“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

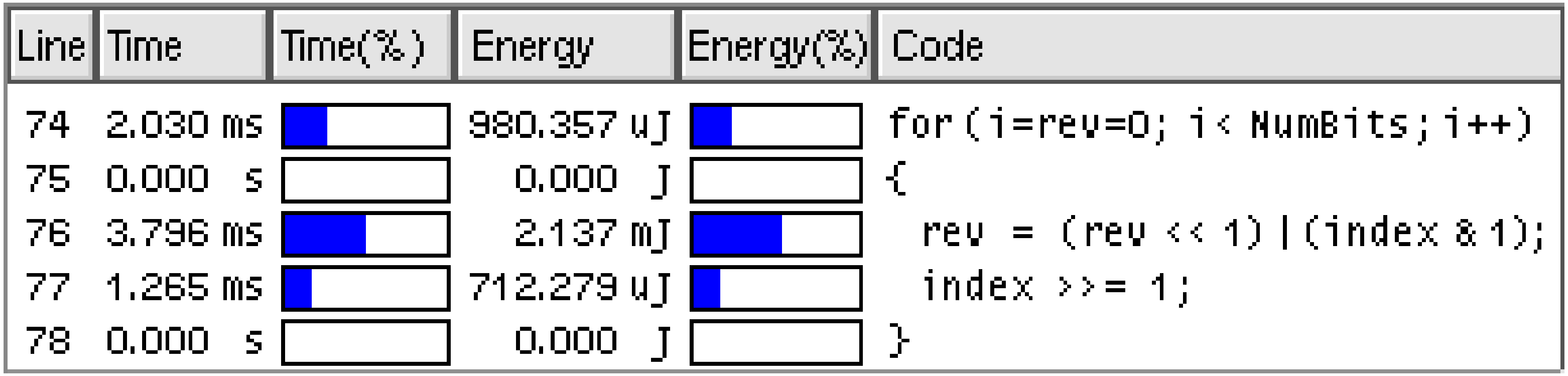
- Static Timing Analys (STA) cannot deal with dynamism: [Puschner89,..., Chen01]
  - its main objective is the determination of the WCET
  - cannot deal with dynamic features:
    unbounded loops, recursion, dynamic fn ref;
  - unfortunately, code is becoming more and more dynamic
    (e.g. object based video coding, wireless ad-hoc networks, ...)
- Instruction-Set Simulation (ISS) is slow and at a low level: [Brooks00, Sinha01, Qin03]
  - 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 [Simunic01]
- *Atomium/PowerEscape* is source-level, but only for memory aspects [Bormans99, Arnout05]
- *SoftExplorer* is a static technique [Senn02]
  - user interaction required to determine loop iterations: unthinkable for real sized projects
- Compilation-based approaches do not provide link to source level [Lajolo99]
- *SIT* is source level (good!) but still unable to resolve chosen clusters [Ravasi03]
- Black-box techniques do not provide any link with code [Muttreja04]

What we do, and others can't

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

```
74 for (i=rev=0; i < NumBits; i++)
75 {
76   rev = (rev << 1) | (index & 1);
77   index >>= 1;
78 }
```

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



Line	Time	Time(%)	Energy	Energy(%)	Code
74	2.030 ms		980.357 uJ		`for(i=rev=0; i< NumBits; i++)`
75	0.000 s		0.000 J		`{`
76	3.796 ms		2.137 mJ		`rev = (rev << 1)|(index & 1);`
77	1.265 ms		712.279 uJ		`index >>= 1;`
78	0.000 s		0.000 J		`}`

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

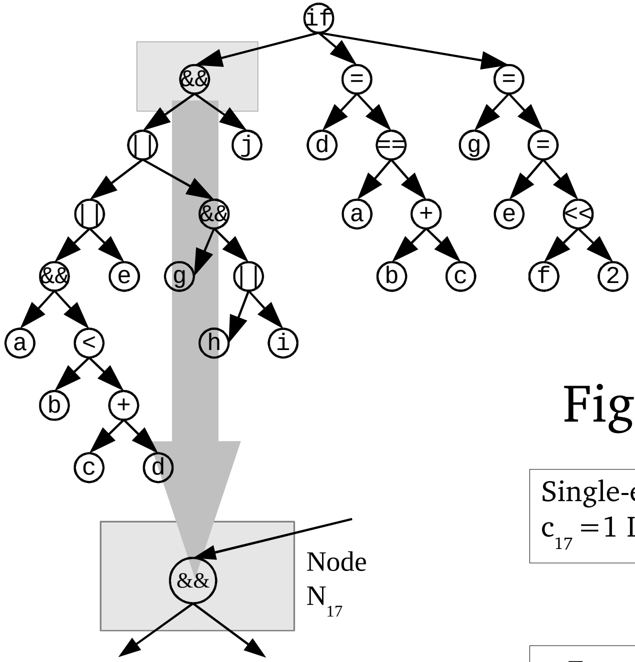
How we perform estimation

Input source code

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

Abstract syntax tree

Atoms



Abstract instructions

Abstract translation model

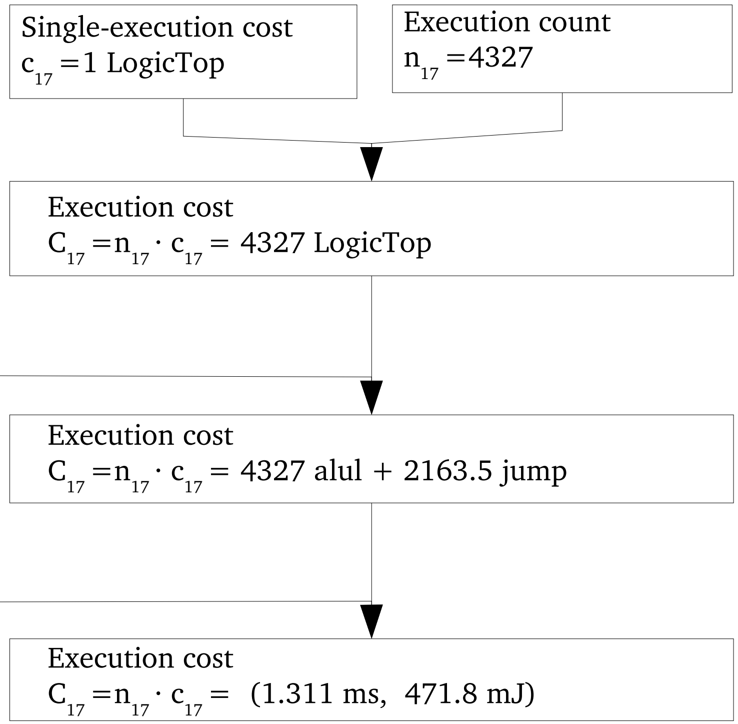
... = ...
LogicLeaf = 1 jump
LogicTop = 1 alul + 0.5 jump
Switch = 2 alul + 1 jump
If = 1 jump
... = ...

Time and energy

Target Platform Characterization

... = ...
alul = (178 mA, 1.715 cycles)
jump = (170 mA, 1.0 cycles)
... = ...

Figure break-up for node 17



Single-execution cost
$c_{17} = 1$ LogicTop

Execution count
$n_{17} = 4327$

Execution cost
$C_{17} = n_{17} \cdot c_{17} =$ 4327 LogicTop

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

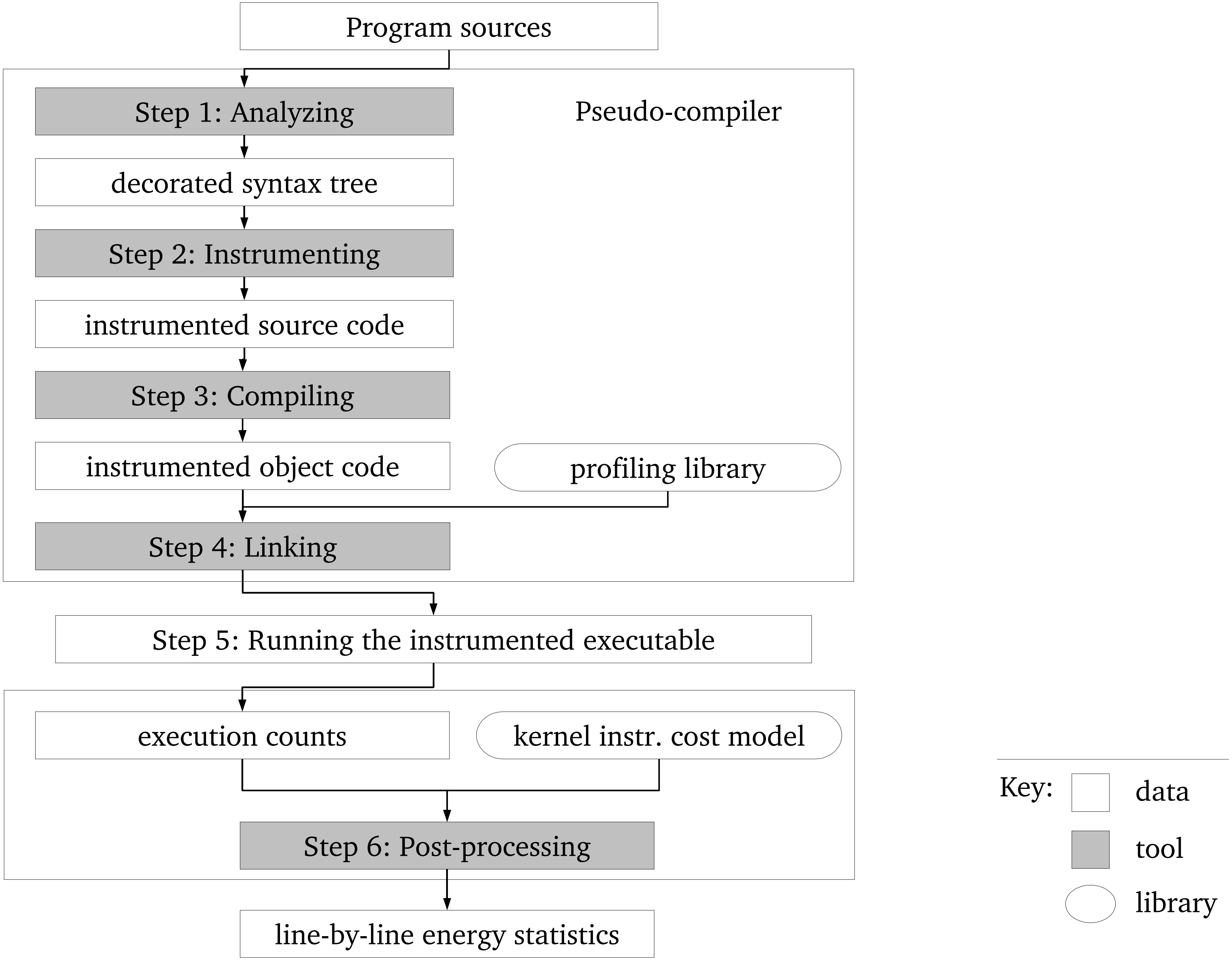
Execution cost
$C_{17} = n_{17} \cdot c_{17} =$ (1.311 ms, 471.8 mJ)

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;

Attribute		**Name**	**Defined for**
k	synthesized	constancy	expressions
e	synthesized	constant value	expressions
t	synthesized	real result type	expressions
v	inherited	valueness	expressions
r	inherited	restricted result type	expressions
f	inherited	translation flavor	expressions and statements
ci	synthesized	inherent cost	expressions and statements
cc	synthesized	conversion cost	expressions and statements
cf	inherited	flow control cost	expressions and statements
c	synthesized	total cost	expressions and statements

Estimation: the tool flow



Program sources

Pseudo-compiler

Step 1: Analyzing

decorated syntax tree

Step 2: Instrumenting

instrumented source code

Step 3: Compiling

instrumented object code

profiling library

Step 4: Linking

Step 5: Running the instrumented executable

execution counts

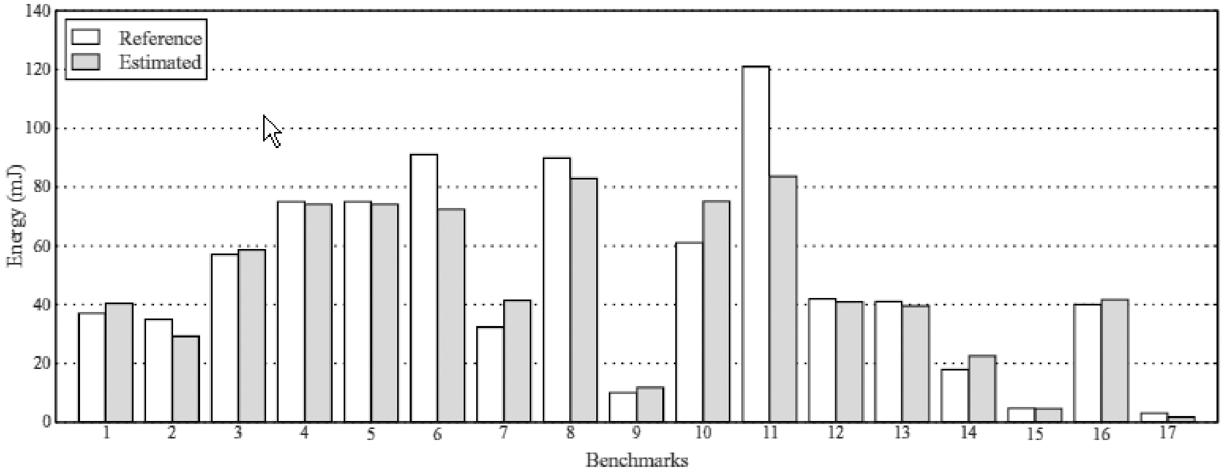
kernel instr. cost model

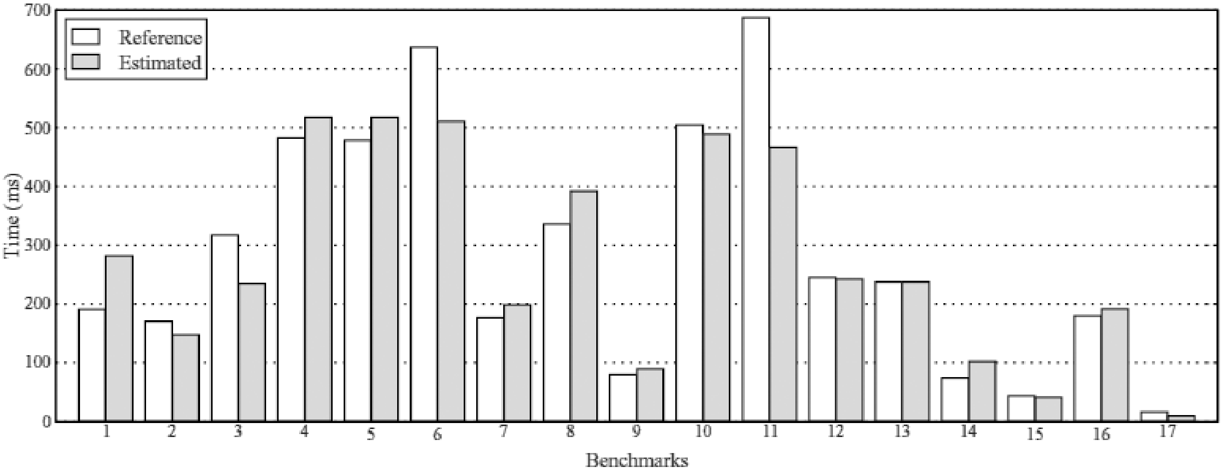
Step 6: Post-processing

line-by-line energy statistics

Key:
☐ data
▨ tool
○ library

Results: accuracy and speed



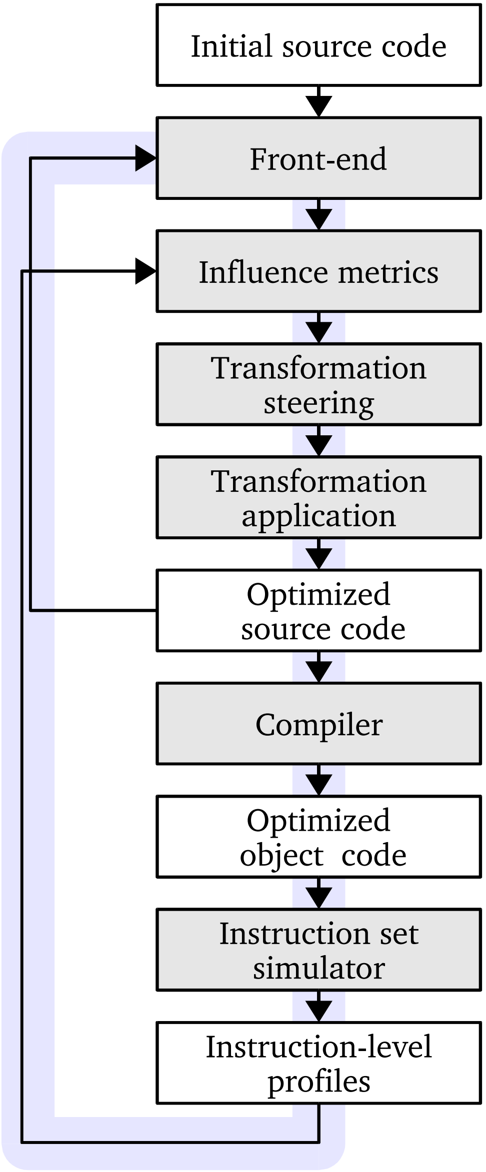


- Experimental Setup
  - comparison against SimIt-Arm (cycles) [Qin03]
  - current figures from JouleTrack (energy) [Sinha01]
  - modelling for SA1100, 206 MHz, 1.5 V
  - 24 benchmark from MiBench [Guthaus01]
- Accuracy
  - avg modulo error =15% E, <17% T
  - coefficients of correlation = 0.978 E, 0.960 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

Long exploration loop



Initial source code

Front-end

Influence metrics

Transformation steering

Transformation application

Optimized source code

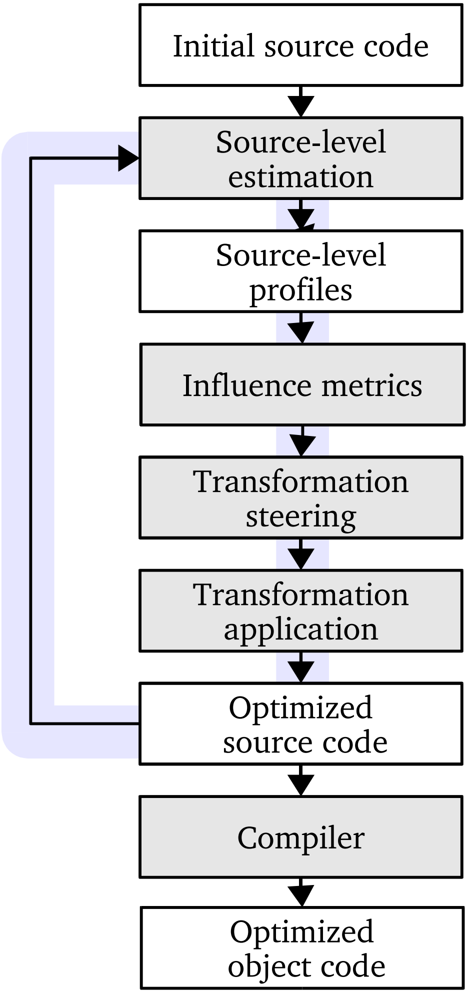
Compiler

Optimized object code

Instruction set simulator

Instruction-level profiles

Short exploration loop



Initial source code

Source-level estimation

Source-level profiles

Influence metrics

Transformation steering

Transformation application

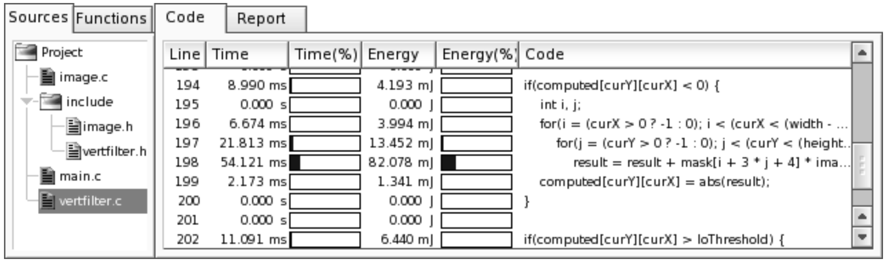
Optimized source code

Compiler

Optimized object code

What we do and others can't

- Import a project

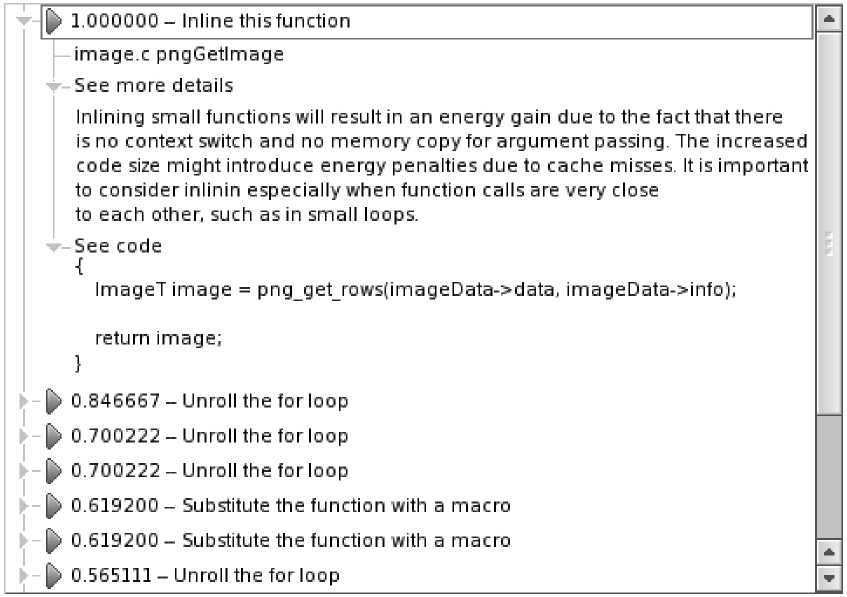


Line	Time	Time(%)	Energy	Energy(%)	Code
194	8.990 ms		4.193 mJ		`if(computed[curY][curX] < 0) {`
195	0.000 s		0.000 J		`int i, j;`
196	6.674 ms		3.994 mJ		`for(i = (curX > 0 ? -1 : 0); i < (curX < (width - ...`
197	21.813 ms		13.452 mJ		`for(j = (curY > 0 ? -1 : 0); j < (curY < (height..`
198	54.121 ms		82.078 mJ		`result = result + mask[i + 3 * j + 4] * ima...`
199	2.173 ms		1.341 mJ		`computed[curY][curX] = abs(result);`
200	0.000 s		0.000 J		`}`
201	0.000 s		0.000 J		
202	11.091 ms		6.440 mJ		`if(computed[curY][curX] > loThreshold) {`

- Analyze it

```
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
```

- **Get source-level optimization directives, generated at the source level**



1.000000 – Inline this function

image.c pngGetImage

See more details

Inlining small functions will result in an energy gain due to the fact that there is no context switch and no memory copy for argument passing. The increased code size might introduce energy penalties due to cache misses. It is important to consider inlinin especially when function calls are very close to each other, such as in small loops.

See code

```
{
  ImageT image = png_get_rows(imageData->data, imageData->info);

  return image;
}
```

0.846667 – Unroll the for loop

0.700222 – Unroll the for loop

0.700222 – Unroll the for loop

0.619200 – Substitute the function with a macro

0.619200 – Substitute the function with a macro

0.565111 – Unroll the for loop

- Apply them
  and see the result

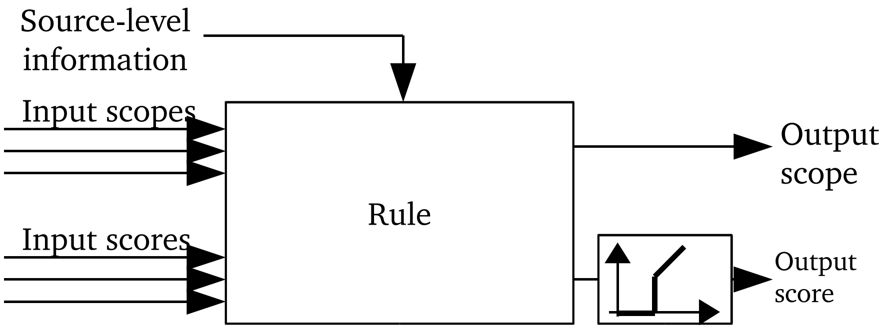
```
File                Time         Energy
image.c              21.638 µs    16.561 µJ
main.c               28.962 µs    21.158 µJ
vertfilter.c        356.222 ms  396.261 mJ
(glibc)             305.800 µs   21.158 µJ
TOTAL               356.509 ms  396.921 mJ
```

What a short-loop methodology needs

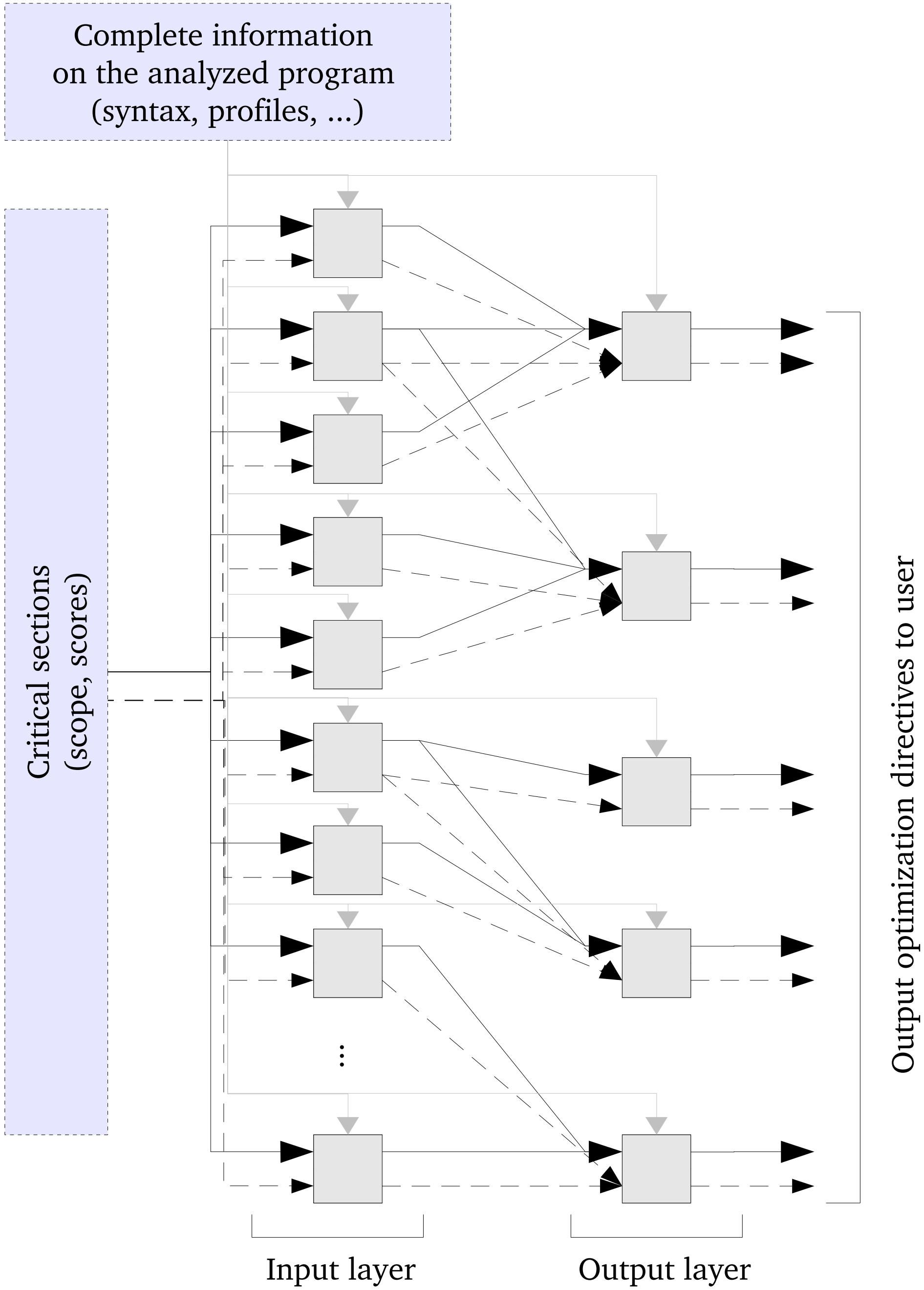
Problem	Task	Additional Requirements
source code analysis	analyze the code and determine which are the critical sections	analysis must be performed at source level; profile data must be available at source level
		SLE is the first approach
influence metrics	determine what is the gain in applying a trf over a section	
		Many exist, e.g. [Brandolese03]
transformation steering	decide which transformation to apply and where	steering engine must operate **automatically** on **source-level** data provided by above analysis and metrics
		→ **None exist!**
transformation application	apply transformation on the source code	
		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

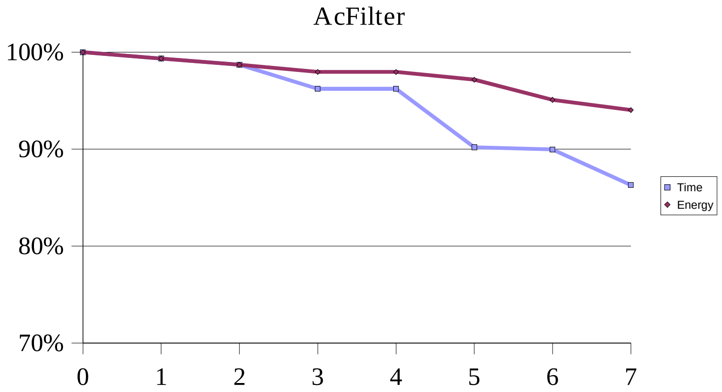


- Advantages:
  - scalable O(n·Q)
  - modular (no IP disclosed)

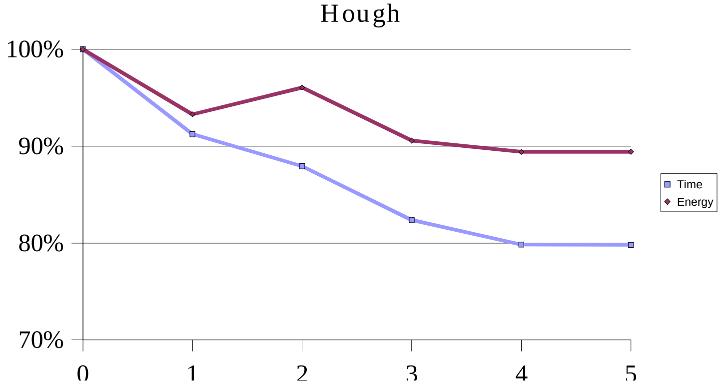


Experimental results

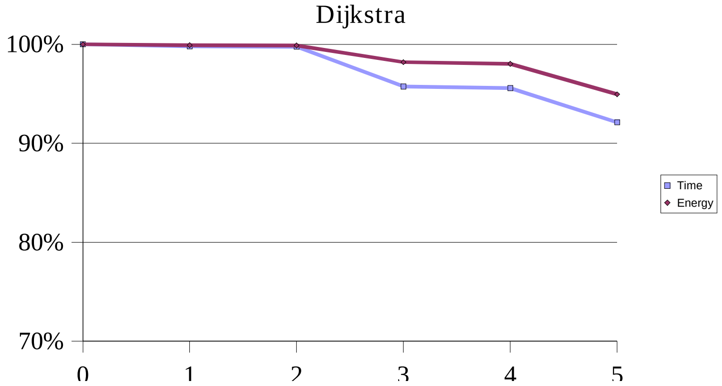
- Modelled transformations:
  1) loop unrolling
  2) function inlining
  3) function replacement with macro
  4) common subexpression elimination
  5) strength reduction
  6) **type conversion elimination**
  7) **standard library function factorization**
  8) **memory allocation factorization**
  9) **argument passing via pointer**
  10) **function specialization**
- Benchmarks:
  4 applications (audio filter, hough transform, dijkstra, FFT);
- Energy gains: 5 – 22 %
- Time gains: 8 – 20 %



T: -13.68%
E: -5.96%



T: -20.18%
E: -10.58%



T: -7.88%
E: -5.05%

T: -22.27%
E: -22.01%