A source-level estimation and optimization methodology for execution time and energy consumption of embedded software

Daniele Paolo Scarpazza
PhD candidate,
Dipartimento di Elettronica e Informazione,
Politecnico di Milano

scarpaz@scarpaz.com

Doctoral dissertation
May 18th, 2006
The main ideas:

1. Need: Why this research was needed
designers need fast, dynamic, fine-detail, source-level estimation techniques; current techniques do not satisfy these requirements;

2. Theory: How my technique works
I assign a (time-, energy-) cost to each AST node in a C program;

3. Results: The technique is accurate and fast
an ANSI-C compliant tool flow implementation is available; mean modulo error within 8%; 10,000x faster than ISS;

4. Uses and developments
optimization: an automated transformation exploration flow is available; extension for VWR architectures is ready, for VLIW coming; prospective extension to C++ language possible;
1. The need

- **1.1 Requirements:**
  designers need fast, dynamic, fine-detail, source-level techniques to estimate the energy consumed by their software;

- **1.2 Focus:**
  I focus on the core of single-issue CPUs (no memory hierarchy, no VLIW, ...)

- **1.3 State of the Art:**
  current techniques do not satisfy the above requirements;
1.1. Requirements

1. fast
2. dynamic
3. source-level
4. fine-detail
1.1. Requirements

1. **fast**
   - the size and complexity of modern embedded applications is increasing quickly;

2. **dynamic**
   - simulating non-toy apps at the circuit level or gate level is unaffordable;

3. **source-level**
   - instruction-set simulation is also unaffordable for apps of sufficient complexity (e.g. video decoders);

4. **fine-detail**
   - whichever technique is cycle-accurate, or close to cycle accuracy is doomed to obsolescence very soon;

   - estimation techniques with a high performance are needed, even at the expenses of inferior accuracy;
1.1. Requirements

1. fast
   • modern applications are becoming more and more dynamic in nature;

2. dynamic
   • the behavior of multimedia en-/de-coders depends more and more on the contents of the streams they process;

3. source-level

4. fine-detail
   • the variability in workload is high and increasing;
   • the gap between typical and worst case is very large;
   • static techniques are worst-case techniques, and lead to expensive, oversized systems which are underutilized most of the time;

![Graphs showing: uncompressed, constant resolution, I,P,B (MPEG-2), object-based encoding (MPEG-4).]
1.1. Requirements

1. **fast**
   - many energy estimation flows operate at the assembly level, but designers do not code in assembly any more;

2. **dynamic**
   - designers use high-level languages instead, estimation flows should provide information at the same abstraction level;

3. **source-level**
   - compilation is a (more and more) complex process; lot of skill and experience required to relate instruction-level estimates to the source-level causes;

4. **fine-detail**
   - source-level optimizing transformations have been showed to lead to the highest gains; only source-level analysis can guide them;
1.1. Requirements

1. **fast**
   - most of the time and energy are spent in small computational kernels;

2. **dynamic**
   - “small” is much smaller than a program and a function, potentially smaller than inner loops;

3. **source-level**
   - many estimation techniques (even source-level ones) cannot “look inside functions”

4. **fine-detail**
   - fine-detail analysis techniques are needed;
     “fine-detail” = individual operator instance;
What I mean by fine-detail source-level
1.3. Current techniques are not ok

- **Static Timing Analysis (STA)** cannot deal with dynamism:
  - its main objective is the determination of the WCET
  - cannot deal with dynamic features:
    unbounded loops, recursion, dynamic function reference;
  - 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
1.3. Current techniques are not ok

- Static Timing Analysis (STA) techniques cannot deal with dynamism;  
- Instruction-Set Simulation (ISS) is slow and at a low level;  
- ISS + gprof provide estimates only at a function level;  
- Atomium/PowerEscape is source-level, but only for memory aspects (not our focus);  
- SoftExplorer is a static technique;  
- 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 source code;
2. How my technique works

• 2.1 Divide and conquer:

\[ C_i = n_i \cdot c_i \]

- Cost of executing the i-th node in the AST
- Execution count
- Single-execution cost

• 2.2 Determine single-execution costs via an attribute grammar, founded on an abstract translation model

• 2.3 Determine execution counts by instrumenting the original program in an efficient way and running the instrumented program over real input data
2.1. Divide and conquer: \( C_i = n_i \cdot c_i \)
2.2. Determining single-execution costs

- the cost is due to 3 contributions:
  - inherent cost
  - conversion costs
  - flow-control cost

- I compute costs with an attribute grammar:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Name</th>
<th>Defined for which AST nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>synthesized</td>
<td>total cost</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions and statements</td>
</tr>
<tr>
<td>ci</td>
<td>synthesized</td>
<td>inherent cost</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions and statements</td>
</tr>
<tr>
<td>cc</td>
<td>synthesized</td>
<td>conversion cost</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions and statements</td>
</tr>
<tr>
<td>cf</td>
<td>inherited</td>
<td>flow control cost</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions and statements</td>
</tr>
<tr>
<td>k</td>
<td>synthesized</td>
<td>constancy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions</td>
</tr>
<tr>
<td>e</td>
<td>synthesized</td>
<td>constant value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions</td>
</tr>
<tr>
<td>t</td>
<td>synthesized</td>
<td>real result type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions</td>
</tr>
<tr>
<td>v</td>
<td>inherited</td>
<td>valueness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions</td>
</tr>
<tr>
<td>r</td>
<td>inherited</td>
<td>restricted result type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions</td>
</tr>
<tr>
<td>b</td>
<td>synthesized</td>
<td>register-boundedness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions</td>
</tr>
<tr>
<td>f</td>
<td>inherited</td>
<td>translation flavor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expressions and statements</td>
</tr>
</tbody>
</table>
2.2. Determining single-execution costs
Why all these attributes?

- Full C type system needed (attribute \( t \))
  - cost of operations depend on the operands' types
  - conversions depend on types;

- Full constant expression evaluation needed (attributes \( k,e \))
  - constant expressions are resolved at static time (no translation, no runtime cost)
  - constant expressions appear in type declarations, and influence operator costs;

- Example:

```c
struct tag
{
    int  field1;
    char field2 [sizeof(type_x)*5];
} s1, s2;

int main()
{
    ...
    s1 = s2;  // s1 = [struct tag] W(t) = ...
    ...
}
```
Why attribute $r$ (restricted type) is needed

$\ast a = s;$

$(*a).m = b;$

(the cost of a star operator depends on its type)

(not really!)
Why attribute \( r \) (restricted type) is needed

\[ *a = s; \]

\[ (*a).m = b; \]

(the cost of a star operator depends on its type)  
(not really!)
Why attribute \( v \) (valueness) is needed

\[ *a = \ldots; \]

\[ \ldots = *a; \]
Why attribute $v$ (valueness) is needed

$$****p = ***q;$$

double **** p
double *** q;

$v = L$

$t = [\text{double}]$
$r = [\text{double}]$
$ci = 1 \text{LValueStar} + 1 \text{LValueStarNext}$

$v = R$

$t = [\text{pointer}][\text{double}]$
$r = [\text{pointer}][\text{double}]$
$ci = 1 \text{RValueStar}$

$v = R$

$t = [\text{pointer}][\text{pointer}][\text{double}]$
$r = [\text{pointer}][\text{pointer}][\text{double}]$
$ci = 1 \text{RValueStar}$

$v = R$

$t = [\text{pointer}][\text{pointer}][\text{pointer}][\text{double}]$
$r = [\text{pointer}][\text{pointer}][\text{pointer}][\text{double}]$
$ci = 1 \text{RValueStar}$

$v = R$

$t = [\text{pointer}][\text{pointer}][\text{pointer}][\text{pointer}][\text{double}]$
$r = [\text{pointer}][\text{pointer}][\text{pointer}][\text{pointer}][\text{double}]$
$ci = 1 \text{RValueStar}$
The dot operator's anomaly

\[ (*a).n.m = b; \]

The dot operator propagates valueness and restricted type to its left child.
2.3. Determining execution counts

- optimal strategy to select probe insertion points
  - I insert only one probe per each generalized basic block (g.b.b.);
  - a g.b.b. is a maximal set of nodes which are all executed the same number of times (possibly larger than basic blocks); example:

```c
/*section 1*/ ...
if (f())
{
    /*section 2*/
    ...
} else {
    /*section 3*/
    ...
}
/*section 4*/
...
```

- transparent, probe-inserting source-to-source transformations:
  - expressions: `e ( __profile__(137), e )`
  - statements: `s; { __profile__(137); s; }`
  - functions: `int f(args) { __profile__(151); { ... } } __profile__(152);`
3. The technique is accurate and fast

3.1 ANSI-C compliant flow implementation

3.2. New experiments – Setup:

- Simulator: SimIt-ARM v2.0.3 with cache latency = 0  [Qin03]
- Platform: SA-1100 @ 206 MHz, 1.5 Vdd
- Parameters: avg. currents for each instruction, from JouleTrack  [Sinha01]
- Compiler: gcc v2.95 -O2/-O3
- Benchmarks: from MiBench  [Guthaus01]

3.3. New experiments – Results:

- accuracy: average modulo error within 8%;
  correlation between estimates and reference > 0.995;
- performance: simulation times 10,350 times shorter than ISS;
  simulation only 2.2x slower than normal execution;
3.1 Tool flow

Program sources

Step 1: Analyzing
- decorated syntax tree

Step 2: Instrumenting
- instrumented source code

Step 3: Compiling
- instrumented object code

Step 4: Linking

Step 5: Running the instrumented executable
- execution counts
- abstract instr. costs

Step 6: Post-processing
- time, energy statistics

Key:
- data
- tool
- library
- pseudo-compiler

~gcc
### 3.3. Accuracy results

<table>
<thead>
<tr>
<th>SimIt</th>
<th>e3tools</th>
<th>Estimation error</th>
</tr>
</thead>
<tbody>
<tr>
<td>E (mJ)</td>
<td>T (ms)</td>
<td>E (mJ)</td>
</tr>
<tr>
<td>adpcm-s</td>
<td>46,1</td>
<td>41,9</td>
</tr>
<tr>
<td>adpcm-l</td>
<td>910,2</td>
<td>722,1</td>
</tr>
<tr>
<td>bitcount-s</td>
<td>65,7</td>
<td>55,0</td>
</tr>
<tr>
<td>bitcount-l</td>
<td>981,9</td>
<td>977,1</td>
</tr>
<tr>
<td>blowfish</td>
<td>1067,0</td>
<td>748,3</td>
</tr>
<tr>
<td>CRC32</td>
<td>38,3</td>
<td>35,4</td>
</tr>
<tr>
<td>FFT-s</td>
<td>207,9</td>
<td>207,1</td>
</tr>
<tr>
<td>FFT-l</td>
<td>3213,2</td>
<td>3264,8</td>
</tr>
<tr>
<td>IFFT-s</td>
<td>205,1</td>
<td>207,3</td>
</tr>
<tr>
<td>IFFT-l</td>
<td>3181,8</td>
<td>3266,2</td>
</tr>
<tr>
<td>jpeg</td>
<td>87,9</td>
<td>91,2</td>
</tr>
<tr>
<td>rijndael</td>
<td>63,8</td>
<td>71,4</td>
</tr>
<tr>
<td>sha-s</td>
<td>22,1</td>
<td>21,9</td>
</tr>
<tr>
<td>sha-l</td>
<td>229,4</td>
<td>224,7</td>
</tr>
</tbody>
</table>

**Quality of result:**

- $\rho(E, \hat{E}) = 0,9960$, $|E - \hat{E}| = 7,49\%$
- $\rho(T, \hat{T}) = 0,9987$, $|T - \hat{T}| = 5,65\%$, 

- $\hat{E}$ and $\hat{T}$ are the estimated values.
3.3. Accuracy results

- Sim-It (reference)
- e3tools

Bar chart showing energy consumption (mJ) for various benchmarks and algorithms, including adpcm, bit-count, blowfish, CRC32, FFT, IFFT, jpeg, rijndael, sha, sha-l, and susan.
4. Uses & developments

1. Opt.: Automated source-code optimization
2. VWR: support for VWR architectures
3. VLIW: support for VLIW architectures
4. C++: estimating C++ sources
4.2. Uses & developments: Optimization

   - The need for source-level optimization:
     - applications are becoming larger and larger;
     - the degree of optimization influences feasibility, performance, usability, cost and commercial success of the product;
     - current optimization techniques involve a long exploration loop, with many, slow steps;
   - Goal:
     - an automatic technique for the source-to-source optimizing transformation steering
     - steering:
       where to optimize?
       which transformation to apply?
   - Limitations:
     suitable for local transformation
     with loose mutual interaction
4.2. Uses & developments: Optimization

2. VWR
3. VLIW
4. C++

Long vs. short exploration loop:

Previous approaches:
- Initial source code
  - Front-end
  - Influence metrics
  - Transformation steering
  - Transformation application
  - Optimized source code
  - Compiler
  - Optimized object code
  - Instruction set simulator
  - Instruction-level profiles

This approach:
- Initial source code
  - Source-level estimation
  - Source-level profiles
  - Influence metrics
  - Transformation steering
  - Transformation application
  - Optimized source code
  - Compiler
  - Optimized object code
4.2. Uses & developments: Optimization

What the new approach offers:

- Import a project
- Analyze it
- Get source-level optimization directives, generated at the source level
- Apply them and measure the result

File | Time | Energy
--- | --- | ---
image.c | 21.638 µs | 16.561 µJ
main.c | 28.962 µs | 21.158 µJ
vertfilter.c | 377.672 ms | 421.048 mJ
(glibc) | 305.800 µs | 622.000 µJ
TOTAL | 378.029 ms | 421.708 mJ
## 4.2. Uses & developments: Optimization

### 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>steering engine must operate automatically on source-level data provided by above analysis and metrics</td>
</tr>
<tr>
<td>transformation application</td>
<td>apply transformation on the source code</td>
<td>None exists!</td>
</tr>
</tbody>
</table>

- **Problem**: Various methodologies require different aspects to be considered.
- **Task**: Each problem has a corresponding task to address.
- **Additional Requirements**: Requirements for the task to be effective.

- **SLE**: Short Loop Engine, the first approach to short-loop methodology.
- **Brandolese03**: A reference for short-loop methodologies.
- **SUIF94**: Another reference, highlighting the absence of existing solutions.
- **None exists!**: No existing methodologies for short-loop optimization, with SLE as the first attempt.

### Problem

- **Source Code Analysis**: Analysis of the code to identify critical sections.
- **Influence Metrics**: Determining the benefits of applying transformations.
- **Transformation Steering**: Deciding which transformations to apply.
- **Transformation Application**: Applying the transformations to the source code.

### Task

- **Source Code Analysis**: Analyze the code to find critical sections.
- **Influence Metrics**: Assess the impact of transformations.
- **Transformation Steering**: Choose the right transformations.
- **Transformation Application**: Implement the chosen transformations.

### Additional Requirements

- **Source Code Analysis**: Analysis must be performed at source level; profile data must be available.
- **Influence Metrics**: Steering engine must operate automatically on source-level data.
- **Transformation Application**: Requires the use of existing methodologies like SLE.
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)
4.2. Uses & developments: Optimization

   - Results:
     - Energy reduction: \(-5.1\% - 22.0\%\)
     - Execution time reduction: \(-7.8\% - 22.3\%\)

2. VWR

3. VLIW

4. C++
4.2. Uses & developments: VWR

   - Very wide register (VWR) architectures achieve extreme low power via:

2. VWR
   - a wide data-path (e.g. 256 bit) and very wide registers (e.g. 2048 bit) with SIMD instructions;
   - a software controlled scratchpad in place of a L1 cache;
   - a loop buffer (32 instructions);

3. VLIW

4. C++
   - We have augmented our technique with features to:
     1. map code to different executors
     2. mark concurrent code
     3. define intrinsics to map scratchpad transfer costs;
     4. define intrinsics for SIMD operations;

     support for simulation and estimation at the same time;
     all these features are ANSI C-transparent;
Multiple CPUs

- Now, users can define multiple CPUs, each with distinct abstract assembly parameters and operating conditions;

- To map code on a different CPU, use a pragma:
  
  ```c
  #pragma e3tools CPU n
  ```

- Example:

  ```c
  int main() {
      int i, j;
      #pragma e3tools CPU 1
      for (i=0; i<20; i++) {
          printf("This code is executed on CPU 1");
      }
      #pragma e3tools CPU 0
      for (j=0; j<20; j++) {
          printf("This code is executed on CPU 0");
      }
      printf("This code is also executed on CPU 0");
      return 0;
  }
  ```
4.2. Uses & developments: VWR

Concurrent code

- create split/join paths, using a pragma before a compound statement:
  ```c
  #pragma e3tools concurrent
  ```
- All the statements inside this block will start concurrently; implied rendez-vous at the end of the block (simulation remains additive)
- Example:

  ```c
  ...
  #pragma e3tools Concurrent
  {
    #pragma e3tools CPU 0
    printf("I run on CPU 0");
    #pragma e3tools CPU 1
    for (j=0; j<20; j++) {
      printf("I run on CPU 1");
    }
    #pragma e3tools CPU 2
    {
      printf("Everything inside this block...");
      ...
      printf("... will run on CPU 2");
    }
  }
  ...
  ```
4.2. Uses & developments: VWR

User definable-intrinsics

- prepend a “#pragma e3tools intrinsic” directive;
- provide code implementing the simulation semantics (e.g. perform a real complex multiplication, if needed)
- provide declaration for an atom with the same name: \( \text{ComplexMul} = 2 \text{ rfrd} + 4 \text{ aluh} + 2 \text{ alul} + 1 \text{ rfrw}; \)
- Example:

```cpp
#include <e3tools.h>

#pragma e3tools intrinsic
complex ComplexMul(complex a, complex b)
{
    complex result;
    result.real = (a.real * b.real - a.imag * b.imag);
    result.imag = (a.real * b.imag + a.imag * b.real);
    return result;
}

int main(int argc, char** argv)
{
    ...
    for (a = 0; a < CHAN_HEIGHT; a++) {
        ...
        Out[a][index] = ComplexAddShr(
            ComplexMul(F[a*2][0], Data[a][index]),
            ComplexMul(F[a*2+1][0], Data[a+52][index]), DEC_SDM );
        ...
    }
    ...
}
```
4.3. Uses & developments: VLIW

1. Opt. Extending the e3tools to VLIW architectures.

2. VWR Goals:

3. VLIW • trace-based: model exactly the per-trace compilation results of VLIW compilers;

4. C++ • incremental rebuild: rebuild only the intermediate products actually needed by changes made in the source code, architecture, input data;

• keep the current efficiency;
4.3. Uses & developments: VLIW

2. VWR
3. VLIW
4. C++

The new flow.
4.3. Uses & developments: VLIW

Rewriting code to generate all the traces:

- conditional expressions

\[
\begin{align*}
\text{... ( condition ? expression1 : expression2 ) ...} \\
\text{... ( (condition, TRUE) ? expression1 : expression2 ) ...} \\
\text{... ( (condition, FALSE) ? expression1 : expression2 ) ...}
\end{align*}
\]

- if statements:

\[
\begin{align*}
\text{if (condition)} \\
\text{    { ... /* then branch */} else { ... /* else branch */}}
\end{align*}
\]

- switch statements:

\[
\begin{align*}
\text{switch (condition)} \\
\text{    { case value1: /* code for value 1*/} case value2: /* code for value 1*/} \\
\text{    { ... default: /* code for value 1*/}} \\
\text{    { switch (value1) \\
\text{        { case value1: /* code for value 1*/} case value2: /* code for value 1*/} \\
\text{        { ... default: /* code for value 1*/}} \\
\text{        { switch (value2) \\
\text{            { case value1: /* code for value 1*/} case value2: /* code for value 1*/} \\
\text{            { ... default: /* code for value 1*/}} \\
\text{            { switch (valueN) \\
\text{                { case value1: /* code for value 1*/} case value2: /* code for value 1*/} \\
\text{                { ... default: /* code for value 1*/}} \\
\end{align*}
\]

Note: a table is required to store all the possible cases (<=256 by std) and select one among the unused ones.
4.3. Uses & developments: VLIW

   - Trace-based profiling: how many times each trace was executed?
   - It can be solved with current, node-based instrumentation technique

2. VWR
   - Need to determine trace counts from node counts

3. VLIW
   - Node counts
   - Collapsing basic blocks
   - Enumerating blocks per each trace
   - Solving the corresponding equations

\[
\begin{align*}
  t_1 + t_2 + t_3 &= n_1 \\
  t_1 &= n_2 \\
  t_1 + t_2 &= n_3 \\
  t_2 + t_3 &= n_4 \\
  t_3 &= n_5 \\
  t_1 + t_2 + t_3 &= n_6
\end{align*}
\]

\[
\begin{bmatrix}
  1 & 1 & 1 & 22 \\
  1 & 0 & 0 & 20 \\
  1 & 1 & 0 & 21 \\
  0 & 1 & 1 & 2 \\
  0 & 0 & 1 & 1 \\
  1 & 1 & 1 & 22
\end{bmatrix}
\]

\[
\text{At} = b
\]

```octave
# octave script
>a = [1 1 1; 1 0 0; 1 1 0; 0 1 1; 0 0 1; 1 1 1]
b = [22; 20; 21; 2; 1; 22]
t = a \ b
```

\[
\begin{bmatrix}
  20.0 \\
  1.0 \\
  1.0
\end{bmatrix}
\]
4.4. Prospective extension to C++

   • Extending the technique to the C++ language is possible and involves reasonable effort;

2. VWR
   • Tasks required:

3. VLIW
   • lexer (28 new keywords, negligible effort);
   • parser: $213 \ll 560$ syntax rules;

4. C++
   • new type system and scoping rules (significant effort);
   • parser needs some semantic-level disambiguation techniques;
   • overloading / templates / late binding (current instrumentation technique is sufficient to determine which function has been actually called);
   • extension of theoretical abstract translation model (significant effort);
   • Required effort: 1 “me-year”
Reference: the POET project

Part (approx. 1/3) of WorkPackage 2 of project “POET”,
http://poet.offis.de

EU-funded integrated project IST-2000-30125,
Sep 2001 – Mar 2005;
Reference: the POET consortium

Consortium:
- OFFIS
- Politecnico di Torino
- Cefriel
- BullDAST
- ChipVision
- OSC
- ARM
- Alcatel
- Atmel
- Motorola

Research

EDAs Vendors

Users

WP1: Design flow

WP2: Software power estimation
- Instr. Level Estimation
- Source Level Estimation
- e'tools
- Library Level Estimation

WP3: Algorithm level power estimation

WP4: RTL power estimation

WP5: Optimizers

WP6: Integration Evaluation

WP7: Dissemination, Exploitation

WP8: Management
Selected Scientific Publications

- **Book chapters:**

- **Journal papers:**
  - with Carlo Brandolese, “A source-level software analysis methodology able to resolve clusters of operations and finer details”, Journal on Low-power Electronics (JOLPE) [accepted];

- **Conference papers:**
  - with C. Brandolese, “A fast, dynamic, source-level and fine-detail technique to estimate the energy consumed by embedded software on single-issue processor cores”, CODES+ISSS’06, Seoul, Korea [submitted];
  - with P. Raghavan, D. Novo, C. Brandolese, F. Catthoor, D. Verkest, “Software Simultaneous Multi-Threading, a technique to exploit Task-level Parallelism to improve Instruction and Data-level Parallelism”, PATMOS’06, Montpellier, France [submitted];
The End.
~
Questions welcome.
Backup slides follow
What e³ tools can and cannot do

- The e³ tools perform source level estimation of the ALU and control flow contributions of {time, energy} consumption of a ANSI C program.

- They are NOT designed for data transfer and storage exploration and optimization (although: possible estimation for software-controlled memories, e.g. Feenecs SPM + VWR).

- In this sense, e³ tools are perfectly complementary with Atomium/PowerEscape.
4.3. Uses & developments: VLIW

Minimal incremental rebuild.
Example: when the input data changes:

2. VWR
3. VLIW
4. C++
4.3. Uses & developments: VLIW

   Example: when architecture changes:

   - Minimal incremental rebuild.
   - E, T assembly model

2. VWR
   - trace source generator
   - original C source code
   - input data
   - decorated AST
   - exact node profiles
   - flow graph solver
   - exact trace profiles

3. VLIW
   - C source code + LU + SIMD + IVR + fixed wordlength
   - per-trace compile [CRISP]
   - trace binary
   - assembly translation
   - instruction counter
   - cost accumulator
   - final E, T estimates

4. C++
4.3. Uses & developments: VLIW

   Minimal incremental rebuild.
   Example: when source code changes
2. VWR
3. VLIW
4. C++
User-definable models

- Parsing (1) is defined by the language;
- Cost association (2, in atoms) to syntax nodes:
  - theoretically founded, not user "serviceable"
  - see Chapter 4 of my Thesis;
    warning: implementation is not yet aligned with the theoretical developments!
- Mapping of atoms to abstract-instructions (3):
  - also theoretically founded on some assumptions
  - user can refine model: 
    `/scratch/scarpaz/poet/4.3/root/lib/compiler`
- Cost of abstract instructions (4):
  - must be characterized:
    `/scratch/scarpaz/poet/4.3/root/lib/tech/processor`
Atoms to abstract instructions:

See directories and associated files under:
/scratch/scarpaz/poet/4.3/root/lib/compiler

- \text{IntAdd} = 1 \text{ alul};
- \text{IntSub} = 1 \text{ alul};
- \text{IntMul} = 1 \text{ aluh};
- \text{BitwiseOperation} = 1 \text{ alul};
- \text{IntRelation} = 1 \text{ cmpl} + 1 \text{ jump};
- \text{IntImplicitRelation} = 1 \text{ cmpl} + 1 \text{ jump};
- \text{RValueStar} = 1 \text{ mvld};
- \text{LValueStar} = 1 \text{ mvst};
- \text{RLValueStar} = 1 \text{ mvld} + 1 \text{ mvst};
- \text{RValueStarNext} = 1 \text{ mvld} + 1 \text{ alul};
- \text{LValueStarNext} = 1 \text{ mvst} + 1 \text{ alul};
- \text{RLValueStarNext} = 1 \text{ mvld} + 1 \text{ mvst} + 1 \text{ alul};
- \text{Break} = 1 \text{ jump};
- \text{Continue} = 1 \text{ jump};
- \text{Goto} = 1 \text{ jump};
- \text{While} = 1 \text{ cjt};
- \text{WhileBody} = 1 \text{ cjn} + 1 \text{ jump};
- \text{Do} = -1 \text{ cjt} + 1 \text{ cjn};
- \text{DoBody} = 1 \text{ cjt};
- \text{For} = 1 \text{ cjt};
- \text{ForBody} = 1 \text{ cjn} + 1 \text{ jump};
Abstract instructions to time/energy

See directories and associated files under:

- **Cost of abstract instructions:**
  
  ```
  /scratch/scarpaz/poet/4.3/root/lib/tech/processor/arm7tdmi-new/default/kis.dat
  ```

<table>
<thead>
<tr>
<th>Abstract instruction</th>
<th>Average absorbed current (mA)</th>
<th>Average CPI (clock cycles)</th>
<th>Encoded instruction size (bytes) [future use]</th>
</tr>
</thead>
<tbody>
<tr>
<td>aluh</td>
<td>196</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>cmpl</td>
<td>178</td>
<td>0.950</td>
<td>0</td>
</tr>
<tr>
<td>cmph</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>call</td>
<td>170</td>
<td>7.430</td>
<td>0</td>
</tr>
<tr>
<td>mvst</td>
<td>229</td>
<td>22.0</td>
<td>0</td>
</tr>
<tr>
<td>mvld</td>
<td>196</td>
<td>0.75</td>
<td>0</td>
</tr>
<tr>
<td>jump</td>
<td>170</td>
<td>0.98</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Operating conditions:**
  
  ```
  /scratch/scarpaz/poet/4.3/root/lib/tech/processor/arm7tdmi-new/default/oc.dat
  ```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VDD</td>
<td>1.5</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>FCK</td>
<td>206.4</td>
<td>MHz</td>
<td></td>
</tr>
<tr>
<td>MAINI</td>
<td>0.0</td>
<td>uJ</td>
<td></td>
</tr>
<tr>
<td>MAINT</td>
<td>0.0</td>
<td>us</td>
<td></td>
</tr>
</tbody>
</table>
Practical usage of the tools

• Prepare your project:
  - must be ANSI C  (make sure it compiles with gcc  -ansi)
  - must have a Makefile and use gcc

• An experimental installation is available on pc3643:
  - ssh pc3643
  - bash
  - cd /scratch/scarpaz/poet/4.3
  - . fake.sh
  - cd /your-project-dir/
  - make clean
  - make
  - <run your project>
  - taylor -c gcc -t arm7tdmi *.e3.count
Loop pre-conditioning is needed

- **Issue:** Conditions may not be extracted inside loops
- **Solution:**
  - we assume that functions are compiled individually, and
  - we perform a loop preconditioning step
  - we do NOT perform condition extraction inside surviving loops
- **Loop conditioning:**
  - case 1) small loop body, few iterations:
    fully unroll the loop, perform condition extraction after unroll
  - case 2) small loop body, many/unpredictable iterations:
    partially unroll code
  - case 3) large body, few large conditional codes, few interactions with remaining code:
    function-export the code (pessimistic, acceptable under constraints)
  - case 4) large body, many large conditioned statements:
    group them together and function-export them cumulatively
- **Prototype implementation:**
  - SUIF2 tested successfully to unroll loops;
  - a modified version of current instrumentation tool can be used for loop body exportation;
Assumptions on the compiler:
- it is capable of basic constant folding
- it performs no interprocedural optimization;
- it generates code on a per-function basis;
- inline functions already expanded;

Issues ok:
- gotos,
- short circuit evaluation, ...
- conditions inside loop (preconditioning)

Open issues:
- exponential explosion:
  number of function traces is:
  \[
  \sum_{\text{functions}} 2^{N_{\text{if}}} \prod_{j=0}^{N_{\text{switch}}} \text{Choices}
  \]
  assuming per-function separation;
  otherwise even worse:
  \[
  \prod_{\text{functions}} 2^{N_{\text{if}}} \prod_{j=0}^{N_{\text{switch}}} \text{Choices}.
  \]

Development tasks:
- implementation of CEE:
  as an extension to e3tools/democritos;
- implementation of CR:
  as a modified version of e3tools/stradivari