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 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
2. dynamic
3. source-level
4. fine-detail

- the size and complexity of modern embedded applications is increasing quickly;
- simulating non-toy apps at the circuit level or gate level is unaffordable;
- instruction-set simulation is also unaffordable for apps of sufficient complexity (e.g. video decoders);
- 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
2. **dynamic**
3. source-level
4. fine-detail

- modern applications are becoming more and more dynamic in nature;
- the behavior of multimedia en-/de-coders depends more and more on the contents of the streams they process;

uncompressed, constant resolution

I,P,B (MPEG-2)

object-based encoding (MPEG-4)

- 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;

1.1. Requirements

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

- many energy estimation flows operate at the assembly level, but designers do not code in assembly any more;
- designers use high-level languages instead, estimation flows should provide information at the same abstraction 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;
- 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
2. dynamic
3. source-level
4. **fine-detail**

- most of the time and energy are spent in small computational kernels;
- "small" is much smaller than a program and a function, potentially smaller than inner loops;
- many estimation techniques (even source-level ones) cannot "look inside functions"
- fine-detail analysis techniques are needed; "fine-detail" = individual operator instance;

What I mean by fine-detail source-level

```
56 for ( BlockSize = 2; BlockSize <= NumSamples; B
57 {
58  float delta_angle, sm2, sm1, cm2, cm1, w, ar[3
59
60  delta_angle = angle_numerator / (float)BlockSize;
61  sm2 = sinf ( -2 * delta_angle );
62  sm1 = sinf ( -delta_angle );
63  cm2 = cosf ( -2 * delta_angle );
64  cm1 = cosf ( -delta_angle );
65  w = 2 * cm1;
66
67      for ( i=0; i< NumSamples; i += BlockSize )
68      {
69            ar[2] = cm2;
70            ar[1] = cm1;
71
72            ai[2] = sm2;
73            ai[1] = sm1;
74
75            for ( j=i, n=0; n < BlockEnd; j++, n++ )
76            {
77                  ar[0] = w*ar[1] - ar[2];
78                  ar[2] = ar[1];
79                  ar[1] = ar[0];
80
81                  ai[0] = w*ai[1] - ai[2];
82                  ai[2] = ai[1];
83                  ai[1] = ai[0];
```

#284: 77,24 -- 77,39 Operator –
CPU=0 (Pivot is #408)
k=0, v=R, b=0, t='[float]'
n=53248, c = 1 FloatSub
T = 88*alul
c_{1E} = 1.13837uJ c_{1T} = 4.26357us
c_{nE} = 6.0616mJ c_{nT} = 22.7026ms
CcE = 12.206mJ CcT = 45.6633ms

#280: 77,24 -- 77,31 Operator *
CPU=0 (Pivot is #408)
k=0, v=R, b=0, t='[float]'
n=53248, c = 1 FloatMul
T = 87*alul
c_{1E} = 1.12544uJ c_{1T} = 4.21512us
c_{nE} = 5.99272mJ c_{nT} = 22.4447ms
CcE = 6.06857mJ CcT = 22.7026ms

#276: 77,24 -- 77,25 Identifier "w"
CPU=0 (Pivot is #408)
k=0, v=R, b=1, c=0, t='[float]'
n=53248
CcE = 0mJ CcT = 0ms

#279: 77,26 -- 77,31 Operator []
CPU=0 (Pivot is #408)
k=0, v=R, b=0, t='[float]'
n=53248, c = 1 RValueIndex
T = 1*mvld
c_{1E} = 0.0142442uJ c_{1T} = 0.0484496us
c_{nE} = 0.075874mJ c_{nT} = 0.257984ms
CcE = 0.075874mJ CcT = 0.257984ms

#277: 77,26 -- 77,28 Identifier "ar"
CPU=0 (Pivot is #408)
k=0, v=R, b=1, c=0, t='[array[3]] [f
n=53248
CcE = 0mJ CcT = 0ms

#278: 77,29 -- 77,30 Constant "1"

1.3. Current techniques are not ok

- Static Timing Analysis (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 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: [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 [Ravasio03]
- Black-box techniques do not provide any link with code [Muttreja04]

1.3. Current techniques are not ok

• Static Timing Analysis (STA) techniques cannot deal with dynamism;	Fast ~~Dyn~~ Src Fine
• Instruction-Set Simulation (ISS) is slow and at a low level:	~~Fast~~ Dyn ~~Src~~ ~~Fine~~
• ISS + gprof provide estimates only at a function level;	~~Fast~~ Dyn Src ~~Fine~~
• Atomium/PowerEscape is source-level, but only for memory aspects (not our focus);	(Fast Dyn Src Fine)
• SoftExplorer is a static technique;	Fast ~~Dyn~~ ~~Src~~ ~~Fine~~
• Compilation-based approaches do not provide link to source level;	Fast Dyn ~~Src~~ ~~Fine~~
• SIT is source level (good!) but still unable to resolve chosen clusters;	Fast Dyn Src ~~Fine~~
• Black-box techniques do not provide any link with source code;	Fast Dyn ~~Src~~ ~~Fine~~

2. How my technique works

- 2.1 Divide and conquer:

$$C_i = n_i \cdot c_i$$

C_i: cost of executing the i-th node in the AST

n_i: execution count

c_i: 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

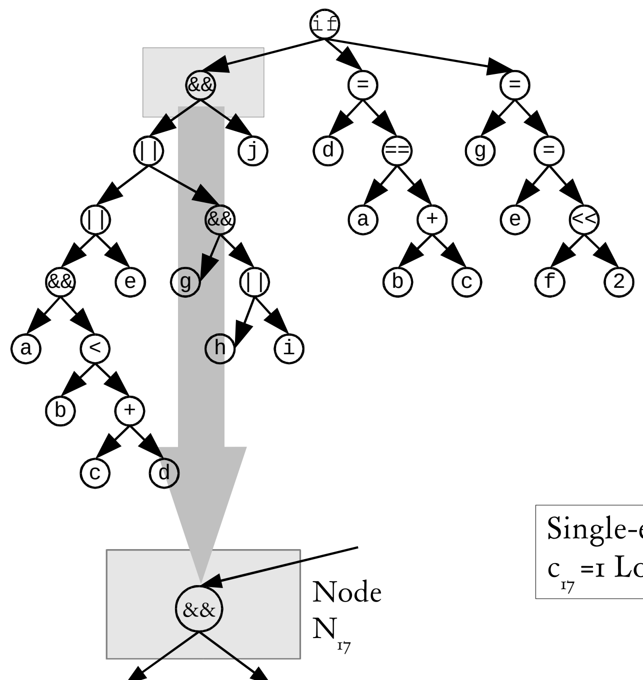
2.1. Divide and conquer: $C_i = n_i \cdot c_i$

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



Single-execution cost
c_{17} =1 LogicTop

Execution count
n_{17} =4327

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

Abstract instructions

Abstract translation model

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

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

Time and energy

Target Platform Characterization

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

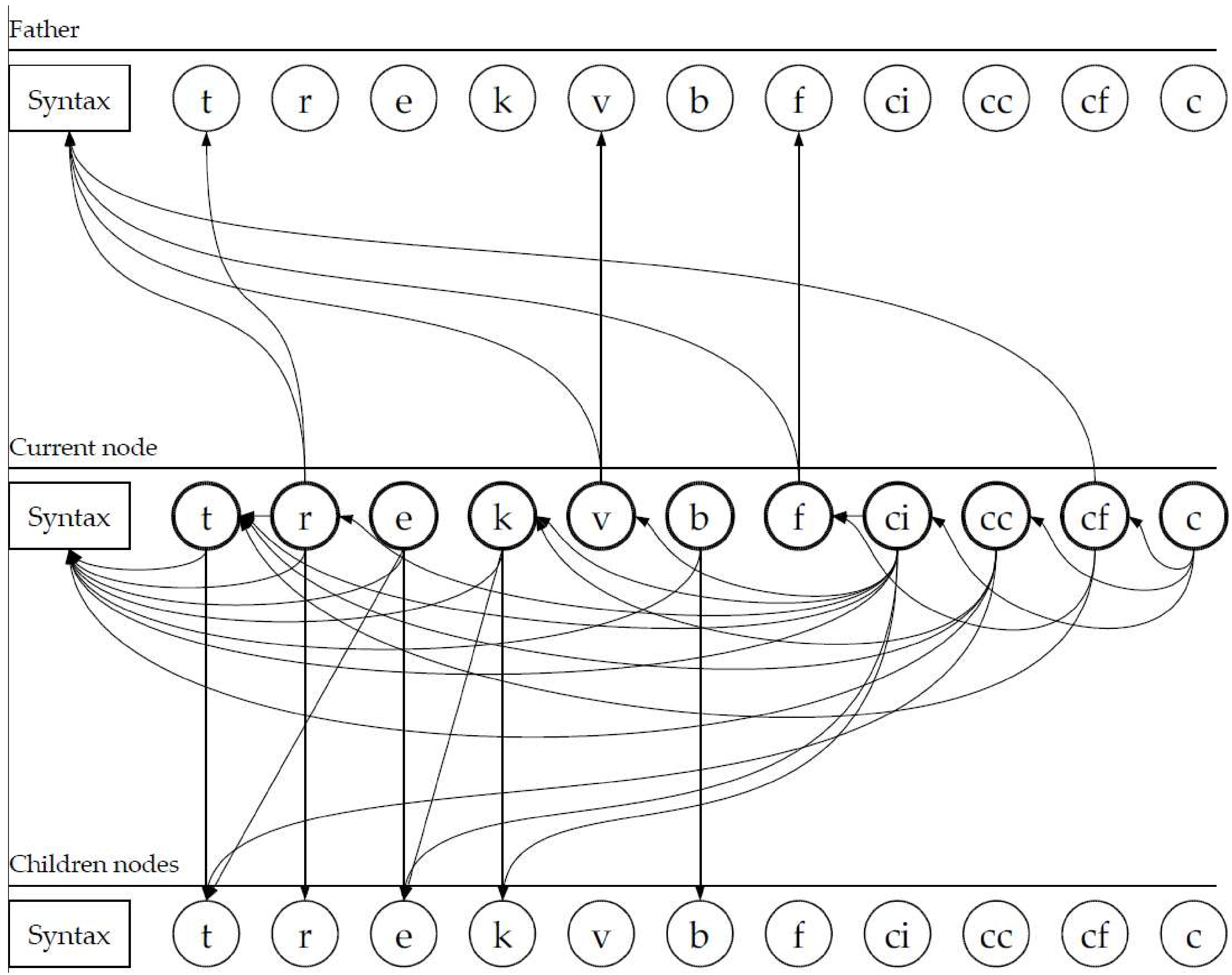
Execution cost
$C_{17} = n_{17} \cdot c_{17}$ = (1.311 ms, 471.8 mJ)

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:

Attribute		Name	Defined for which AST nodes
c	synthesized	total cost	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
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
b	synthesized	register-boundedness	expressions
f	inherited	translation flavor	expressions and statements

2.2. Determining single-execution costs

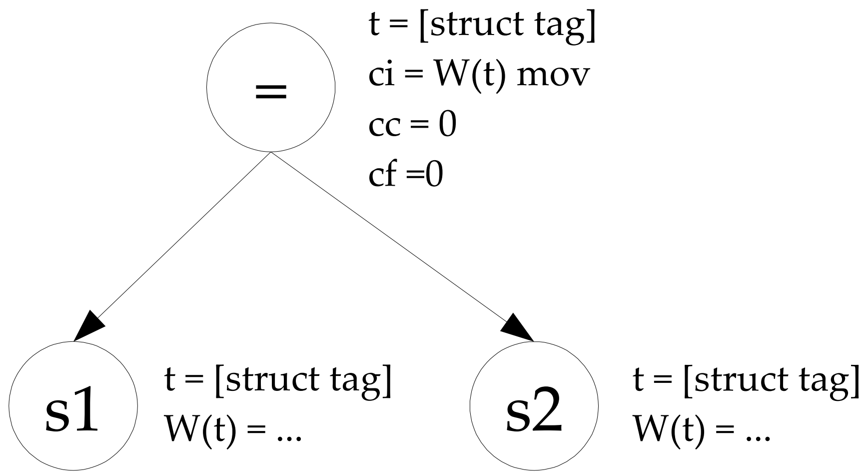


Why all these attributes?

- 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:

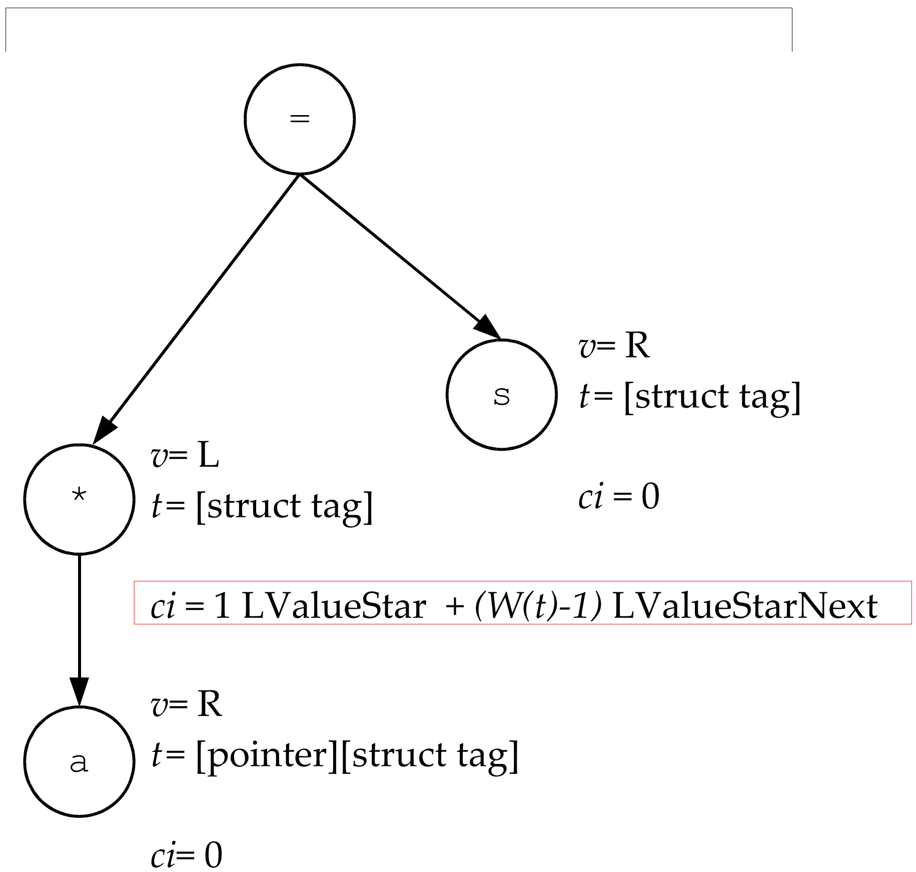
```
struct tag
{
    int  field1;
    char field2 [sizeof(type_x)*5];
} s1, s2;

int main()
{
    ...
    s1 = s2;
    ...
}
```



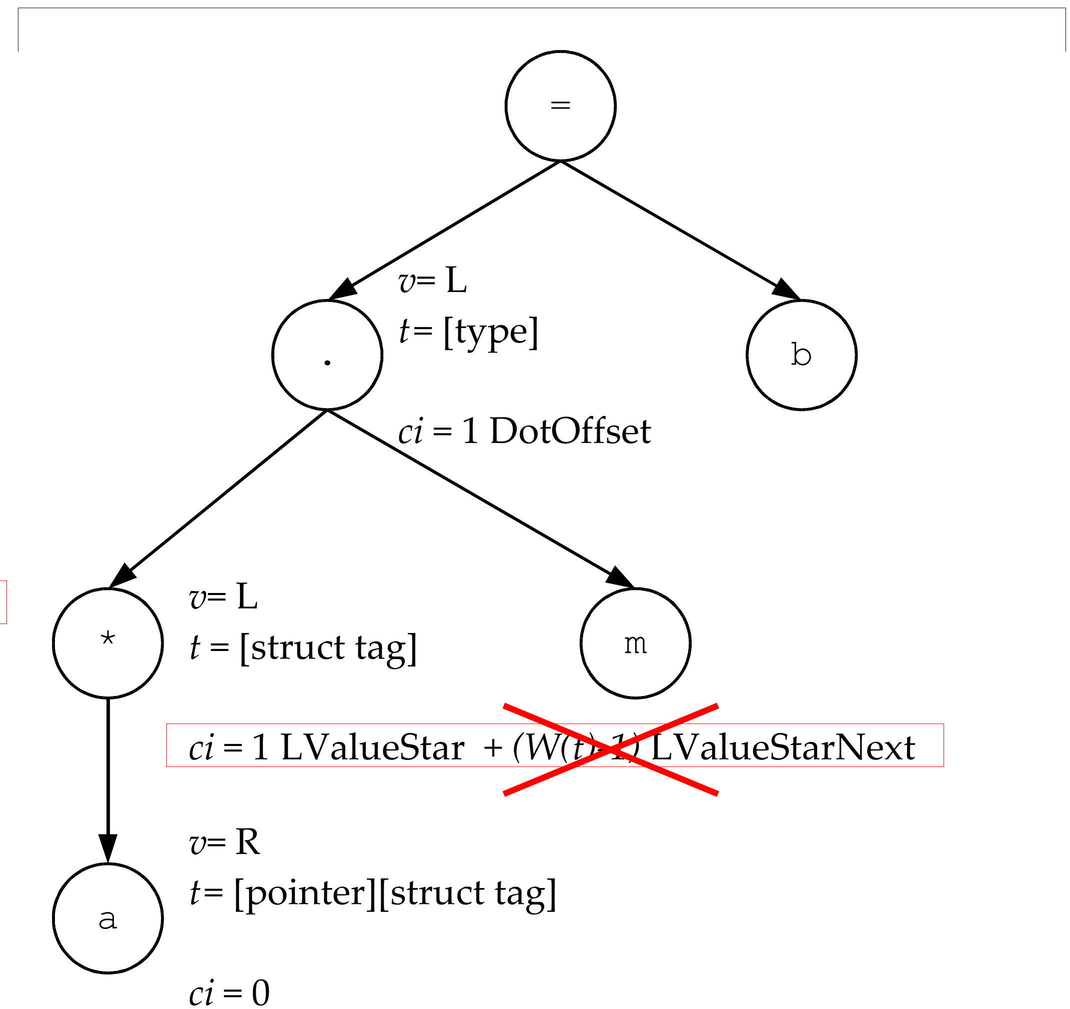
Why attribute r (restricted type) is needed

`*a = s;`



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

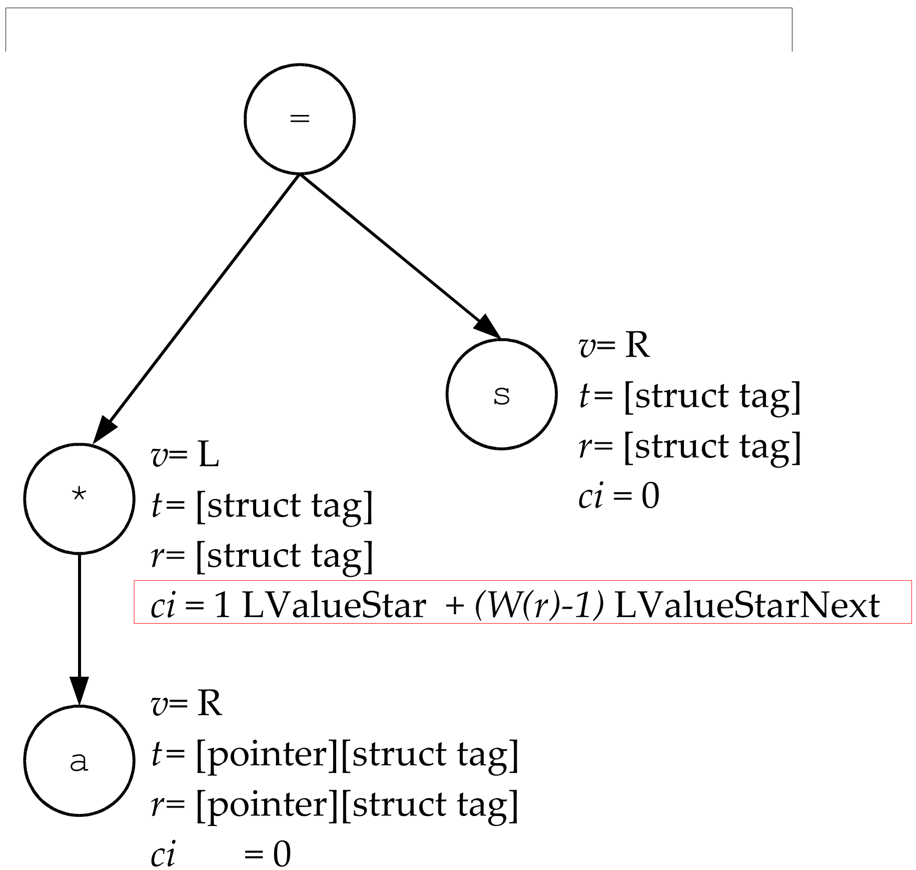
`(*a).m = b;`



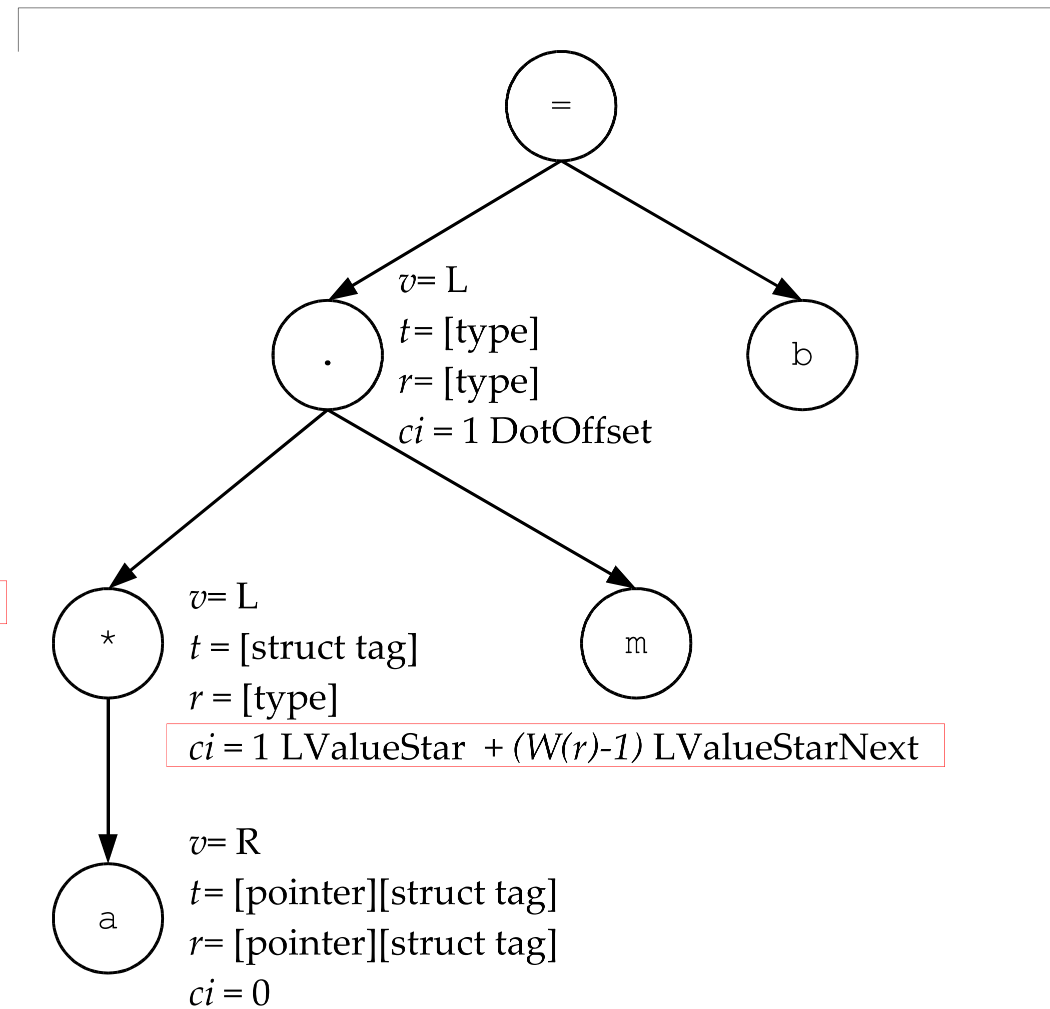
(not really!)

Why attribute r (restricted type) is needed

`*a = s;`



`(*a).m = b;`

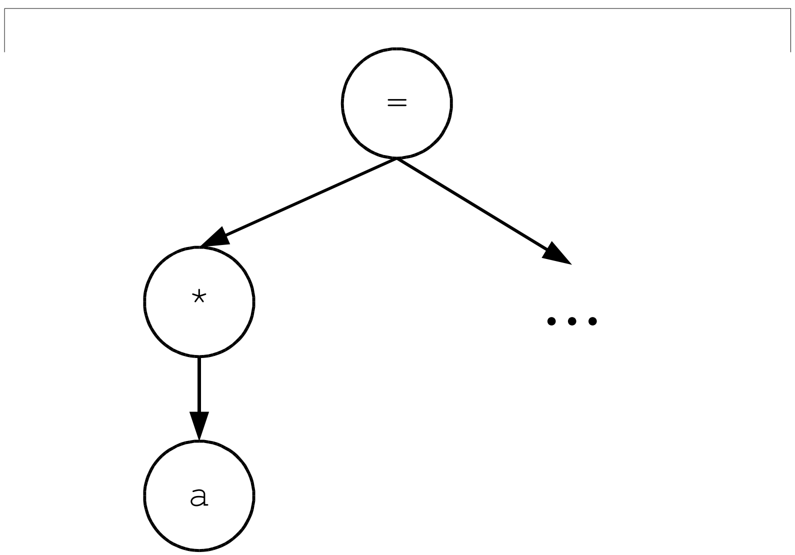


(the cost of a star operator depends on its type)

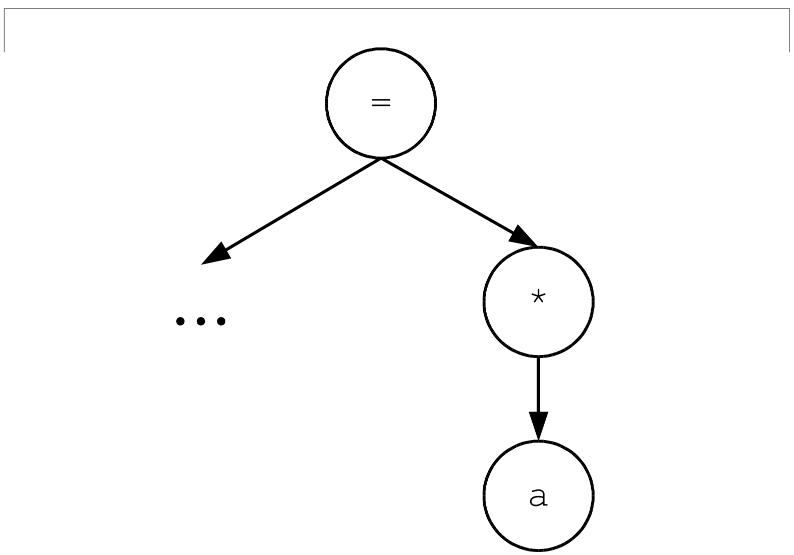
(not really!)

Why attribute v (valueness) is needed

`*a = ...;`



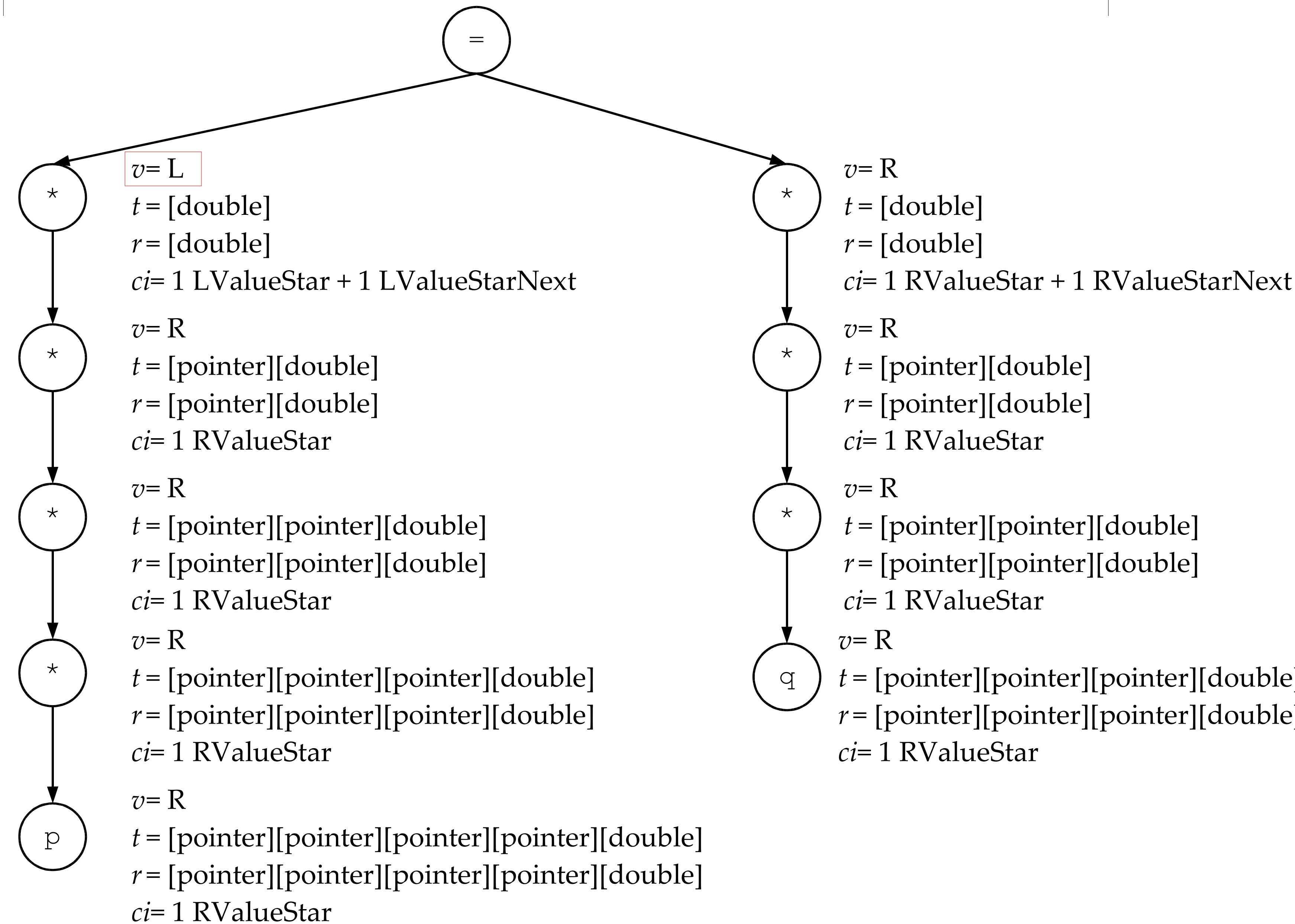
`... = *a;`



Why attribute v (valueness) is needed

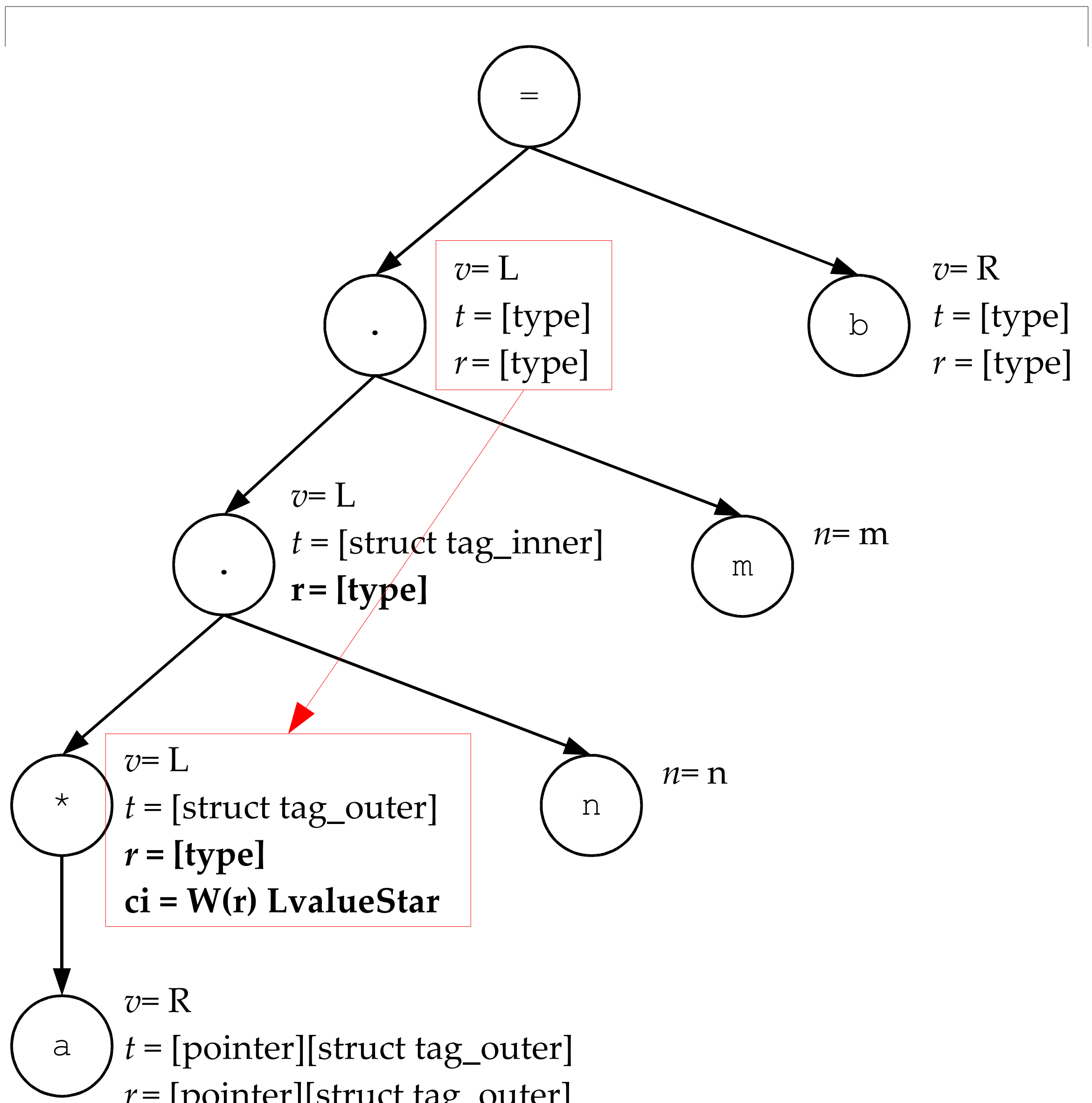
```
****p = ***q;
```

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



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:

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

```
/*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:

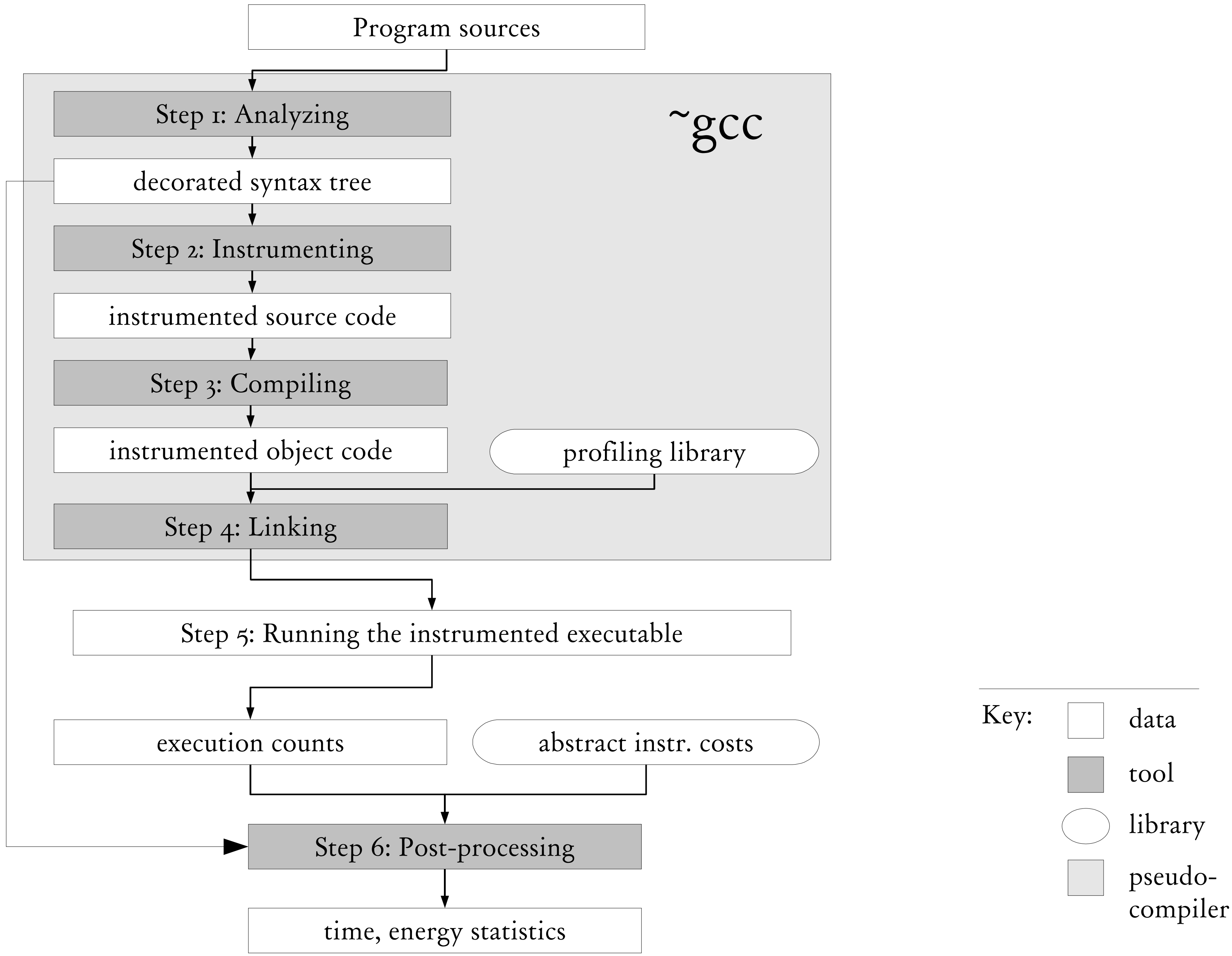
```
int f(args)
{
        ...
};
```

```
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
~gcc
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
abstract instr. costs
Step 6: Post-processing
time, energy statistics
Key:
data
tool
library
pseudo-compiler

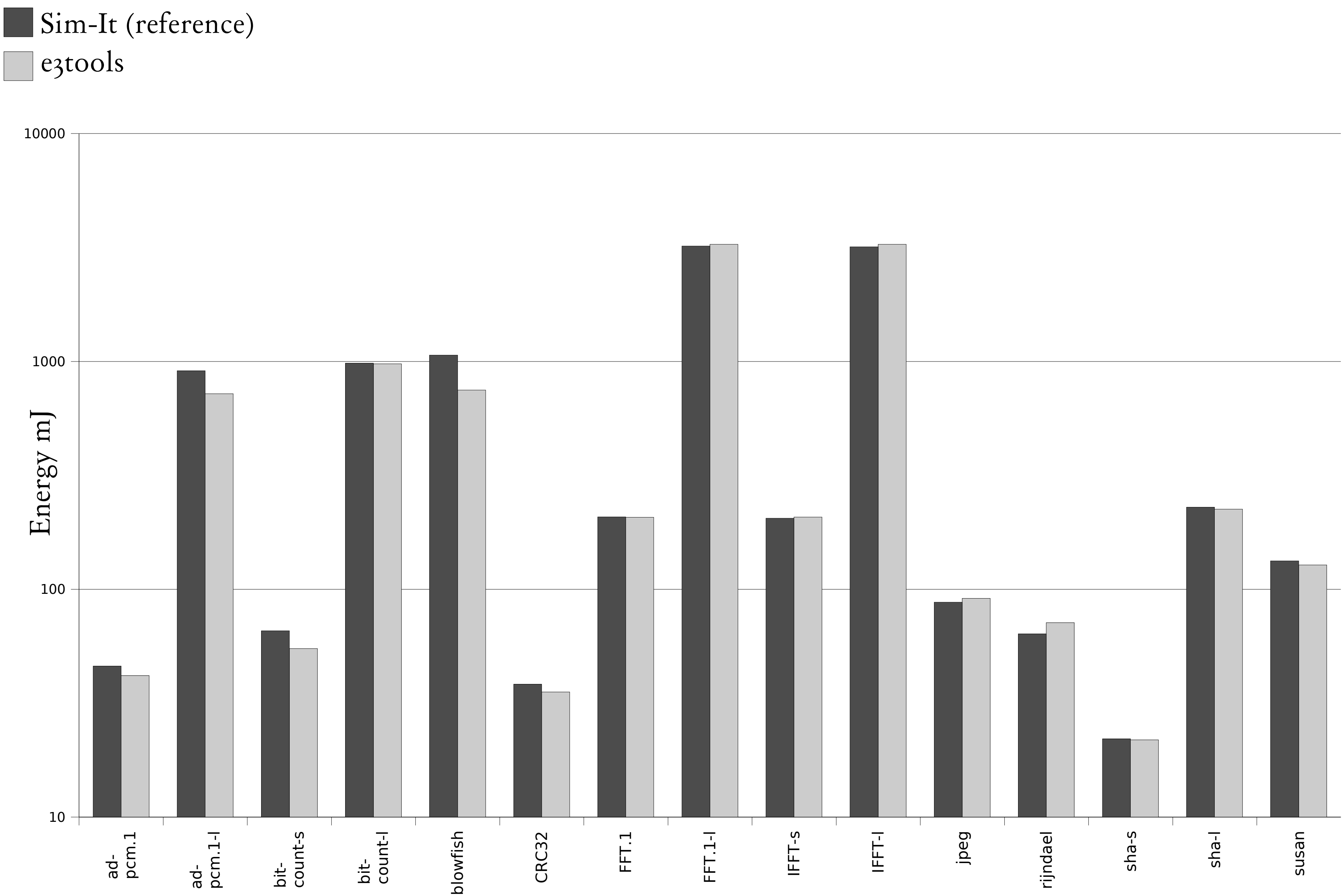
3.3. Accuracy results

	SimIt		e3tools		Estimation error	
	E (mJ)	T (ms)	E (mJ)	T (ms)	E	T
adpcm-s	46,1	166,3	41,9	156,4	-9,1%	-6,0%
adpcm-l	910,2	3289,9	722,1	2710,5	-20,7%	-17,6%
bitcount-s	65,7	242,8	55,0	204,0	-16,3%	-16,0%
bitcount-l	981,9	3628,6	977,1	3649,2	-0,5%	+0,6%
blowfish	1067,0	3742,7	748,3	3371,0	-29,9%	-9,9%
CRC32	38,3	132,2	35,4	129,6	-7,5%	-2,0%
FFT-s	207,9	764,6	207,1	770,3	-0,4%	+0,7%
FFT-l	3213,2	11851,5	3264,8	12142,5	+1,6%	+2,5%
IFFT-s	205,1	755,1	207,3	771,0	+1,1%	+2,1%
IFFT-l	3181,8	11744,7	3266,2	12147,8	+2,7%	+3,4%
jpeg	87,9	309,9	91,2	328,5	+3,8%	+6,0%
rijndael	63,8	221,3	71,4	257,3	+12,0%	+16,3%
sha-s	22,1	78,9	21,9	78,6	-0,9%	-0,4%
sha-l	229,4	820,0	224,7	818,3	-2,1%	-0,2%

Quality of result:

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

3.3. Accuracy results



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

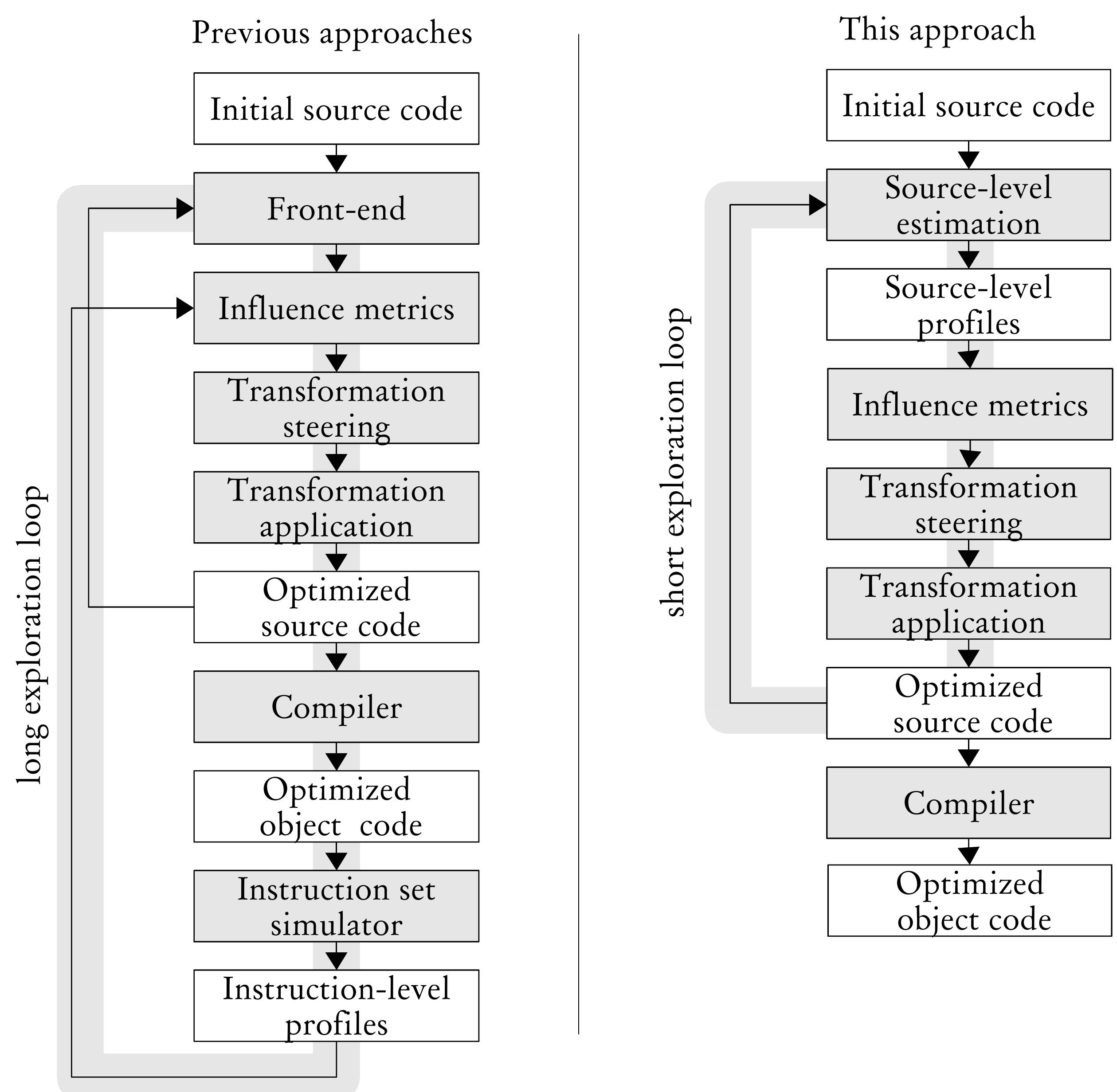
1. Opt.
2. VWR
3. VLIW
4. C++

- 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

1. **Opt.**
2. VWR
3. VLIW
4. C++

Long vs. short exploration loop:

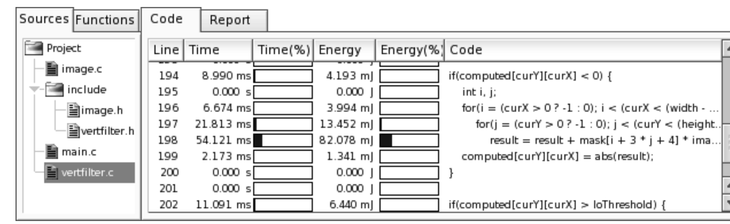


4.2. Uses & developments: Optimization

1. Opt.
2. VWR
3. VLIW
4. C++

What the new approach offers:

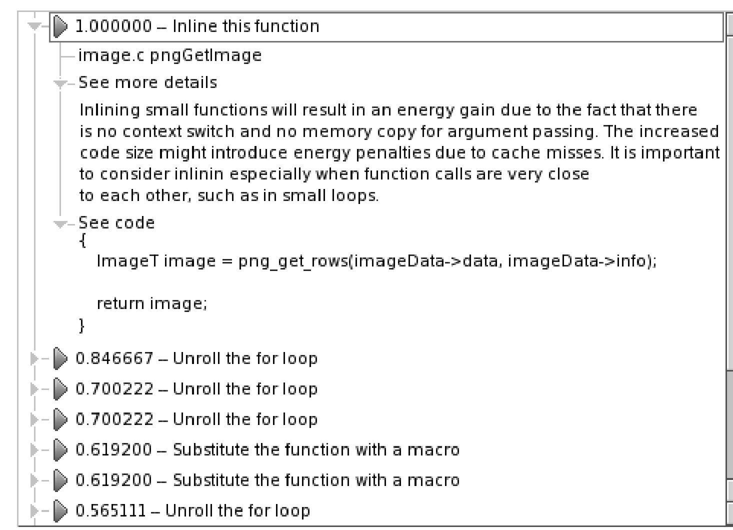
- Import a project



- 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



- 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	356.222 ms	396.261 mJ
(glibc)	305.800 µs	21.158 µJ
TOTAL	**356.509 ms**	**396.921 mJ**

4.2. Uses & developments: Optimization

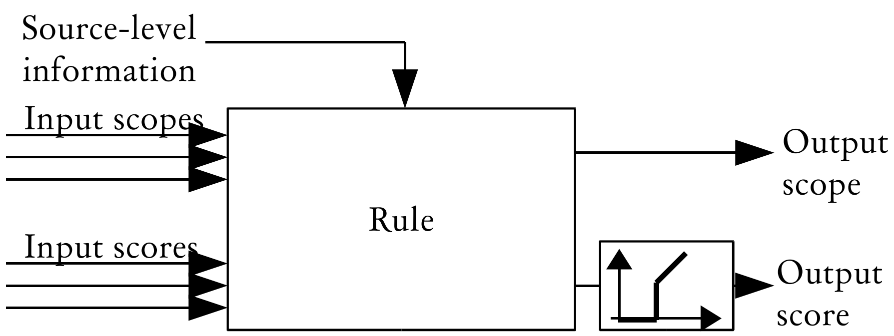
What a short-loop methodology needs:

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 exists!
transformation application	apply transformation on the source code	
		e.g. [SUIF94]

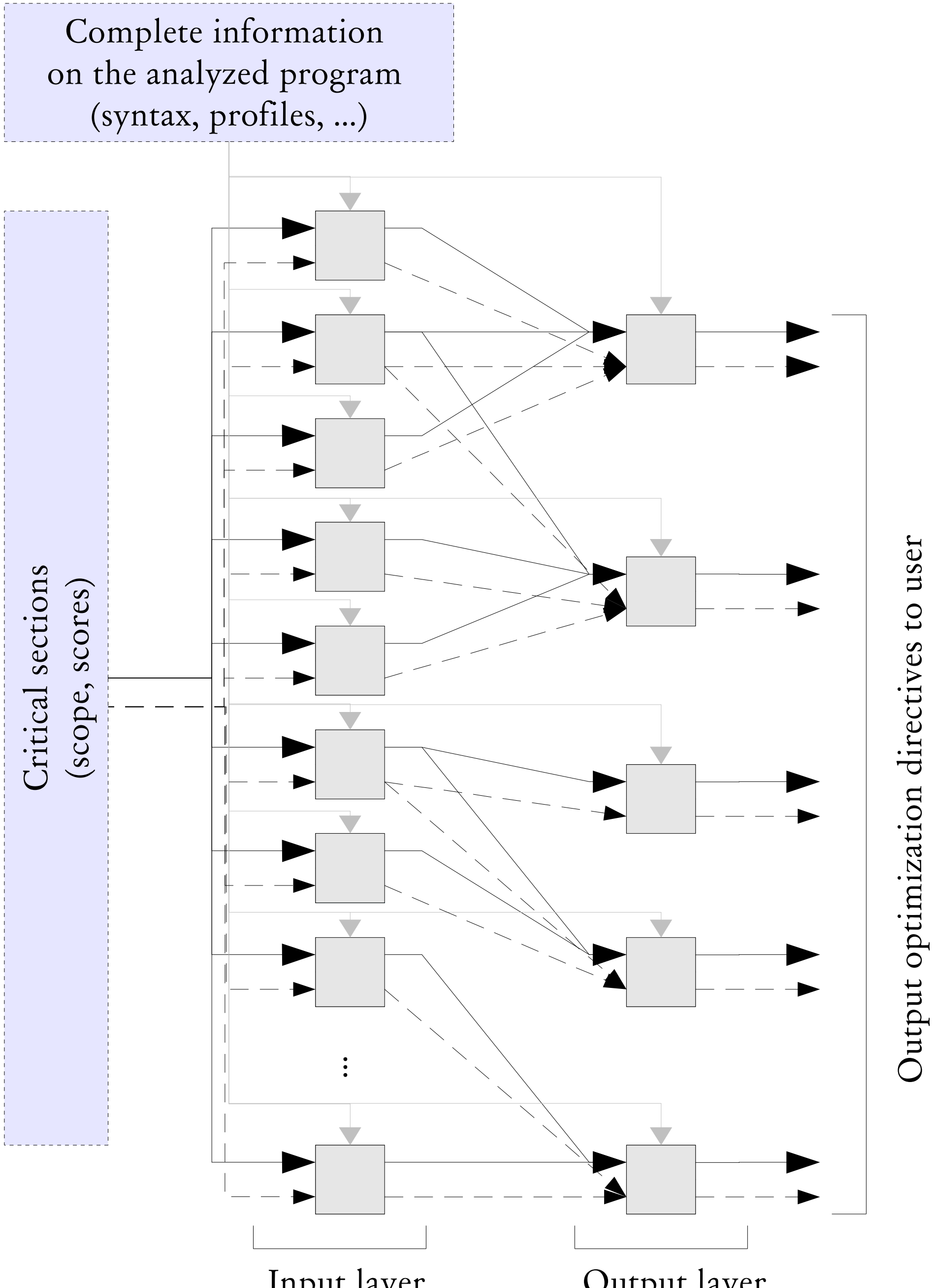
4.2. Uses & developments: Optimization

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)

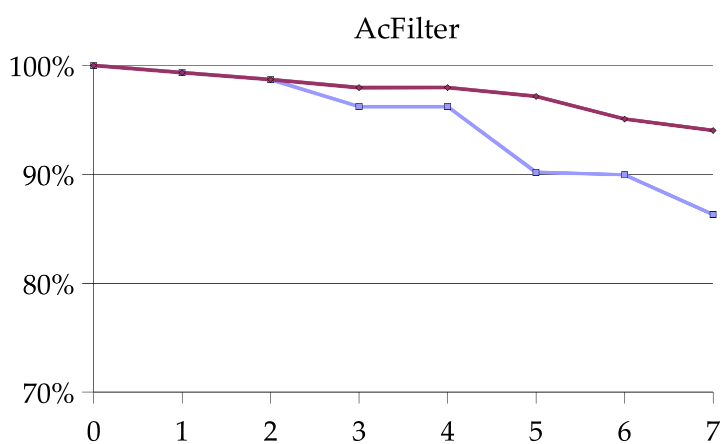


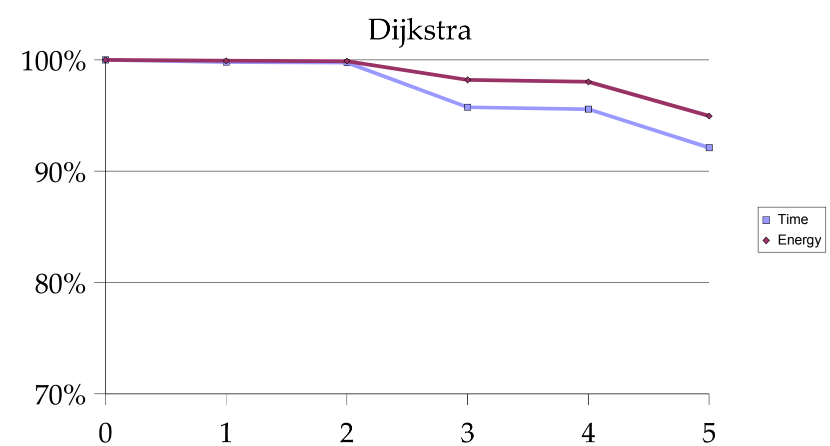
4.2. Uses & developments: Optimization

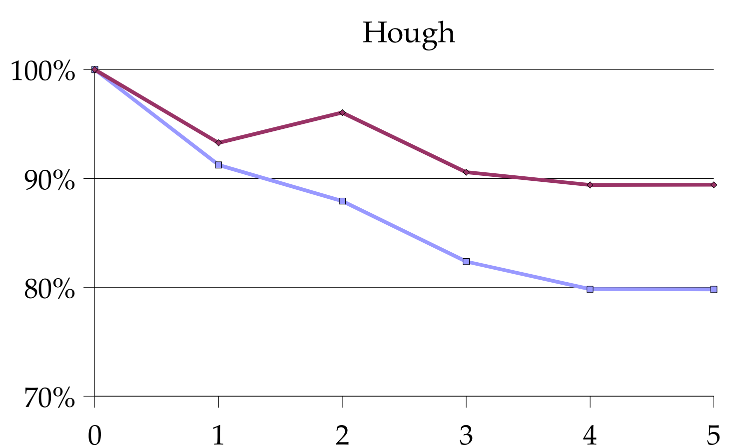
1. Opt.
2. VWR
3. VLIW
4. C++

- Results:

energy reduction: -5.1 – -22.0%
execution time reduction: -7.8 – -22.3%



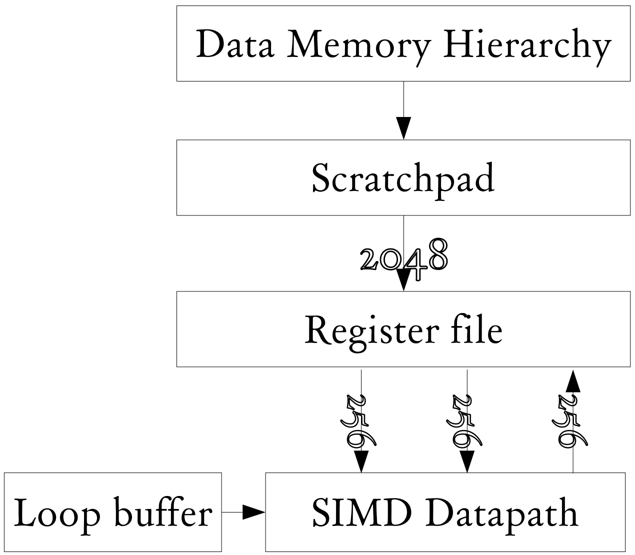




4.2. Uses & developments: VWR

1. Opt.
2. **VWR**
3. VLIW
4. C++

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



- 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;

4.2. Uses & developments: VWR

1. Opt.
2. VWR
3. VLIW
4. C++

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: `#pragma e3tools CPU n`
- Example:

```
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: `#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:

```
...
#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: ComplexMul = 2 rfrd + 4 aluh + 2 alul + 1 rfrw;
- Example:

```
#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.
2. VWR
3. VLIW
4. C++

Extending the e3tools to VLIW architectures.

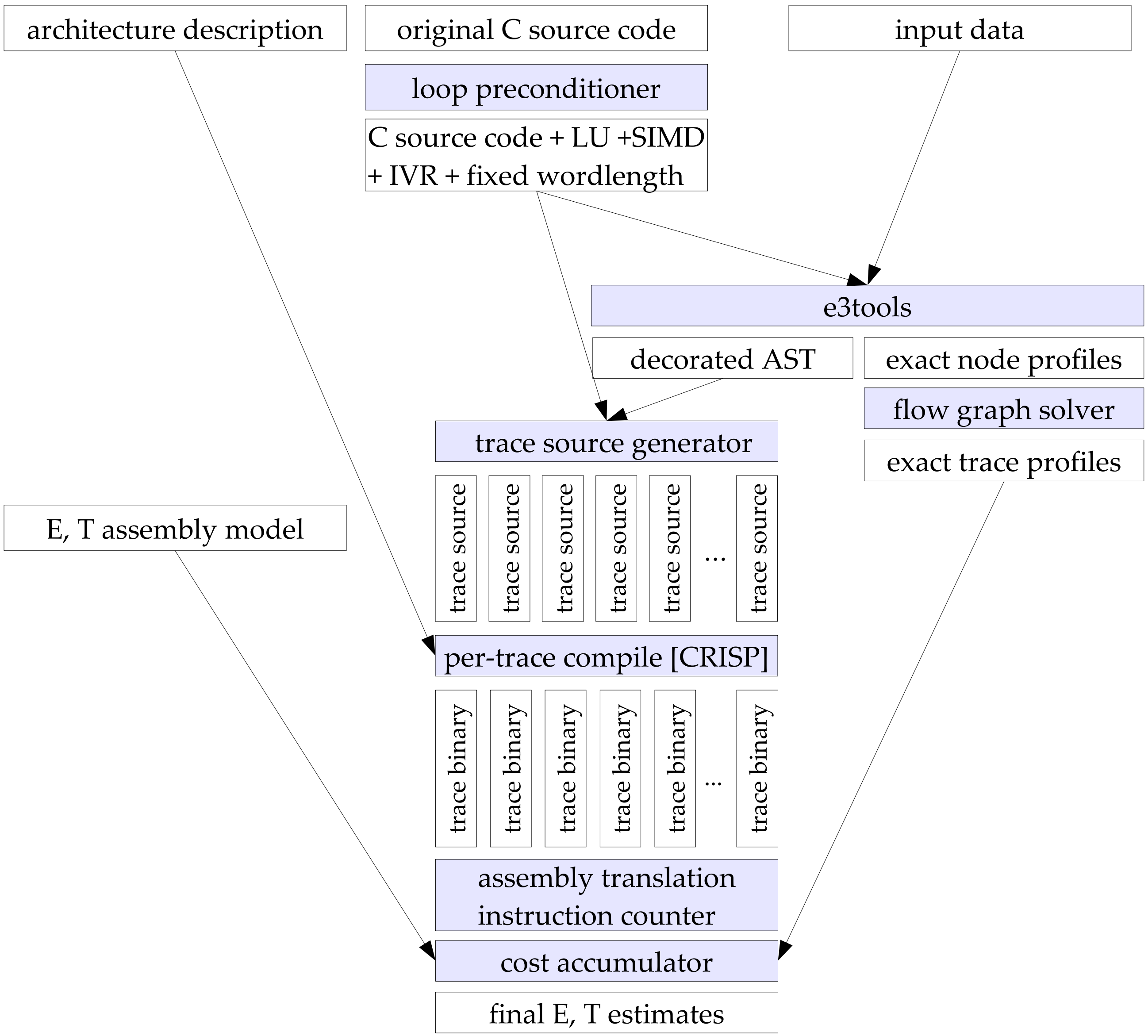
Goals:

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

1. Opt.
2. VWR
3. VLIW
4. C++

The new flow.



4.3. Uses & developments: VLIW

Rewriting code to generate all the traces:

- conditional expressions

```
... ( condition ?
      expression1 :
      expression2 ) ...
```

```
... ( (condition, TRUE) ?
  expression1 :
      expression2 ) ...
```

```
... ( (condition, FALSE) ?
      expression1 :
      expression2 ) ...
```

- if statements:

```
if (condition)
{
   ... /* then branch */
} else {
   ... /* else branch */
}
```

```
trash = condition;
if (TRUE)
{
   ... /* then branch */
} else {
   ... /* else branch */
}
```

```
trash = condition;
if (FALSE)
{
   ... /* then branch */
} else {
   ... /* else branch */
}
```

- switch statements:

```
switch (condition)
{
  case value1:
     /* code for value 1*/
  case value2:
     /* code for value 1*/
  ...
  default:
     /* code for value 1*/
}
```

```
trash = condition;
switch (value1)
{
  case value1:
     /* code for value 1*/
  case value2:
     /* code for value 1*/
  ...
  default:
     /* code for value 1*/
}
```

```
trash = condition;
switch (value2)
{
  case value1:
     /* code for value 1*/
  case value2:
     /* code for value 1*/
  ...
  default:
     /* code for value 1*/
}
```

...

```
trash = condition;
switch (valueN)
{
  case value1:
     /* code for value 1*/
  case value2:
     /* code for value 1*/
  ...
  default:
     /* code for value 1*/
}
```

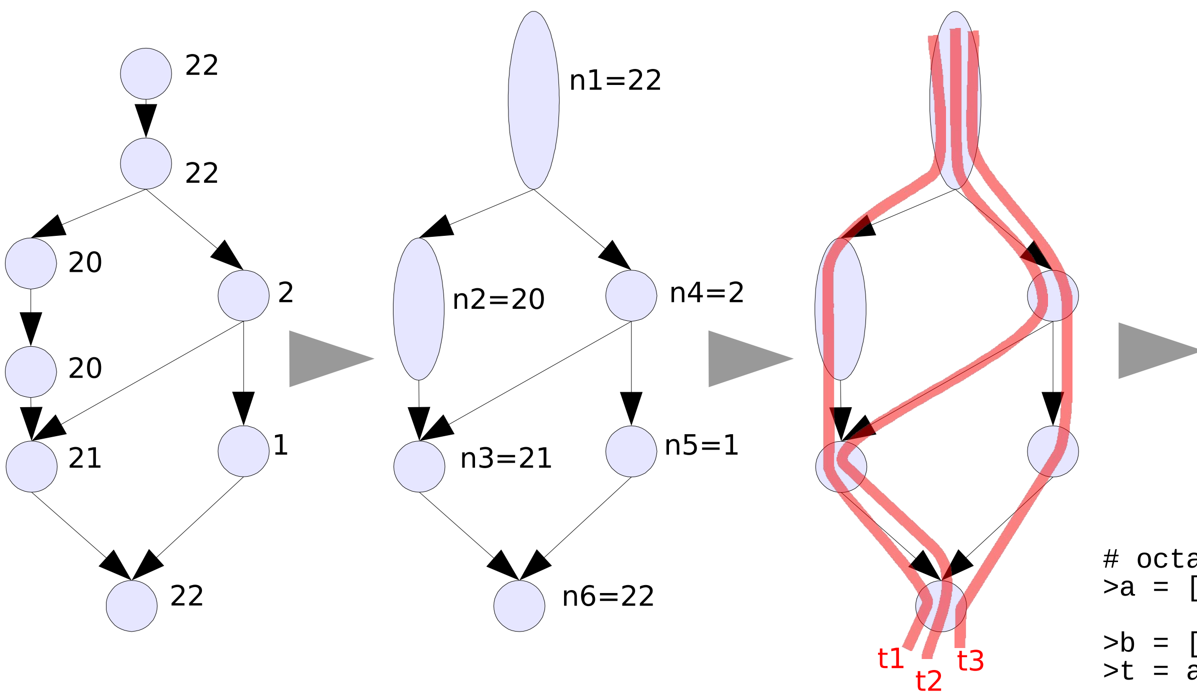
```
trash = condition;
switch (valueN)
{
  case value1:
     /* code for value 1*/
  case value2:
     /* code for value 1*/
  ...
  default:
     /* code for value 1*/
}
```

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 traces was executed?
- It can be solved with current, node-based instrumentation technique
- Need to determine trace counts from node counts

node counts — collapsing basic blocks — enumerating blocks per each trace — solving the corresponding equations



$$
\begin{aligned}
t1 + t2 + t3 &= n1\\
t1 &= n2\\
t1 + t2 &= n3\\
t2 + t3 &= n4\\
t3 &= n5\\
t1 + t2 + t3 &= n6
\end{aligned}
$$

$$
\begin{pmatrix} 1 & 1 & 1\\ 1 & 0 & 0\\ 1 & 1 & 0\\ 0 & 1 & 1\\ 0 & 0 & 1\\ 1 & 1 & 1 \end{pmatrix} t = \begin{pmatrix} 22\\ 20\\ 21\\ 2\\ 1\\ 22 \end{pmatrix}
$$

$$At = b$$

```
# 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
t =
  20.0
   1.0
   1.0
```

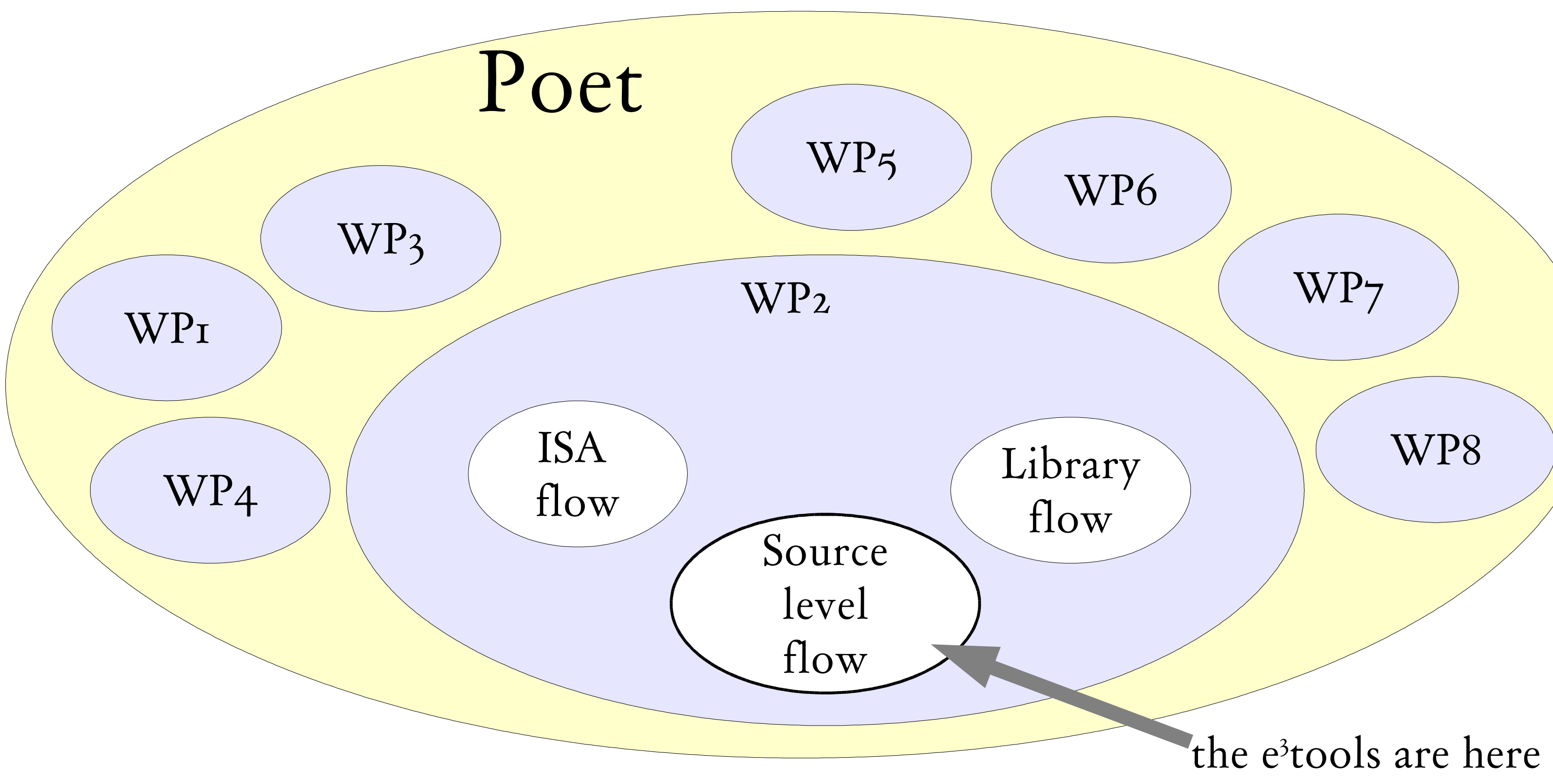
4.4. Prospective extension to C++

- Extending the technique to the C++ language is possible and involves reasonable effort;
- Tasks required:
  - lexer (28 new keywords, negligible effort);
  - parser: 213 << 560 syntax rules;
  - 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;



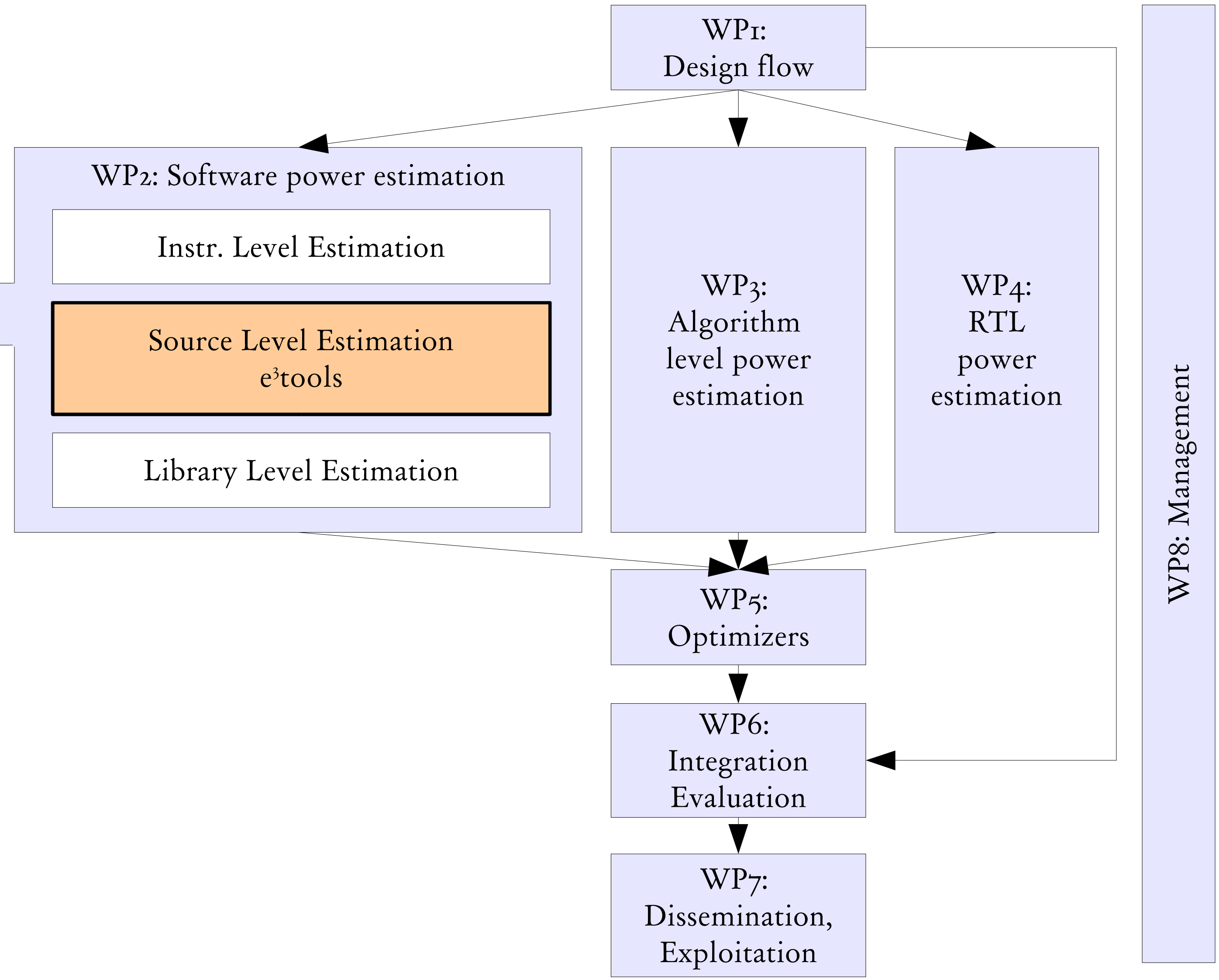
Reference: the POET consortium

Consortium:

Research
- OFFIS
- Politecnico di Torino
- Cefriel

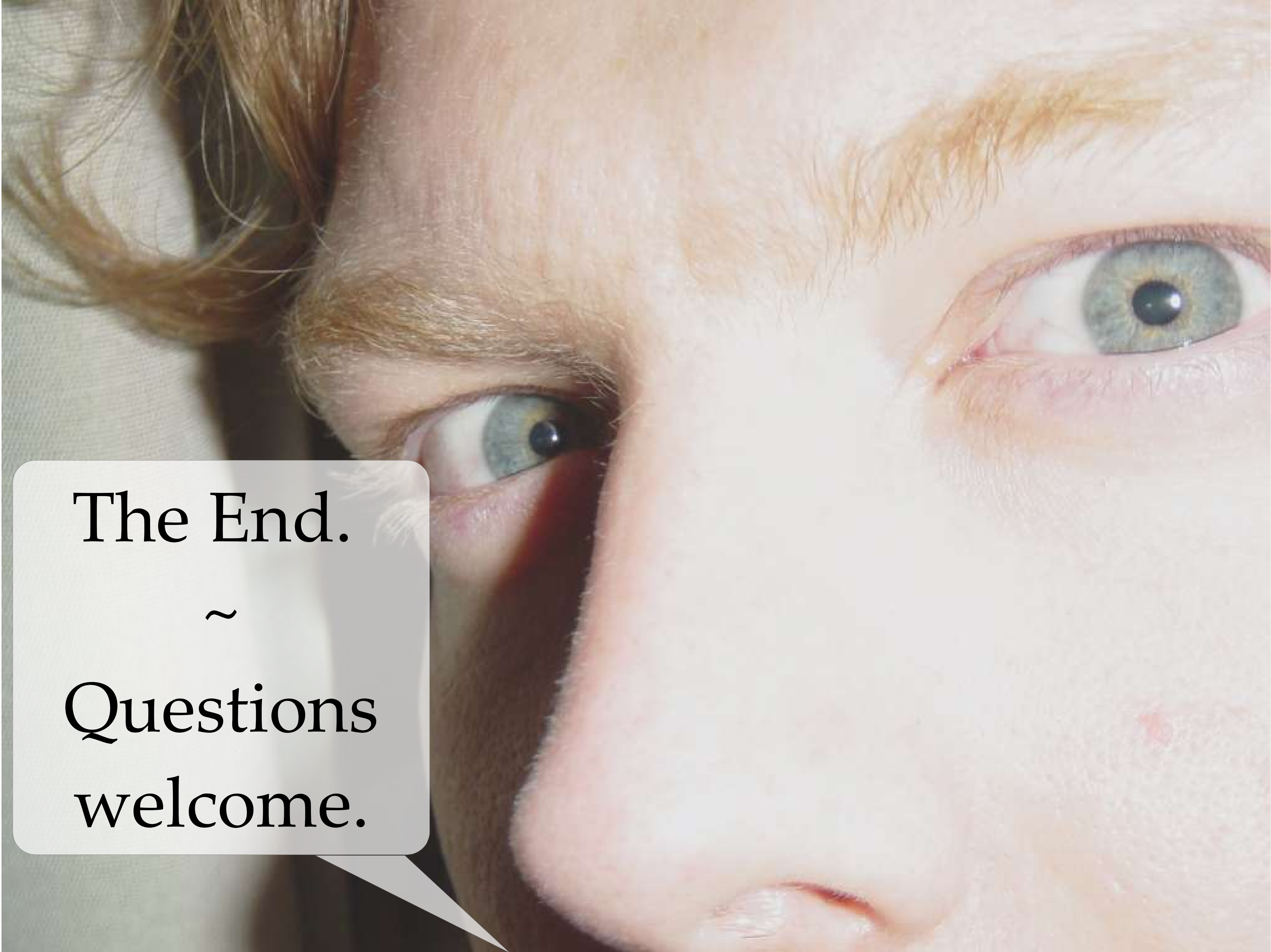
EDA Vendors
- BullDAST
- ChipVision
- OSC

Users
- ARM
- Alcatel
- Atmel
- Motorola



Selected Scientific Publications

- Book chapters:
  - “Estimation of the execution time and energy consumption at source code”, in F. Catthoor, J. I. Gomez, S. Himpe, Z. Ma, P. Marchal, D. P. Scarpazza, C. Wong, P. Yang, “Systematic methodology for real-time cost-effective mapping of dynamic concurrent task-based systems on heterogeneous platforms”, Springer Verlag [accepted];
- 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];
  - with Carlo Brandolese, “Energy estimation for Embedded Software”, IEEE Transactions on Computers;
- 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 e^3tools can and cannot do

- The e^3tools perform **source level estimation**
  of the **ALU and control flow** contributions
  of {time, energy} consumption of a ANSI C program

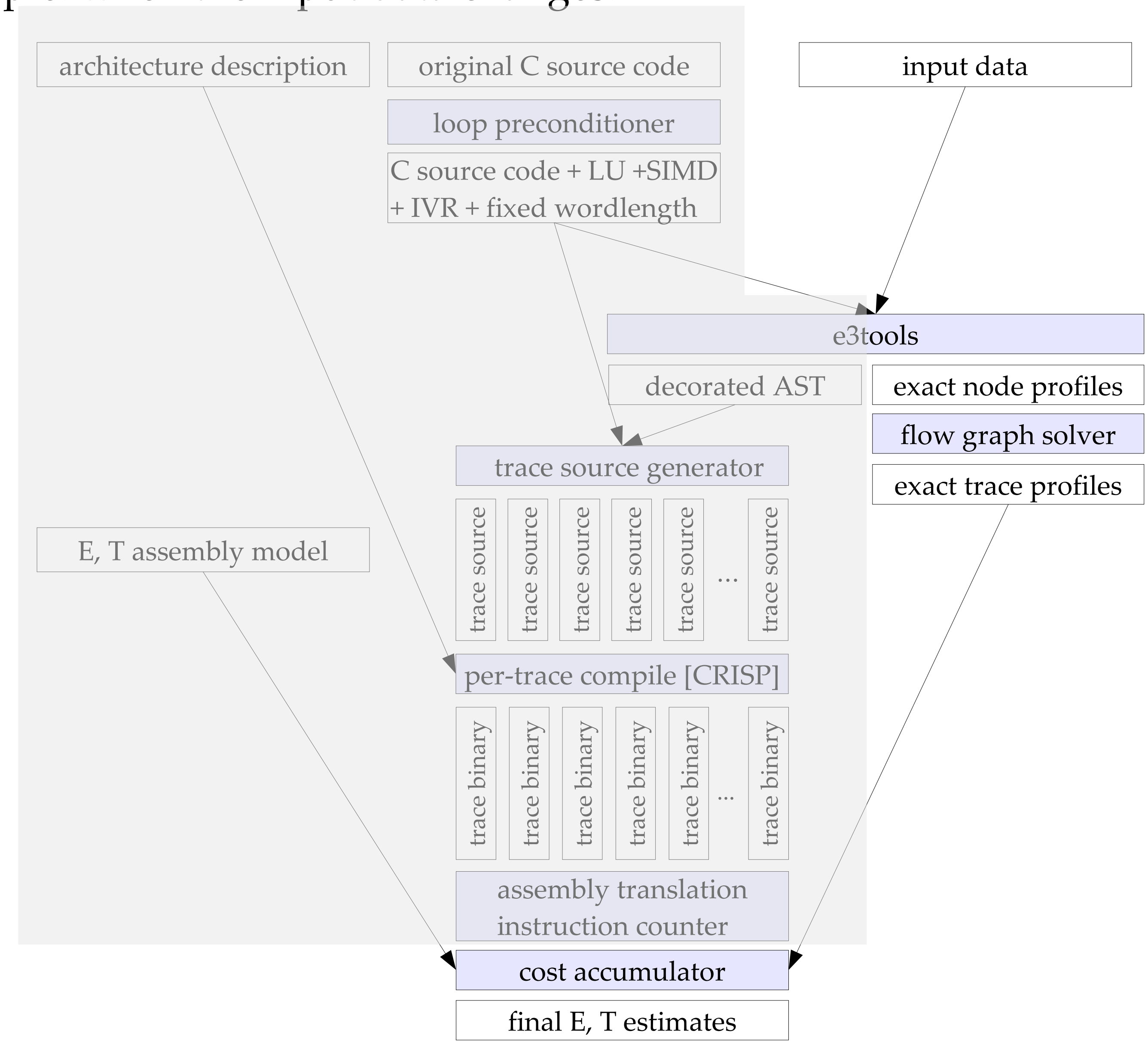
- They are NOT designed for data transfer and storage
  exploration and optimization
  (although: possible estimation for software-controlled memories,
  e.g. Feenecs SPM + VWR)

- In this sense, e^3tools are perfectly complementary
  with *Atomium/PowerEscape*

4.3. Uses & developments: VLIW

1. Opt.
2. VWR
3. VLIW
4. C++

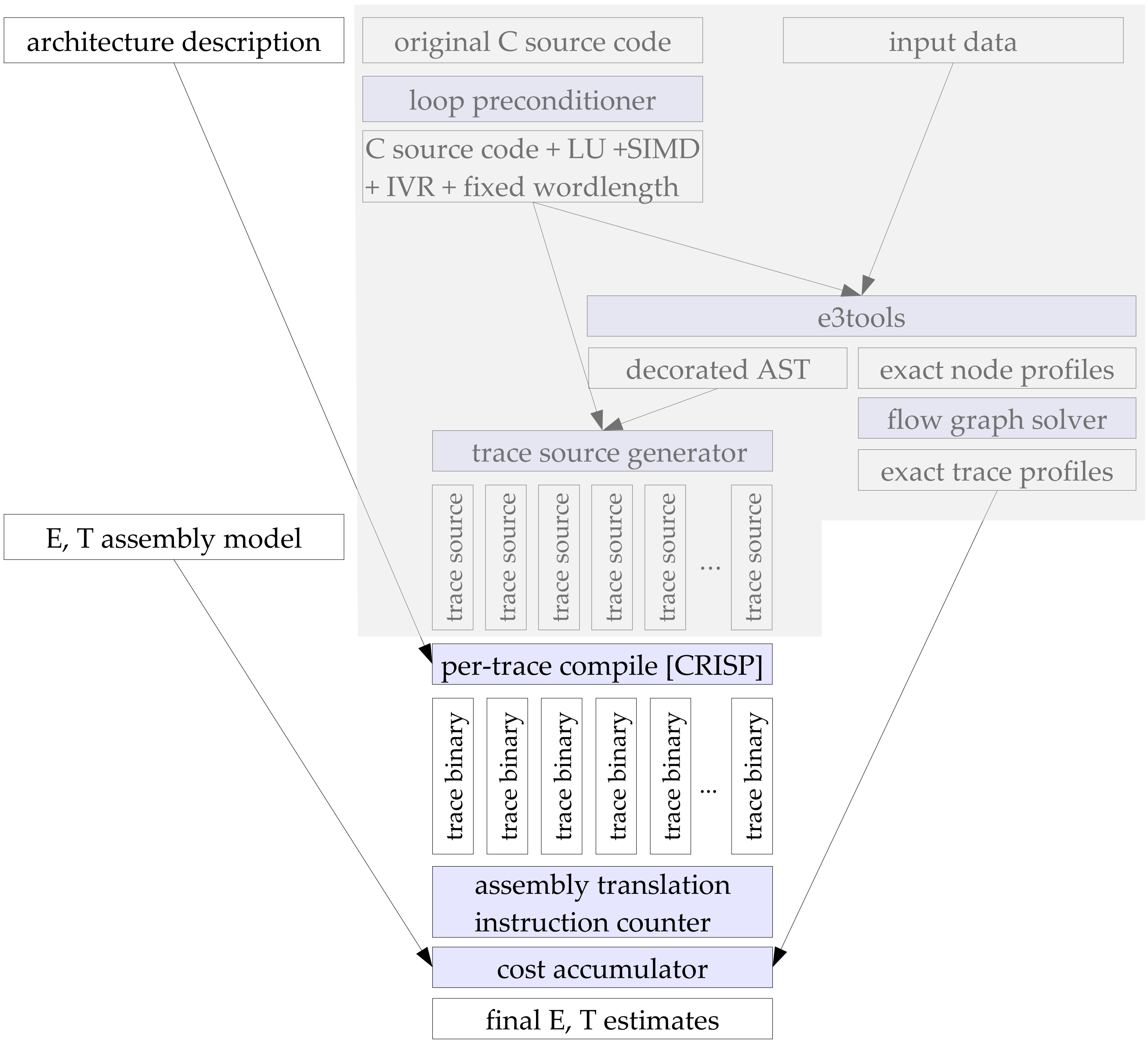
Minimal incremental rebuild.
Example: when the input data changes:



4.3. Uses & developments: VLIW

1. Opt.
2. VWR
3. VLIW
4. C++

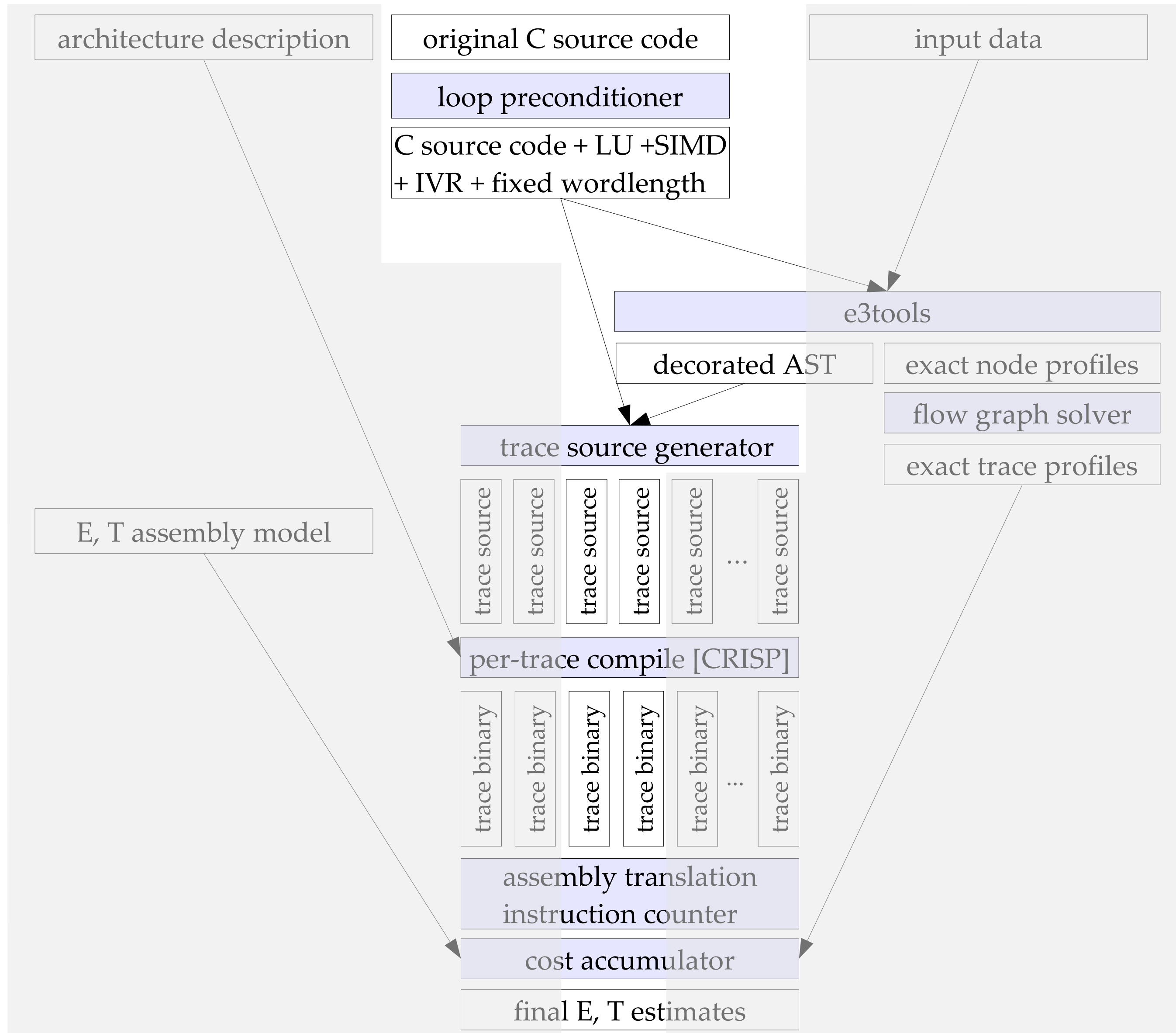
Minimal incremental rebuild.
Example: when architecture changes:



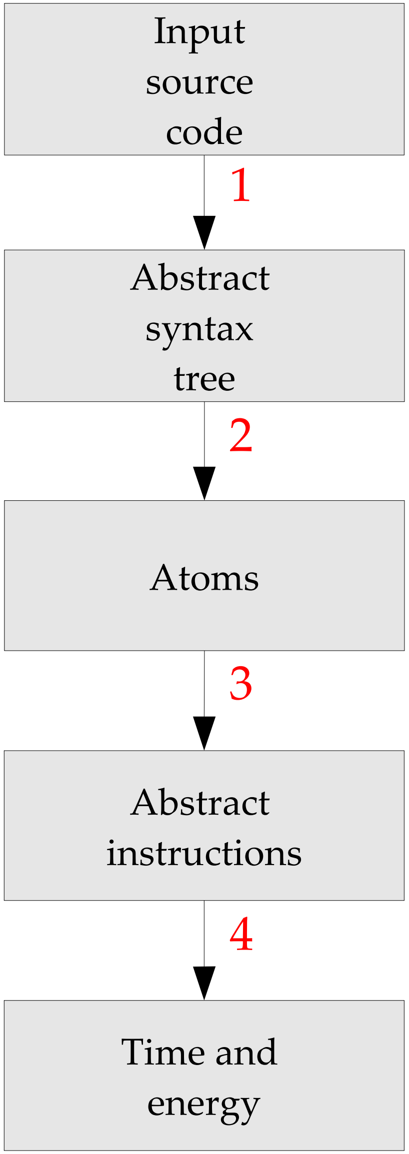
4.3. Uses & developments: VLIW

1. Opt.
2. VWR
3. VLIW
4. C++

Minimal incremental rebuild.
Example: when source code changes

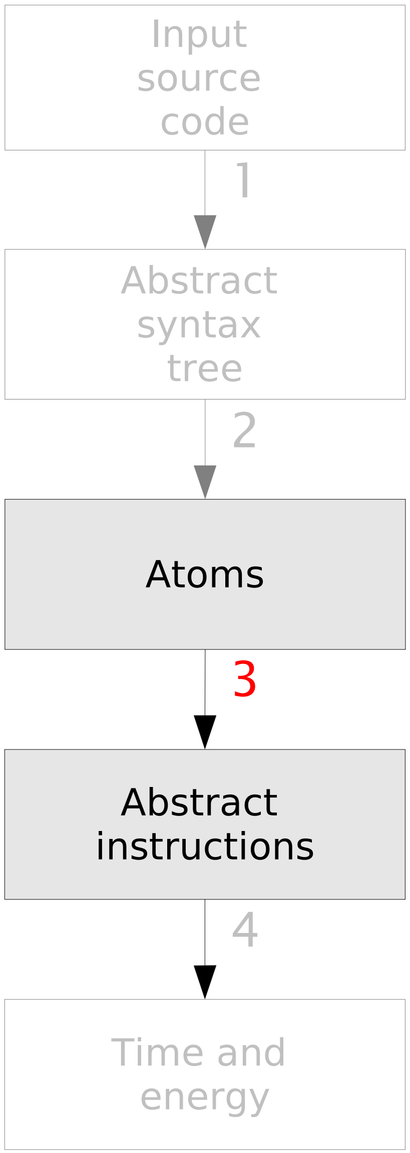


User-definable models



- 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:

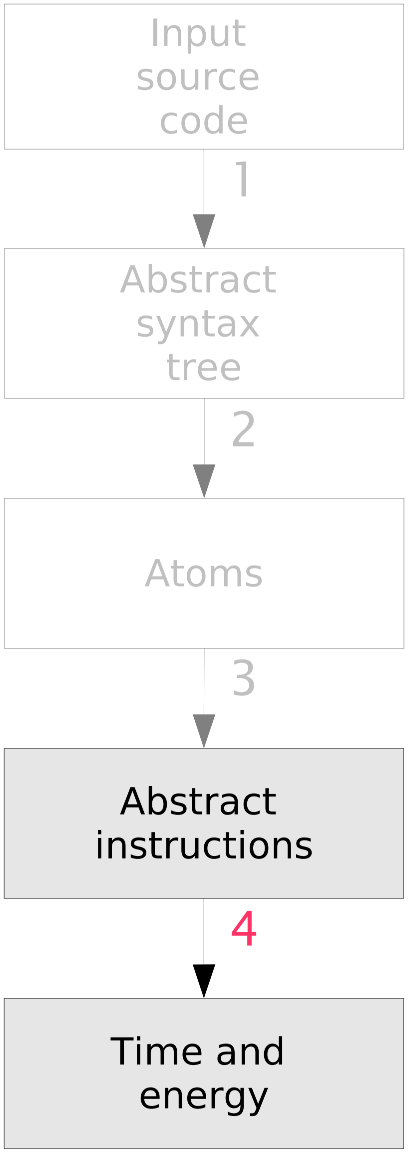


See directories and associated files under:

```
/scratch/scarpaz/poet/4.3/root/lib/compiler
```

```
IntAdd                  = 1 alul;
IntSub                  = 1 alul;
IntMul                  = 1 aluh;
BitwiseOperation        = 1 alul;
IntRelation             = 1 cmpl + 1 jump;
IntImplicitRelation     = 1 cmpl + 1 jump;
...
RValueStar              = 1 mvld;
LValueStar              = 1 mvst;
RLValueStar             = 1 mvld + 1 mvst;
RValueStarNext          = 1 mvld           + 1 alul;
LValueStarNext          = 1 mvst           + 1 alul;
RLValueStarNext         = 1 mvld + 1 mvst  + 1 alul;
...
Break                   = 1 jump;
Continue                = 1 jump;
Goto                    = 1 jump;
...
While                   = 1 cjt;
WhileBody               = 1 cjn + 1 jump;
Do                      = -1 cjt + 1cjn;
DoBody                  = 1 cjt;
For                     = 1 cjt;
ForBody                 = 1 cjn + 1 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`

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

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

VDD	1.5	V
FCK	206.4	MHz
MAINI	0.0	uJ
MAINT	0.0	us

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 remaning 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;

Trace source code generator

- 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:

$$\sum_{functions} 2^{N_{if}} \prod_{j \geq 0}^{N_{switch}} Choices$$

    assuming per-function separation;
    otherwise even worse:

$$\prod_{functions} 2^{N_{if}} \prod_{j \geq 0}^{N_{switch}} Choices$$

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