Benoit Baudry

WASP Professor in Software Technology

baudry@kth.se
Background - CV

• 2017: WASP Professor at KTH (SE)
• 2004 – 17: Research scientist at INRIA (FR)
• 2008: Visiting scientist at CSU (USA)
• 2003 – 2004: Postdoc at CEA (FR)
• 2003: PhD Université de Rennes 1
Background - area

• Software testing
• Model-based design

• Automatic search and analysis
• Bio inspired software evolution
• Empirical methods
Background - collaborations

• European projects coordination
  • Collective adaptive systems (FP7)
  • Software technology (H2020)

• Industry bilateral collaborations
  • Tech transfer

• amiunique.org
  • Public outreach
Context: Large, open software systems

• Large-scale distributed systems
• High frequency computation and data streams
• Fragile software monocultures
Research agenda

Software diversification

Break asymmetry between unstable environments and rigid computation

Evolution
specialization, randomization, runtime diversity

Approximate computing
optimization, performance, unsound transformations

Testing
code analysis, transformation, qualification, specification
Approximate Loop Unrolling

M. Rodriguez-Cancio, B. Combemale, B. Baudry
Approximate computing

• Trade computation quality for performance
• Hypothesis
  • Performance vs. precision in hardware
  • Continuous vs. discrete nature of software
• Four ingredients
  • Forgiving zone
  • Domain-specific accuracy metric
  • Approximation strategy
  • Determine performance trade-off

Two examples


Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, Martin C. Rinard: Quality of service profiling. ICSE 2010: 25-34
# FlexJava - EnerJ

<table>
<thead>
<tr>
<th>Operation</th>
<th>Technique</th>
<th>Mild</th>
<th>Medium</th>
<th>Aggressive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Arithmetic/Logic</td>
<td>Voltage Overscaling</td>
<td>Timing Error Probability</td>
<td>$10^{-6}$</td>
<td>$10^{-4}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Energy Reduction</td>
<td>12%</td>
<td>22%</td>
</tr>
<tr>
<td>Floating Point Arithmetic</td>
<td>Bit-width Reduction</td>
<td>Mantissa Bits (float)</td>
<td>16 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mantissa Bits (double)</td>
<td>32 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Energy Reduction</td>
<td>32%</td>
<td>78%</td>
</tr>
<tr>
<td>SRAM Read/Write (Reg File/Data Cache)</td>
<td>Voltage Overscaling</td>
<td>Read Upset Probability</td>
<td>$10^{-16.7}$</td>
<td>$10^{-7.4}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Write Failure Probability</td>
<td>$10^{-5.6}$</td>
<td>$10^{-4.9}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Energy Reduction</td>
<td>70%</td>
<td>80%</td>
</tr>
<tr>
<td>DRAM (Memory)</td>
<td>Reduced Refresh Rate</td>
<td>Per-Second Bit Flip  Probability</td>
<td>$10^{-9}$</td>
<td>$10^{-5}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory Power Reduction</td>
<td>17%</td>
<td>22%</td>
</tr>
</tbody>
</table>
float computeLuminance (float r, float g, float b) {
    float luminance = r*0.3f+g*0.6f+b*0.1f;
    loosen(luminance);
    return luminance;
}
FlexJava - EnerJ

(a) Energy Reduction

(b) Quality Loss
Loop perforation (Rinard and colleagues)

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

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

source code → instrumented binary → running program

Compile → In memory → Execution

Instrumentation → Monitoring and perforation
Loop perforation
Our contribution

Under review.
Approximate Loop Unrolling

• **Forgiving zone: for loops**
  - (i) have an induction variable incremented by a constant stride,
  - (ii) contain an array assignment inside their body, and
  - (iii) have an indexing expression of the array assignment that is value-dependent on the loop’s induction variable.

• **Domain-specific accuracy metric**

• **Approximation strategy**
  - Nearest neighbour
  - Linear regression

• **Determine performance trade-off**
  - Speed and energy
for ( int i = 0; i < N ; i++ ) {
  phase = phase + phaseInc;
  double v = Math.sin(phase) * ampl;
  A[i] = v;
}
for ( int i = 0; i < N ; i+= 2 ) {
    phase = phase + phaseInc;
    double v = Math.sin(phase) * ampl;
    A[i] = v;

    // UNROLLED ITERATION:
    phase = phase + phaseInc;
    double v1 = Math.sin(phase)*ampl;
    A[i + 1] = v1;
}
for ( i = 0; i < N - 1; i += 2 )
{
    phase = phase + phaseInc;
    double v = Math.sin(phase) * ampl;
    A[i] = v;
    phase = phase + phaseInc;
    //v1=Math.sin(phase)*ampl; <-- REMOVED
    A[i + 1] = A[i];
}

// Guard Loop:
for (j = i; j < N; j++)
{
    phase = phase + phaseInc;
    double v = Math.sin(phase) * ampl;
    A[i] = v;
}

phase = phase + phaseInc;
double v0 = Math.sin(phase) * ampl;
A[0] = v0;
int i = 1;
if ( N > 2 )
   for (i = 2; i < N - 1; i += 2 ) { // <-- Main Loop
       phase = phase + phaseInc;
       //v=Math.sin(phase)*ampl; <-- CODE REMOVED
       //Exact iteration:
       phase = phase + phaseInc;
       double v = Math.sin(phase) * ampl;
       A[i] = v;
       //Approximate iteration:
       A[i - 1] = (A[i] + A[i-2]) * 0.5f;
   }
// Guard Loop:
for (j = i; j < N; j++) {
   phase = phase + phaseInc;
   double v = Math.sin(phase) * ampl;
   A[i] = v;
}
Case studies

<table>
<thead>
<tr>
<th>Case study</th>
<th>Domain</th>
<th>Accuracy</th>
<th>#candidate loops</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSyn</td>
<td>Sound synthesizer</td>
<td>PEAQ</td>
<td>8</td>
</tr>
<tr>
<td>OpenImaJ</td>
<td>Image analysis</td>
<td>Dice</td>
<td>16</td>
</tr>
<tr>
<td>Lucene</td>
<td>Text mining</td>
<td>Rank of hit</td>
<td>9</td>
</tr>
<tr>
<td>Smile</td>
<td>Machine learning</td>
<td>Recall</td>
<td>12</td>
</tr>
</tbody>
</table>
Effectiveness of Approximate Loop Unrolling

<table>
<thead>
<tr>
<th>Project</th>
<th>Loops</th>
<th>Good</th>
<th>Rec. Loops</th>
<th>Good</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSyn</td>
<td>8</td>
<td>7 (88%)</td>
<td>7</td>
<td>7 (100%)</td>
</tr>
<tr>
<td>OpenImaJ</td>
<td>16</td>
<td>7 (44%)</td>
<td>8</td>
<td>7 (75%)</td>
</tr>
<tr>
<td>Lucene</td>
<td>9</td>
<td>5 (56%)</td>
<td>6</td>
<td>6 (100%)</td>
</tr>
<tr>
<td>Smile</td>
<td>12</td>
<td>5 (42%)</td>
<td>5</td>
<td>5 (100%)</td>
</tr>
<tr>
<td>Total</td>
<td>45</td>
<td>24 (53%)</td>
<td>26</td>
<td>24 (92%)</td>
</tr>
</tbody>
</table>
Comparison with Loop Perforation

<table>
<thead>
<tr>
<th></th>
<th>Approx. Unrolling</th>
<th>Tie</th>
<th>Loop perforation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Better accuracy</td>
<td>29</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Higher speedup</td>
<td>14</td>
<td>0</td>
<td>31</td>
</tr>
<tr>
<td>Lower energy consumption</td>
<td>16</td>
<td>0</td>
<td>29</td>
</tr>
</tbody>
</table>
Good example

```java
for (int i = 0; i < N; i++) {
    float sum = 0.0F;
    for (int j = 0, jj = kernel.length - 1; j < kernel.length; j++, jj--)
        sum += buffer[i + j] * kernel[jj];
    buffer[i] = sum;
}
```
Another one

```java
for (int i = 0; i < (imp.length); i++)
    importance[i] += imp[i];
```
Counter example

```c
for (int l = 0; l < (k); l++)
    falseCount[l] = count[l]) - (trueCount[l];
```
Conclusion

• Approximate computing trades accuracy for performance
• Approximate Loop Unrolling targets counted loops
• Implementation in the C2 JIT compiler