Energy Consumption Analysis and Verification by Translation to Horn Clauses and Abstract Interpretation

Manuel Hermenegildo [1,2] P. López-García[1,3] M. Klemen[1] U. Liqat[1]

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

VPT Workshop @ ETAPS, Uppsala, April 29, 2017

Analysis/Debugging/Verification of Resources

Statically and automatically infer upper/lower bounds on the usage that a program makes of a general notion of (*user-definable*) resources.

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

- Key observations about resource consumption:
  - Undecidable → infer safe **bounds** (as accurately as possible) → **AI**
  - Dependent on input data metrics → infer the bounds **as functions** of input data sizes (list length, array dimensions, numerical values, ...). (Difference with WCET and related methods.)

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

[DLH90, DLGHL94, DLGHL97, NMLGH07, MLGCH08, NMLH08, NMLH09, LGDB10, SLBH13, LKSGL13, SLH14]

Analysis/Debugging/Verification of Resources

Statically and automatically infer upper/lower bounds on the usage that a program makes of a general notion of (*user-definable*) resources.

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

- Key observations about resource consumption:
  - Undecidable → infer safe **bounds** (as accurately as possible) → **AI**
  - Dependent on input data metrics → infer the bounds **as functions** of input data sizes (list length, array dimensions, numerical values, ...). (Difference with WCET and related methods.)

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

[DLH90, DLGHL94, DLGHL97, NMLGH07, MLGCH08, NMLH08, NMLH09, LGDB10, SLBH13, LKSGL13, SLH14]

Analysis/Debugging/Verification of Resources

Statically and automatically infer upper/lower bounds on the usage that a program makes of a general notion of (*user-definable*) resources.

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

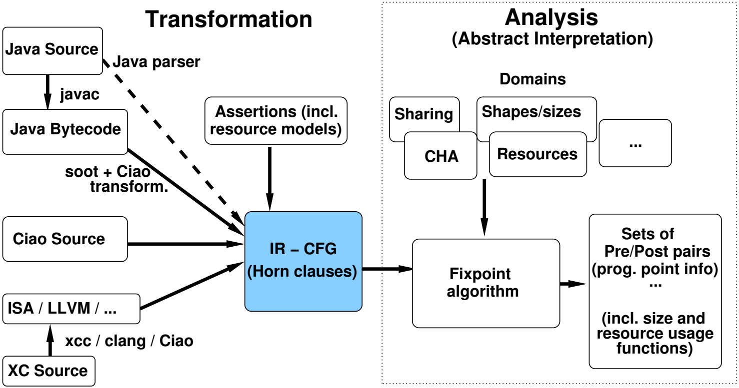
- Key observations about resource consumption:
  - Undecidable → infer safe **bounds** (as accurately as possible) → **AI**
  - Dependent on input data metrics → infer the bounds **as functions** of input data sizes (list length, array dimensions, numerical values, ...). (Difference with WCET and related methods.)

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

[DLH90, DLGHL94, DLGHL97, NMLGH07, MLGCH08, NMLH08, NMLH09, LGDB10, SLBH13, LKSGL13, SLH14]

CiaoPP Intermediate Repr.: (Constraint) Horn Clauses

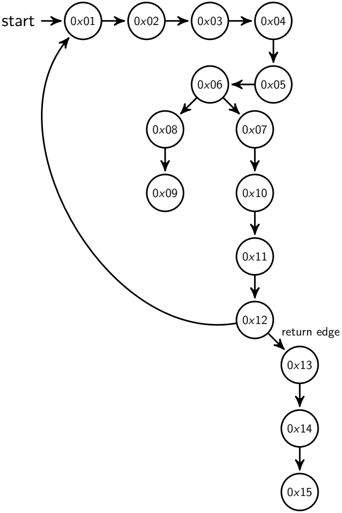
[MLNH07]



- Transformation:
  - Source: Program P in L_P + (possibly abstract) Semantics of L_P
  - Target: A (C) Horn Clause program capturing $[\![P]\!]$ (or, possibly, $[\![P]\!]^{\alpha}$)
- Block-based CFG. Each block represented as a *Horn clause*.
- Used for all analyses: aliasing, CHA/shape/types, data sizes, resources, etc.
- Allows supporting multiple languages.

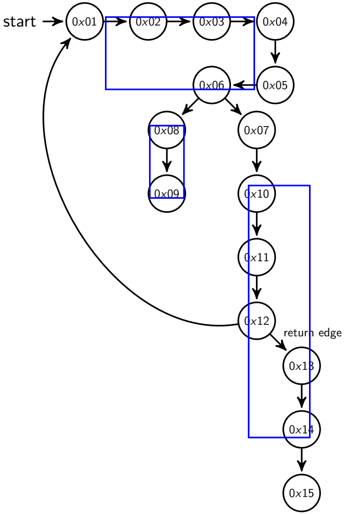
Xcore Example: Control Flow Graph (CFG)

```
<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
```



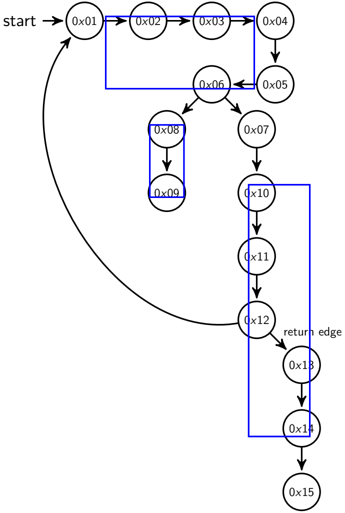
Xcore Example: Block Representation

```
<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: Constrained Horn Clauses IR

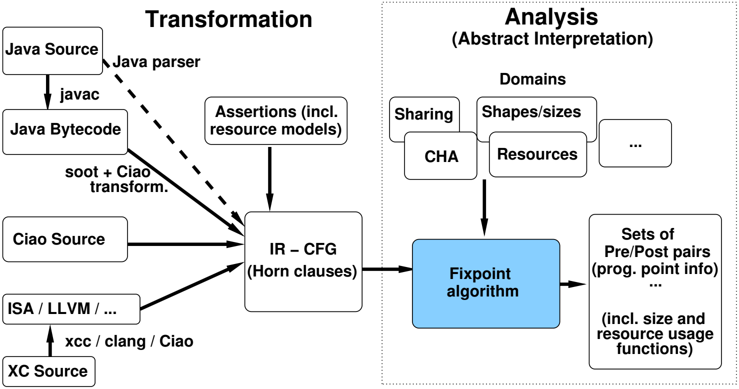
```
:- entry fact/2.
fact(R0,R0_3):-
  entsp(_),
  stw(R0,Sp0x1),
  ldw(R1,Sp0x1),
  ldc(R0_1,0x0),
  lss(R0_2,R0_1,R1),
  bf(R0_2,_),
  bf01(R0_2,Sp0x1,R0_3,R1_1).

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

- Typical 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.
- Some specifics for Java:
  - Control flow graph is constructed from bytecode.
  - Elimination of stack variables.
  - Conversion to three-address statements.
  - Explicit representation of this and ret as extra block parameters.
- Some specifics for XC:
  - Control flow graph is constructed from ISA or LLVM IR representation.
  - Inferring block parameters.
  - Resolving branching to predicates with multiple clauses.
- Can be done via **partial evaluation of an interpreter** (implementing the semantics of the low-level code) w.r.t. the concrete low-level program or **directly** (cf. Futamura projections).

Analysis: CiaoPP Parametric AI Framework



- Analysis *parametric* w.r.t. abstractions, resources, ... (and languages).
- Efficient fixpoint algorithm for (C)HC IR.

[MH92, MGH94, BGH99, PH96, HPMS00, NMLH07]

[MH89, MH91, DLGH97, VB02, BLGH04, LGBH05, NBH06, MSHK07]

[MLH08, MKSH08, MMLH^{+}08, MHKS08, MKH09, LGBH10, MLLH08]

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing HC-based analyses: given P (as a set of HCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC's.

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.
- Characteristics:
  - **Precision:** context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Modular and *incremental*.
  - Handles library calls, externals, ...

[MH89, MH92, MGH94, PH96, HPMS00, NMLH07]

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing HC-based analyses: given P (as a set of HCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC's.

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.
- Characteristics:
  - **Precision:** context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Modular and *incremental*.
  - Handles library calls, externals, ...

[MH89, MH92, MGH94, PH96, HPMS00, NMLH07]

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing HC-based analyses: given P (as a set of HCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC's.

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.
- Characteristics:
  - **Precision:** context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Modular and *incremental*.
  - Handles library calls, externals, ...

[MH89, MH92, MGH94, PH96, HPMS00, NMLH07]

Efficient, Parametric Fixpoint Algorithm

- *Generic framework* for implementing HC-based analyses: given P (as a set of HCs) and abstract domain(s), computes $\mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha$, s.t. $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$.

→ Essentially efficient, incremental, abstract OLDT resolution of HC's.

- It maintains and computes as a result (simplified):
  - **A call-answer table**: with (multiple) entries $\{block : \lambda_{in} \mapsto \lambda_{out}\}$.
    - Exit states for calls to *block* satisfying precond λ_{in} meet postcond λ_{out}.
  - **A dependency arc table**: $\{A : \lambda_{inA} \Rightarrow B : \lambda_{inB}\}$.
    - Answers for call $A : \lambda_{inA}$ depend on the answers for $B : \lambda_{inB}$: (if exit for $B : \lambda_{inB}$ changes, exit for $A : \lambda_{inA}$ possibly also changes).
    - $Dep(B : \lambda_{inB})$ = the set of entries depending on $B : \lambda_{inB}$.
- Characteristics:
  - **Precision:** context-sensitivity / multivariance, prog. point info, ...
  - **Efficiency:** memoization, dependency tracking, SCCs, base cases, ...
  - **Genericity:** abstract domains are plugins, configurable, widening, ...
  - Handles mutually recursive methods.
  - Modular and *incremental*.
  - Handles library calls, externals, ...

[MH89, MH92, MGH94, PH96, HPMS00, NMLH07]

Bottom-up vs. top-down

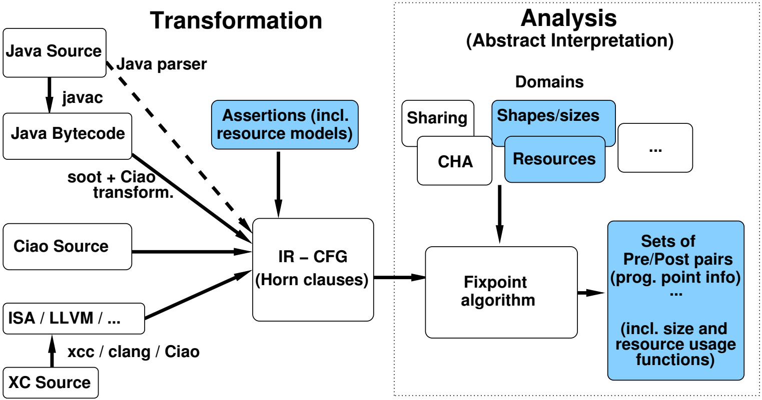
- Only bottom-up information (success):

```
:- true pred sort(X,Y) => list(X), list(Y),
:- true pred sort(X,Y) => sorted(Y), perm(Y,X).
```

- With call information (top-down):

```
:- true pred sort(X,Y) :  list(X), var(Y)
                       => list(Y), sorted(Y), perm(Y,X)
                       +  resource(steps, len(X)**2),
                           no_fail.

:- true pred sort(X,Y) :  ground(X)
                       => indep(Y,X).
```

CiaoPP Resource Analysis



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

[NMLH09, LGDB10, SLBH13, LKSGL13, SLH14] [LGK^{+}16, LBLGH16] [LGKLH16, HLGL^{+}16]

CiaoPP Resource Analysis

- The objective of the resource analysis is to obtain for each predicate/block *call - resource usage function pairs*:
  - Arithmetic functions providing lower/upper bounds on the resource usage of the predicate/block given the sizes of its input data for a particular entry condition.

Example

```
#pragma true nrev(x) :  list(x) ==>
    ( 0 <= resource(energy) && resource(energy) <= 1+length(x)**2) )
```

- **x** points to a list → energy consumed $\leq 1 + length(x)^2$

1. Programmer defines the resource consumption for basic elements (e.g., instructions, bytecodes, libraries, ...) – the "cost model."
2. System infers resource usage bound functions for rest of program. (Can be polynomial, exponential, logarithmic, ...)

CiaoPP Resource Analysis

- The objective of the resource analysis is to obtain for each predicate/block *call - resource usage function pairs*:
  - Arithmetic functions providing lower/upper bounds on the resource usage of the predicate/block given the sizes of its input data for a particular entry condition.

Example

```
#pragma true nrev(x) :  list(x) ==>
     ( 0 <= resource(energy) && resource(energy) <= 1+length(x)**2) )
```

- **x** points to a list → energy consumed $\leq 1 + length(x)^2$

1. Programmer defines the resource consumption for basic elements (e.g., instructions, bytecodes, libraries, ...) – the "cost model."
2. System infers resource usage bound functions for rest of program. (Can be polynomial, exponential, logarithmic, ...)

Some Examples of Resource Functions Inferred

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

- Different complexity functions, resources, size metrics, types of loops/recursion, etc.

Overview of the Analysis ("classical" view)

1. Supporting analyses (examples):
    - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
    - 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 (relevant) output argument as a function of the input data sizes.
    - Size metrics are derived from inferred shape (type) information.
    - Data dependency graphs determine *relative* sizes of variable contents.
3. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
    - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.

Overview of the Analysis ("classical" view)

1. Supporting analyses (examples):
   - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
   - 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 (relevant) output argument as a function of the input data sizes.
   - Size metrics are derived from inferred shape (type) information.
   - Data dependency graphs determine *relative* sizes of variable contents.
3. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.

Overview of the Analysis ("classical" view)

1. Supporting analyses (examples):
    - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
    - 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 (relevant) output argument as a function of the input data sizes.
    - Size metrics are derived from inferred shape (type) information.
    - Data dependency graphs determine *relative* sizes of variable contents.
3. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
    - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.

Overview of the Analysis ("classical" view)

1. Supporting analyses (examples):
   - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
   - 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 (relevant) output argument as a function of the input data sizes.
   - Size metrics are derived from inferred shape (type) information.
   - Data dependency graphs determine *relative* sizes of variable contents.
3. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
   - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.
4. E.g.:

```
#pragma true conc(x,y) : list * list
      ==> ( length(x)+length(y) <= size(ret) &&
            size(ret) <= length(x)+length(y) )
```

Overview of the Analysis ("classical" view)

1. Supporting analyses (examples):
    - Sized types/shapes for size metrics (heap manipulating programs), and to simplify CFG and improve precision (class hierarchy analysis).
    - 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 (relevant) output argument as a function of the input data sizes.
    - Size metrics are derived from inferred shape (type) information.
    - Data dependency graphs determine *relative* sizes of variable contents.
3. Compute bounds to the solutions of these recurrence equations to obtain output argument sizes as functions of input sizes.
    - Using internal recurrence solver, or the interfaces with Mathematica, Parma, PUBS, Matlab, etc.

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

Resource Analysis as an Abstract Interpretation

[SLH14, SLBH13]

- In classical CiaoPP resource analysis the last steps (setting up and solving recurrences) were not implemented as an abstract domain.
- We have recently integrated resource analysis as an *abstract domain "plug-in"* of the generic analysis fixpoint –we get 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*.

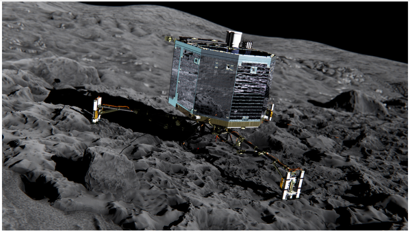
listnum	*sized*(*listnum*)
`listnum -> []` `listnum -> [num | listnum]`	$listnum^{(\alpha,\beta)}\left(num^{(\gamma,\delta)}_{\langle \cdot,1\rangle}\right)$

- Competitive results with state-of-the-art systems (e.g., RAML).
- Used in the XC energy analysis.

Energy Consumption Analysis – Motivation

Energy consumption of computing technologies 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 models – 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:

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Total Cost in in μJ
`iadd`	.957860	2.273580	3.23144
`isub`	.957360	2.273580	3.230.94
...	...	...	...

  - Encouraging results: meaningful functions inferred in many cases.
  - But no comparison with actual device consumption.

Energy Consumption Analysis – Approach

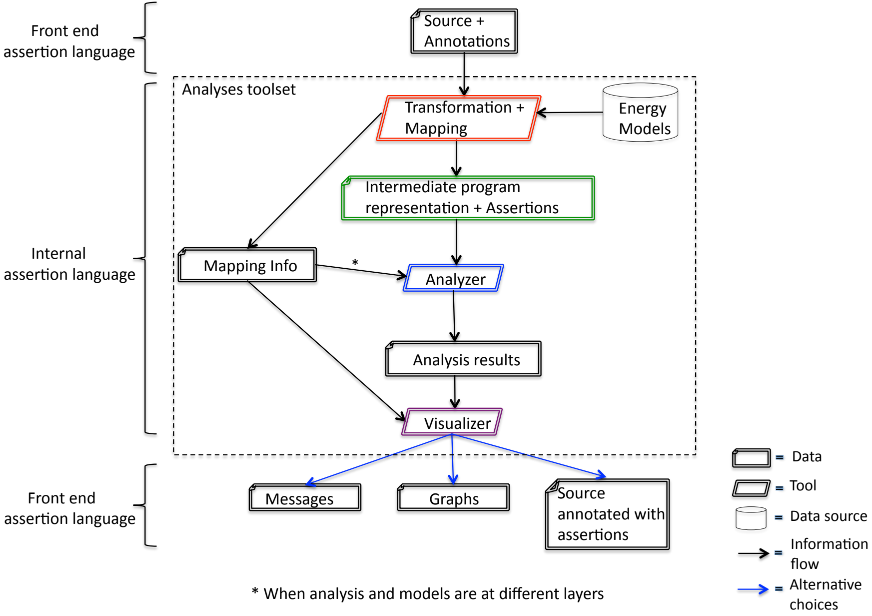
Requires low-level models – approach: [NMLH08]

- 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.

⇒ Addressed recently: [LKSGL13, LGK+16, LBLGH16]

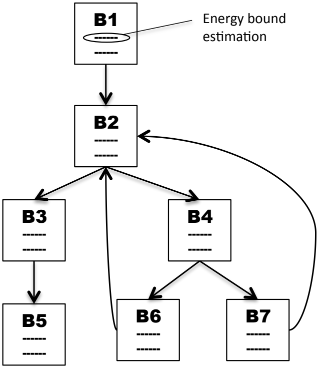
- Analysis of (embedded) programs written in XC, on XMOS processors.
- Using more sophisticated *ISA-level energy models* for XMOS XS1, developed by Bristol & XMOS.
- Comparing to measured energy consumption.

Energy Consumption Analysis – Approach



* When analysis and models are at different layers

Modeling at the Instruction Level

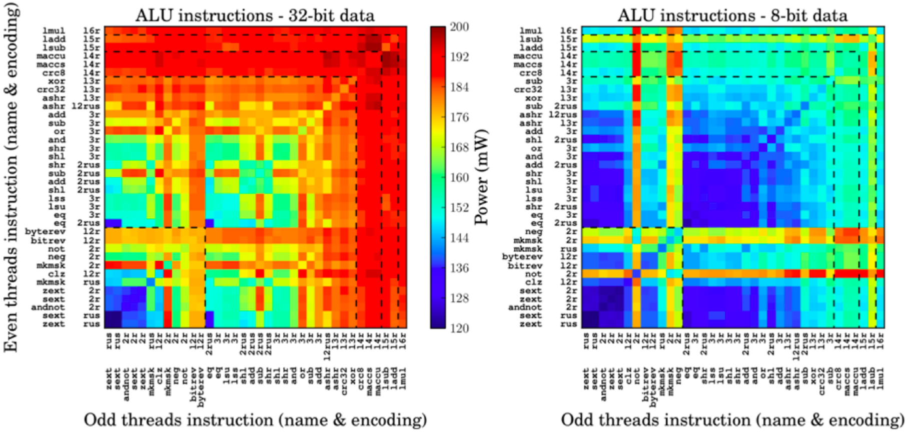


- Each instruction is profiled (using, e.g., an Evolutionary Algorithm – EA) to derive upper- and lower-bound energy estimates.
- These are combined using static analysis.

+ Very compositional.

Low-level ISA characterization – interference

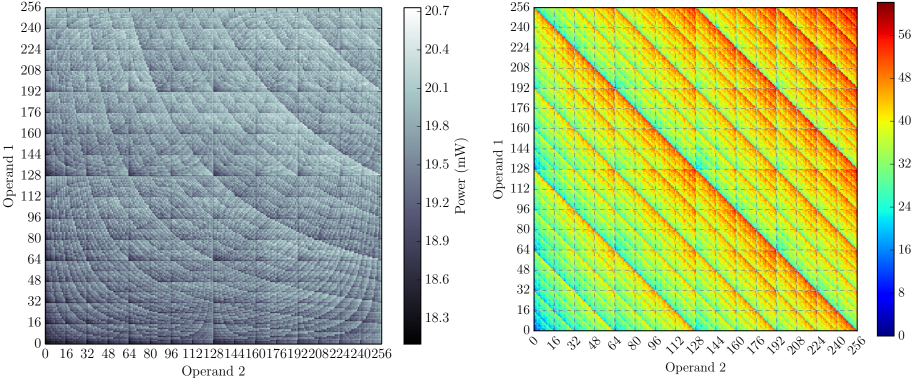
Obtaining the cost model: energy consumption/instruction; interference.



Eder, Kerrison – Bristol U / XMOS.

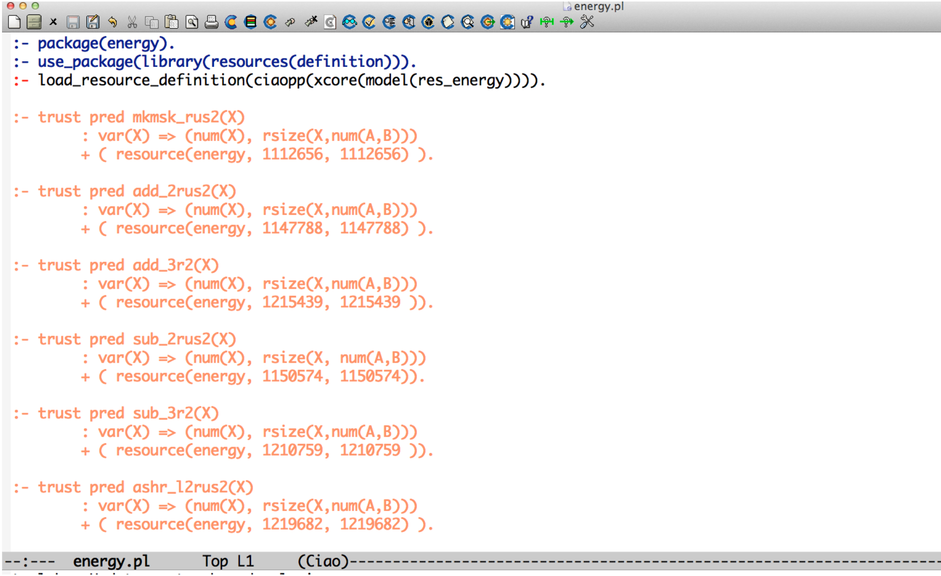
Low-level ISA characterization – operand size

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



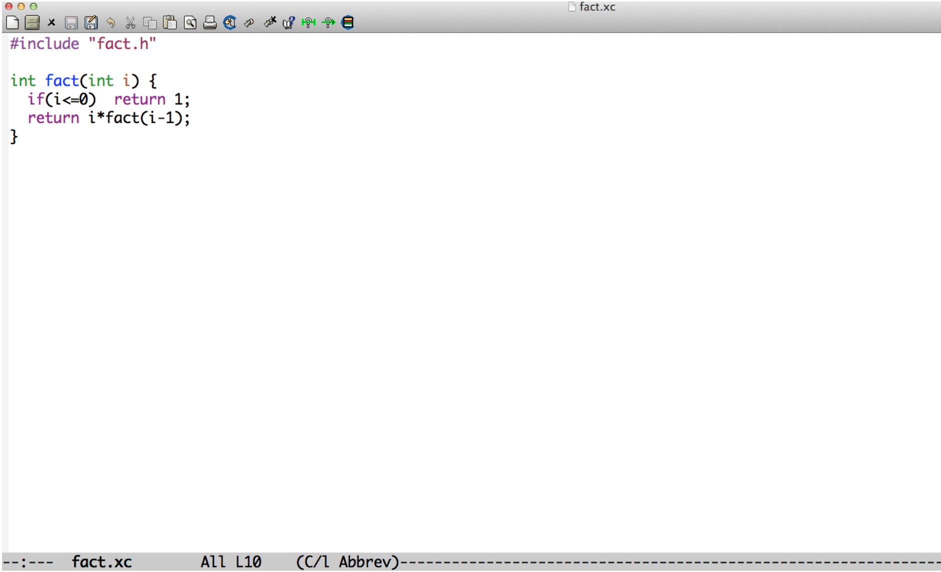
Eder, Kerrison – Bristol U / XMOS.

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_l2rus2(X)
        : var(X) => (num(X), rsize(X,num(A,B)))
        + ( resource(energy, 1219682, 1219682) ).
```

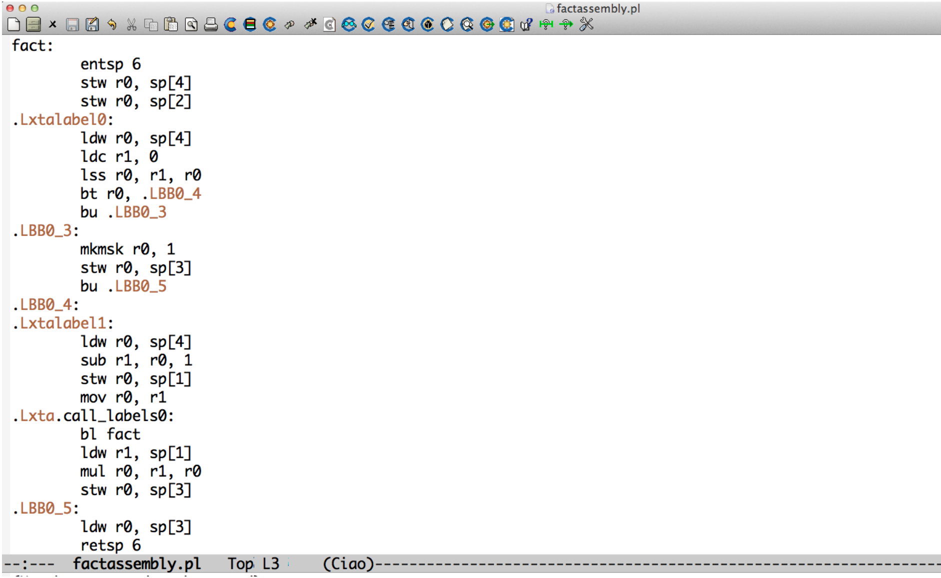
XC Source



```
#include "fact.h"

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

```
--:---  fact.xc        All L10    (C/l Abbrev)------------------------------------------------------
```

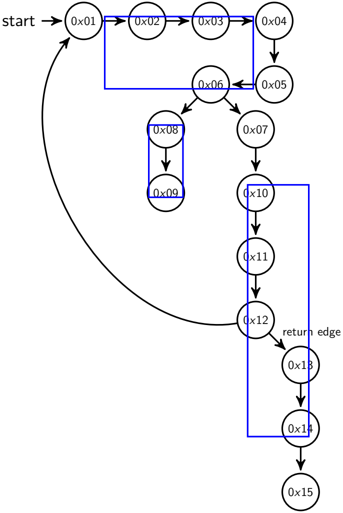
Assembly Code



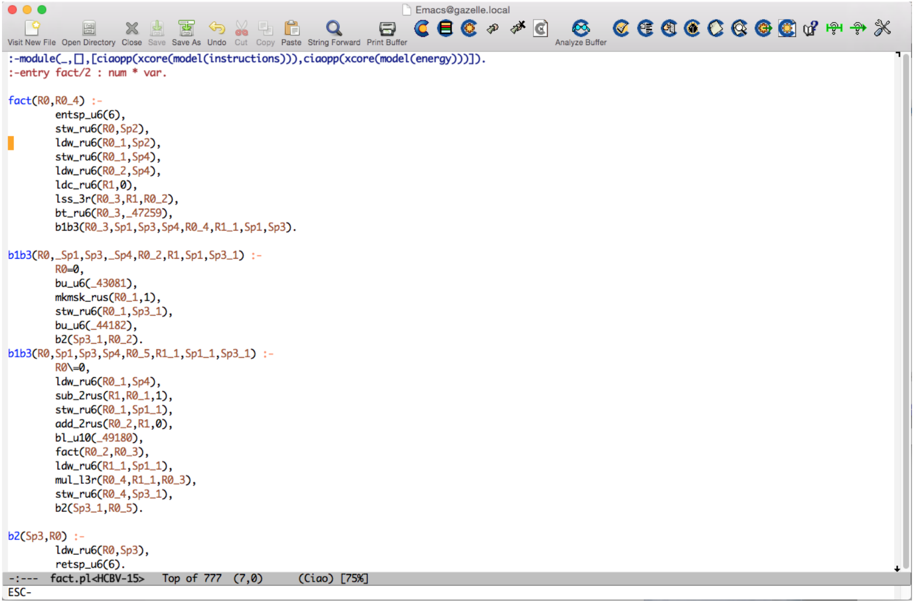
```
fact:
        entsp 6
        stw r0, sp[4]
        stw r0, sp[2]
.Lxtalabel0:
        ldw r0, sp[4]
        ldc r1, 0
        lss r0, r1, r0
        bt r0, .LBB0_4
        bu .LBB0_3
.LBB0_3:
        mkmsk r0, 1
        stw r0, sp[3]
        bu .LBB0_5
.LBB0_4:
.Lxtalabel1:
        ldw r0, sp[4]
        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
--:---  factassembly.pl   Top L3    (Ciao)------------------------------------------------
```

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
```

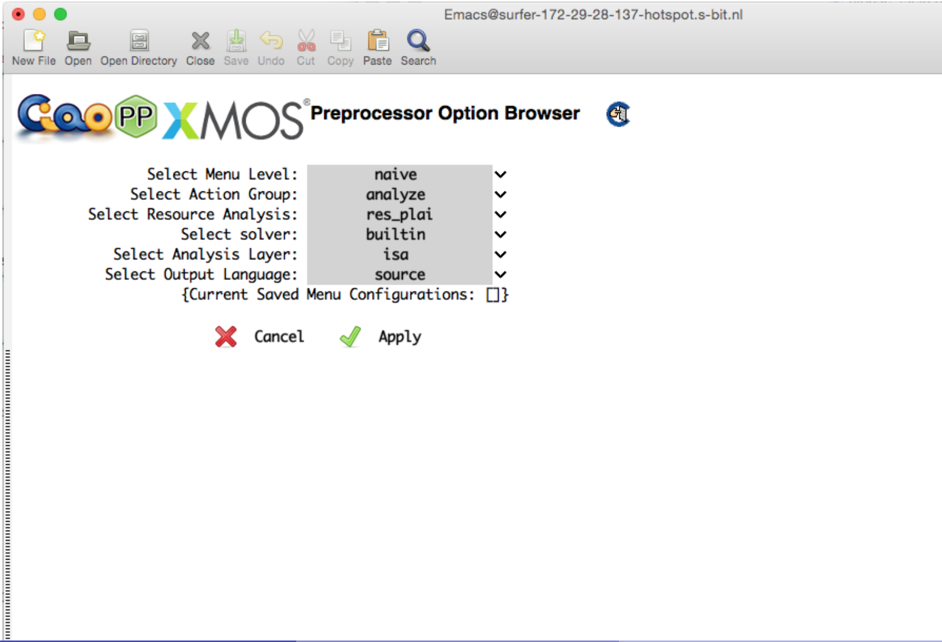


Horn Clause Representation



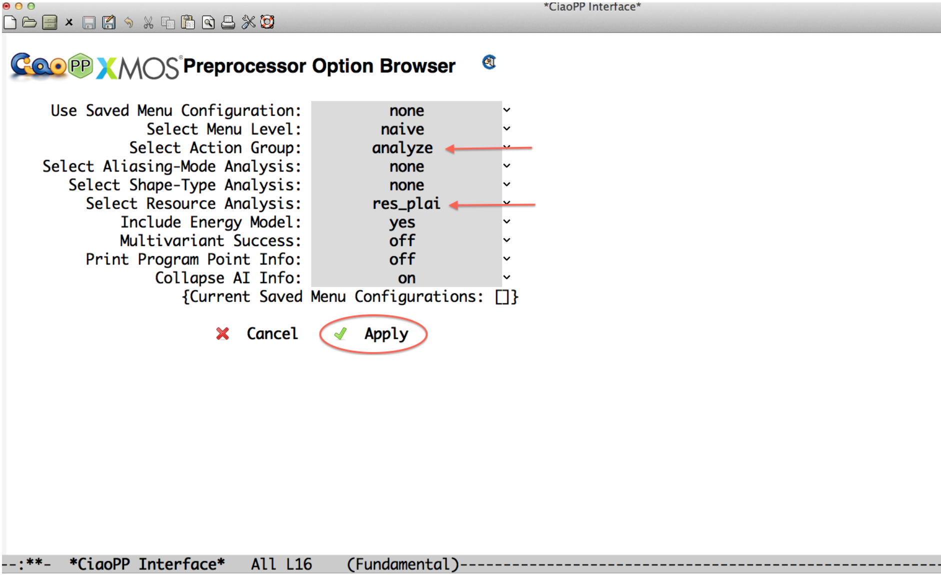
```
:-module(_,[],[ciaopp(xcore(model(instructions))),ciaopp(xcore(model(energy)))]).
:-entry fact/2 : num * var.

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),
        mkmsk_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),
        add_2rus(R0_2,R1,0),
        bl_u10(_49180),
        fact(R0_2,R0_3),
        ldw_ru6(R1_1,Sp1_1),
        mul_l3r(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%]
ESC-
```

CiaoPP Menu



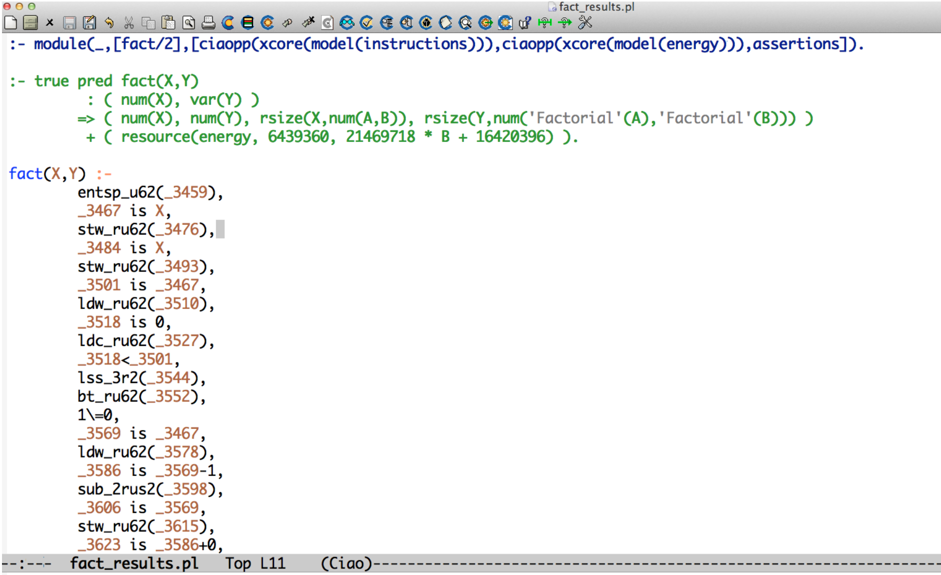
Select Menu Level: naive
Select Action Group: analyze
Select Resource Analysis: res_plai
Select solver: builtin
Select Analysis Layer: isa
Select Output Language: source
{Current Saved Menu Configurations: []}

Select Resource Analysis



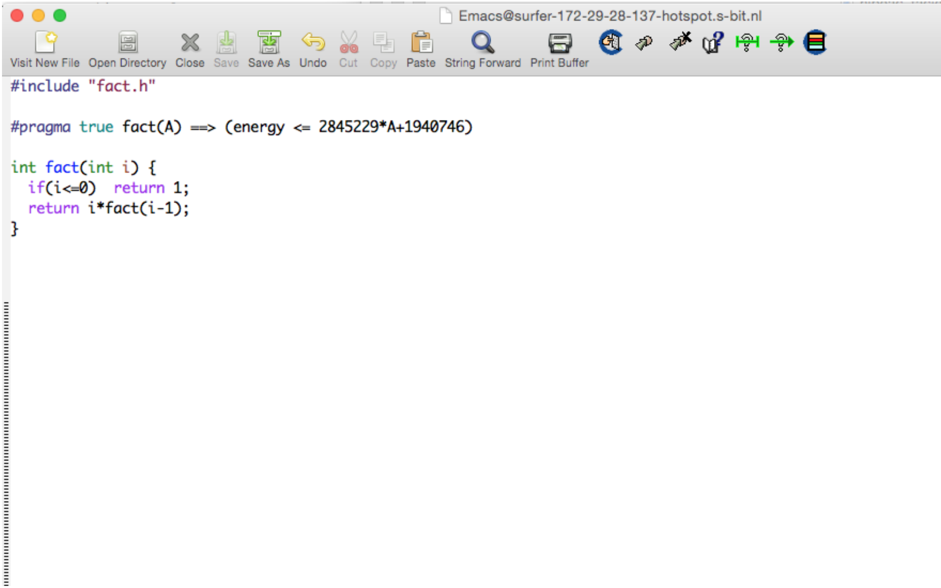
```
    Use Saved Menu Configuration:   none
               Select Menu Level:   naive
             Select Action Group:   analyze
  Select Aliasing-Mode Analysis:   none
      Select Shape-Type Analysis:   none
        Select Resource Analysis:   res_plai
            Include Energy Model:   yes
           Multivariant Success:   off
          Print Program Point Info:   off
               Collapse AI Info:   on
         {Current Saved Menu Configurations: []}
```

Analysis Results



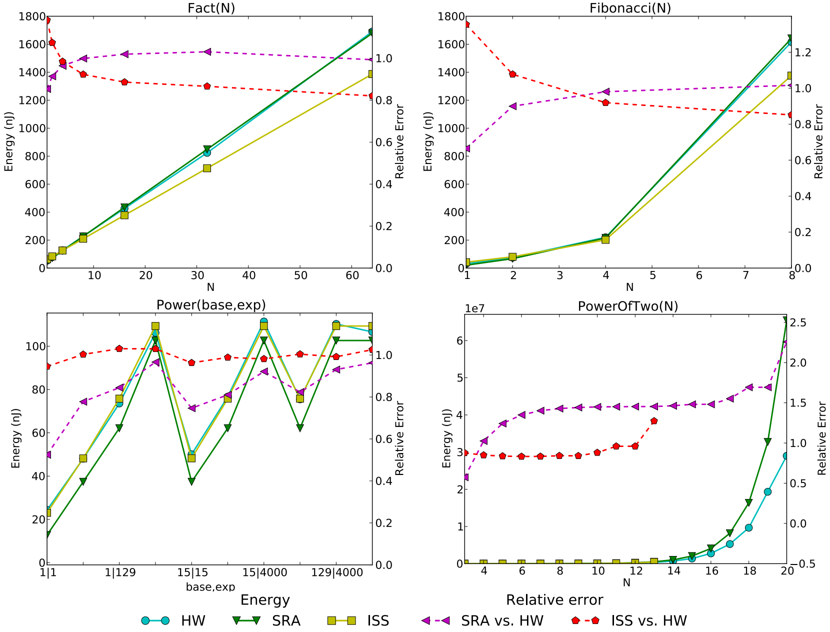
```
:- module(_,[fact/2],[ciaopp(xcore(model(instructions))),ciaopp(xcore(model(energy))),assertions]).

:- 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

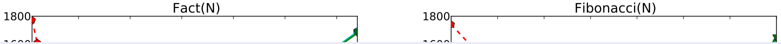


```
#include "fact.h"

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

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

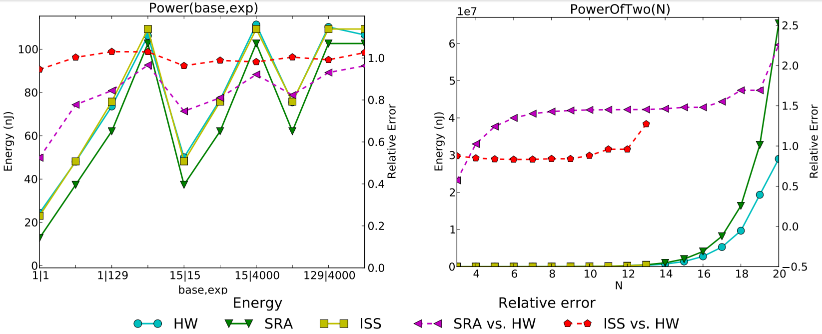
Some Results [LKSGL13]



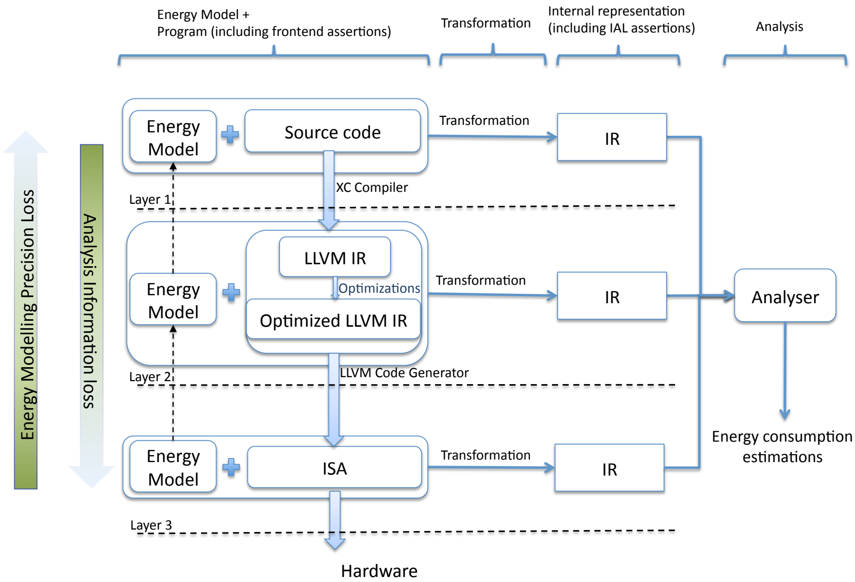
Some Results [LKSGL13]



- SRA provides results *beyond what is possible with simulation* (as test run-time increases, ISS becomes impractically long).
- SRA showed already promising accuracy in comparison with ISS and the HW (but still relatively simple benchmarks).
- Simulation time limits the usefulness of ISS method, whereas equation solving limits SRA.



IR Level Trade-offs



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;
}
```

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;
}
```

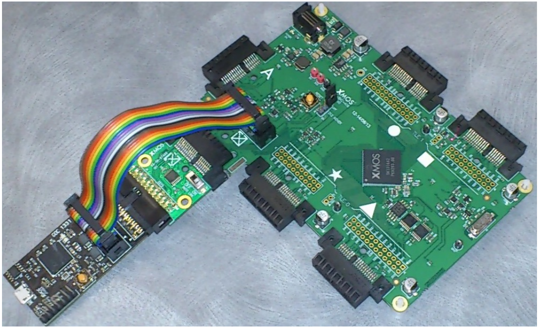
XC Energy Analysis Results – LLVMIR level

[LGKL+15]

Program	**Analysis at LLVMIR level** **Energy consumption funtions (in NJ)**
`fact(N)`	$28.4\ N + 22.4$
`fibonacci(N)`	$37.53 + 42.3 \times 1.62^N + 11.68 \times (-0.62)^N$
`sqr(N)`	$10.52\ N^2 + 55.79\ N + 16.5$
`power_of_two(N)`	$49.2 \times 2^N - 31.5$
`reverse(N,M)`	$20.50\ N + 72.98$
`concat(N,M)`	$69.14\ N + 69.14\ M + 14.12$
`mat_mult(N,M)`	$44.71\ N^3 + 72.47\ N^2 + 52.52\ N + 25.49$
`sum_facts(N,M)`	$69.14\ N + 69.14\ M + 14.12$
`fir(N)`	$33.47\ N + 141.6$
`biquad(N)`	$165.3\ N + 54.45$

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) [LGKL+15]

Program	**Error vs. HW**		**ISA/LLVMIR**
	isa	**llvmir**	
`fact(N)`	2.86%	4.50%	0.94
`fibonacci(N)`	5.41%	11.94%	0.92
`sqr(N)`	1.49%	9.31%	0.91
`power_of_two(N)`	4.26%	11.15%	0.93
`Average`	**3.50%**	**9.20%**	**0.92**
`reverse(N,M)`	N/A	2.18%	N/A
`concat(N,M)`	N/A	8.71%	N/A
`mat_mult(N,M)`	N/A	1.47%	N/A
`sum_facts(N,M)`	N/A	2.42%	N/A
`fir(N)`	N/A	0.63%	N/A
`biquad(N)`	N/A	2.34%	N/A
`Average`	**N/A**	**3.0%**	**N/A**
`Gobal Avg.`	**3.50%**	**5.48%**	**0.92**

Accuracy vs. HW measurements (ISA and LLVMIR) [LGKL+15]

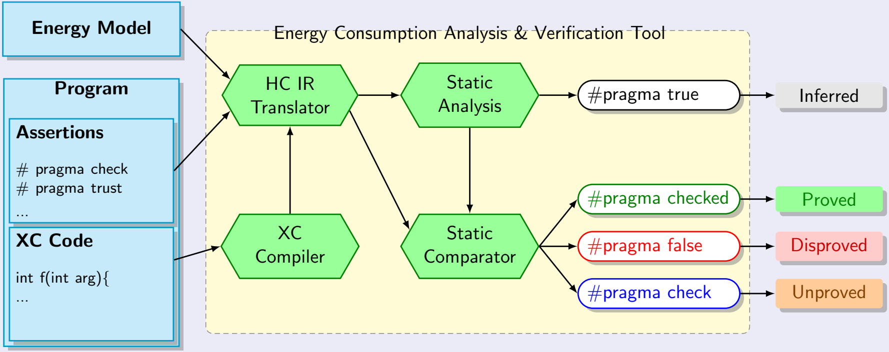
Program	**Error vs. HW**		**ISA/LLVMIR**
	isa	**llvmir**	
`fact(N)`	2.86%	4.50%	0.94
`fibonacci(N)`	5.41%	11.94%	0.92
`sqr(N)`	1.49%	9.31%	0.91
`power_of_two(N)`	4.26%	11.15%	0.93
`Average`	**3.50%**	**9.20%**	**0.92**
`Average`	**N/A**	**3.0%**	**N/A**
`Gobal Avg.`	**3.50%**	**5.48%**	**0.92**

- 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.

Applications

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

XC Energy Consumption Verification Tool (based on CiaoPP)



[LHKLH15]

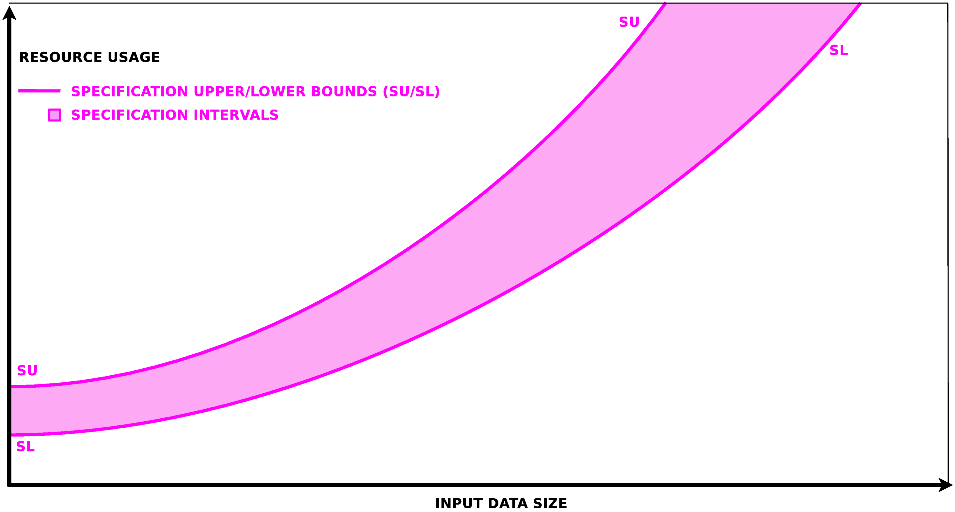
XC Program (FIR Filter) w/Energy Specification [LHKLH15]

```
#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);}
```

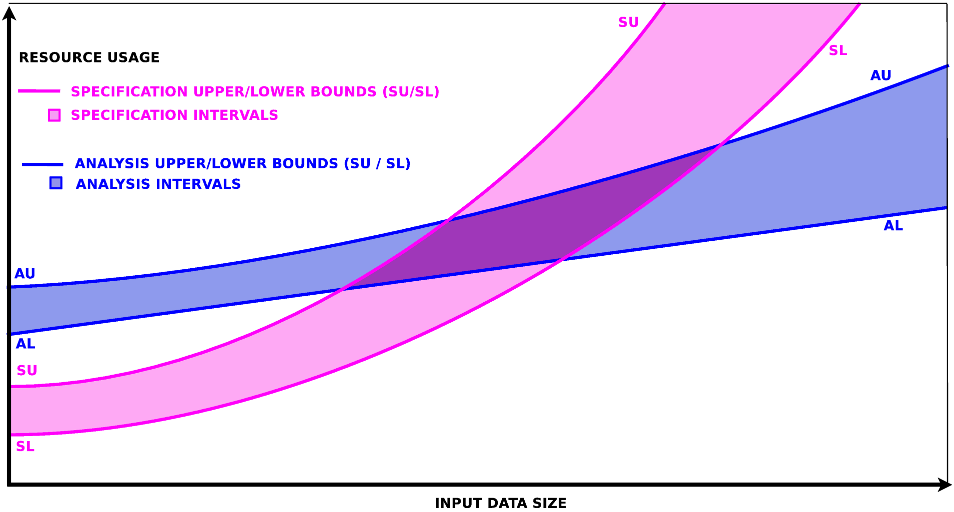
XC Program (FIR Filter) w/Energy Specification [LHKLH15]

```
#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);}
```

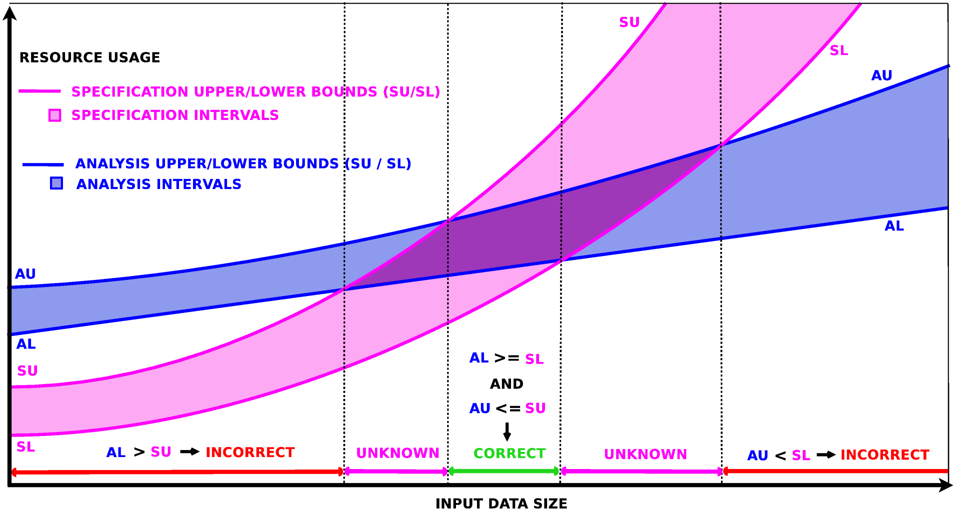
Resource Usage Verification – Function Comparisons



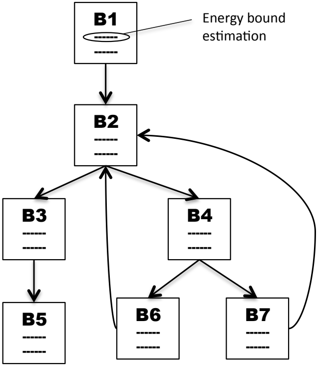
Resource Usage Verification – Function Comparisons



Resource Usage Verification – Function Comparisons



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.

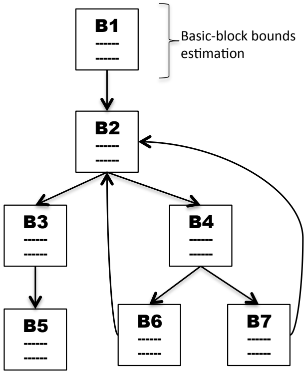
\+ Very compositional.

– Bounds obtained are *very conservative*.

– Dependence among instructions is not modeled (or complex).

Modeling at the Basic Block Level

[LBLGH16]



- 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.

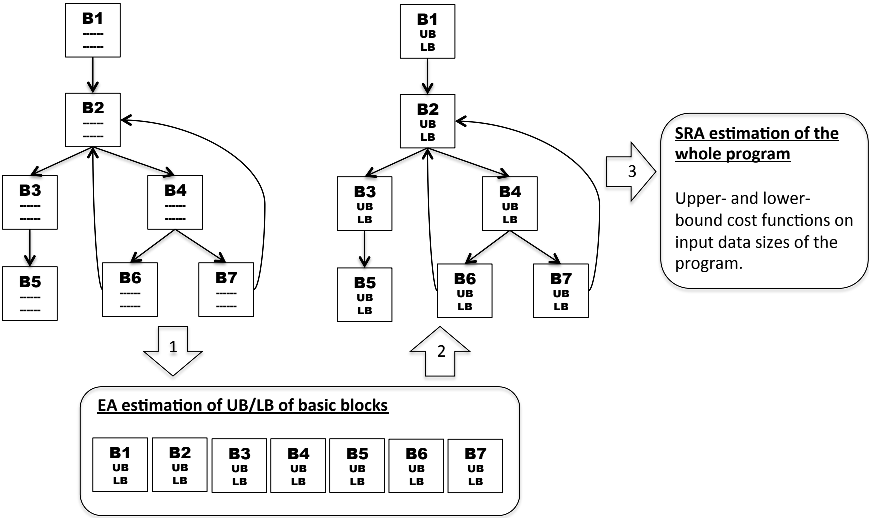
\+ Inter-instruction dependence is captured within the blocks: more precise bounds.

\+ 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

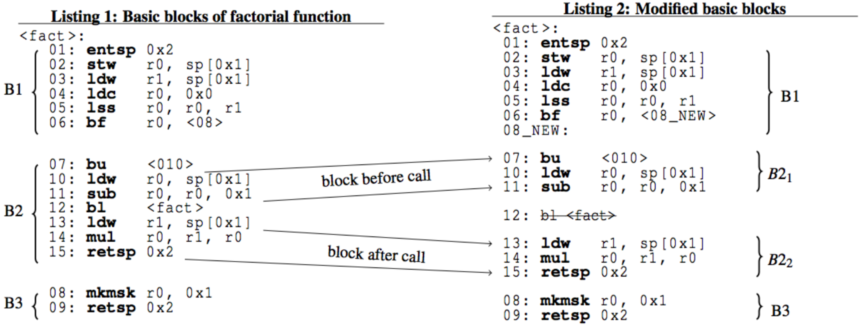


SRA estimation of the whole program

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

EA estimation of UB/LB of basic blocks

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]
B1   04: ldc   r0, 0x0
     05: lss   r0, r0, r1
     06: bf    r0, <08>

     07: bu    <010>
     10: ldw   r0, sp[0x1]
     11: sub   r0, r0, 0x1
B2   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               B1
 05: lss   r0, r0, r1
 06: bf    r0, <08_NEW>
 08_NEW:

 07: bu    <010>
 10: ldw   r0, sp[0x1]           B2_1
 11: sub   r0, r0, 0x1

 12: ~~bl <fact>~~

 13: ldw   r1, sp[0x1]
 14: mul   r0, r1, r0            B2_2
 15: retsp 0x2

 08: mkmsk r0, 0x1               B3
 09: retsp 0x2
```

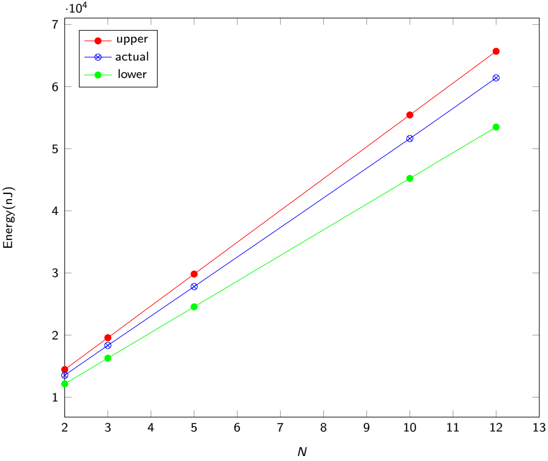
Experimental Evaluation (XMOS XS1 architecture)

Program	**Upper/Lower Bounds (nJ)**$\times 10^3$	**vs. HW**
$fact(N)$	$E_u = 5.1\ N + 4.2$ $E_l = 4.1\ N + 3.8$	7% -11.7%
$fibonacci(N)$	$E_u = 5.2\ lucas(N) + 6\ fib(N) - 6.6$ $E_l = 4.5\ lucas(N) + 5\ fib(N) - 4.2$	8.71% -4.69%
$reverse(N)$	$E_u = 3.7\ N + 13.3$ $E_l = 2.95\ N + 12$	8% -8.8%
$findMax(N)$	$E_u = 5\ N + 6.9$ $E_l = 3.3\ N + 5.6$	8.7% -9.1%
$fir(N)$	$E_u = 6\ N + 26.4$ $E_l = 4.8\ N + 22.9$	8.9% -9.7%
$biquad(N)$	$E_u = 29.6\ N + 10$ $E_l = 23.5\ N + 9$	9.8% -11.9%

- EA times vary depending upon the initialization parameters.
  - On average within 150-200 min.
- Static analysis times are relatively small $\approx 4sec$.

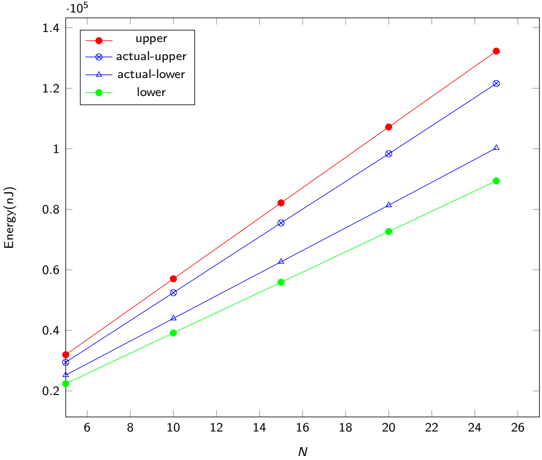
Experimental Results (Benchmark with no Data Dependent Branching)

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

[LGKLH16, HLGL$^+$16]

- For verification purposes *safe* upper and lower bounds are required → standard/classical cost, but
- for optimization → accumulated cost:
  - Guides the program developer to identify parts that should be optimized, because of their greater impact on the total cost.
- The *standard cost* of a procedure q is

  → the cost of a *single call* to q, denoted $\mathcal{C}_q(\bar{n})$.
- The *accumulated cost* of a procedure q is defined in the context of a call to another (or the same) procedure p.
  - It is the *addition of the (accumulated) costs of all calls to* q originated during the computation of a (single) call to p.
  - Denoted $\mathcal{C}_p^q(\bar{n})$, *"the accumulated cost of q when called from p"*.

Example: Variance of an Array of Integers

Assume that both mean() and variance () are declared as cost centers.

Naive implementation: mean() is responsible for most of the cost of the call to variance ().

```
int  variance(int * Arr, int N){
  int tmp[N], i = N;
  while(i > 0) {
    i--;
    tmp[i] = (Arr[i] - mean(Arr, N));
    tmp[i] = tmp[i] * tmp[i];
    }
  return mean(tmp, N);
}
```

Standard Cost:

$$\mathcal{C}_{mean}(N') \in \mathcal{O}(N')$$

$$\mathcal{C}_{variance}(N) \in \mathcal{O}(N^2)$$

Accumulated Cost:

$$\mathcal{C}^{mean}_{variance}(N) \in \mathcal{O}(N^2)$$

$$\mathcal{C}^{variance}_{variance}(N) \in \mathcal{O}(N)$$

Obvious improvement: move mean(Arr, N) outside the loop.

```
int  variance(int * Arr, int N){
  int tmp[N], int i = N;
  int m = mean(Arr, N);
  while(i > 0) {
    i--;
    tmp[i] = (Arr[i] - m);
    tmp[i] = tmp[i] * tmp[i];
  }
  return mean(tmp, N);
}
```

Standard Cost:

$$\mathcal{C}_{mean}(N') \in \mathcal{O}(N')$$

$$\mathcal{C}_{variance}(N) \in \mathcal{O}(N)$$

Accumulated Cost:

$$\mathcal{C}^{mean}_{variance}(N) \in \mathcal{O}(N)$$

$$\mathcal{C}^{variance}_{variance}(N) \in \mathcal{O}(N)$$

Our system infers that the costs accumulated in the variance () and mean() are linear.

Example: Variance of an Array of Integers

Assume that both mean() and variance () are declared as cost centers.

Naive implementation: mean() is responsible for most of the cost of the call to variance ().

```
int  variance(int * Arr, int N){
  int tmp[N], i = N;
  while(i > 0) {
    i--;
    tmp[i] = (Arr[i] - mean(Arr, N));
    tmp[i] = tmp[i] * tmp[i];
    }
  return mean(tmp, N);
}
```

Standard Cost:

$$\mathcal{C}_{mean}(N') \in \mathcal{O}(N')$$
$$\mathcal{C}_{variance}(N) \in \mathcal{O}(N^2)$$

Accumulated Cost:

$$\mathcal{C}^{mean}_{variance}(N) \in \mathcal{O}(N^2)$$
$$\mathcal{C}^{variance}_{variance}(N) \in \mathcal{O}(N)$$

Obvious improvement: move mean(Arr, N) outside the loop.

```
int  variance(int * Arr, int N){
  int tmp[N], int i = N;
  int m = mean(Arr, N);
  while(i > 0) {
    i--;
    tmp[i] = (Arr[i] - m);
    tmp[i] = tmp[i] * tmp[i];
  }
  return mean(tmp, N);
}
```

Standard Cost:

$$\mathcal{C}_{mean}(N') \in \mathcal{O}(N')$$
$$\mathcal{C}_{variance}(N) \in \mathcal{O}(N)$$

Accumulated Cost:

$$\mathcal{C}^{mean}_{variance}(N) \in \mathcal{O}(N)$$
$$\mathcal{C}^{variance}_{variance}(N) \in \mathcal{O}(N)$$

Our system infers that the costs accumulated in the variance () and mean() are linear.

Example: Variance of an Array of Integers

Assume that both mean() and variance() are declared as cost centers.

Naive implementation: mean() is responsible for most of the cost of the call to variance().

```
int  variance(int * Arr, int N){
  int tmp[N], i = N;
  while(i > 0) {
    i--;
    tmp[i] = (Arr[i] - mean(Arr, N));
    tmp[i] = tmp[i] * tmp[i];
    }
  return mean(tmp, N);
}
```

Standard Cost:

$$\mathcal{C}_{mean}(N') \in \mathcal{O}(N')$$

$$\mathcal{C}_{variance}(N) \in \mathcal{O}(N^2)$$

Accumulated Cost:

$$\mathcal{C}^{mean}_{variance}(N) \in \mathcal{O}(N^2)$$

$$\mathcal{C}^{variance}_{variance}(N) \in \mathcal{O}(N)$$

Obvious improvement: move mean(Arr, N) outside the loop.

```
int  variance(int * Arr, int N){
  int tmp[N], int i = N;
  int m = mean(Arr, N);
  while(i > 0) {
    i--;
    tmp[i] = (Arr[i] - m);
    tmp[i] = tmp[i] * tmp[i];
  }
  return mean(tmp, N);
}
```

Standard Cost:

$$\mathcal{C}_{mean}(N') \in \mathcal{O}(N')$$

$$\mathcal{C}_{variance}(N) \in \mathcal{O}(N)$$

Accumulated Cost:

$$\mathcal{C}^{mean}_{variance}(N) \in \mathcal{O}(N)$$

$$\mathcal{C}^{variance}_{variance}(N) \in \mathcal{O}(N)$$

Our system infers that the costs accumulated in the variance() and mean() are linear.

Example: Variance of an Array of Integers

Assume that both mean() and variance () are declared as cost centers.

Naive implementation: mean() is responsible for most of the cost of the call to variance ().

```
int  variance(int * Arr, int N){
  int tmp[N], i = N;
  while(i > 0) {
    i--;
    tmp[i] = (Arr[i] - mean(Arr, N));
    tmp[i] = tmp[i] * tmp[i];
    }
  return mean(tmp, N);
}
```

Standard Cost:

$$\mathcal{C}_{mean}(N') \in \mathcal{O}(N')$$
$$\mathcal{C}_{variance}(N) \in \mathcal{O}(N^2)$$

Accumulated Cost:

$$\mathcal{C}^{mean}_{variance}(N) \in \mathcal{O}(N^2)$$
$$\mathcal{C}^{variance}_{variance}(N) \in \mathcal{O}(N)$$

Obvious improvement: move mean(Arr, N) outside the loop.

```
int  variance(int * Arr, int N){
  int tmp[N], int i = N;
  int m = mean(Arr, N);
  while(i > 0) {
    i--;
    tmp[i] = (Arr[i] - m);
    tmp[i] = tmp[i] * tmp[i];
  }
  return mean(tmp, N);
}
```

Standard Cost:

$$\mathcal{C}_{mean}(N') \in \mathcal{O}(N')$$
$$\mathcal{C}_{variance}(N) \in \mathcal{O}(N)$$

Accumulated Cost:

$$\mathcal{C}^{mean}_{variance}(N) \in \mathcal{O}(N)$$
$$\mathcal{C}^{variance}_{variance}(N) \in \mathcal{O}(N)$$

Our system infers that the costs accumulated in the variance () and mean() are linear.

Accumulated Cost: Experimental Results

Cost-Centers & Input Sizes	Accumulated Cost UB	Static vs. Dyn	Standard Cost UB	#Calls
$variance(n)^*$	1	0%	$2n^2$	1
$sq\_diff(m_1, m_2)$	$n-1$	0%	$2m_1m_2 - 2m_2$	$n-1$
$mean(u)$	$2n^2 - n$	0%	$2u+1$	n
$is\_prime(n)^*$	1	0%	$(n-1)! + n + 3$	1
$fact(m)$	n	0%	m	n
$mult(u)$	$(n-1)! + 2$	0%	$u+1$	$(n-1)! + 2$
$app1(n_1, n_2, n_3)^*$	n_1	0%	$\mathcal{O}(n_1n_2n_3)^\dagger$	1
$app2(m_1, m_2)$	n_1n_2	0%	m_1m_2	n_1
$app3(u)$	$2n_1n_2n_3$	0%	u	$n_1n_2 + n_1$
$dyade(n_1, n_2)^*$	n_1	0%	$n_1(n_2+1)$	1
$mult(m)$	n_1n_2	0%	m	n_1
$minsort(n)^*$	$n+1$	0%	$\frac{(n+1)^2}{2} + \frac{n+1}{2}$	1
$findmin(m)$	$\frac{(n+1)^2}{2} + \frac{n-1}{2}$	7%	m	$n+1$
$hanoi(n)^*$	$2^n - 1$	0%	$2^{n+1} - 2$	1
$move(m)$	$2^n - 1$	0%	1	$2^n - 1$
$coupled(n)^*$	1	0%	$n+2$	1
$p(m)$	$\frac{n}{2} + \frac{(-1)^n}{4} + \frac{3}{4}$	1.2%	$m+1$	$\frac{n}{2} - \frac{(-1)^n}{4} + \frac{1}{4}$
$q(u)$	$\frac{n}{2} - \frac{(-1)^n}{4} + \frac{1}{4}$	0%	$u+1$	$\frac{n}{2} + \frac{(-1)^n}{4} - \frac{1}{4}$
$search(n)^*$	1	0%	$2n+2$	1
$member(m)$	$2n+1$	0%	$2m+1$	$2n+1$
$sublist(n_1, n_2)^*$	$n_2 + 3$	5%	$n_1n_2 + 3n_2 + 2$	2
$append(m)$	$n_1n_2 + 2n_2 - 1$	40%	$2m-1$	$n_1n_2 + 2n_2 - 1$

Experimental Results: Times (milliseconds)

Cost-Center	**Accumulated Cost UB** Cost Relations	Transformation (FLOPS'16)	**Standard Cost UB**	**Acc / Std**
*variance** *sq_diff* *mean*	3283 (-45%)	6038	3066	1.07
*isprime** *fact* *mult*	1245 (-42%)	2172	1231	1.01
*app1** *app2* *app3*	4150 (-34%)	6328	3757	1.11
*minsort** *findmin*	3400 (-29%)	4845	3300	1.03
*dyade** *mult*	3097 (-24%)	4117	2853	1.08
*hanoi** *move*	1605 (-19%)	1996	1376	1.16
*coupled** *f* *g*	2407 (-14%)	3112	1877	1.28
*search** *member*	1079	N/A	1071	1.00
*sublist** *append*	3674	N/A	3610	1.01
Average	**2652 (-33%)**	**4125**	**2542**	**1.05**

Tools / timeline

'83	Parallel abstract machines → motivation: auto-parallelization.
'88	**MA3 analyzer**: memo tables (cf. OLDT resolution), practicality established.
'89	**PLAI analyzer**: accelerated fixpoint, abstract domains as plugins. Sharing analysis, side-effect analysis.
90's	Incremental analysis, concurrency (dynamic scheduling), automatic domain combinations, scalability, auto-parallelization, extension to constraints.
'90	**GraCos analyzer**: fully automatic cost analysis (upper bounds).
early 90's	Automatic parallelization with task granularity control.
mid 90's	*CiaoPP model: Integrated verification/debugging/optimization w/assertions.*
'97-present	**CiaoPP tool**:
'91-'06	Combined **abstract interpretation and partial evaluation**.
late 90's	**Lower bounds** cost analysis. Non-failure (no exceptions), determinacy.
'01	**Verification** of cost, additional resources, ...
'01-05	Modularity/scalability. Diagnosis (locating origin of assrt. violations). New shape/type domains, widenings. Polyhedra, convex hulls.
'03	Abstraction carrying code, reduced certificates.
'04	Verification/debugging/optimization of **user-defined** resources.
'05	**Multi-language support** using CLP as IR: Java, C# (shapes, resources, ...).
'08	Verification of exec. **time**. First results in **energy** (Java), heap models, ...
'12	(X)C program energy analysis/verification, ISA-level energy models.
'13	**Cost anal. as Abs. Int.** Sized shapes. LLVM. **Accumulated Cost**

Thank you!

Experimental Results

Prog.	Resource An. (LB)			Resource An. (UB)					An. Time (s)	
	New	*Prev.*		*New*	*Prev.*		*RAML*		*New*	*Prev.*
`append`	α	α	$=$	β	β	$=$	β	$=$	1.00	0.53
`appAll`	$a_1 a_2 a_3$	a_1	$+$	$b_1 b_2 b_3$	∞	$+$	$b_1 b_2 b_3$	$=$	2.41	0.67
`coupled`	μ	0	$+$	ν	∞	$+$	ν	$=$	1.37	0.64
`dyade`	$\alpha_1\alpha_2$	$\alpha_1\alpha_2$	$=$	$\beta_1\beta_2$	$\beta_1\beta_2$	$=$	$\beta_1\beta_2$	$=$	1.66	0.62
`erathos`	α	α	$=$	β^2	β^2	$=$	β^2	$=$	2.25	0.77
`fib`	ϕ^μ	ϕ^μ	$=$	ϕ^ν	ϕ^ν	$=$	infeas.	$+$	1.06	0.67
`hanoi`	1	0	$+$	2^ν	∞	$+$	infeas.	$+$	0.82	0.60
`isort`	α^2	α^2	$=$	β^2	β^2	$=$	β^2	$=$	1.68	0.62
`isortl`	a_1^2	a_1^2	$=$	$b_1^2 b_2$	∞	$+$	$b_1^2 b_2$	$=$	2.55	0.67
`lisfact`	$\alpha\gamma$	α	$+$	$\beta\delta$	∞	$+$	unkn.	?	1.39	0.64
`listnum`	μ	μ	$=$	ν	ν	$=$	unkn.	?	1.19	0.58
`minsort`	α^2	α	$+$	β^2	β^2	$=$	β^2	$=$	1.94	0.67
`nub`	a_1	a_1	$=$	$b_1^2 b_2$	∞	$+$	$b_1^2 b_2$	$=$	3.61	0.91
`part`	α	α	$=$	β	β	$=$	β	$=$	1.70	0.65
`zip3`	$\min(\alpha_i)$	0	$+$	$\min(\beta_i)$	∞	$+$	β_3	$+$	2.48	0.57

IR Issues: Approaches to Performing the Transformation

- The transformation (akin to *Abstract Compilation*):
  - Source: Program P in L_P + (possibly abstract) Semantics of L_P
  - Target: A (C) Horn Clause program capturing $[\![P]\!]$ (or, possibly, $[\![P]\!]^\alpha$)

- Some approaches to performing the transformation:
  - Partial evaluation of instrumented interpreters + slicing.
    - Systematic construction from small- and big-step semantics.
    - Correctness proof more direct.
    - Not always fully automatic?
  - Direct transformation into block-based intermediate representation.
    - More control but correctness proof more indirect.
    - Used in the following (translation to a Ciao program).
    - Can add assertions to help analysis (sizes, metrics, resource models, ..).

The two approaches can produce similar results.

CFG traversal

- Blocks are nodes; edges are invocations.
- Top-down traversal of this CFG, starting from entry point.
- Within each block: sequence of builtins, handled in the domain.
- Inter-block calls/edges: *project*, *extend*, etc. (next slide).
- As graph is traversed, triples $(block, \lambda_{in}, \lambda_{out})$ are stored for each block in a *memo table*.
- Memo table entries have status $\in \{fixpoint, approx., complete\}$.
- Iterate until all *complete*.

Interprocedural analysis / recursion support

- **Project** the caller state over the actual parameters,
- find all the **compatible implementations** (blocks),
- **rename** to their formal parameters,

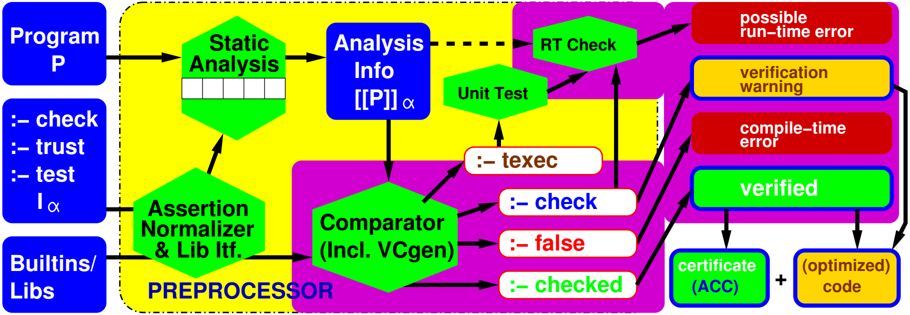
... abstractly execute each compatible block, ...

- calculate the **least upper bound** of the partial results of each block (if "monovariant on success" flag),
- **rename back** to the actual parameters and, finally
- **extend** (reconcile) return state into calling state.

Speeding up convergence

- Analyze non-recursive blocks first,
  use as starting λ_{out} in recursions.
- Blocks derived from conditionals treated specially
  (no *project* or *extend* operations required).
- The $(block, \lambda_{in}, \lambda_{out})$ tuples
  act as a cache that avoids recomputation.
- Use strongly-connected components (on the fly).

Integrated Static/Dynamic Debugging and Verification



Program P

:– check
:– trust
:– test
I_α

Builtins/ Libs

Static Analysis

Assertion Normalizer & Lib Itf.

PREPROCESSOR

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

Comparator (Incl. VCgen)

RT Check

Unit Test

:– texec

:– check

:– false

:– checked

possible run–time error

verification warning

compile–time error

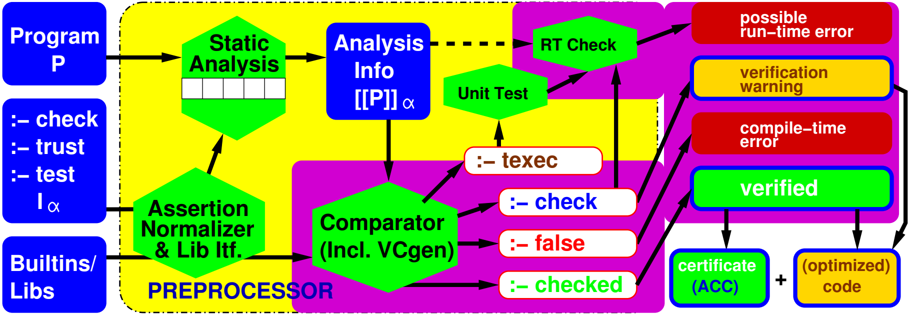
verified

certificate (ACC) + **(optimized) code**

	Definition	Sufficient condition
P is prt. correct w.r.t. $\mathcal{I}_\alpha$ if	$\alpha(\llbracket P \rrbracket) \leq \mathcal{I}_\alpha$	$\llbracket P \rrbracket_{\alpha^+} \leq \mathcal{I}_\alpha$
P is complete w.r.t. $\mathcal{I}_\alpha$ if	$\mathcal{I}_\alpha \leq \alpha(\llbracket P \rrbracket)$	$\mathcal{I}_\alpha \leq \llbracket P \rrbracket_{\alpha^=}$
P is incorrect w.r.t. $\mathcal{I}_\alpha$ if	$\alpha(\llbracket P \rrbracket) \not\leq \mathcal{I}_\alpha$	$\llbracket P \rrbracket_{\alpha^=} \not\leq \mathcal{I}_\alpha$, or $\llbracket P \rrbracket_{\alpha^+} \cap \mathcal{I}_\alpha = \emptyset \wedge \llbracket P \rrbracket_\alpha \neq \emptyset$
P is incomplete w.r.t. $\mathcal{I}_\alpha$ if	$\mathcal{I}_\alpha \not\leq \alpha(\llbracket P \rrbracket)$	$\mathcal{I}_\alpha \not\leq \llbracket P \rrbracket_{\alpha^+}$

[BDD$^+$97, HPB99, PBH00c, PBH00a, HPBLG03, HALGP05, PCPH06, PCPH08, MLGH09, SMH14, SMH15, SMH16]

Integrated Static/Dynamic Debugging and Verification



- 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, SMH15, SMH16]

References – Analysis and Verification of Energy

[LBLGH16] U. Liqat, Z. Banković, P. Lopez-Garcia, and M. V. Hermenegildo.
Inferring Energy Bounds Statically by Evolutionary Analysis of Basic Blocks.
In *(HIP3ES'16)*, 2016. arXiv:1601.02800.

[LGK$^+$16] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and K. Eder.
Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVM IR.
In M. Van Eekelen and U. Dal Lago, editors, *Foundational and Practical Aspects of Resource Analysis: 4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers*, volume 9964 of *Lecture Notes in Computer Science*, pages 81–100. Springer, 2016.

[LHKLH15] P. Lopez-Garcia, R. Haemmerlé, M. Klemen, U. Liqat, and M. V. Hermenegildo
Towards Energy Consumption Verification via Static Analysis.
In *HIPEAC Workshop on High Performance Energy Efficient Embedded Systems (HIP3ES 2015)*, Amsterdam.

[LGKL$^+$15] U. Liqat, K. Georgiou, S. Kerrison, P. Lopez-Garcia, M. V. Hermenegildo, J. P. Gallagher, and Kerstin Eder.
Inferring Parametric Energy Consumption Functions at Different Software Levels: ISA vs. LLVMIR. Under review.

[LKSGL13] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. López-Garcia, N. Grech, M.V. Hermenegildo, and K. Eder.
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)*, LNCS, Springer, September 2013.

[NMLH08] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications.
In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, April 2008. Extended Abstract.

References – Analysis and Verification of Resources in General

[LGKLH16] P. Lopez-Garcia, M. Klemen, U. Liqat, and M.V. Hermenegildo.
A General Framework for Static Profiling of Parametric Resource Usage.
Theory and Practice of Logic Programming, 32nd Int'l. Conference on Logic Programming (ICLP'16) Special Issue, 16(5-6):849–865, October 2016.

[HLGL$^+$16] R. Haemmerlé, P. Lopez-Garcia, U. Liqat, M. Klemen, J. P. Gallagher, and M. V. Hermenegildo.
A Transformational Approach to Parametric Accumulated-cost Static Profiling.
In *FLOPS'16*, volume 9613 of *LNCS*, pages 163–180. Springer, 2016.

[SLH14] A. Serrano, P. López-Garcia, and M. Hermenegildo.
Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types.
In *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, Vol. 14, Num. 4-5, pages 739-754, Cambridge U. Press, 2014.

[SLBH13] A. Serrano, P. López-Garcia, F. Bueno, and M. Hermenegildo.
Sized Type Analysis of Logic Programs (Technical Communication).
In *Theory and Practice of Logic Programming, 29th Int'l. Conference on Logic Programming (ICLP'13) Special Issue, On-line Supplement,* pages 1–14, Cambridge U. Press, August 2013.

[LDBH123] P. Lopez-Garcia, L. Darmawan, F. Bueno, and M. Hermenegildo
Interval-Based Resource Usage Verification: Formalization and Prototype
In *Foundational and Practical Aspects of Resource Analysis. Second Iternational Workshop FOPARA 2011, Revised Selected Papers*. Lecture Notes in Computer Science, 2012, 7177, 54–71, Springer.

[LGDB10] P. López-García, L. Darmawan, and F. Bueno.
A Framework for Verification and Debugging of Resource Usage Properties.
In *Technical Communications of the 26th ICLP. Leibniz Int'l. Proc. in Informatics (LIPIcs)*, Vol. 7, pages 104–113, Dagstuhl, Germany, July 2010.

[NMLH09] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
User-Definable Resource Usage Bounds Analysis for Java Bytecode.
In *Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *ENTCS*, pages 6–86. Elsevier, March 2009.

[MLGCH08] E. Mera, P. López-García, M. Carro, and M. Hermenegildo.
Towards Execution Time Estimation in Abstract Machine-Based Languages.
In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[NMLGH07] J. Navas, E. Mera, P. López-García, and M. Hermenegildo.
User-Definable Resource Bounds Analysis for Logic Programs.
In *23rd International Conference on Logic Programming (ICLP'07)*, LNCS Vol. 4670. Springer, 2007.

[DLGHL97] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin.
Lower Bound Cost Estimation for Logic Programs.
In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[DLGHL94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin.
Estimating the Computational Cost of Logic Programs.
In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Sep 1994. Springer.

[LGHD96] P. López-García, M. Hermenegildo, and S. K. Debray.
A Methodology for Granularity Based Control of Parallelism in Logic Programs.
Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation, 21(4–6):715–734, 1996.

[LGHD94] P. López-García, M. Hermenegildo, and S.K. Debray.
Towards Granularity Based Control of Parallelism in Logic Programs.
In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

[DLH90] S. K. Debray, N.-W. Lin, and M. Hermenegildo.
Task Granularity Analysis in Logic Programs.
In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

References – Intermediate Representation / Multi-Lingual Support

[MLNH07] M. Méndez-Lojo, J. Navas, and M. Hermenegildo.
A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs.
In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2007)*, number 4915 in LNCS, pages 154–168. Springer-Verlag, August 2007.

References – Assertion Language

[SMH14] N. Stulova, J. F. Morales, M. V. Hermenegildo.
Assertion-based Debugging of Higher-Order (C)LP Programs.
In *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*, ACM Press, September 2014.

[MLGH09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.
In *25th International Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.

[PBH00b] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Constraint Logic Programs.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[PBH97] G. Puebla, F. Bueno, and M. Hermenegildo.
An Assertion Language for Debugging of Constraint Logic Programs.
In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997. Available from
`ftp://cliplab.org/pub/papers/assert_lang_tr_discipldeliv.ps.gz` as tech. report CLIP2/97.1.

References – Overall Debugging and Verification Model

[SMH16] N. Stulova, J. F. Morales, and M. V. Hermenegildo.
Reducing the Overhead of Assertion Run-time Checks via static analysis.
In *18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16)*, pages 90–103. ACM Press, September 2016.

[SMH15] N. Stulova, J. F. Morales, and M. V. Hermenegildo.
Practical Run-time Checking via Unobtrusive Property Caching.
Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue, 15(04-05):726–741, September 2015.

[MLGH09] E. Mera, P. López-García, and M. Hermenegildo.
Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework.
In *25th International Conference on Logic Programming (ICLP'09)*, number 5649 in LNCS, pages 281–295. Springer-Verlag, July 2009.

[HPBLG05] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García.
Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation
Science of Computer Programming, 58(1–2), 2005.

[HPBLG03] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García.
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.

[PBH00a] G. Puebla, F. Bueno, and M. Hermenegildo.
A Generic Preprocessor for Program Validation and Debugging.
In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[PBH00c] G. Puebla, F. Bueno, and M. Hermenegildo.
Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs.
In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

[HPB99] M. Hermenegildo, G. Puebla, and F. Bueno.
Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging.
In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[BDD$^+$97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla.
On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs.
In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

References – Abstraction Carrying Code

[AAPH06] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo.
Reduced Certificates for Abstraction-Carrying Code.
In *22nd International Conference on Logic Programming (ICLP 2006)*, number 4079 in LNCS, pages 163–178. Springer-Verlag, August 2006.

[HALGP05] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla.
Abstraction Carrying Code and Resource-Awareness.
In *PPDP*. ACM Press, 2005.

[APH05] E. Albert, G. Puebla, and M. Hermenegildo.
Abstraction-Carrying Code.
In *Proc. of LPAR'04*, volume 3452 of *LNAI*. Springer, 2005.

References – Fixpoint-based Analyzer (Abstract Interpreter)

[NMLH07] J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
An Efficient, Context and Path Sensitive Analysis Framework for Java Programs.
In *9th Workshop on Formal Techniques for Java-like Programs FTfJP 2007*, July 2007.

[HPMS00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey.
Incremental Analysis of Constraint Logic Programs.
ACM Transactions on Programming Languages and Systems, 22(2):187–223, March 2000.

[BGH99] F. Bueno, M. García de la Banda, and M. Hermenegildo.
Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming.
ACM Transactions on Programming Languages and Systems, 21(2):189–238, March 1999.

[PH96] G. Puebla and M. Hermenegildo.
Optimized Algorithms for the Incremental Analysis of Logic Programs.
In *International Static Analysis Symposium (SAS 1996)*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[MGH94] K. Marriott, M. García de la Banda, and M. Hermenegildo.
Analyzing Logic Programs with Dynamic Scheduling.
In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[MH92] K. Muthukumar and M. Hermenegildo.
Compile-time Derivation of Variable Dependency Using Abstract Interpretation.
Journal of Logic Programming, 13(2/3):315–347, July 1992.

References – Modular Analysis, Analysis of Concurrency

[PCPH08] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo.
A Practical Type Analysis for Verification of Modular Prolog Programs.
In *PEPM'08*, pages 61–70. ACM Press, January 2008.

[PCPH06] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo.
Context-Sensitive Multivariant Assertion Checking in Modular Programs.
In *LPAR'06*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

[BdlBH$^+$01] F. Bueno, M. García de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. Stuckey.
A Model for Inter-module Analysis and Optimizing Compilation.
In *Logic-based Program Synthesis and Transformation*, number 2042 in LNCS, pages 86–102. Springer-Verlag, March 2001.

[PH00] G. Puebla and M. Hermenegildo.
Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs.
In *Special Issue on Optimization and Implementation of Declarative Programming Languages*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier - North Holland, March 2000.

[MGH94] K. Marriott, M. García de la Banda, and M. Hermenegildo.
Analyzing Logic Programs with Dynamic Scheduling.
In *20th. Annual ACM Conf. on Principles of Programming Languages*, pages 240–254. ACM, January 1994.

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla.
Global Analysis of Standard Prolog Programs.
In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

References – Domains: Sharing/Aliasing

[MKH09] M. Marron, D. Kapur, and M. Hermenegildo.
Identification of Logically Related Heap Regions.
In *ISMM'09: Proceedings of the 8th international symposium on Memory management*, New York, NY, USA, June 2009. ACM Press.

[MLLH08] M. Méndez-Lojo, O. Lhoták, and M. Hermenegildo.
Efficient Set Sharing using ZBDDs.
In *21st Int'l. WS on Languages and Compilers for Parallel Computing (LCPC'08)*, LNCS. Springer-Verlag, August 2008.

[MKSH08] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo.
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.

[MMLH$^+$08] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur.
Sharing Analysis of Arrays, Collections, and Recursive Structures.
In *ACM WS on Program Analysis for SW Tools and Engineering (PASTE'08)*. ACM, November 2008.

[MLH08] M. Méndez-Lojo and M. Hermenegildo.
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.

[NBH06] J. Navas, F. Bueno, and M. Hermenegildo.
Efficient top-down set-sharing analysis using cliques.
In *Eight International Symposium on Practical Aspects of Declarative Languages*, number 2819 in LNCS, pages 183–198. Springer-Verlag, January 2006.

[MH91] K. Muthukumar and M. Hermenegildo.
Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation.
In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[MH89] K. Muthukumar and M. Hermenegildo.
Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation.
In *1989 North American Conf. on Logic Programming*, pages 166–189. MIT Press, October 1989.

References – Domains: Shape/Type Analysis

[MHKS08] M. Marron, M. Hermenegildo, D. Kapur, and D. Stefanovic.
Efficient context-sensitive shape analysis with graph-based heap models.
In Laurie Hendren, editor, *International Conference on Compiler Construction (CC 2008)*, Lecture Notes in Computer Science. Springer, April 2008.

[MSHK07] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur.
Heap Analysis in the Presence of Collection Libraries.
In *ACM WS on Program Analysis for Software Tools and Engineering (PASTE'07)*. ACM, June 2007.

[VB02] C. Vaucheret and F. Bueno.
More Precise yet Efficient Type Inference for Logic Programs.
In *International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 102–116. Springer-Verlag, September 2002.

References – Domains: Non-failure, Determinacy

[LGBH10] P. López-García, F. Bueno, and M. Hermenegildo.
Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information.
New Generation Computing, 28(2):117–206, 2010.

[LGBH05] P. López-García, F. Bueno, and M. Hermenegildo.
Determinacy Analysis for Logic Programs Using Mode and Type Information.
In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 19–35. Springer-Verlag, August 2005.

[BLGH04] F. Bueno, P. López-García, and M. Hermenegildo.
Multivariant Non-Failure Analysis via Standard Abstract Interpretation.
In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[DLGH97] S.K. Debray, P. López-García, and M. Hermenegildo.
Non-Failure Analysis for Logic Programs.
In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

References – Automatic Parallelization, (Abstract) Partial Evaluation, Other Optimizations

[MKSH08] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo.
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.

[CCH08] A. Casas, M. Carro, and M. Hermenegildo.
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.

[CMM$^+$06] M. Carro, J. Morales, H.L. Muller, G. Puebla, and M. Hermenegildo.
High-Level Languages for Small Devices: A Case Study.
In Krisztian Flautner and Taewhan Kim, editors, *Compilers, Architecture, and Synthesis for Embedded Systems*, pages 271–281. ACM Press / Sheridan, October 2006.

[PAH06] G. Puebla, E. Albert, and M. Hermenegildo.
Abstract Interpretation with Specialized Definitions.
In *SAS'06*, number 4134 in LNCS, pages 107–126. Springer-Verlag, 2006.

[MCH04] J. Morales, M. Carro, and M. Hermenegildo.
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.

[PH03] G. Puebla and M. Hermenegildo.
Abstract Specialization and its Applications.
In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003.
Invited talk.

[PHG99] G. Puebla, M. Hermenegildo, and J. Gallagher.
An Integration of Partial Evaluation in a Generic Abstract Interpretation Framework.
In O Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, number NS-99-1 in BRISC Series, pages 75–85. University of Aarhus, Denmark, January 1999.

[PH99] G. Puebla and M. Hermenegildo.
Abstract Multiple Specialization and its Application to Program Parallelization.
J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs, 41(2&3):279–316, November 1999.

[MBdlBH99] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo.
Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism.
Journal of Logic Programming, 38(2):165–218, February 1999.

[PH97] G. Puebla and M. Hermenegildo.
Abstract Specialization and its Application to Program Parallelization.
In J. Gallagher, editor, *Logic Program Synthesis and Transformation*, number 1207 in LNCS, pages 169–186. Springer-Verlag, 1997.

[GH91] F. Giannotti and M. Hermenegildo.
A Technique for Recursive Invariance Detection and Selective Program Specialization.
In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.