Energy Analysis, Debugging, and Verification
(with the CaoPP system)

Manuel Hermenegildo\textsuperscript{1,2} \hfill P. López-García\textsuperscript{1,3}
R. Haemmerlé\textsuperscript{1} \hfill M. Klemen\textsuperscript{1} \hfill U. Liqat\textsuperscript{1} \hfill J. Morales\textsuperscript{1}

\textsuperscript{1}IMDEA Software Institute
\textsuperscript{2}Technical University of Madrid (UPM)
\textsuperscript{3}Spanish Research Council (CSIC)

In collaboration with:
\textsuperscript{4}U. Bristol, XMOS, Roskilde U.

K. Georgiou\textsuperscript{4}, N. Grech\textsuperscript{4}, S. Kerrison\textsuperscript{4}, and K. Eder\textsuperscript{4}

EACO Workshop, Bristol, September 10-11, 2014
Analysis/Debugging/Verification of Resources

Automatically infer upper/lower bounds on the usage that a program makes of a general notion of various (user-definable) resources.

- Memory, execution time, execution steps, data sizes, ...
- Bits sent or received over a socket, SMSs sent or received, accesses to a database, calls to a procedure, files left open, money spent, ...
- **Energy consumed**, ...

Key observations about resource consumption:
- Imprecision $\rightarrow$ infer bounds that are safe and as accurate as possible.
- Dependent on input data metrics $\rightarrow$ infer the bounds as functions.
  (Difference with WCET and related methods.)

Developed methodology and tool (CiaoPP) for such analysis where:

1. Programmer defines the resource consumption for basic elements (e.g., instructions, bytecodes, libraries, ...).
2. System infers resource usage bound functions for rest of program.

Applications: performance debugging and verification, resource-oriented optimization, *granularity control in parallelism*, ...

[DLH90, DLGH94, DLGH97, NMLGH07, MLGCH08, NMLH08, NMLH09, LGDB10, SLBH13, LKSGL13, SLH14]
Intermediate Representation:

[MLNH07]

Transformation

Analysis

- Allows supporting multiple languages.
- Block-based CFG. Each block represented as a Horn clause.
- Used for all analyses: aliasing, CHA/shape/types, data sizes, resources, etc.
- Analysis is parametric w.r.t. languages, abstractions, resources, etc.
Xcore Example: Control Flow Graph (CFG)

<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>
0x08: mkmsk (rus) r0, 0x1
0x09: retsp (u6) 0x2
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
Xcore Example: Block Representation

<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
Xcore Example: Block Representation

fact :-
0x01: entsp(0x2)
0x02: stw(r0, sp[0x1])
0x03: ldw(r1, sp[0x1])
0x04: ldc(r0, 0x0)
0x05: lss(r0, r0, r1)
0x06: bf(r0, 0x1 <0x08>)
    branch(bf0, bf1)

bf1 :-
0x07: bu(0x2 <0x10>)
0x10: ldw(r0, sp[0x1])
0x11: sub(r0, r0, 0x1)
0x12: bl(-0xc <fact>)
    call(fact)
0x13: ldw(r1, sp[0x1])
0x14: mul(r0, r1, r0)
0x15: retsp(0x2)

bf0 :-
0x08: mkmsk(r0, 0x1)
0x09: retsp(0x2)

Block Control Flow Graph
Xcore Example: Horn Clause IR

:- entry fact/2 : int * var.
fact(R0,R0_3):-
    entsp(_),
    stw(R0,Sp0x1),
    ldw(R1,Sp0x1),
    ldc(R0_1,0x0),
    lss(R0_2,R0_1,R1),
    bf(R0_2,_),
    bf01(R0_2,Sp0x1,R0_3,R1_1).

bf01(1,Sp0x1,R0_4,R1):-
    bu(_),
    ldw(R0_1,Sp0x1),
    sub(R0_2,R0_1,0x1),
    bl(_),
    fact(R0_2,R0_3),
    ldw(R1,Sp0x1),
    mul(R0_4,R1,R0_3),
    retsp(_).

bf01(0,Sp0x1,R0,R1):-
    mkmsk(R0,0x1),
    retsp(_).
Generating the Intermediate Representation

- **Specifics for Java:**
  - Control flow graph construction from bytecode representation.
  - Elimination of stack variables.
  - Conversion to three-address statements.
  - Explicit representation of this and ret as extra block parameters.

- **Specifics for XC:**
  - Control flow graph construction from ISA (or LLVM IR) representation.
  - Resolving branching to predicates with multiple clauses.
  - Inferring block parameters.

- **Some common tasks:**
  - Generation of block-based CFG.
  - SSA transformation (e.g., splitting of input/output param).
  - Conversion of loops into recursions among blocks.
  - Branching, cases, dynamic dispatch → blocks w/same signature.
  - Representation as Horn clauses.

- Can be done directly or via partial evaluation of an interpreter (implementing the semantics of the low level code) w.r.t. the concrete low-level program.
Transformation

Java Source → javac → Java Bytecode

Java Bytecode → soot + Ciao transform. → IR – CFG (Horn clauses)

Ciao Source → IR – CFG (Horn clauses)

Xcore assembly → xobjdump

Analysis

Fixpoint algorithm (AI-based) → Resource Usage → Resource Model

Sets of Pre/Post pairs → Shapes → Pre/Post pairs

Sharing → CHA → Shapes

Resource Info. → Sizes and Resource Info.

[MH92, BGH99, PH96, HPMS00, NMLH07] [MGH94, BCPH96, PH00, BdIBH+01, PCPH06, PCPH08]
[MH89, MH91, DLGH97, VB02, BLGH04, LGBH05, NBH06, MSHK07]
[MLH08, MKSH08, MMLH+08, MHKS08, MKH09, LGBH10, MLLH08]
Resource Analysis

Transformation

Java Source

Java Bytecode

xobjdump

Ciao Source

Xcore assembly

XC Source

Java parser

soot + Ciao transform.

Analysis

IR – CFG (Horn clauses)

Fixpoint algorithm (AI-based)

Resource Usage

Resource Model

Sets of Pre/Post pairs

Prog. Point Info

Shape

CHA

Sharing

Sizes and Resource Info.

[DLH90, LGHD94, LGHD96, DLGHL94, DLGHL97, NMLGH07, MLNH07, MLGCH08, NMLH08]

[NMLH09, LGDB10, SLBH13, LKSGL13, SLH14]
The objective of the resource analysis is to obtain for each predicate/block *resource usage functions*:

Monotonic arithmetic functions that return lower/upper bounds on the resource usage of the predicate/block given the sizes of its input data.

**Example output**

\[
\text{true } \text{nrev}(x) : \text{list}(x) \rightarrow \text{resource(energy, 0, 1+length}(x)**2)
\]

- \(x\) points to a list \(\Rightarrow\) number of execution steps \(\leq 1 + \text{length}(A)^2\)

To define a new resource and its analysis:

- User just needs to provide (via assertions) the resource usage functions for relevant primitive operations.
- The system provides libraries of predefined resources.
Overview of the Analysis ("classical" approach)

1. Supporting analyses:
   - Class hierarchy analysis simplifies CFG and improves overall precision. Types/shapes for size metrics.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.
   - *Determinacy* (mutual exclusion) to obtain tighter bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine *relative* sizes of variable contents.
   - Size measures are derived from inferred shape (type) information.

3. Compute upper bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - We have an internal recurrence solver, and the system also interfaces with tools like Mathematica, Parma, PUBS, Matlab, etc.
Overview of the Analysis (“classical” approach)

1. Supporting analyses:
   - Class hierarchy analysis simplifies CFG and improves overall precision. Types/shapes for size metrics.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.
   - Determinacy (mutual exclusion) to obtain tighter bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine relative sizes of variable contents.
   - Size measures are derived from inferred shape (type) information.

3. Compute upper bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - We have an internal recurrence solver, and the system also interfaces with tools like Mathematica, Parma, PUBS, Matlab, etc.
Overview of the Analysis (“classical” approach)

1. Supporting analyses:
   - Class hierarchy analysis simplifies CFG and improves overall precision. Types/shapes for size metrics.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - *Non-failure* (no exceptions) inferred for non-trivial lower bounds.
   - *Determinacy* (mutual exclusion) to obtain tighter bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine *relative* sizes of variable contents.
   - Size measures are derived from inferred shape (type) information.

3. Compute upper bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - We have an internal recurrence solver, and the system also interfaces with tools like Mathematica, Parma, PUBS, Matlab, etc.
Overview of the Analysis ("classical" approach)

1. Supporting analyses:
   - Class hierarchy analysis simplifies CFG and improves overall precision. Types/shapes for size metrics.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.
   - Determinacy (mutual exclusion) to obtain tighter bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine relative sizes of variable contents.
   - Size measures are derived from inferred shape (type) information.

3. Compute upper bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - We have an internal recurrence solver, and the system also interfaces with tools like Mathematica, Parma, PUBS, Matlab, etc.

4. E.g.: true conc(x,y) : list * list
   
   => ( size_lb(ret, length(x)+length(y) ), size_ub(ret, length(x)+length(y) ) )
Overview of the Analysis ("classical" approach)

1. Supporting analyses:
   - Class hierarchy analysis simplifies CFG and improves overall precision. Types/shapes for size metrics.
   - Sharing analysis for correctness (conservative: only when there is no sharing among data structures).
   - Non-failure (no exceptions) inferred for non-trivial lower bounds.
   - Determinacy (mutual exclusion) to obtain tighter bounds.

2. Set up recurrence equations representing the size of each output argument as a function of the input data sizes.
   - Data dependency graphs determine relative sizes of variable contents.
   - Size measures are derived from inferred shape (type) information.

3. Compute upper bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - We have an internal recurrence solver, and the system also interfaces with tools like Mathematica, Parma, PUBS, Matlab, etc.

5. Use the size information to set up recurrence equations representing the computational cost of each block and compute upper bounds to their solutions to obtain **resource usage functions**.
## Some results

<table>
<thead>
<tr>
<th>Program</th>
<th>Resource</th>
<th>Usage Function</th>
<th>Metrics</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>client</td>
<td>“bits received”</td>
<td>( \lambda x \cdot 8 \cdot x )</td>
<td>length</td>
<td>186</td>
</tr>
<tr>
<td>color_map</td>
<td>“unifications”</td>
<td>39066</td>
<td>size</td>
<td>176</td>
</tr>
<tr>
<td>copy_files</td>
<td>“files left open”</td>
<td>( \lambda x \cdot x )</td>
<td>length</td>
<td>180</td>
</tr>
<tr>
<td>eight_queen</td>
<td>“queens movements”</td>
<td>19173961</td>
<td>length</td>
<td>304</td>
</tr>
<tr>
<td>eval_polynom</td>
<td>“FPU usage”</td>
<td>( \lambda x \cdot 2.5 \cdot x )</td>
<td>length</td>
<td>44</td>
</tr>
<tr>
<td>fib</td>
<td>“arith. operations”</td>
<td>( \lambda x \cdot 2.17 \cdot 1.61^x + 0.82 \cdot (-0.61)^x - 3 )</td>
<td>value</td>
<td>116</td>
</tr>
<tr>
<td>grammar</td>
<td>“phrases”</td>
<td>24</td>
<td>length/size</td>
<td>227</td>
</tr>
<tr>
<td>hanoi</td>
<td>“disk movements”</td>
<td>( \lambda x \cdot 2^x - 1 )</td>
<td>value</td>
<td>100</td>
</tr>
<tr>
<td>insert_stores</td>
<td>“accesses Stores”</td>
<td>( \lambda n, m \cdot n + k )</td>
<td>length</td>
<td>292</td>
</tr>
<tr>
<td>perm</td>
<td>“bytecode instructions”</td>
<td>( \lambda x \cdot \left( \sum_{i=1}^{x} 18 \cdot \frac{x!}{i!} \right) + \left( \sum_{i=1}^{x} 14 \cdot \frac{x!}{i!} \right) + 4 \cdot x! )</td>
<td>length</td>
<td>98</td>
</tr>
<tr>
<td>power_set</td>
<td>“output elements”</td>
<td>( \lambda x \cdot \frac{1}{2} \cdot 2^{x+1} )</td>
<td>length</td>
<td>119</td>
</tr>
<tr>
<td>qsort</td>
<td>“lists parallelized”</td>
<td>( \lambda x \cdot 4 \cdot 2^x - 2x - 4 )</td>
<td>length</td>
<td>144</td>
</tr>
<tr>
<td>send_files</td>
<td>“bytes read”</td>
<td>( \lambda x, y \cdot x \cdot y )</td>
<td>length/size</td>
<td>179</td>
</tr>
<tr>
<td>subst_exp</td>
<td>“replacements”</td>
<td>( \lambda x, y \cdot 2xy + 2y )</td>
<td>size/length</td>
<td>153</td>
</tr>
<tr>
<td>zebra</td>
<td>“steps”</td>
<td>30232844295713061</td>
<td>size</td>
<td>292</td>
</tr>
</tbody>
</table>
### Some results (Java)

<table>
<thead>
<tr>
<th>Program</th>
<th>Resource(s)</th>
<th>t</th>
<th>Resource Usage Func. / Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>BST</td>
<td>Heap usage</td>
<td>367</td>
<td>( O(2^n) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( n \equiv ) tree depth</td>
</tr>
<tr>
<td>CellPhone</td>
<td>SMS monetary cost</td>
<td>386</td>
<td>( O(n^2) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( n \equiv ) packets length</td>
</tr>
<tr>
<td>Client</td>
<td>Bytes received and Bandwidth required</td>
<td>527</td>
<td>( O(n) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( n \equiv ) stream length</td>
</tr>
<tr>
<td></td>
<td>Bandwidth required</td>
<td></td>
<td>( O(1) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>—</td>
</tr>
<tr>
<td>Dhrystone</td>
<td>Energy consumption</td>
<td>759</td>
<td>( O(n) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( n \equiv ) int value</td>
</tr>
<tr>
<td>Divbytwo</td>
<td>Stack usage</td>
<td>219</td>
<td>( O(\log_2(n)) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( n \equiv ) int value</td>
</tr>
<tr>
<td>Files</td>
<td>Files left open and Data stored</td>
<td>649</td>
<td>( O(n) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( n \equiv ) number of files</td>
</tr>
<tr>
<td></td>
<td>Data stored</td>
<td></td>
<td>( O(n \times m) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( m \equiv ) stream length</td>
</tr>
<tr>
<td>Join</td>
<td>DB accesses</td>
<td>460</td>
<td>( O(n \times m) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( n, m \equiv ) table records</td>
</tr>
<tr>
<td>Screen</td>
<td>Screen width</td>
<td>536</td>
<td>( O(n) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( n \equiv ) stream length</td>
</tr>
</tbody>
</table>

- Different complexity functions, resources, types of loops/recursion, etc.
## Observed and Estimated Execution Time (Intel)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>110</td>
<td>110</td>
<td>113</td>
<td>-2.4</td>
<td>-2.4</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>69</td>
<td>69</td>
<td>71</td>
<td>-2.3</td>
<td>-2.3</td>
</tr>
<tr>
<td>3</td>
<td>E</td>
<td>1525</td>
<td>1525</td>
<td>1576</td>
<td>-3.3</td>
<td>-3.3</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>1501</td>
<td>1501</td>
<td>1589</td>
<td>-5.7</td>
<td>-5.7</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>2569</td>
<td>2569</td>
<td>2638</td>
<td>-2.7</td>
<td>-2.7</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>1875</td>
<td>1875</td>
<td>2027</td>
<td>-7.8</td>
<td>-7.8</td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td>1868</td>
<td>1868</td>
<td>1931</td>
<td>-3.3</td>
<td>-3.3</td>
</tr>
<tr>
<td>8</td>
<td>L</td>
<td>43</td>
<td>68</td>
<td>81</td>
<td>-67.2</td>
<td>-17.8</td>
</tr>
<tr>
<td>U</td>
<td>3414</td>
<td>3569</td>
<td>3640</td>
<td></td>
<td>-6.4</td>
<td>-2.0</td>
</tr>
<tr>
<td>9</td>
<td>L</td>
<td>54</td>
<td>79</td>
<td>91</td>
<td>-54.6</td>
<td>-14.8</td>
</tr>
<tr>
<td>U</td>
<td>3414</td>
<td>3694</td>
<td>4011</td>
<td></td>
<td>-16.2</td>
<td>-8.2</td>
</tr>
<tr>
<td>10</td>
<td>L</td>
<td>135</td>
<td>142</td>
<td>124</td>
<td>8.6</td>
<td>13.7</td>
</tr>
<tr>
<td>U</td>
<td>7922</td>
<td>2937</td>
<td>2858</td>
<td></td>
<td>120.6</td>
<td>2.7</td>
</tr>
<tr>
<td>11</td>
<td>L</td>
<td>216</td>
<td>138</td>
<td>111</td>
<td>72.3</td>
<td>22.5</td>
</tr>
<tr>
<td>U</td>
<td>226</td>
<td>216</td>
<td>162</td>
<td></td>
<td>34.0</td>
<td>29.5</td>
</tr>
</tbody>
</table>
Energy Consumption Analysis

Approach: [NMLH08]

- Specialize generic 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: [NMLH08]

- 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>.957860</td>
<td>2.273580</td>
<td>3.23144</td>
</tr>
<tr>
<td>isub</td>
<td>.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:** [NMLH08]

- Specialize generic 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.

Much more ambitious (within ENTRA Project):

- Analysis of (embedded) programs written in XC, on XMOS processors.
- Use more sophisticated *ISA-level energy models* for XMOS XS1, developed by Bristol & XMOS (based on “Tiwari” model).
Example: An XC source, its ISA (left) and its IR (right)

```java
int fact(int N) {
    if (N <= 0) return 1;
    return N * fact(N - 1);
}
```

<table>
<thead>
<tr>
<th>1</th>
<th>fact(y):</th>
<th>1</th>
<th>fact(R0,R0_3):-</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0x01: entsp 0x2</td>
<td>2</td>
<td>entsp(0x2),</td>
</tr>
<tr>
<td>3</td>
<td>0x02: stw r0, sp[0x1]</td>
<td>3</td>
<td>stw(R0,Sp0x1),</td>
</tr>
<tr>
<td>4</td>
<td>0x03: ldw r1, sp[0x1]</td>
<td>4</td>
<td>ldw(R1,Sp0x1),</td>
</tr>
<tr>
<td>5</td>
<td>0x04: ldc r0, 0x0</td>
<td>5</td>
<td>ldc(R0_1,b0x0),</td>
</tr>
<tr>
<td>6</td>
<td>0x05: lss r0, r0, r1</td>
<td>6</td>
<td>lss(R0_2,bR0_1,R1),</td>
</tr>
<tr>
<td>7</td>
<td>0x06: bf r0, &lt;0x08&gt;</td>
<td>7a</td>
<td>bf(R0_2,0x8),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7b</td>
<td>fact_aux(R0_2,Sp0x1,R0_3,R1_1).</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>9</td>
<td>fact_aux(1,Sp0x1,R0_4,R1):-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bu(0x10),</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>10</td>
<td>ldw(R0_1,Sp0x1),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>sub(R0_2,R0_1,0x1),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
<td>bl(fact),</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>13a</td>
<td>fact(R0_2,R0_3),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13b</td>
<td>fact(R0_2,R0_3),</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>lsw(R1,Sp0x1),</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>14</td>
<td>mul(R0_4,R1,R0_3),</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
<td>retp(0x2).</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>16</td>
<td>fact_aux(0,Sp0x1,R0,R1):-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
<td>mkmsk(R0,R0_1),</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>18</td>
<td>retsp(0x2).</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>retsp(0x2).</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The energy model is obtained via profiling (once and for all).

- Gets energy data for each ISA instruction by using a test harness:
  - Slave processor executing test kernels and
  - A master processor collecting power samples.
Low-level ISA characterization

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

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

```ciao
:- 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_l2rus2(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 Code

fact:
    entsp 6
    stw r0, sp[4]
    stw r0, sp[2]
.Lxtalabel0:
    ldw r0, sp[4]
    ldc r1, 0
    lss 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
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
- Multivariant Success: off
- Print Program Point Info: off
- Collapse AI Info: on

{Current Saved Menu Configurations: □}

Cancel  Apply
Analysis Results

```
:- module(_, [fact/2], [ciaop(xcore(model(instructions))), ciaop(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),
  l1\=0,
  _3569 is _3467,
  ldw_ru62(_3578),
  _3586 is _3569-1,
  sub_2rus2(_3598),
  _3606 is _3569,
  stw_ru62(_3615),
  _3623 is _3586+0,
```

Hermenegildo et al. ()
Resource Analysis/Debugging/Verification
EACO WS, Sep 10-11, 2014
Analysis Results

:- module(_, [fact/2], [ciao(app(xcore(model(instructions))), ciao(app(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),
   l1\0,

Other results

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
<th>Energy function</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact(N)</td>
<td>Calculates N!</td>
<td>26.0  N + 19.4</td>
</tr>
<tr>
<td>fibonacci(N)</td>
<td>Nth Fibonacci no.</td>
<td>30.1 + 35.6 (\phi^N) + 11.0 ((1 - \phi)^N)</td>
</tr>
<tr>
<td>sqr(N)</td>
<td>Computes (N^2)</td>
<td>103.0 (N^2) + 205.8 (N + 188.32)</td>
</tr>
<tr>
<td>isprime(N)</td>
<td>Checks if N is prime</td>
<td>58.6 (N - 35.5)</td>
</tr>
<tr>
<td>power(base, exp)</td>
<td>Calculates (base^{exp})</td>
<td>6.3 (\log_2 exp + 1) + 6.5</td>
</tr>
</tbody>
</table>
Measuring Power Consumption on the Hardware

- XMOS Ltd. provides the XTAG3 measurement circuit.
- XTAG3 plugs into the XMOS XS1 board.

We compare to:
- ISS (Instruction Set Simulation) and
- SRA (Static Resource Analysis).
Some Results

Some Results

- SRA provides results beyond what is possible with simulation (as test run-time increases, ISS becomes impractically long).
- SRA shows promising accuracy in comparison with ISS and the HW (at least for the simple cases studied so far).
- Simulation time limits the usefulness of ISS method, whereas equation solving limits SRA.

Hermenegildo et al. ()
IR Level Trade-offs

![Diagram of IR Level Trade-offs]

- Energy Model + Program (including frontend assertions)
- Source code
- Transformation
- Internal representation (including IAL assertions)
- Analysis

- Energy Model
- Source code
- Transformation
- Optimized LLVM IR
- Analysis

- Energy Model
- LLVM IR
- Transformation
- Optimized LLVM IR
- Analysis

- Energy Model
- ISA
- Transformation
- Optimized LLVM IR
- Analysis

- Hardware
- Energy consumption estimations

**Hermenegildo et al.**
### Accuracy of Energy Analyses: LLVM vs. ISA layer

<table>
<thead>
<tr>
<th>Program</th>
<th>Error vs. HW</th>
<th>ISA/LLVM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>llvm</td>
<td>isa</td>
</tr>
<tr>
<td>fact</td>
<td>4.5%</td>
<td>2.86%</td>
</tr>
<tr>
<td>fibonacci</td>
<td>11.94%</td>
<td>5.41%</td>
</tr>
<tr>
<td>sqr</td>
<td>9.31%</td>
<td>1.49%</td>
</tr>
<tr>
<td>power_of_two</td>
<td>11.15%</td>
<td>4.26%</td>
</tr>
<tr>
<td>reverse</td>
<td>2.18%</td>
<td>N/A</td>
</tr>
<tr>
<td>concat</td>
<td>8.71%</td>
<td>N/A</td>
</tr>
<tr>
<td>mat_mult</td>
<td>1.47%</td>
<td>N/A</td>
</tr>
<tr>
<td>sum_facts</td>
<td>2.42%</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>6.46%</strong></td>
<td><strong>3.50%</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 layer but can be analyzed at the LLVM layer.
Energy consumption verification / debugging

\textbf{check fact}(x) : \ int(x) + \ resource(energy, 0, 100)

1 Resource analysis infers upper and lower bounds for resource “energy.” The analysis results produced are:

\textbf{true fact}(x) : \ int(x)
\begin{align*}
\Rightarrow & \ (\text{int}(x), \text{int}(\text{ret}), \text{rsize}(x, \text{num}(lx,ux))), \\
& \ \text{rsize}(\text{ret}, \text{num}(\text{Factorial}(lx),\text{Factorial}(ux)))) \\
& + \ resource(energy, 21 \times lx + 16, 21 \times ux + 16)
\end{align*}

2 Then, the analysis results are compared with the “check” assertion (the specification) and the following assertions are produced:

\textbf{checked fact}(x) : \ (\text{int}(x), \text{intervals(\text{int}(x), [i(0,4)]))}
\begin{align*}
& + \ resource(energy, 0, 100)
\end{align*}

\textbf{false fact}(x) : \ (\text{int}(x), \text{intervals(\text{int}(x), [i(5,\infty)]))}
\begin{align*}
& + \ resource(energy, 0, 100)
\end{align*}
Resource Usage Verification – Function Comparisons

RESOURCE USAGE

- SPECIFICATION UPPER/LOWER BOUNDS (SU/SL)
- SPECIFICATION INTERVALS

INPUT DATA SIZE

SU
SL
Resource Usage Verification – Function Comparisons

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

**INPUT DATA SIZE**
In the classical CiaoPP resource analysis the last steps (setting up and solving recurrences) were not implemented as an abstract domain.

We have now defined, implemented and integrated the resource analysis as an abstract domain (a plugin of the generic fixpoint).

We get all the good features of the AI framework for free:

- Multivariance: e.g., separate different call patterns for same block: sort(lst(int),var) ... sort(lst(flt),var) ... sort(var,lst(int))
- Easier combination with other domains.
- Easier integration w/static debugging/verification and rt-checking.
- Many other engineering advantages.

- New domain for size analysis (sized types) that infers bounds on the size of data structures and substructures.
  - Size: number of rule applications in type/shape definition.
- Used in the XC energy analysis.
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>append</td>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>$\beta$</td>
</tr>
<tr>
<td>appAll</td>
<td>$a_1a_2a_3$</td>
<td>$a_1$</td>
<td>$b_1b_2b_3$</td>
</tr>
<tr>
<td>coupled</td>
<td>$\mu$</td>
<td>0</td>
<td>$\nu$</td>
</tr>
<tr>
<td>dyade</td>
<td>$\alpha_1\alpha_2$</td>
<td>$\alpha_1\alpha_2$</td>
<td>$\beta_1\beta_2$</td>
</tr>
<tr>
<td>erathos</td>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>$\beta^2$</td>
</tr>
<tr>
<td>fib</td>
<td>$\phi^\mu$</td>
<td>$\phi^\mu$</td>
<td>$\phi^\nu$</td>
</tr>
<tr>
<td>hanoi</td>
<td>1</td>
<td>0</td>
<td>$2^\nu$</td>
</tr>
<tr>
<td>isort</td>
<td>$\alpha^2$</td>
<td>$\alpha^2$</td>
<td>$\beta^2$</td>
</tr>
<tr>
<td>isortl</td>
<td>$a_1^2$</td>
<td>$a_1^2$</td>
<td>$b_1^2b_2$</td>
</tr>
<tr>
<td>lisfact</td>
<td>$\alpha\gamma$</td>
<td>$\alpha$</td>
<td>$\beta\delta$</td>
</tr>
<tr>
<td>listnum</td>
<td>$\mu$</td>
<td>$\mu$</td>
<td>$\nu$</td>
</tr>
<tr>
<td>minsert</td>
<td>$\alpha^2$</td>
<td>$\alpha$</td>
<td>$\beta^2$</td>
</tr>
<tr>
<td>nub</td>
<td>$a_1$</td>
<td>$a_1$</td>
<td>$b_1^2b_2$</td>
</tr>
<tr>
<td>part</td>
<td>$\alpha$</td>
<td>$\alpha$</td>
<td>$\beta$</td>
</tr>
<tr>
<td>zip3</td>
<td>$\min(\alpha_i)$</td>
<td>0</td>
<td>$\min(\beta_i)$</td>
</tr>
</tbody>
</table>
Some Future Work

- Analysis of multi-threaded applications.
  - We can handle at least 'fork-join' and 'coroutining' in most other analyses, but not (yet) for resources.

- Average cost analysis (based on some earlier work).

- Increase (always?) the classes of programs for which accurate results can be obtained.

- Try richer energy models.

- Improve analysis accuracy at the ISA layer:
  - Propagating high-level program information such as types, loop bounds, and communication into the lower-level representations.

- Explore more analysis/modeling layer choices, e.g.:
  - Try stand-alone models at the LLVM IR layer.
  - Analysis of source code.
Thank you for your attention!
Integrated Static/Dynamic Debugging and Verification

**Program** $P$

- check
- trust
- test $I_\alpha$

**Builtins/ Libs**

**PREPROCESSOR**

**Static Analysis**

- Assertion Normalizer & Lib Itf.

**Analysis Info** $[[P]]_\alpha$

**Comparator** (Incl. VCgen)

**Analysis Info** $[[P]]_\alpha$

**RT Check**

- possible run-time error
  - verification warning
  - compile-time error

**verified**

- certificate (ACC) + (optimized) code

**Definition**

<table>
<thead>
<tr>
<th>Sufficient condition</th>
<th>Sufficient condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is prt. correct w.r.t. $I_\alpha$ if $\alpha([[P]]) \leq I_\alpha$</td>
<td>$[[P]]<em>{\alpha^+} \leq I</em>\alpha$</td>
</tr>
<tr>
<td>$P$ is complete w.r.t. $I_\alpha$ if $I_\alpha \leq \alpha([[P]])$</td>
<td>$I_\alpha \leq [[P]]_{\alpha^=}$</td>
</tr>
<tr>
<td>$P$ is incorrect w.r.t. $I_\alpha$ if $\alpha([[P]]) \not\leq I_\alpha$</td>
<td>$[[P]]<em>{\alpha^=} \not\leq I</em>\alpha$, or</td>
</tr>
<tr>
<td>$P$ is incomplete w.r.t. $I_\alpha$ if $I_\alpha \not\leq \alpha([[P]])$</td>
<td>$[[P]]<em>{\alpha^+} \cap I</em>\alpha = \emptyset \land [[P]]_{\alpha^=} \neq \emptyset$</td>
</tr>
</tbody>
</table>

**References**

- BDD+97, HPB99, PBH00c, PBH00a, HPBLG03, HALGP05, PCPH06, PCPH08, MLGH09, SMH14
Based throughout on the notion of safe approximation (abstraction).

Run-time checks generated for parts of asserts, not verified statically.

Diagnosis (for both static and dynamic errors).

Comparison not always trivial: e.g., resource debugging/certification

- Need to compare functions.
- “Segmented” answers.

[BDD⁺97, HPB99, PBH00c, PBH00a, HPBLG03, HALGP05, PCPH06, PCPH08, MLGH09, SMH14]
References – Analysis and Verification of Resources

Task Granularity Analysis in Logic Programs.

Towards Granularity Based Control of Parallelism in Logic Programs.

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

Estimating the Computational Cost of Logic Programs.

Lower Bound Cost Estimation for Logic Programs.

User-Definable Resource Bounds Analysis for Logic Programs.

Towards Execution Time Estimation in Abstract Machine-Based Languages.
**[NMLH08]** J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications.
Extended Abstract.

**[NMLH09]** J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
User-Definable Resource Usage Bounds Analysis for Java Bytecode.

**[LGDB10]** P. López-García, L. Darmawan, and F. Bueno.

Sized Type Analysis Logic Programs (Technical Communication).

Energy Consumption Analysis of Programs based on XMOS ISA-Level Models.
In *Pre-proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR’13)*, September 2013.

**[SLH14]** A. Serrano, P. López-Garcia, M. Hermenegildo.
Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types.
References – Overall Model


References – Assertion Language

An Assertion Language for Debugging of Constraint Logic Programs.

An Assertion Language for Constraint Logic Programs.

[MLGH09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.

Assertion-based Debugging of Higher-Order (C)LP Programs.

References – Intermediate Representation

A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.
References – Abstraction Carrying Code


References – Fixpoint-based Analyzers (Abstract Interpreters)

Compile-time Derivation of Variable Dependency Using Abstract Interpretation.

Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming.

Optimized Algorithms for the Incremental Analysis of Logic Programs.

Incremental Analysis of Constraint Logic Programs.

An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.
References – Modular Analysis, Analysis of Concurrency


References – Domains: Sharing/Aliasing

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

Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation.

Efficient top-down set-sharing analysis using cliques.

Precise Set Sharing Analysis for Java-style Programs.
In *9th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI’08)*, number 4905 in LNCS, pages 172–187. Springer-Verlag, January 2008.

Sharing Analysis of Arrays, Collections, and Recursive Structures.

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.
References – Domains: Shape/Type Analysis

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 Logically Related Heap Regions.

More Precise yet Efficient Type Inference for Logic Programs.

Heap Analysis in the Presence of Collection Libraries.

Efficient context-sensitive shape analysis with graph-based heap models.
References – Domains: Non-failure, Determinacy

Non-Failure Analysis for Logic Programs.

Multivariant Non-Failure Analysis via Standard Abstract Interpretation.

Determinacy Analysis for Logic Programs Using Mode and Type Information.

Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.
References – Automatic Parallelization, (Abstract) Partial Evaluation, Other Optimizations

A Technique for Recursive Invariance Detection and Selective Program Specialization.

Abstract Specialization and its Application to Program Parallelization.

Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism.

Abstract Multiple Specialization and its Application to Program Parallelization.

An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework.
Abstract Specialization and its Applications. 
Invited talk.

Abstract Interpretation with Specialized Definitions. 

A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. 
In M. García de la Banda and E. Pontelli, editors, 24th International Conference on Logic Programming (ICLP’08), volume 5366 of LNCS, pages 651–666. Springer-Verlag, December 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.

Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. 
In PADL’04, number 3057 in LNCS, pages 86–103. Springer-Verlag, June 2004.

High-Level Languages for Small Devices: A Case Study. 