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

Daniele P. Scarpazza
scarpaz@scarpaz.com
Dipartimento di Elettronica e Informazione
Politecnico di Milano

1. The need: Why this research was needed

- 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;

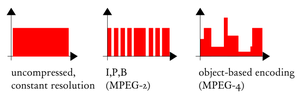
1. fast
2. dynamic
3. source-level
4. fine-detail

1. fast

- size and complexity of modern embedded applications is increasing quickly;
- instruction-set simulation is 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;

2. dynamic

- 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;



- workload variability 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;

3. source-level

- 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; their steering need source-level analysis;

4. fine detail

- most of the time and energy is 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"

1.3 Current techniques do not satisfy these requirements

- Static Timing Analysis (STA) techniques cannot deal with dynamism; [Puschner89,..., Chen01] ~~Fast~~ Dyn Src Fine
- Instruction-Set Simulation (ISS) is slow and at a low level: [Brooks00, Sinha01, Qin03] Fast Dyn ~~Src~~ Fine
- ISS + gprof provide estimates only at a function level; [Simunic01] Fast Dyn Src ~~Fine~~
- Atomium/PowerEscape is source-level, but only for memory aspects (not our focus); [Bormans99, Arnout05] (Fast Dyn Src Fine)
- SoftExplorer is a static technique; [Senn02] ~~Fast~~ Dyn Src Fine
- Compilation-based approaches do not provide link to source level; [Lajolo99] Fast Dyn ~~Src~~ ~~Fine~~
- SIT is source level (good!) but still unable to resolve chosen clusters; [Ravasi03] Fast Dyn Src ~~Fine~~
- Black-box techniques do not provide any link with source code; [Muttreja04] Fast Dyn ~~Src~~ ~~Fine~~

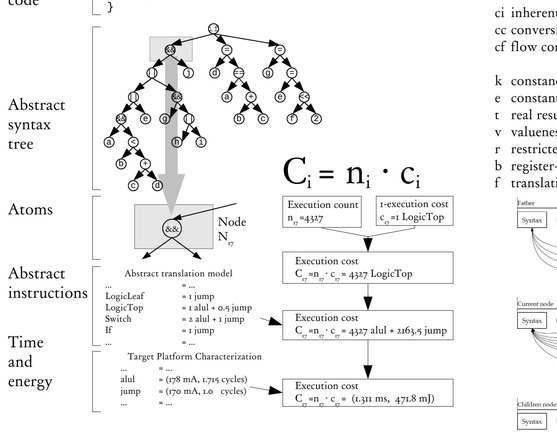
2. Theory: how this technique works

- 2.1 Divide and conquer:

$$\underbrace{C_i}_{\substack{\text{cost of executing}\\ \text{the i-th node in the AST}}} = \underbrace{n_i}_{\substack{\text{execution}\\ \text{count}}} \cdot \underbrace{c_i}_{\substack{\text{single-execution}\\ \text{cost}}}$$

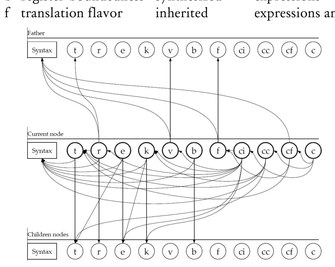
- 2.2 Determine single-execution costs
  via an attribute grammar, founded on an abstract translation model
- 2.3 Determine execution counts
  by instrumenting the original program in an efficient way and running the instrumented program over real input data

2.1. Divide and conquer



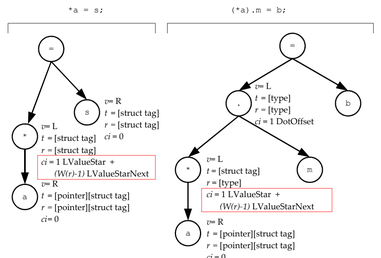
2.2. Determining single-execution costs

Attribute	Computation	Defined for which AST nodes
c total cost	synthesized	expressions and statements
ci inherent cost	synthesized	expressions and statements
cc conversion cost	synthesized	expressions and statements
cf flow control cost	inherited	expressions and statements
k constancy	synthesized	expressions
e constant value	synthesized	expressions
t real result type	synthesized	expressions
v valueness	inherited	expressions
r restricted result type	inherited	expressions
b register-boundedness	synthesized	expressions
f translation flavor	inherited	expressions and statements



- Full C type system needed (attribute t)
  cost of operations and conversions depend on the operands' types
- Full constant expression evaluation needed (attributes k,e)
  constant expressions are resolved at static time (zero runtime cost) and appear in type declarations, and influence operator costs;

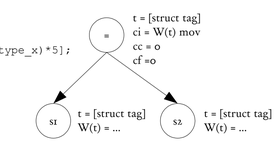
Why attribute r (restricted type) is needed



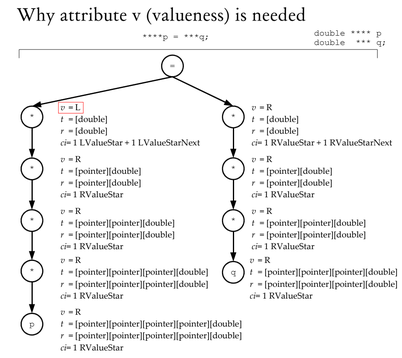
Why attributes t,k,e are needed

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

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



Why attribute v (valueness) is needed



2.3. Determining execution counts

- optimal strategy to select probe insertion points
  - only one probe per each generalized basic block (g.b.b.);
  - a g.b.b. is a maximal set of nodes, 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:

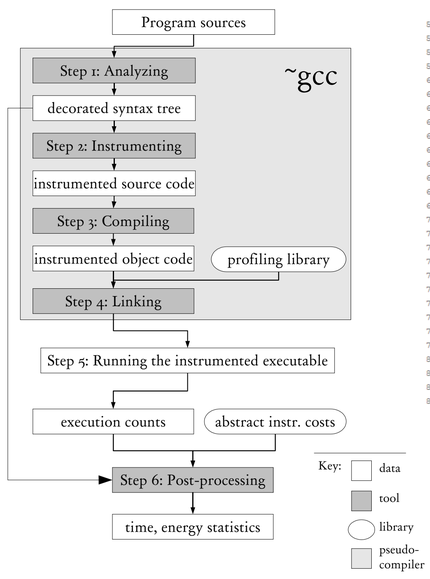
```
int f(args)
{
    ...
}
```

```
int f(args)
{ __profile__(151);
  { ... }
  __profile__(152);
}
```

3. Results: The technique is accurate and fast

- 3.1 ANSI-C compliant flow implementation available
- 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 <8%; correlation with reference > 0.995;
  - performance: simulation times 10,350x shorter than ISS; only 2.2x slower than normal;

3.1 ANSI C-compliant tool flow available




```
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];
```

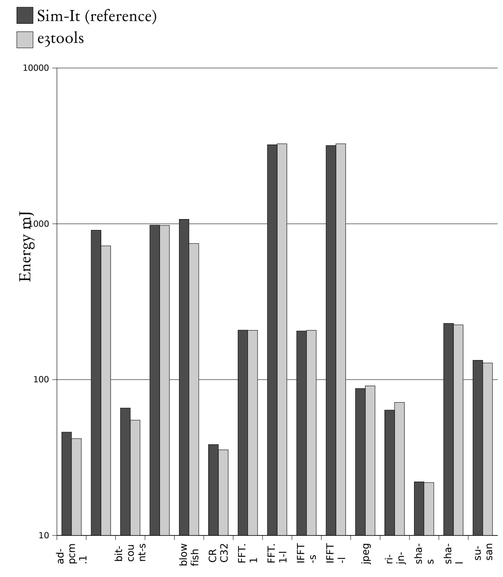
```
#284: 77.24 -- 77.39 Operator -
CPU=0 (Pivot is #408)
k=0, v=R, b=0, t='[float] '
n=53248, c = 1 FloatSub
T = 88*alul
c1E = 1.13837uJ c1T = 4.26357us
cnE = 6.0616mJ cnT = 22.7026ms
CcE = 12.206mJ CcT = 45.6633ms
    #285: 77.24 -- 77.31 Operator *
    CPU=0 (Pivot is #408)
    k=0, v=R, b=0, t='[float] '
    n=53248, c = 1 FloatMul
    T = 87*alul
    c1E = 1.12544uJ c1T = 4.21512us
    cnE = 5.99272mJ cnT = 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
        c1E = 0.0.42442uJ c1T = 0.0484496us
        cnE = 0.0758474mJ cnT = 0.257984ms
        CcE = 0.0758474mJ 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]] [
            n=53248
            CcE = 0mJ CcT = 0ms
            #278: 77.29 -- 77.30 Constant "1"
```

3.3. Accuracy results

	SimIt		e3tools		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:

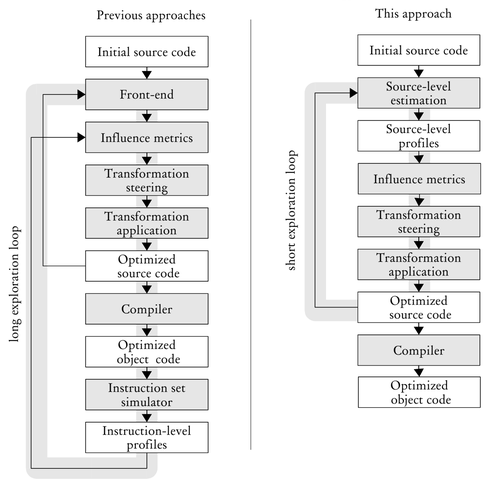
- $\rho(E,\hat{E}) = 0{,}9960$, $\overline{|E-\hat{E}|} = 7{,}49\%$
- $\rho(T,\hat{T}) = 0{,}9987$, $\overline{|T-\hat{T}|} = 5{,}65\%$,



4. Uses and developments

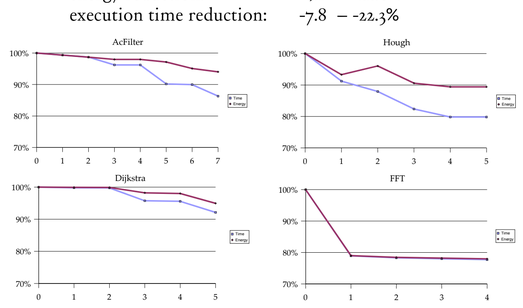
1. Automated source-code optimization
2. Support for VWR architectures
3. Support for VLIW architectures
4. Estimating C++ sources

4.1 Automated source optimization



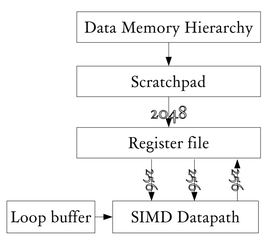
Results:

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



4.2 Support for VWR

- 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);

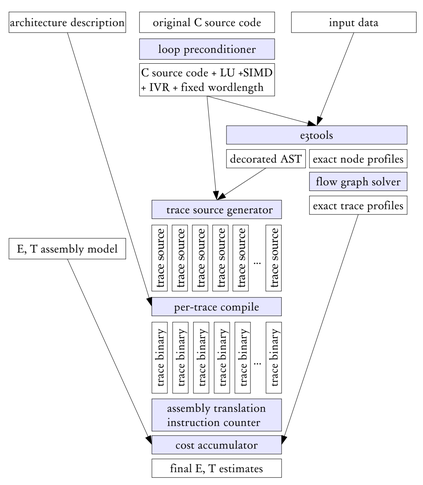


- We added features to:
  1. map code to different executors
  2. mark concurrent code
  3. define intrinsics for scratchpad transfer costs;
  4. define intrinsics for SIMD operations;

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

4.3 Support for VLIW

- 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;
  obtain trace-based profiles by current node-based profiles;



4.4 Extension to C++

- The extension to C++ is feasible with acceptable effort;
- Tasks required:
  - lexer adaptation (28 new keywords, negligible effort);
  - parser syntax adaptation 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 / method has been actually called);
  - extension of theoretical abstract translation model (significant effort);
- Required effort: 1 "me-year"

Selected 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];