“A Source-Level Estimation and Optimization Methodology for the Execution Time and Energy Consumption of the Embedded Software”

Daniele Paolo Scarpazza
Politecnico di Milano
January 31st, 2006
This thesis at a glance:

• **Estimation:**
  - designers frequently need to estimate the \{time, energy\} consumption of significant clusters of operations;
  - current approaches (ISS, STA, SLI) do not solve the problem effectively;
  - we propose a new method (SLE) which is flexible, fast, accurate

• **Optimization:**
  - exploring source-level optimizing transformation is a slow task
  - many approaches involve ISS
  - we propose a new flow which is short-loop, scalable, modular
Estimation
Previous approaches are inadequate

- **Static Timing Analys (STA)** cannot deal with dynamism:
  - its main objective is the determination of the WCET
  - cannot deal with dynamic features:
    unbounded loops, recursion, dynamic fn ref;
  - unfortunately, code is becoming more and more dynamic
    (e.g. object based video coding, wireless ad-hoc networks, ...)

- **Instruction-Set Simulation (ISS)** is slow and at a low level:
  - it is 10k-100k times slower than application execution;
  - provides estimate at assembly level whereas developer works at source level;
  - estimates are difficult to interpret: not much helpful for optimization:
    (deep pipelines, superscalarity, wide-issue, speculation, branch prediction, ...)

- ISS + gprof provide estimates only at a function level

- *Atomium/PowerEscape* is source-level, but only for memory aspects

- *SoftExplorer* is a static technique
  - user interaction required to determine loop iterations: unthinkable for real sized projects

- Compilation-based approaches do not provide link to source level

- *SIT* is source level (good!) but still unable to resolve chosen clusters

- Black-box techniques do not provide any link with code
What we do, and others can't

- Motivational example: we consider a sample fragment of real code (FFT implementation, [Guthaus01])

```c
for (i=rev=0; i < NumBits; i++)
{
    rev = (rev << 1) | (index & 1);
    index >>= 1;
}
```

- After the analysis, we provide estimates for the individual operator instances

<table>
<thead>
<tr>
<th>Line</th>
<th>Time</th>
<th>Time(%)</th>
<th>Energy</th>
<th>Energy(%)</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>74</td>
<td>2.030 ms</td>
<td></td>
<td>980.357 uJ</td>
<td></td>
<td><code>for (i=rev=0; i &lt; NumBits; i++)</code></td>
</tr>
<tr>
<td>75</td>
<td>0.000 s</td>
<td></td>
<td>0.000 J</td>
<td></td>
<td><code>{</code></td>
</tr>
<tr>
<td>76</td>
<td>3.796 ms</td>
<td></td>
<td>2.137 mJ</td>
<td></td>
<td>`rev = (rev &lt;&lt; 1)</td>
</tr>
<tr>
<td>77</td>
<td>1.265 ms</td>
<td></td>
<td>712.279 uJ</td>
<td></td>
<td><code>index &gt;&gt;= 1;</code></td>
</tr>
<tr>
<td>78</td>
<td>0.000 s</td>
<td></td>
<td>0.000 J</td>
<td></td>
<td><code>}</code></td>
</tr>
</tbody>
</table>

- Currently, no other method can provide this detailed results
- Estimation at the source-level is 10,000 x faster than an ISS
How we perform estimation

Input source code

```c
if ( (a && (b < c+d)) && e && g && (h || i) ) && j) {
    d = (a == b+c);
} else {
    g = e = f << 2;
}
```

Abstract syntax tree

Figure break-up for node 17

Single-execution cost
\( c_{17} = 1 \text{ LogicTop} \)

Execution count
\( n_{17} = 4327 \)

Execution cost
\( C_{17} = n_{17} \cdot c_{17} = 4327 \text{ LogicTop} \)

Execution cost
\( C_{17} = n_{17} \cdot c_{17} = 4327 \text{ alul} + 2163.5 \text{ jump} \)

Execution cost
\( C_{17} = n_{17} \cdot c_{17} = (1.311 \text{ ms}, 471.8 \text{ mJ}) \)

Atoms

Abstract instructions

Target Platform Characterization
\( \cdots \) alul = (178 mA, 1.715 cycles)
\( \cdots \) jump = (170 mA, 1.0 cycles)
\( \cdots \) = ...

Abstract translation model
\( \cdots \)
LogicLeaf = 1 jump
LogicTop = 1 alul + 0.5 jump
Switch = 2 alul + 1
jump = 1 jump
\( \cdots \) = ...

Time and energy

Target Platform Characterization
\( \cdots \) alul = (178 mA, 1.715 cycles)
\( \cdots \) jump = (170 mA, 1.0 cycles)
\( \cdots \) = ...

Input source code

```
if ( (a && (b < c+d)) && e && g && (h || i) ) && j) {
    d = (a == b+c);
} else {
    g = e = f << 2;
}
```
The cost of syntax elements

- Step 1 (Analysis) associates a single-execution cost $c(i)$ to each syntax node, expressed as sum of atoms.
- The cost is due to 3 contributions: $c(i) = ci(i) + cf(i) + cc(i)$
- Contributions are calculated by an attribute grammar over the AST.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Name</th>
<th>Defined for</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>synthesized constancy</td>
<td>expressions</td>
</tr>
<tr>
<td>$e$</td>
<td>synthesized constant value</td>
<td>expressions</td>
</tr>
<tr>
<td>$t$</td>
<td>synthesized real result type</td>
<td>expressions</td>
</tr>
<tr>
<td>$v$</td>
<td>inherited valueness</td>
<td>expressions</td>
</tr>
<tr>
<td>$r$</td>
<td>inherited restricted result type</td>
<td>expressions</td>
</tr>
<tr>
<td>$f$</td>
<td>inherited translation flavor</td>
<td>expressions and statements</td>
</tr>
<tr>
<td>$ci$</td>
<td>synthesized inherent cost</td>
<td>expressions and statements</td>
</tr>
<tr>
<td>$cc$</td>
<td>synthesized conversion cost</td>
<td>expressions and statements</td>
</tr>
<tr>
<td>$cf$</td>
<td>inherited flow control cost</td>
<td>expressions and statements</td>
</tr>
<tr>
<td>$c$</td>
<td>synthesized total cost</td>
<td>expressions and statements</td>
</tr>
</tbody>
</table>
Estimation: the tool flow

1. Step 1: Analyzing
   - decorated syntax tree

2. Step 2: Instrumenting
   - instrumented source code

3. Step 3: Compiling
   - instrumented object code

4. Step 4: Linking

5. Step 5: Running the instrumented executable
   - execution counts
   - kernel instr. cost model

6. Step 6: Post-processing
   - line-by-line energy statistics

Key:
- Data
- Tool
- Library
Results: accuracy and speed

- **Experimental Setup**
  - comparison against SimIt-Arm (cycles) [Qin03]
  - current figures from JouleTrack (energy) [Sinha01]
  - modelling for SA1100, 206 MHz, 1.5 V
  - 24 benchmark from MiBench [Guthaus01]

- **Accuracy**
  - avg modulo error =15% E, <17% T
  - coefficients of correlation = 0.978 E, 0.960 T

- **Speed**
  - simulation times 10,350 x shorter than ISS
  - simulation only 2.2x slower than normal execution

- **Robustness**
  - 24/24 MiBench projects successfully processed
Optimization
A short-loop exploration methodology is needed
What we do and others can't

- Import a project
- Analyze it
- Get source-level optimization directives, generated at the source level
- Apply them and see the result
# What a short-loop methodology needs

<table>
<thead>
<tr>
<th>Problem</th>
<th>Task</th>
<th>Additional Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>source code analysis</td>
<td>analyze the code and determine which are the critical sections</td>
<td>analysis must be performed at source level; profile data must be available at source level</td>
</tr>
<tr>
<td>influence metrics</td>
<td>determine what is the gain in applying a trf over a section</td>
<td>SLE is the first approach</td>
</tr>
<tr>
<td>transformation steering</td>
<td>decide which transformation to apply and where</td>
<td>Many exist, e.g. [Brandolese03]</td>
</tr>
<tr>
<td>transformation application</td>
<td>apply transformation on the source code</td>
<td>steering engine must operate automatically on source-level data provided by above analysis and metrics</td>
</tr>
</tbody>
</table>

- SLE is the first approach
- Many exist, e.g. [Brandolese03]
- steering engine must operate **automatically** on source-level data provided by above analysis and metrics
- None exist!
- e.g. [suif94]
How we perform transformation steering

- We employ a Network of Fuzzy Rules
- It is a modified version of a neural network; differences:
  - weights and connections model explicitly transformation influence metrics;
  - each rule (~neuron) accesses complete syntactic and profiling information;
- Base component: NFR rule
- Advantages:
  - scalable \( O(n \cdot Q) \)
  - modular (no IP disclosed)
Experimental results

- Modelled transformations:
  1) loop unrolling
  2) function inlining
  3) function replacement with macro
  4) common subexpression elimination
  5) strength reduction
  6) type conversion elimination
  7) standard library function factorization
  8) memory allocation factorization
  9) argument passing via pointer
  10) function specialization

- Benchmarks:
  4 applications (audio filter, hough transform, dijkstra, FFT);

- Energy gains: 5 – 22 %
- Time gains: 8 – 20 %