Energy Consumption Analysis and Verification

Manuel Hermenegildo\(^1,2\) \quad P. López-García\(^{1,3}\) \quad M. Klemen\(^1\) \quad U. Liqat\(^1\)

\(^1\)IMDEA Software Institute
\(^2\)T. U. of Madrid (UPM)
\(^3\)Spanish Research Council (CSIC)

Dagstuhl 17291 – Jul 17-21, 2017
Energy consumption of computing is a major concern

From high-perf. computing and cloud servers to mobile phones, wearables, implantable/portable medical devices, micro-spacecraft, sensors ...
Energy Consumption Analysis – Approach

Requires low-level modeling – approach: [NASA FM’08]

- Specialize our parametric resource analysis with instruction-level models:
  - Provide energy and data size assertions for each individual instruction. (Energy and data sizes can be constants or functions.)
- CiaoPP then generates statically safe upper- and lower-bound energy consumption functions.

Initially applied to Java bytecode: [NASA FM’08]

- Java bytecode energy consumption models available for simple processors – upper bound consumption per bytecode in joules:

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Inst. Cost in $\mu$J</th>
<th>Mem. Cost in $\mu$J</th>
<th>Total Cost in $\mu$J</th>
</tr>
</thead>
<tbody>
<tr>
<td>iadd</td>
<td>0.957860</td>
<td>2.273580</td>
<td>3.23144</td>
</tr>
<tr>
<td>isub</td>
<td>0.957360</td>
<td>2.273580</td>
<td>3.230.94</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

- Encouraging results: meaningful functions inferred in many cases.
- But no comparison with actual device consumption.
Energy Consumption Analysis – Approach

Requires low-level modeling – approach: [NASA FM’08]

- Specialize our parametric resource analysis with instruction-level models:
  - Provide energy and data size assertions for each individual instruction.
    (Energy and data sizes can be constants or functions.)
- CiaoPP then generates statically safe upper- and lower-bound energy consumption functions.

⇒ Addressed recently: [LOPSTR’13, FOPARA’15, HIP3ES’16]

- Analysis of (embedded) programs written in XC, on XMOS processors.
- Using more sophisticated ISA- and LLVM-level energy models for XMOS XS1 (Bristol & XMOS).
- Comparing to measured energy consumption.
For a description of how resource analysis is performed in CiaoPP, please see the slides of the tutorial, also at this Dagstuhl.
Each instruction is profiled (using, e.g., an Evolutionary Algorithm – EA) to derive upper- and lower-bound energy estimates.

- These are combined using static analysis.

+ Very compositional.
Low-level ISA characterization – interference

Obtaining the cost model: energy consumption/instruction; interference.

Eder, Kerrison – Bristol U / XMOS.
Low-level ISA characterization – operand size

Obtaining the cost model: energy consumption/instruction; operand size.

Eder, Kerrison – Bristol U / XMOS.
Energy model, expressed in the Ciao assertion language

:- package(energy).
:- use_package(library(resources(definition))).
:- load_resource_definition(ciaopp(xcore(model(res_energy)))).

:- trust pred mkmsk_rus2(X)
    : var(X) => (num(X), rsize(X,num(A,B)))
    + ( resource(energy, 1112656, 1112656) ).

:- trust pred add_2rus2(X)
    : var(X) => (num(X), rsize(X,num(A,B)))
    + ( resource(energy, 1147788, 1147788) ).

:- trust pred add_3r2(X)
    : var(X) => (num(X), rsize(X,num(A,B)))
    + ( resource(energy, 1215439, 1215439 )).

:- trust pred sub_2rus2(X)
    : var(X) => (num(X), rsize(X, num(A,B)))
    + ( resource(energy, 1150574, 1150574)).

:- trust pred sub_3r2(X)
    : var(X) => (num(X), rsize(X,num(A,B)))
    + ( resource(energy, 1210759, 1210759 )).

:- trust pred ashr_12rus2(X)
    : var(X) => (num(X), rsize(X,num(A,B)))
    + ( resource(energy, 1219682, 1219682) ).
#include "fact.h"

int fact(int i) {
    if(i<=0) return 1;
    return i*fact(i-1);
}
```assembly
Fact:
entsp 6
stw r0, sp[4]
stw r0, sp[2]
.Lxtalabel0:
ldw r0, sp[4]
ldc r1, 0
lsl r0, r1, r0
bt r0, .LBB0_4
bu .LBB0_3
.LBB0_3:
mkmsk r0, 1
stw r0, sp[3]
bu .LBB0_5
.LBB0_4:
.Lxtalabel1:
ldw r0, sp[4]
sub r1, r0, 1
stw r0, sp[1]
mov r0, r1
.Lxta.call_labels0:
bl fact
ldw r1, sp[1]
mul r0, r1, r0
stw r0, sp[3]
.LBB0_5:
ldw r0, sp[3]
retsp 6
```
Xcore Example: Block Representation

\[\text{fact}\]
0x01: entsp (u6) 0x2
0x02: stw (ru6) r0, sp[0x1]
0x03: ldw (ru6) r1, sp[0x1]
0x04: ldc (ru6) r0, 0x0
0x05: lss (3r) r0, r0, r1
0x06: bf (ru6) r0, 0x1 <0x08>
0x07: bu (u6) 0x2 <0x10>
0x10: ldw (ru6) r0, sp[0x1]
0x11: sub (2rus) r0, r0, 0x1
0x12: bl (u10) -0xc <fact>
0x13: ldw (ru6) r1, sp[0x1]
0x14: mul (l3r) r0, r1, r0
0x15: retsp (u6) 0x2
0x08: mkmsk (rus) r0, 0x1
0x09: retsp (u6) 0x2
Horn Clause Representation

```prolog
:-module(_,[]), [ciaopp(xcore(model(instructions))), ciaopp(xcore(model(energy)))].
:-entry fact/2 : num * var.

fact(R0,R0_4) :-
  entsp_u6(6),
  stw_ru6(R0,Sp2),
  ldw_ru6(R0_1,Sp2),
  stw_ru6(R0_1,Sp4),
  ldw_ru6(R0_2,Sp4),
  ldc_ru6(R1,0),
  lss_3r(R0_3,R1,R0_2),
  bt_ru6(R0_3,47259),
  b1b3(R0_3,Sp1,Sp3,Sp4,R0_4,R1_1,Sp1,Sp3).

b1b3(R0,Sp1,Sp3,Sp4,R0_2,R1,Sp1,Sp3_1) :-
  R0=0,
  bu_u6(43081),
  mkmrk_rus(R0_1,1),
  stw_ru6(R0_1,Sp3_1),
  bu_u6(44182),
  b2(Sp3_1,R0_2).

b1b3(R0,Sp1,Sp3,Sp4,R0_5,R1_1,Sp1_1,Sp3_1) :-
  R0=0,
  ldw_ru6(R0_1,Sp4),
  sub_2rus(R1,R0_1,1),
  stw_ru6(R0_1,Sp1_1),
  odd_2rus(R0_2,R1,0),
  bl_u6(49980),
  fact(R0_2,R0_3),
  ldw_ru6(R1_1,Sp1_1),
  mul_13r(R0_4,R1_1,R0_3),
  stw_ru6(R0_4,Sp3_1),
  b2(Sp3_1,R0_5).

b2(Sp3,R0) :-
  ldw_ru6(R0,Sp3),
  retsp_u6(6).

:-;-- fact.pl <HCBV-15> Top of 777 (7,0) (Ciao) [75%]
```

Hermenegildo, Lopez-Garcia, Klemen, Liqat
CiaoPP Menu

Select Menu Level: naive
Select Action Group: analyze
Select Resource Analysis: res_plai
Select solver: builtin
Select Analysis Layer: isa
Select Output Language: source

{Current Saved Menu Configurations: }

Cancel  Apply
Select Resource Analysis

Preprocessor Option Browser

- Use Saved Menu Configuration: none
- Select Menu Level: naive
- Select Action Group: analyze
- Select Aliasing-Mode Analysis: none
- Select Shape-Type Analysis: none
- Select Resource Analysis: res_plai
- Include Energy Model: yes
- Multivariate Success: off
- Print Program Point Info: off
- Collapse AI Info: on

{Current Saved Menu Configurations: □}

Cancel  Apply
Analysis Results

:- module(_, [fact/2], [ciaopp(xcore(model(instructions))), ciaopp(xcore(model(energy))), assertions]).

:- true pred fact(X, Y)
    : ( num(X), var(Y) )
    => ( num(X), num(Y), rsize(X, num(A, B)), rsize(Y, num('Factorial'(A), 'Factorial'(B))) )
    + ( resource(energy, 6439360, 21469718 * B + 16420396) ).

fact(X, Y) :-
    entsp_u62(_3459),
    _3467 is X,
    stw_ru62(_3476),
    _3484 is X,
    stw_ru62(_3493),
    _3501 is _3467,
    ldw_ru62(_3510),
    _3518 is 0,
    ldc_ru62(_3527),
    _3518<_3501,
    lss_3r2(_3544),
    bt_ru62(_3552),
    1\=0,
    _3569 is _3467,
    ldw_ru62(_3578),
    _3586 is _3569-1,
    sub_2rus2(_3598),
    _3606 is _3569,
    stw_ru62(_3615),
    _3623 is _3586+0,
Analysis Output

```
#include "fact.h"

#pragma true fact(A) ==> (energy <= 2845229*A+1940746)

int fact(int i) {
    if(i<=0) return 1;
    return i*fact(i-1);
}
```
Some Results [LOPSTR’13]

- **Fact(N)**
  - Energy (nJ) vs. N
  - Relative Error

- **Fibonacci(N)**
  - Energy (nJ) vs. N
  - Relative Error

- **Power(base,exp)**
  - Energy (nJ) vs. base,exp
  - Relative Error

- **PowerOfTwo(N)**
  - Energy (nJ) vs. N
  - Relative Error

Legend:
- HW
- SRA
- ISS
- SRA vs. HW
- ISS vs. HW
IR Level Trade-offs

Energy Model + Program (including frontend assertions) → Transformation → Energy Model

Source code → Transformation → LLVM IR

Optimizations → Transformation → Optimized LLVM IR

Energy Model → Optimization → ISA

LLVM IR → Transformation → IR

ISA → Transformation → IR

Hardware → Energy Model

Energy Modelling Precision Loss → Analysis Information loss

Energy Analysis and Verification

Dagstuhl 17291 Jul 17-21’17 18 / 42
<table>
<thead>
<tr>
<th>Program</th>
<th>Analysis at LLVMIR level</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Energy consumption functions (in NJ)</td>
</tr>
<tr>
<td>fact(N)</td>
<td>$28.4 \times N + 22.4$</td>
</tr>
<tr>
<td>fibonacci(N)</td>
<td>$37.53 + 42.3 \times 1.62^N + 11.68 \times (-0.62)^N$</td>
</tr>
<tr>
<td>sqr(N)</td>
<td>$10.52 \times N^2 + 55.79 \times N + 16.5$</td>
</tr>
<tr>
<td>power_of_two(N)</td>
<td>$49.2 \times 2^N - 31.5$</td>
</tr>
<tr>
<td>reverse(N,M)</td>
<td>$20.50 \times N + 72.98$</td>
</tr>
<tr>
<td>concat(N,M)</td>
<td>$69.14 \times N + 69.14 \times M + 14.12$</td>
</tr>
<tr>
<td>mat_mult(N,M)</td>
<td>$44.71 \times N^3 + 72.47 \times N^2 + 52.52 \times N + 25.49$</td>
</tr>
<tr>
<td>sum_facts(N,M)</td>
<td>$69.14 \times N + 69.14 \times M + 14.12$</td>
</tr>
<tr>
<td>fir(N)</td>
<td>$33.47 \times N + 141.6$</td>
</tr>
<tr>
<td>biquad(N)</td>
<td>$165.3 \times N + 54.45$</td>
</tr>
</tbody>
</table>
Measuring Power Consumption on the Hardware

- XMOS XTAG3 measurement circuit.
- Plugs into XMOS XS1 board.

We compare these HW measurements with:
- Static Resource Analysis (SRA).
- Instruction Set Simulation (ISS).
Accuracy vs. HW measurements (ISA and LLVMIR)

[FOPARA'15]

<table>
<thead>
<tr>
<th>Program</th>
<th>Error vs. HW</th>
<th>ISA/LLVMIR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>isa</td>
<td>llvmir</td>
</tr>
<tr>
<td>fact (N)</td>
<td>2.86%</td>
<td>4.50%</td>
</tr>
<tr>
<td>fibonacci (N)</td>
<td>5.41%</td>
<td>11.94%</td>
</tr>
<tr>
<td>sqr (N)</td>
<td>1.49%</td>
<td>9.31%</td>
</tr>
<tr>
<td>power_of_two (N)</td>
<td>4.26%</td>
<td>11.15%</td>
</tr>
<tr>
<td>Average</td>
<td>3.50%</td>
<td>9.20%</td>
</tr>
<tr>
<td>reverse (N,M)</td>
<td>N/A</td>
<td>2.18%</td>
</tr>
<tr>
<td>concat (N,M)</td>
<td>N/A</td>
<td>8.71%</td>
</tr>
<tr>
<td>mat_mult (N,M)</td>
<td>N/A</td>
<td>1.47%</td>
</tr>
<tr>
<td>sum_facts (N,M)</td>
<td>N/A</td>
<td>2.42%</td>
</tr>
<tr>
<td>fir (N)</td>
<td>N/A</td>
<td>0.63%</td>
</tr>
<tr>
<td>biquad (N)</td>
<td>N/A</td>
<td>2.34%</td>
</tr>
<tr>
<td>Average</td>
<td>N/A</td>
<td>3.0%</td>
</tr>
<tr>
<td>Gobal Avg.</td>
<td>3.50%</td>
<td>5.48%</td>
</tr>
</tbody>
</table>
### Accuracy vs. HW measurements (ISA and LLVMIR)

[FOPARA’15]

<table>
<thead>
<tr>
<th>Program</th>
<th>Error vs. HW isa</th>
<th>Error vs. HW llvmir</th>
<th>ISA/LLVMIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact (N)</td>
<td>2.86%</td>
<td>4.50%</td>
<td>0.94</td>
</tr>
<tr>
<td>fibonacci (N)</td>
<td>5.41%</td>
<td>11.94%</td>
<td>0.92</td>
</tr>
<tr>
<td>sqr (N)</td>
<td>1.49%</td>
<td>9.31%</td>
<td>0.91</td>
</tr>
<tr>
<td>power_of_two (N)</td>
<td>4.26%</td>
<td>11.15%</td>
<td>0.93</td>
</tr>
<tr>
<td>reverse (N, M)</td>
<td>N/A</td>
<td>2.18%</td>
<td>N/A</td>
</tr>
<tr>
<td>concat (N, M)</td>
<td>N/A</td>
<td>8.71%</td>
<td>N/A</td>
</tr>
<tr>
<td>mat mult (N, M)</td>
<td>N/A</td>
<td>1.47%</td>
<td>N/A</td>
</tr>
<tr>
<td>sum facts (N, M)</td>
<td>N/A</td>
<td>2.42%</td>
<td>N/A</td>
</tr>
<tr>
<td>fir (N)</td>
<td>N/A</td>
<td>0.63%</td>
<td>N/A</td>
</tr>
<tr>
<td>biquad (N)</td>
<td>N/A</td>
<td>2.34%</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>N/A</strong></td>
<td><strong>3.0%</strong></td>
<td><strong>N/A</strong></td>
</tr>
<tr>
<td><strong>Global Avg.</strong></td>
<td><strong>3.50%</strong></td>
<td><strong>5.48%</strong></td>
<td><strong>0.92</strong></td>
</tr>
</tbody>
</table>

- ISA analysis estimations are reasonably accurate.
- ISA estimations are more accurate than LLVM estimations.
- LLVM estimations are close to ISA estimations.
- Some programs cannot be analysed at the ISA level but can be analyzed at the LLVM level.
@pragma true fir(xn, coeffs, state, N) :
(3347178*N + 13967829 <= energy &&
energy <= 3347178*N + 14417829)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{
  unsigned int ynl; int ynh;
  ynl = (1<<23); ynh = 0;
  for(int j=ELEMENTS-1; j!=0; j--) {
    state[j] = state[j-1];
    {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
  }
  state[0] = xn;
  {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
  if (sext(ynh,24) == ynh) {
    ynh = (ynh << 8) | ((unsigned) ynl) >> 24);
  } else if (ynh < 0) { ynh = 0x80000000; }
  else { ynh = 0x7fffffff; }
  return ynh;
}
XC Analysis Results (FIR Filter, LLVM IR level)

@#pragma true fir(xn, coeffs, state, N) :
    (3347178*N + 13967829 <= energy &&
    energy <= 3347178*N + 14417829)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{
    unsigned int ynl; int ynh;
    ynl = (1<<23); ynh = 0;
    for(int j=ELEMENTS-1; j!=0; j--) {
        state[j] = state[j-1];
        {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
    }
    state[0] = xn;
    {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
    if (sext(ynh,24) == ynh) {
        ynh = (ynh << 8) | ((unsigned) ynl) >> 24);
    } else if (ynh < 0) { ynh = 0x80000000; }
    else { ynh = 0x7fffffff; }
    return ynh;
}
Applications

Performance debugging and verification, resource-oriented optimization, heterogeneous computers, QoS, ...

XC Energy Consumption Verification Tool (using CiaoPP)

Energy Model

Program

Assertions
# pragma check
# pragma trust
...

XC Code
int f(int arg)
...

Energy Consumption Analysis & Verification Tool

HC IR Translator

Static Analysis
# pragma true
Inferred

XC Compiler

Static Comparator
# pragma checked
Proved

# pragma false
Disproved

# pragma check
Unproved

[HIP3ES’15]
Resource Usage Verification – Function Comparisons
[ICLP’10, FOPARA’12]

![Diagram showing resource usage and input data size comparison]

- ** RESOURCE USAGE 
- ** SPECIFICATION UPPER/LOWER BOUNDS (SU/SL) 
- ** SPECIFICATION INTERVALS 

**SU**

**SL**

**INPUT DATA SIZE**
Resource Usage Verification – Function Comparisons
[ICLP’10, FOPARA’12]
Resource Usage Verification – Function Comparisons
[ICLP’10, FOPARA’12]
XC Program (FIR Filter) w/Energy Specification [HIP3ES’15]

```c
#pragma check fir(xn, coeffs, state, N) :
    (1 <= N) ==> (energy <= 416079189)

#pragma true fir(xn, coeffs, state, N) :
    (3347178*N + 13967829 <= energy &&
     energy <= 3347178*N + 14417829)

#pragma checked fir(xn, coeffs, state, N) :
    (1 <= N && N <= 120) ==> (energy <= 416079189)

#pragma false fir(xn, coeffs, state, N) :
    (121 <= N) ==> (energy <= 416079189)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{
    unsigned int ynl; int ynh;
    ynl = (1<<23); ynh = 0;
    for(int j=ELEMENTS-1; j!=0; j--) {
        state[j] = state[j-1];
        {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
    }
    state[0] = xn;
    {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
    if (sext(ynh,24) == ynh) {
        ynh = (ynh << 8) | (((unsigned) ynl) >> 24);
    } else if (ynh < 0) { ynh = 0x80000000; }
    else { ynh = 0x7fffffff; }
    return ynh;
}
```

Hermenegildo, Lopez-Garcia, Klemen, Liqat

Energy Analysis and Verification

Dagstuhl 17291 Jul 17-21'17 27 / 42
XC Program (FIR Filter) w/Energy Specification [HIP3ES’15]

#pragma check fir(xn, coeffs, state, N) :
    (1 <= N) ==> (energy <= 416079189)

#pragma true fir(xn, coeffs, state, N) :
    (3347178*N + 13967829 <= energy &&
     energy <= 3347178*N + 14417829)

#pragma checked fir(xn, coeffs, state, N) :
    (1 <= N && N <= 120) ==> (energy <= 416079189)

#pragma false fir(xn, coeffs, state, N) :
    (121 <= N) ==> (energy <= 416079189)

int fir(int xn, int coeffs[], int state[], int ELEMENTS)
{
    unsigned int ynl, int ynh;
    ynl = (1<<23); ynh = 0;
    for(int j=ELEMENTS-1; j!=0; j--) {
        state[j] = state[j-1];
        {ynh, ynl} = macs(coeffs[j], state[j], ynh, ynl);
    }
    state[0] = xn;
    {ynh, ynl} = macs(coeffs[0], xn, ynh, ynl);
    if (sext(ynh,24) == ynh) {
        ynh = (ynh << 8) | (((unsigned) ynl) >> 24);
    } else if (ynh < 0) { ynh = 0x80000000; }
    else { ynh = 0x7fffffff; }
    return ynh;
}
Modeling at the Instruction Level

- Each instruction is profiled (using, e.g., an Evolutionary Algorithm – EA) to derive upper- and lower-bound energy estimates.
- These are combined using static analysis.

+ Very compositional.
- Bounds obtained are very conservative.
- Dependence among instructions is not modeled (or complex).
Each basic block is profiled using the EA and upper/lower bounds estimated for each block.

Bounds over basic blocks are composed (by static analysis) to infer the bounds over whole program.

- Inter-instruction dependence is captured within the blocks: more precise bounds.
- The EA is precise and practical since no data dependent branching within a block.
- Infers functions of input data sizes.

- Inter-block dependence may be over- or under-estimated.
Overview of our Approach

SRA estimation of the whole program

Upper- and lower-bound cost functions on input data sizes of the program.

EA estimation of UB/LB of basic blocks

<table>
<thead>
<tr>
<th>Basic Block</th>
<th>UB</th>
<th>LB</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Dividing the Program into Basic Blocks

Listing 1: Basic blocks of factorial function

```
<fact>:
01: entsp 0x2
02: stw r0, sp[0x1]
03: ldw r1, sp[0x1]
04: ldc r0, 0x0
05: lss r0, r0, r1
06: bf r0, <08>
```

B1

```
07: bu <010>
10: ldw r0, sp[0x1]
11: sub r0, r0, 0x1
12: bl <fact>
13: ldw r1, sp[0x1]
14: mul r0, r1, r0
15: retsp 0x2
```

B2

```
block before call
07: bu <010>
10: ldw r0, sp[0x1]
11: sub r0, r0, 0x1
12: bl <fact>
13: ldw r1, sp[0x1]
14: mul r0, r1, r0
15: retsp 0x2
```

B3

```
08: mkmsk r0, 0x1
09: retsp 0x2
```

Listing 2: Modified basic blocks

```
<fact>:
01: entsp 0x2
02: stw r0, sp[0x1]
03: ldw r1, sp[0x1]
04: ldc r0, 0x0
05: lss r0, r0, r1
06: bf r0, <08_NEW>
08_NEW:
```

B1

```
```

B2

```
```

B2

```
block after call
08: mkmsk r0, 0x1
09: retsp 0x2
```

B3
### Experimental Evaluation (XMOS XS1 architecture)

<table>
<thead>
<tr>
<th>Program</th>
<th>Upper/Lower Bounds (nJ) $\times 10^3$</th>
<th>vs. HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact($N$)</td>
<td>$E_u = 5.1 , N + 4.2$\n$E_l = 4.1 , N + 3.8$</td>
<td>7%</td>
</tr>
<tr>
<td>fibonacci($N$)</td>
<td>$E_u = 5.2 , lucas(N) + 6 , fib(N) - 6.6$\n$E_l = 4.5 , lucas(N) + 5 , fib(N) - 4.2$</td>
<td>8.71%</td>
</tr>
<tr>
<td>reverse($N$)</td>
<td>$E_u = 3.7 , N + 13.3$\n$E_l = 2.95 , N + 12$</td>
<td>8%</td>
</tr>
<tr>
<td>findMax($N$)</td>
<td>$E_u = 5 , N + 6.9$\n$E_l = 3.3 , N + 5.6$</td>
<td>8.7%</td>
</tr>
<tr>
<td>fir($N$)</td>
<td>$E_u = 6 , N + 26.4$\n$E_l = 4.8 , N + 22.9$</td>
<td>8.9%</td>
</tr>
<tr>
<td>biquad($N$)</td>
<td>$E_u = 29.6 , N + 10$\n$E_l = 23.5 , N + 9$</td>
<td>9.8%</td>
</tr>
</tbody>
</table>

- EA times vary depending upon the initialization parameters.
  - On average within 150-200 min.
- Static analysis times are relatively small $\approx 4$ sec.
Experimental Results (Benchmark with no Data Dependent Branching)

factorial(x): 7% over- and 11% under-approximation for random runs with different inputs.
Experimental Results (Benchmark with Data Dependent Branching)

findMax(arr,N): 8.7% over- and 9% under-approximation from actual upper- and lower-bounds (ascending vs. descending sorted array).
Inferring Accumulated Cost: Hot Spot Detection!

[TPLP’16, FLOPS’16]

- Helping developers make (resource-related) design decisions:
  - Which parts of the program are the most resource-consuming?
  - Which predicates should be optimized first?

- The standard/classical notion of cost only partially meets these objectives:
  - Predicates w/highest (standard) costs may not need to be optimized first.
  - E.g., perhaps predicates with lower costs but which are called more often.
  - The input sizes to such calls are also relevant.

- Need info resulting from a static profiling of the program to:
  - identify the parts of a program responsible for highest fractions of the cost → accumulated cost.
  - I.e., how the total resource usage of the execution of a program is distributed over selected parts of it (cost centers → predicates).

Static profiling → static inference of the kinds of information that are usually obtained at run-time by profilers.

Main contribution

Novel, general, and flexible framework for setting up cost equations/relations. → can be instantiated for performing a wide range of static resource usage analyses, including both accumulated cost and standard cost.
Definition: Accumulated Cost

The cost of a (single) call \( p(n) \) accumulated in cost center \( q \), denoted \( C_q^p(n) \):

- Is the **sum of the costs** of all the computations that are descendants (in the call stack) of the call \( p(n) \), and are **under the scope of any call to** \( q \).

- We say that a computation is **under the scope** of a call to cost center \( q \), if the **closest ancestor** of such computation in the call stack that is a cost center, is \( q \).

- Expresses how much of the standard cost of the call to \( p \) is attributed to \( q \).
The Team

- Working specifically in CiaoPP resource analysis:
  - Pedro López-García
  - Manuel Hermenegildo
  - Maximiliano Klemen
  - Umer Liqat

- CiaoPP overall:
  - José-Francisco Morales
  - Nataliia Stulova
  - Isabel García-Contreras

- Previous main contributors to CiaoPP resource analysis:
  - Saumya Debray
  - Nai-wei Lin
  - Jorge Navas
  - Alejandro Serrano
  - Mario Méndez-Lojo
  - Edison Mera

Work currently at: IMDEA Software Institute, T.U. Madrid (UPM).
And previously at: U. T. Austin, MCC, U. of Arizona, U. of New Mexico.
Playground at: [http://play.ciao-lang.org](http://play.ciao-lang.org)
Thank you!
Bibliography and Timeline Summary of our Work in the Area
1990 Method for static inference of upper-bound functions on execution cost and data structure sizes [PLDI’90] (building on Wegbreit):

- Techniques for setting up, solving/approximating recurrence relations.
- For Horn-clause programs → used widely as IR for other languages.
- Motivation: task granularity control in automatic parallelization.
- Experimental results (resulting in improved parallel speedups).
- Implementation (leading to CASLOG) but I/O arguments, types, measures, etc. had to be provided by the user.

1993-1994 First fully automatic system, including all auxiliary analyses: GraCos (Granularity Control System), implemented within CiaoPP [SAS’94, PASCO’94].

- Reducing data size computation overhead. [ICLP’95]
- Further improvements. [JSC’96]
- Precision improved w/determinacy, partial eval. . . . [LOPSTR’04, NGC’10]

1997 Lower bounds cost analysis; divide-and-conquer. [ILPS’97]

- Lower bounds required developing non-failure (no-exceptions) analysis, guard coverage, ... [ICLP’97, FLOPS’04]
- Also in [ILPS’97]: proposed non-deterministic recurrence relations, special for divide-and-conquer programs: looking at sets of computation trees and balancing/bounding node cost (e.g., quadratic bound for qsort).
1990  Method for static inference of upper-bound functions on execution cost and data structure sizes [PLDI'90] (building on Wegbreit):

- Techniques for setting up, solving/approximating recurrence relations.
- For Horn-clause programs → used widely as IR for other languages.
- Motivation: task granularity control in automatic parallelization.
- Experimental results (resulting in improved parallel speedups).
- Implementation (leading to CASLOG) but I/O arguments, types, measures, etc. had to be provided by the user.

1993-1994  First fully automatic system, including all auxiliary analyses: GraCos (Granularity Control System), implemented within CiaoPP [SAS'94, PASCO'94].

- Reducing data size computation overhead. [ICLP'95]
- Further improvements. [JSC'96]
- Precision improved w/determinacy, partial eval. . . . [LOPSTR'04, NGC'10]

1997  Lower bounds cost analysis; divide-and-conquer. [ILPS'97]

- Lower bounds required developing non-failure (no-exceptions) analysis, guard coverage, ... [ICLP'97, FLOPS'04]
- Also in [ILPS'97]: proposed non-deterministic recurrence relations, special for divide-and-conquer programs: looking at sets of computation trees and balancing/bounding node cost (e.g., quadratic bound for qsort).
<table>
<thead>
<tr>
<th>Year</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>Abstraction carrying code for resources.</td>
</tr>
<tr>
<td>2006</td>
<td>Probabilistic Cost Analysis.</td>
</tr>
<tr>
<td>2007</td>
<td><strong>User-definable</strong> resources.</td>
</tr>
<tr>
<td>2007</td>
<td>Multi-language support (Java bytecode, C#, FP, CLP) via Horn clause-based IR.</td>
</tr>
<tr>
<td></td>
<td>Combined with user-definable resources: no need to develop specific analyzers for specific languages!</td>
</tr>
<tr>
<td></td>
<td>Instrumental analyses: sharing/nullity/class dependence shape dependence</td>
</tr>
<tr>
<td>2008</td>
<td>Application to execution time (using bytecode-level models, obtained by regression).</td>
</tr>
<tr>
<td>2008</td>
<td>Application to energy consumption of Java bytecode.</td>
</tr>
<tr>
<td>2009</td>
<td>User-definable resources for Java bytecode.</td>
</tr>
<tr>
<td>Year</td>
<td>Event/Description</td>
</tr>
<tr>
<td>------</td>
<td>-------------------</td>
</tr>
<tr>
<td>2004</td>
<td>Abstraction carrying code for resources. [PPDP’05, LPAR’04]</td>
</tr>
<tr>
<td>2006</td>
<td>Probabilistic Cost Analysis. [CLEI’06]</td>
</tr>
<tr>
<td>2007</td>
<td>User-definable resources. [ICLP’07]</td>
</tr>
<tr>
<td>2007</td>
<td>Multi-language support (Java bytecode, C#, FP, CLP) via Horn clause-based IR. [LOPSTR’07]</td>
</tr>
<tr>
<td>2008</td>
<td>Application to execution time (using bytecode-level models, obtained by regression). [PPDP’08]</td>
</tr>
<tr>
<td>2008</td>
<td>Application to energy consumption of Java bytecode. [NASA FM’08]</td>
</tr>
<tr>
<td>2009</td>
<td>User-definable resources for Java bytecode. [Bytecode’09]</td>
</tr>
</tbody>
</table>

- Combined with user-definable resources: no need to develop specific analyzers for specific languages!
- Instrumental analyses: sharing/nullity/class dependence shape [VMCAI’08, PASTE’08, CC’08, SAS’02, LCPC’08].
<table>
<thead>
<tr>
<th>Year</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>Abstraction carrying code for resources.</td>
<td>[PPDP'05, LPAR'04]</td>
</tr>
<tr>
<td>2006</td>
<td>Probabilistic Cost Analysis.</td>
<td>[CLEI'06]</td>
</tr>
<tr>
<td>2007</td>
<td><strong>User-definable</strong> resources.</td>
<td>[ICLP'07]</td>
</tr>
<tr>
<td>2007</td>
<td>Multi-language support (Java bytecode, C#, FP, CLP) via Horn clause-based IR.</td>
<td>[LOPSTR'07]</td>
</tr>
<tr>
<td></td>
<td>Combined with user-definable resources: no need to develop specific analyzers for specific languages!</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instrumental analyses: sharing/nullity/class [VMCAI'08, PASTE'08] dependence [LCPC'08] shape [CC'08, SAS'02].</td>
<td></td>
</tr>
<tr>
<td>2008</td>
<td>Application to <strong>execution time</strong> (using bytecode-level models, obtained by regression).</td>
<td>[PPDP'08]</td>
</tr>
<tr>
<td>2008</td>
<td>Application to <strong>energy consumption</strong> of Java bytecode.</td>
<td>[NASA FM'08]</td>
</tr>
<tr>
<td>2009</td>
<td>User-definable resources for Java bytecode.</td>
<td>[Bytecode'09]</td>
</tr>
</tbody>
</table>
2010-2015 Resource verification: interval-based, improved function comparison. [ICLP’10, FOPARA’12, HIP3ES’15]

2012-2014 Cost analysis as multivariant abstract interpretation. [TPLP’14]

→ Multivariant, integrated with assertion checking, modular, incremental.

Domain: interval (piece-wise) functions. [ICLP’10, FOPARA’12]

2013 Using sized shapes (sized types). [ICLP’13]

2013-2016 Analysis and verification of Energy:

- At the ISA level [LOPSTR’13]
- Comparing LLVM and ISA levels [FOPARA’15]
- At the block level [HIP3ES’16]

2016 Static profiling / accumulated cost. [FLOPS’16, TPLP’16]
2010-2015 Resource verification: interval-based, improved function comparison. [ICLP'10, FOPARA'12, HIP3ES'15]

2012-2014 Cost analysis as multivariant abstract interpretation. [TPLP'14]

→ Multivariant, integrated with assertion checking, modular, incremental.

Domain: interval (piece-wise) functions. [ICLP'10, FOPARA'12]

2013 Using sized shapes (sized types). [ICLP'13]

2013-2016 Analysis and verification of Energy:

- At the ISA level [LOPSTR'13]
- Comparing LLMV and ISA levels [FOPARA'15]
- At the block level [HIP3ES'16]

2016 Static profiling / accumulated cost. [FLOPS'16, TPLP'16]
CiaoPP References – Analysis and Verification of Energy


CiaoPP References – Intermediate Repr. / Multi-Lingual Support

A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.

CiaoPP References – Analysis and Verification of Resources in General


A Transformational Approach to Parametric Accumulated-cost Static Profiling.

Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types.

Sized Type Analysis of Logic Programs (Technical Communication).
Interval-Based Resource Usage Verification: Formalization and Prototype


User-Definable Resource Usage Bounds Analysis for Java Bytecode.

Towards Execution Time Estimation in Abstract Machine-Based Languages.

User-Definable Resource Bounds Analysis for Logic Programs.

Probabilistic Cost Analysis of Logic Programs: A First Case Study.
In XXXII Latin-American Conference on Informatics (CLEI 2006), August 2006.

ICLP 2017 10-year Test of Time Award.
Lower Bound Cost Estimation for Logic Programs.

Estimating the Computational Cost of Logic Programs.

[ICLP’95] P. López-García and M. Hermenegildo.
Efficient Term Size Computation for Granularity Control.

A Methodology for Granularity Based Control of Parallelism in Logic Programs.

Towards Granularity Based Control of Parallelism in Logic Programs.

Task Granularity Analysis in Logic Programs.
CiaoPP References – Assertion Language


CiaoPP References – Overall Debugging and Verification Model

Reducing the Overhead of Assertion Run-time Checks via static analysis.

Practical Run-time Checking via Unobtrusive Property Caching.

[ICLP’09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.

Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation

Program Development Using Abstract Interpretation (and The Ciao System Preprocessor).
In 10th International Static Analysis Symposium (SAS’03), number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

A Generic Preprocessor for Program Validation and Debugging.
Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs.

Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging.

On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs.

**CiaoPP References – Abstraction-Carrying Code**

Reduced Certificates for Abstraction-Carrying Code.

Abstraction Carrying Code and Resource-Awareness.

Abstraction-Carrying Code.
CiaoPP References – Fixpoint-based Analyzer (Abstract Interpreter)


CiaoPP References – Modular Analysis, Analysis of Concurrency

A Practical Type Analysis for Verification of Modular Prolog Programs.  
In PEPM'08, pages 61–70. ACM Press, January 2008.

Context-Sensitive Multivariant Assertion Checking in Modular Programs.  
In LPAR'06, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

A Model for Inter-module Analysis and Optimizing Compilation.  

[ENTCS'00] G. Puebla and M. Hermenegildo.  
Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs.  

Analyzing Logic Programs with Dynamic Scheduling.  

Global Analysis of Standard Prolog Programs.  
CiaoPP References – Domains: Sharing/Aliasing

Identification of Logically Related Heap Regions.

Efficient Set Sharing using ZBDDs.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

Identification of Heap-Carried Data Dependence Via Explicit Store Heap Models.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

Sharing Analysis of Arrays, Collections, and Recursive Structures.

Precise Set Sharing Analysis for Java-style Programs.

Efficient top-down set-sharing analysis using cliques.
Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation.

Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.

CiaoPP References – Domains: Shape/Type Analysis

Sized Type Analysis of Logic Programs (Technical Communication).

Efficient context-sensitive shape analysis with graph-based heap models.

Heap Analysis in the Presence of Collection Libraries.

More Precise yet Efficient Type Inference for Logic Programs.
CiaoPP References – Domains: Non-failure, Determinacy

Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.

Determinacy Analysis for Logic Programs Using Mode and Type Information.

Multivariant Non-Failure Analysis via Standard Abstract Interpretation.

Non-Failure Analysis for Logic Programs.