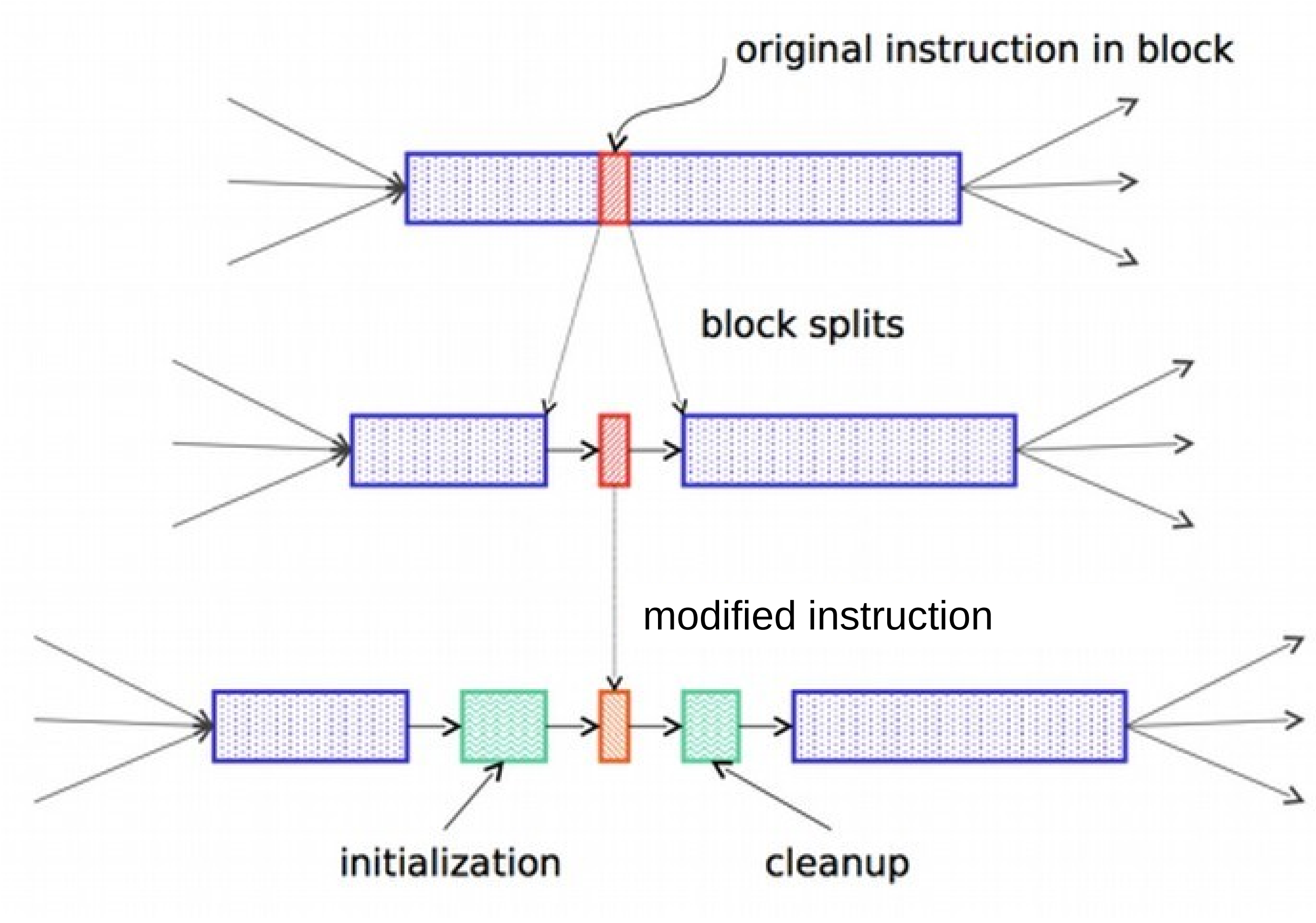
What I do

- Automatic analysis for floating-point programs
- I write and extend binary instrumentation tools
  - Modify existing machine code programs
  - Insert new instrumentation or monitoring code
  - Observe floating-point behavior
  - Report results and make recommendations

Binary Instrumentation



Binary Instrumentation



Binary Instrumentation

- Dyninst: generic binary instrumentation tool
  - Inserts small “snippets” of binary code
  - Good for short, efficient instrumentation
  - Writing the snippets can be a pain

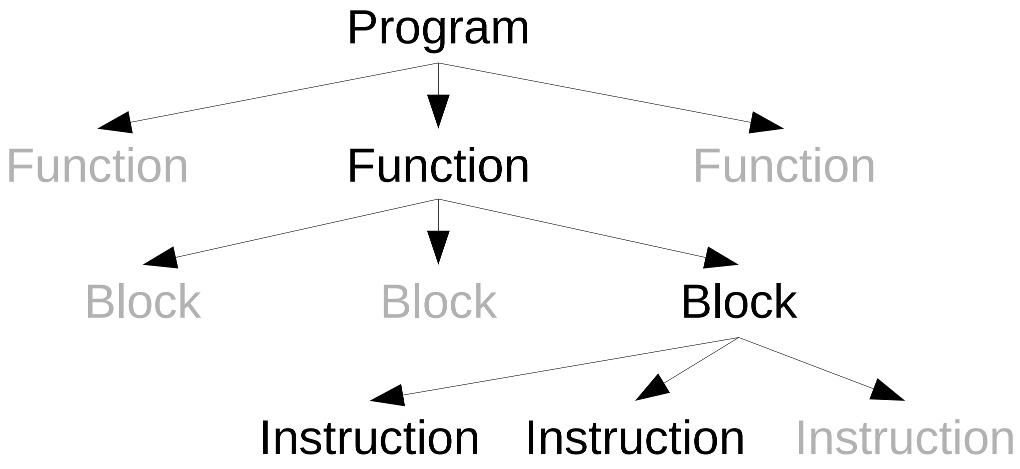
- Intel Pin: x86-64 instrumentation tool
  - Inserts calls to C functions (or inlines them)
  - Good for more complex instrumentation
  - Allows rapid development

Binary Instrumentation

```
gvec[i,j] = gvec[i,j] * lvec[3] + gvar

  movsd 0x601e38(%rax, %rbx, 8), %xmm0
  check/replace -0x78(%rsp) and %xmm0
  mulsd -0x78(%rsp), %xmm0
  check/replace -0x4f02(%rip) and %xmm0
  addsd -0x4f02(%rip), %xmm0

  movsd %xmm0, 0x601e38(%rax, %rbx, 8)
```

Idea: Mixed Precision

- Use 64-bit where necessary
  - 32-bit everywhere else
- How to tell where to switch?
  - User-provided error threshold
  - Complete search w/ pruning



Program

Function Function Function

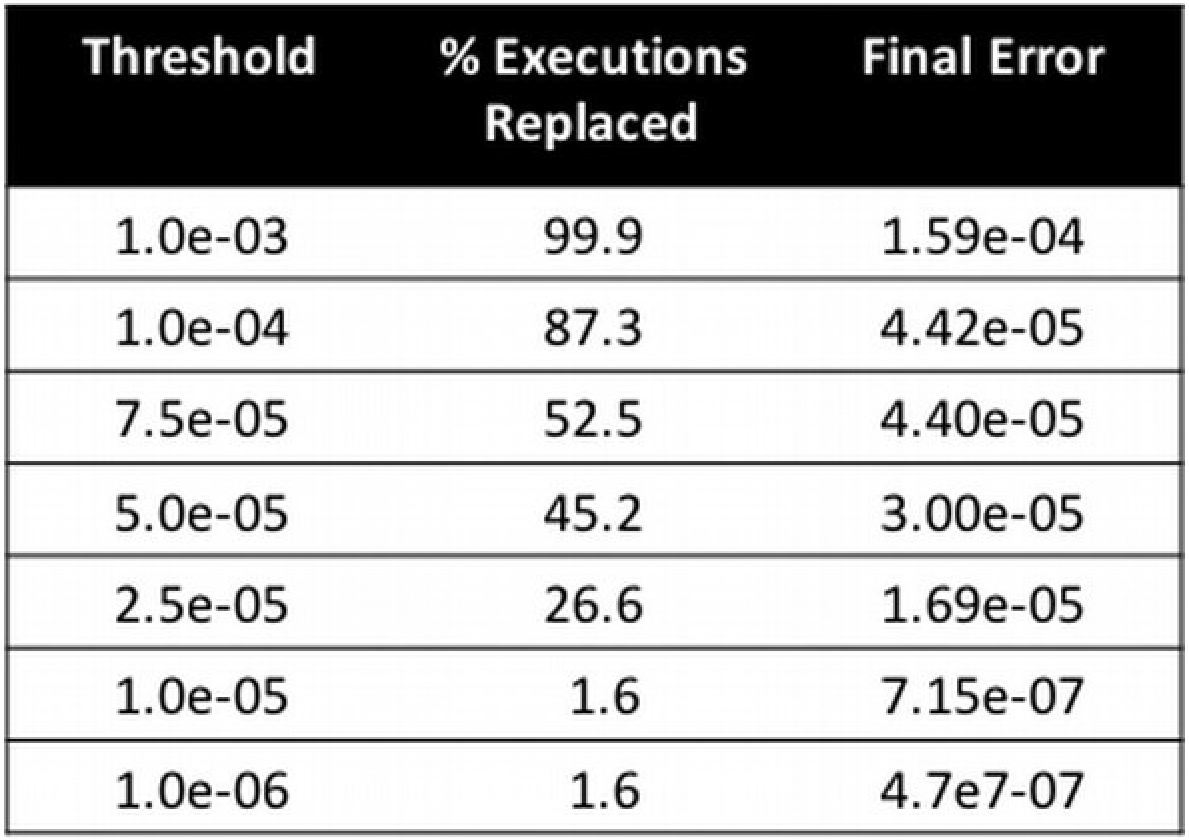
Block Block Block

Instruction Instruction Instruction

```
int sum2pi_x() {
    int i, j, k;
    double x, y, acc;
    double sum;

    double final = PI * OUTER;

    sum = 0.0;
    for (i=0; i<OUTER; i++) {
        acc = 0.0;
        for (j=1; j<INNER; j++) {

            x = 1.0;
            for (k=0; k<j; k++)
                x *= 2.0;

            y = (double)PI / x;
            acc += y;
        }
        sum += acc;
    }

    double err = abs(final-sum) /
                 abs(final);
    if (err < EPS)
        printf(“SUCCESSFUL!\n");
    else
        printf(“FAILED!!!\n");
}
```

Mixed Precision

- Relationship between error threshold and 32-bit percentage
  - More 32-bit computation → more error

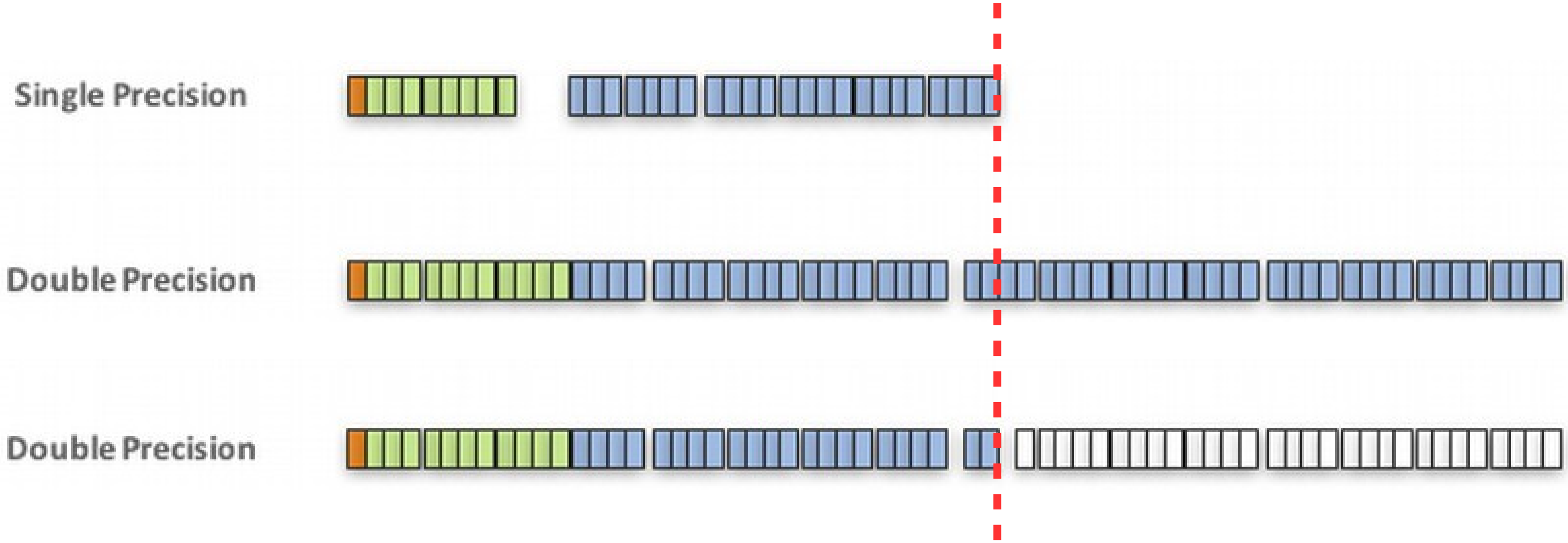


Threshold	% Executions Replaced	Final Error
1.0e-03	99.9	1.59e-04
1.0e-04	87.3	4.42e-05
7.5e-05	52.5	4.40e-05
5.0e-05	45.2	3.00e-05
2.5e-05	26.6	1.69e-05
1.0e-05	1.6	7.15e-07
1.0e-06	1.6	4.7e7-07

SuperLU
proxy app

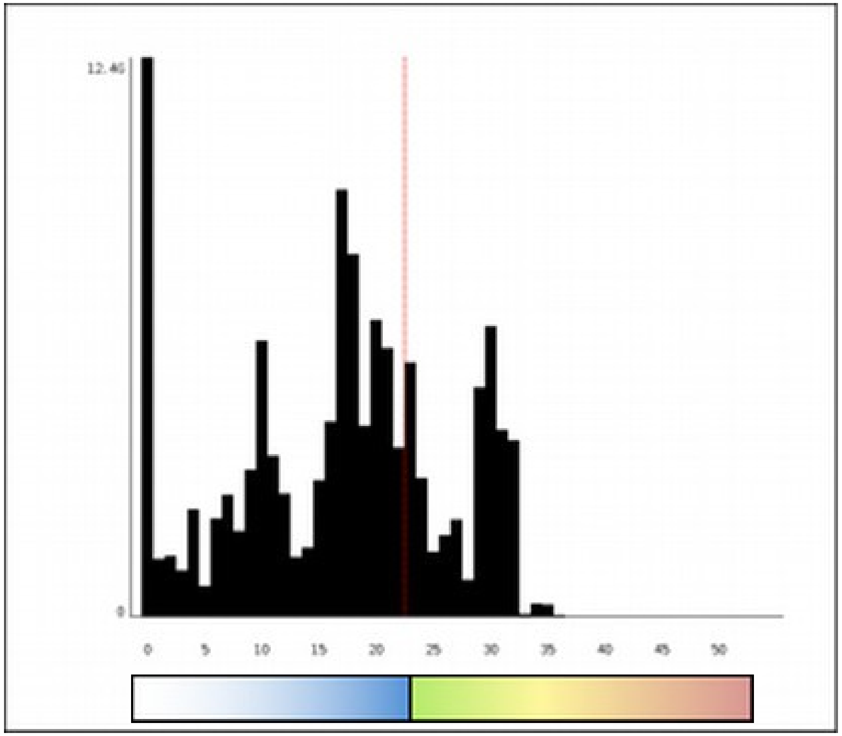
Idea: Reduced Precision

- Don’t restrict ourselves to 32 vs. 64 bits
  - Truncate after an arbitrary # of bits
  - Approximates any precision ≤ 64 bits

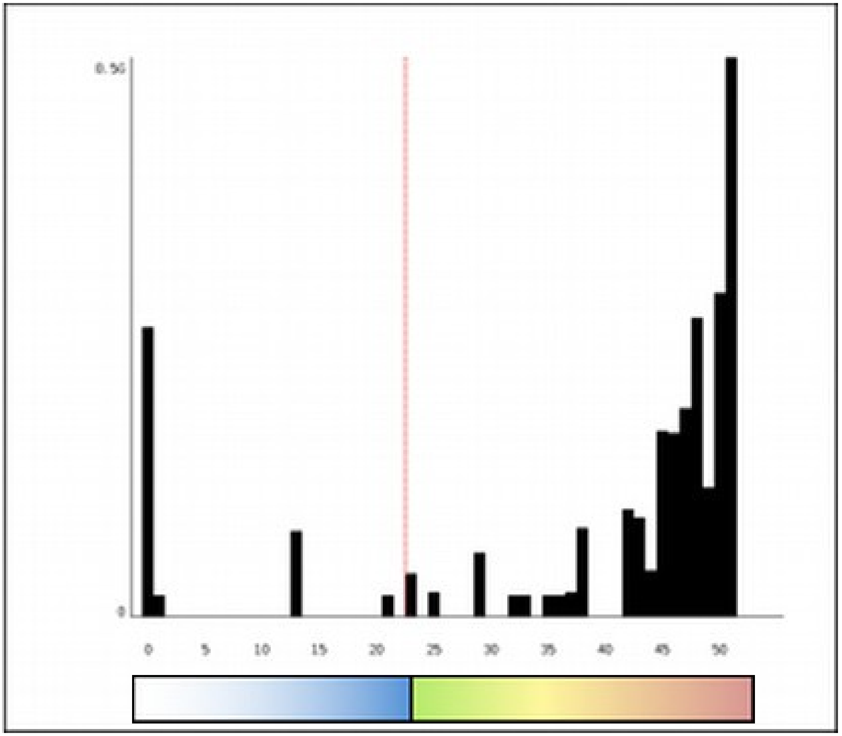


Reduced Precision

- Combine with automated search
  - Generalized precision level requirement profiles



Low sensitivity



High sensitivity

Idea: Shadow Values

- Don’t restrict ourselves to ≤ 64 bits
  - Allow arbitrary online tracking of “shadow” values
    - These values could be of any type
    - 32-bit, 64-bit, 128-bit, or arbitrary precision floating-point
    - Integer, rational number, universal number, interval
  - Track and report relative error
    - Between original value and shadow value
  - Requires rather heavyweight analysis
  - Implemented as a ~2500-LOC Pintool

Example

Original C Code

```
double sum = 0.0;
for (int i = 0; i < 10; i++) {
    sum += 0.1;
}
printf("%25.20f\n", sum);
```

Original Output

```
0.99999999999999988898
```

Compiled x86 Code

```
  pxor    %xmm0, %xmm0          (set to 0.0)
  mov     $10, %eax
  movsd   0x400628, %xmm1       (load 0.1)
loop:
  addsd   %xmm1, %xmm0          (increment)
  sub     $1, %eax
  jnz     loop
  movsd   %xmm0, 0x8(rsp)       (store sum)
```

Inserted Shadow Value Code

```
xmm[0] = convert(0.0)

xmm[1] = convert(*(0x400628))

xmm[0] += xmm[1]


mem[rsp+0x8] = xmm[0]
```

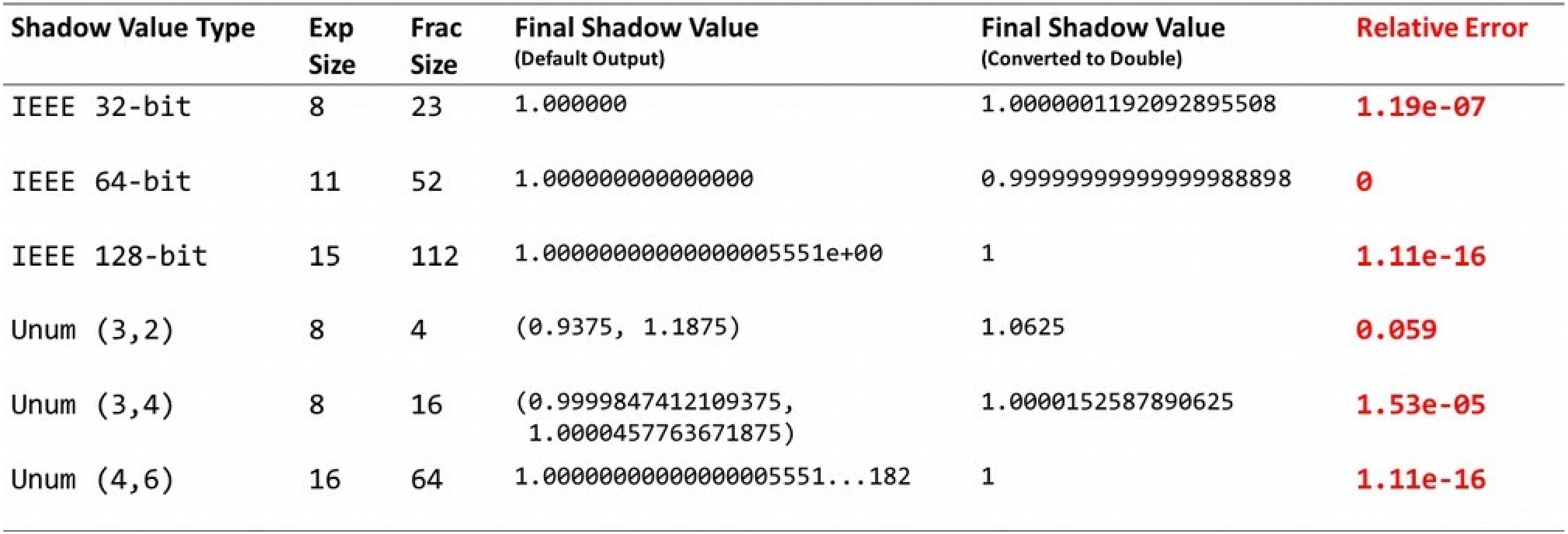
Example

Original C Code

```
double sum = 0.0;
for (int i = 0; i < 10; i++) {
    sum += 0.1;
}
printf("%25.20f\n", sum);
```

Original Output

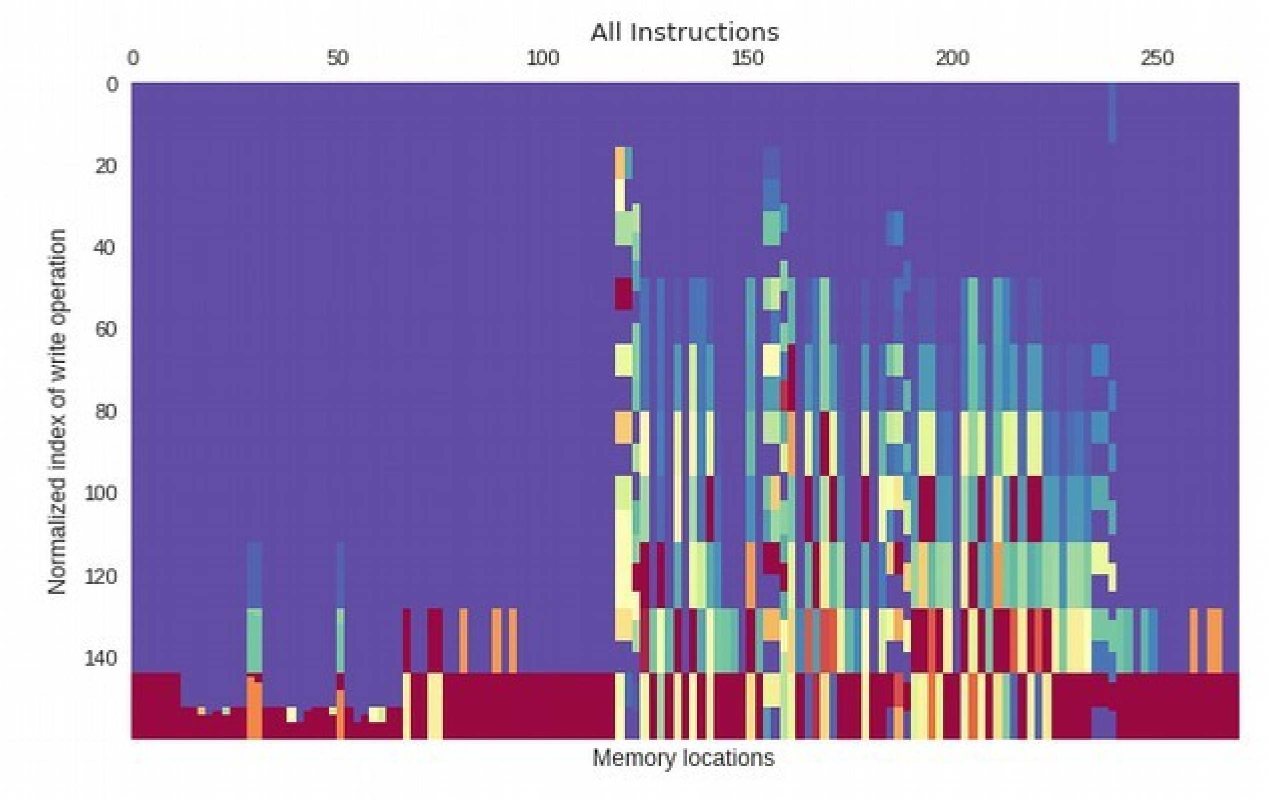
```
0.99999999999999988898
```



Shadow Value Type	Exp Size	Frac Size	Final Shadow Value (Default Output)	Final Shadow Value (Converted to Double)	Relative Error
IEEE 32-bit	8	23	1.000000	1.0000001192092895508	1.19e-07
IEEE 64-bit	11	52	1.000000000000000	0.99999999999999988898	0
IEEE 128-bit	15	112	1.00000000000000005551e+00	1	1.11e-16
Unum (3,2)	8	4	(0.9375, 1.1875)	1.0625	0.059
Unum (3,4)	8	16	(0.9999847412109375, 1.0000457763671875)	1.0000152587890625	1.53e-05
Unum (4,6)	16	64	1.00000000000000005551...182	1	1.11e-16

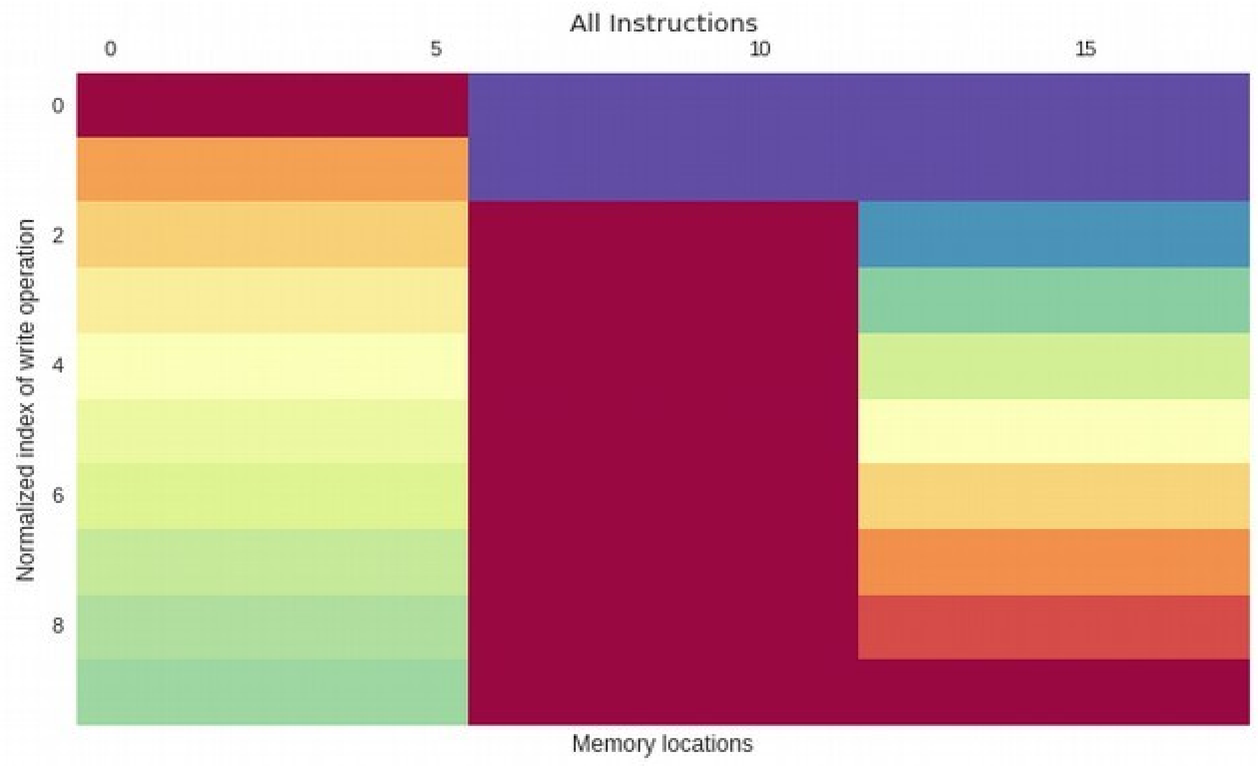
Recent Results

- Change in error per memory location (32-bit vs. 64-bit)
  - Courtesy of Ramy Medhat (Univ. of Waterloo)
  - Extension of shadow value analysis tool
  - 135 memory locations with relative error >1% (max 160 writes per loc)

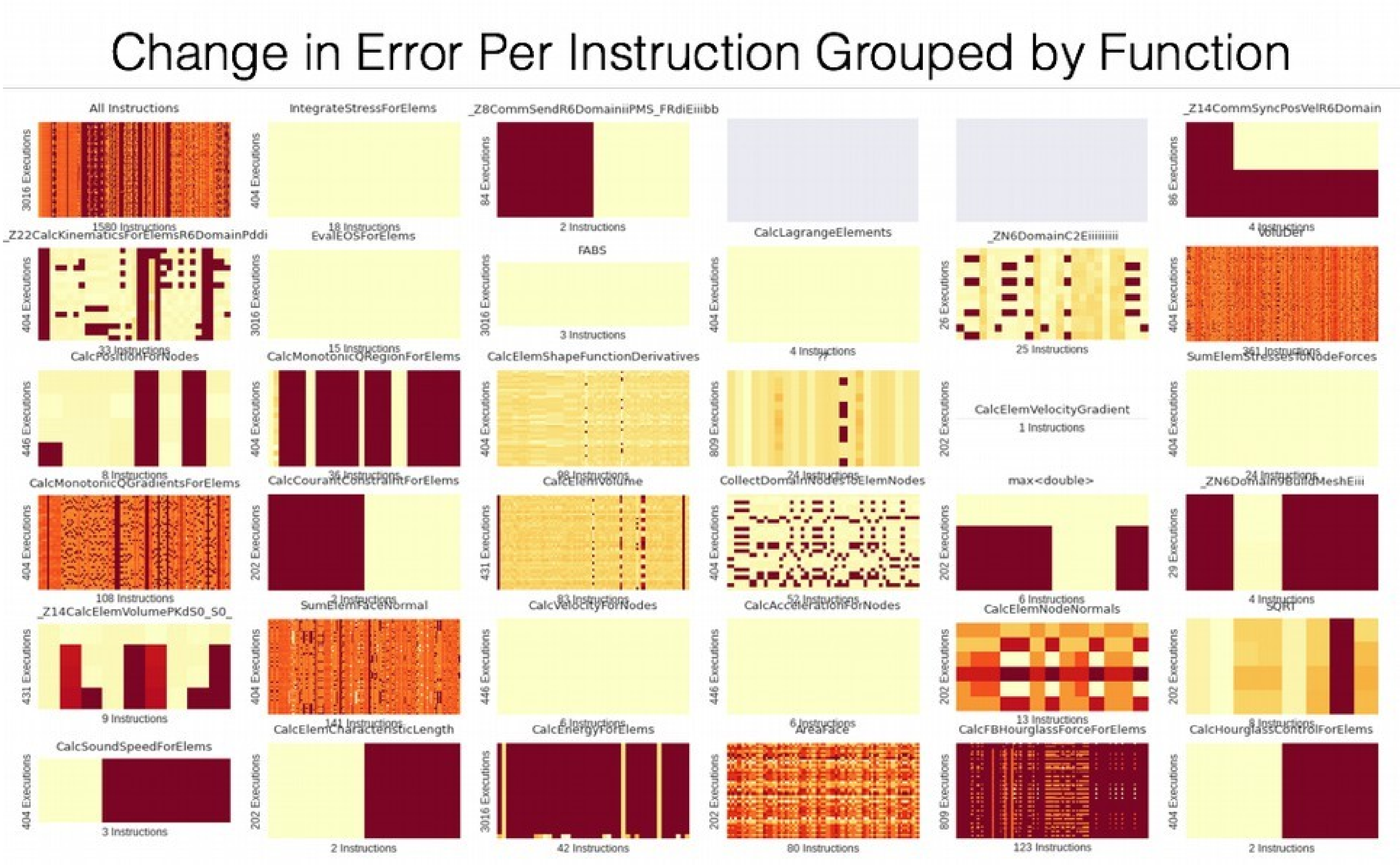


Recent Results

- Change in error per memory location (128-bit vs. 64-bit)
  - Courtesy of Ramy Medhat (Univ. of Waterloo)
  - Extension of shadow value analysis tool
  - 3 memory locations with relative error >1% (max 10 writes per loc)

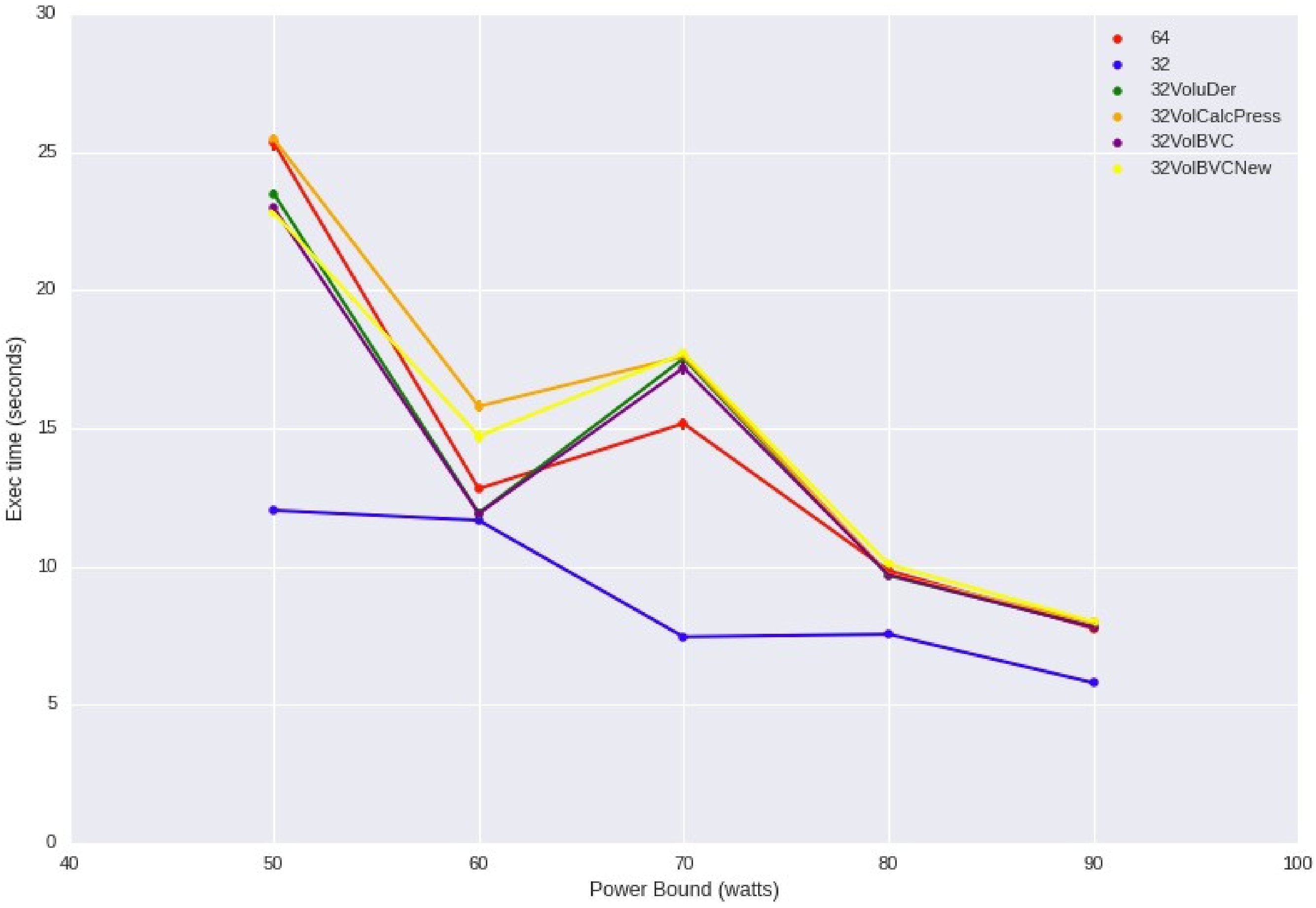


Recent Results



Recent Results

- Various mixed precision configurations
  - Subject to power bounds



Projects: Application

- Applying existing tools
  - More benchmarks and proxy apps
  - More HPC and scientific computing case studies
  - Game/graphics engine (visual fidelity vs. performance)
  - Applications to PDE/ODE solvers or iterative methods
    - (w/ math faculty)

Projects: Extension

- Extend shadow value analysis
  - More profiling and optimization
  - Improve MPI/threading support
  - Improve interval/unum support
  - Improve AVX instruction support
  - Report control flow divergence
  - Report symbol names (from ELF file data?)
  - New shadow value types (rational, fixed-point, stochastic, multi-type)
  - IDE plugin

Projects: New Directions

- Floating-point visualization tool
    - Educational application (useful for CS 261)
- Source-level analysis
    - What can we do at the source level that wouldn't be possible at the binary level? What would we lose?
    - What could we do at the compiler level? (e.g., LLVM)
- Runtime management system
    - Goal: maximize performance with a lower bound on accuracy and an upper bound on power use

Opportunities

- If you’re interested:
  - Talk to me!
  - Take CS 261 if you haven’t already
  - Consider taking CS 432 or 470 as your systems elective
  - Consider the LLNL scholars program (Summer 2017)