Probabilistic & Approximate Computing

Sasa Misailovic

UIUC
Previously:
We wrote and analyzed probabilistic programs!
From now on:
Let’s approximate programs
Approximate Compiler Optimizations and Analyses

• Categories of transformations
• Loop Perforation
• Methodology for evaluation
• Summary of Analysis
Software Based Transformations

Skip part of the work
• Loop perforation

Perform different (less expensive) work
• Function substitution

Reduce communication overhead
• Soft barriers
Common Properties

Suited for specific computational patterns in applications that can inherently tolerate noise.

If applied judiciously, can significantly speed up the computation for small accuracy losses.
Loop Perforation

\[
\text{for (} i = 0; i < n; i++ \text{) \{} \ldots \text{\}}
\]

\[
\text{for (} i = 0; i < n; i += 2 \text{) \{} \ldots \text{\}}
\]

Misailovic et al. Quality of Service Profiling, ICSE 2010
Loop Perforation

\[
\text{for } (i = 0; i < n; i++) \{ \ldots \}
\]

\[
\text{for } (i = 0; i < n / 2; i++) \{ \ldots \}
\]

Misailovic et al. Quality of Service Profiling, ICSE 2010
Loop Perforation

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

→

```c
for (i = 0; i < n; i++) {
    if (Bernoulli(0.5)) continue;
    ...
} 
```
Reduction Sampling

for (i = 0; i < n; i++) {
    y = f(x[i]);
    s = s + y;
}

for (i = 0, z = 0; i < n; i++) {
    if (rand(0.75)) {z++; continue;}
    y = f(x[i]);
    s = s + y;
}

s = s * n/(n-z);
Approximate Tiling

InType[] x; OutType[] y;
for (i = 0; i < n; i++) {
    y[i] = f(x[i]);
}

InType prev;
for (i = 0; i < n; i++) {
    if i%2 == 1
        y[i] = prev;
    else {
        y[i] = f(x[i]);
        prev = y[i];
    }
}
Approximate Memoization

```javascript
InType[] x; OutType[] y;
for (i = 0; i < n; i++) { y[i] = f(x[i]); }

var table = new Map<InType, OutType>;
for (i = 0; i < n; i++) {
  if ∃x’,ν. x’∈[x[i]−ε, x[i]+ε] && (x’,ν)∈table
    y[i] = ν;
  else {
    y[i] = f(x[i]);
    table[x[i]] = y[i];
  }
}
```

Chaudhuri et al. Proving Programs Robust, FSE 2011
Function Substitution

\[ y = f(x); \]

\[ y = f'(x); \]

<table>
<thead>
<tr>
<th>Version</th>
<th>TimeSpec</th>
<th>ErrorSpec</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) )</td>
<td>Time1</td>
<td>Err1</td>
</tr>
<tr>
<td>( f'(x) )</td>
<td>Time2</td>
<td>Err2</td>
</tr>
</tbody>
</table>

For instance, polynomial approximation of transcendental functions:

\[ \sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \text{ for } x \text{ near 0} \]

\[ R(x) \leq |x|^{n+1} / (n + 1)! \]
Function Substitution

\[ y = f(x); \]
\[ \downarrow \]
\[ y = f'(x); \]

<table>
<thead>
<tr>
<th>Version</th>
<th>TimeSpec</th>
<th>ErrorSpec</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) )</td>
<td>Time1</td>
<td>Err1</td>
</tr>
<tr>
<td>( f'(x) )</td>
<td>Time2</td>
<td>Err2</td>
</tr>
</tbody>
</table>

Neural Network:

Esmaeilzadeh et al., Neural Acceleration for General-Purpose Approximate Programs, MICRO '12
Dynamic Function Substitution

\[ y = f(x); \]

\[ y = \text{runtime.executeApprox}()? f'(x) : f(x); \]

<table>
<thead>
<tr>
<th>Version</th>
<th>TimeSpec</th>
<th>ErrorSpec</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(x)</td>
<td>Time1</td>
<td>Err1</td>
</tr>
<tr>
<td>f'(x)</td>
<td>Time2</td>
<td>Err2</td>
</tr>
</tbody>
</table>

* Baek et al., Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation, PLDI 2010
* Hoffmann et al., Dynamic Knobs for Efficient Power Aware Computing, APSLOS 2011
Floating Point Optimizations

double[] x, y
double z = f(x,y)

float[] x, y
float z = f(x,y)
Skipping Tasks *(at Barrier Points)*

```plaintext
task {
  x = ...
  y = ...
}

Continue execution after all tasks finish
```

```plaintext
Continue execution after all tasks finish before timeout,
Otherwise kill delayed or non-responsive tasks
```

Rinard, Probabilistic accuracy bounds for fault-tolerant computations that discard tasks, ICS ‘06
Removing Synchronization

\[
\text{lock();} \\
x = f(x,y); \\
y = g(x,y); \\
\text{unlock();}
\]

\[
\text{lock();} \\
x = f(x,y); \\
y = g(x,y); \\
\text{unlock();}
\]

\[
\text{lock();} \\
x = f(x,y); \\
y = g(x,y); \\
\text{unlock();}
\]

\[
\text{lock();} \\
x = f(x,y); \\
y = g(x,y); \\
\text{unlock();}
\]

* Renganarayana et al. Programming with Relaxed Synchronization, RACES ’12
* Misailovic et al. Dancing with Uncertainty, RACES ‘12
More Accuracy-Aware Transformations

**Software-Level Transformations**

**Approximate Parallelization**
- Misailovic, Kim, Rinard, MIT-TR ’10, TECS PEC ‘13

**Dynamic Knobs**
- Hoffmann, Sidiroglou, Carbin, Misailovic, Agarwal, Rinard ASPLOS ’11

**Randomized Function Substitution**
- Zhu, Misailovic, Kelner, Rinard POPL ‘12

**Function Substitution**
- Baek et al., PLDI 10; Ansel et al., CGO ’11; Samadi et al., ASPLOS ’14;

**Approximate Memoization**
- Chaudhuri et al., FSE ’11; Samadi et al., ASPLOS ’14; Mishra et al., WACAS ’14;

**Floating Point Optimization**
- Rubio-Gonzalez et al., SC ‘13; Schkufza et al., PLDI ’14;

**Relaxed Synchronization**
- Rinard ICS ‘06; Rinard OOPSLA ’07; Meng et al., IPDPS’10; Chakradhar et al., DAC ‘10;
- Renganarayana et al., RACES ’12; Rinard, HotPar ’13; Companoni et al., CGO ‘15
(Some of the) **Main Questions** for Approximate Computing

How to…

…**specify** approximate program requirements?

…**identify** code amenable to approximation?

…**find** most profitable combinations of approximations?

…**guarantee** accuracy and safety of approximations?
Optimization Dimensions:

*Main Flavor of the Approach*

1. **Testing**-based approach

2. **Analysis**-based approach
Optimization Dimensions

Main Flavor of the Approach

1. **Off-line** accuracy-aware optimization

2. **On-line** dynamic accuracy-aware optimization
SpeedPress: Optimization Framework for Loop Perforation

Quality of Service Profiling (ICSE 2010)
Misailovic, Sidiroglou, Hoffmann, Rinard

Managing Performance vs. Accuracy Trade-offs With Loop Perforation (FSE 2011)
Sidiroglou, Misailovic, Hoffmann, Rinard
<table>
<thead>
<tr>
<th>Original Program</th>
<th>Typical Inputs</th>
<th>Accuracy Requirement</th>
</tr>
</thead>
</table>

**SpeedPress**

- **Transforms** programs with perforation
- **Validates** new programs using testing

Optimized Program +
x264 Video Encoder Example

Typical Inputs
**x264 Video Encoder Example**

**Accuracy Requirement**

- **Quality Metric:**
e.g. PSNR and bit rate

- **Quality Loss:**
e.g. relative difference <10%
Search for Perforatable Loops

- Run **performance** profiler
  Identify time consuming loops

- Perforate **one loop** at a time
  Filter out loops that do not satisfy accuracy requirement

- Perforate **multiple loops** together
  Find combinations of loops that maximize performance
## Performance Profiling

<table>
<thead>
<tr>
<th>Loop Location</th>
<th>Instruction %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode</td>
<td>82%</td>
</tr>
<tr>
<td>analyse_inter</td>
<td>65%</td>
</tr>
<tr>
<td>pixel_sad, outer</td>
<td>55%</td>
</tr>
<tr>
<td>pixel_sad, inner</td>
<td>54%</td>
</tr>
<tr>
<td>search_ref</td>
<td>25%</td>
</tr>
<tr>
<td>pixel_sad, outer</td>
<td>18%</td>
</tr>
<tr>
<td>pixel_sad, inner</td>
<td>18%</td>
</tr>
<tr>
<td>refine_subpel</td>
<td>12%</td>
</tr>
</tbody>
</table>
Validate Perforated Loops

Filter out loops that do not satisfy requirement

Check for crashes, slowdowns, latent memory errors

\[ \alpha(\ ) = \begin{cases} \text{PSNR} & q(\ ) \leq q_m \\ \text{bitrate} & \end{cases} \]
Accuracy Requirement for x264

Quality metric $\alpha$ computes:

- Peak signal-to-noise ratio
- Bit rate (file size)

Quality loss $q$: $\frac{1}{2} \left( \left| \frac{PSNR - PSNR'}{PSNR} \right| + \left| \frac{bitrate - bitrate'}{bitrate} \right| \right)$

Acceptable quality loss $q_m : 0.1$
Perforating Individual Loops in \textbf{x264}  

( Quality Loss < 0.1 )
Perforating Individual Loops in x264

( Quality Loss < 0.1)
Perforating Individual Loops in x264

( Quality Loss < 0.1)

# loops

Latent Errors
Crash
Perforating Individual Loops in $\times264$

(Quality Loss < 0.1)

### Graph

- **Y-axis**: # loops
- **X-axis**: looses

- **No Speedup**
- **Latent Errors**
- **Crash**
Perforating Individual Loops in \texttt{x264}

( Quality Loss < 0.1)

\# loops

- Low Accuracy
- No Speedup
- Latent Errors
- Crash
Perforating Individual Loops in **x264**

( Quality Loss < 0.1)

![Bar chart showing the distribution of perforable, low accuracy, no speedup, latent errors, and crashes among 6 perforatable loops.]

- **6 perforatable loops**
  - Green: Perforatable
  - Yellow: Low Accuracy
  - Blue: No Speedup
  - Red: Latent Errors
  - Pink: Crash

### Data

- **# loops**
  - 25
  - 20
  - 15
  - 10
  - 5
  - 0
## Individual Loop Profiling Results

<table>
<thead>
<tr>
<th>Loop Location</th>
<th>Instr%</th>
<th>Approximable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encode</td>
<td>82%</td>
<td>✗</td>
</tr>
<tr>
<td>analyse_inter</td>
<td>65%</td>
<td>✗</td>
</tr>
<tr>
<td>pixel_sad, outer</td>
<td>55%</td>
<td>✓</td>
</tr>
<tr>
<td>pixel_sad, inner</td>
<td>54%</td>
<td>✓</td>
</tr>
<tr>
<td>search_ref</td>
<td>25%</td>
<td>✓</td>
</tr>
<tr>
<td>pixel_satd, outer</td>
<td>18%</td>
<td>✓</td>
</tr>
<tr>
<td>pixel_satd, inner</td>
<td>18%</td>
<td>✓</td>
</tr>
</tbody>
</table>
Navigate Tradeoff Space

Perforate **multiple loops** together

- Validates each combination of loops
- Exhaustive search or Greedy search

This example:

- Exhaustive search over 6 loops
- Skipping 25%, 50%, 75%, or 99% of iterations
Navigate Tradeoff Space

Speedup

Quality loss
Navigate Tradeoff Space

![Graph showing the relationship between Speedup and Quality loss](image-url)
Navigate Tradeoff Space

Quality loss vs. Speedup

Quality loss range: 0.0 to 0.15
Speedup range: 0 to 4

Plot showing the relationship between quality loss and speedup.
Navigate Tradeoff Space

Graph showing Speedup vs. Quality loss.
Perforated Encoding 3x faster
Evaluated Programs

From the PARSEC benchmark suite:

- **x264**: video encoder
- **bodytrack**: human pose tracking
- **swaptions**: financial analysis
- **ferret**: image search
- **canneal**: electronic circuit placement
- **streamcluster**: point clustering
Perforated Computations

Total of 24 perforated loops

- x264: motion estimation
- bodytrack: particle filtering
- swaptions: Monte Carlo simulation
- ferret: similarity hashing
- canneal: simulated annealing
- streamcluster: cluster center search
x264 Motion Estimation

Reference Frame

Current Frame
score = 0;

for (i = 0; i < block_height; i++) {
    for (j = 0; j < block_width; j++) {
        idx1 = IDX(i, j, cur_start);
        idx2 = IDX(i, j, prev_start);
        diff = cur_frame[idx1] − prev_frame[idx2];
        adif = abs(diff);
        score = score + adif;
    }
}

return score;
score = 0;

for (i = 0; i < block_height; i+=2) {
    for (j = 0; j < block_width; j+=2) {

        idx1 = IDX(i, j, cur_start);
        idx2 = IDX(i, j, prev_start);
        diff = cur_frame[idx1] - prev_frame[idx2];
        adif = abs(diff);
        score = score + adif;

    }
}

return score;
```c
int bestidx = 0, bestscr = MAXINT;

for (int b = 4*qpel_iters; b > 0; b-- ) {
  int score = block_match(cur, ref[index[b]], data);
  if (score < bestscr) {
    bestscr = score;
    bestidx = index[b];
  }
  if( early_stop(b, bestidx, bestscr, score) )
    break;
}
```
int bestidx = 0, bestscr = MAXINT;

for (int b = 4*qpel_iters; b > 0; b-- ) { 

    int score = block_match(cur, ref[index[b]], data);

    if (score < bestscr) {
        bestscr = score;
        bestidx = index[b];
    }

    if( early_stop(b, bestidx, bestscr, score) )
        break;
}
int bestidx = 0, bestscr = MAXINT;

for (int b = 4*qpel_iters; b > 0; b-=2) {
    int score = block_match(cur, ref[index[b]], data);
    if (score < bestscr) {
        bestscr = score;
        bestidx = index[b];
    }
    if( early_stop(b, bestidx, bestscr, score) )
        break;
}

x264 Block Matching
Perforated Computations

Total of 24 perforated loops

• Distance metrics
• Search-space enumeration
• Iterative improvement
### Effects of Loop Perforation

<table>
<thead>
<tr>
<th>Distance metrics</th>
<th>compares features of two items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search-space enumeration</td>
<td>visits a collection of items</td>
</tr>
<tr>
<td>Iterative improvement</td>
<td>improves approximation in each step</td>
</tr>
</tbody>
</table>

**Common:** Perforation Makes Coarser Approximations

- **Sampling features**
- **Sampling items**
- **Early stopping**
Main Observations

- **Approximate Kernel Computations**
  (have specific structure + functionality)

- **Accuracy vs Performance Knob**
  (tune how aggressively to approximate kernel)

- **Magnitude and Frequency of Errors**
  (kernels rarely cause large output deviations)
Approximate Program Analysis = Accuracy + Safety
Analysis and **Guarantees**

**Logic-Based** *(worst-case)*
“for all inputs… ”

**Probabilistic** *(worst-case or average-case)*
“for all inputs, with probability at least p…”
“for inputs distributed as…”

**Statistical** *(average-case)*
“for inputs distributed as… with confidence c”
“for tested inputs… with confidence c”

**Empirical** *(typical-case)*
“for typical inputs…” (i.e. those we ran tests on)

* using the word guarantees in this case is very liberal
Testing-based Optimization

- Transform original computation
- Validate transformed computation

Optimized Computation
Analysis-based Optimization

- **Statically analyze** computation’s accuracy
- **Transform** computation by solving a mathematical optimization problem
Comparison

(a) Profile-Based Optimization

(b) Analysis-Based Optimization