Recent Developments in the CiaoPP Resource Analysis Tools

Umer Liqat,
Rémy Haemmerlé,
Maximiliano Klemen,
Luthfi Darmawan,
Pedro López-García and
Manuel Hermenegildo

IMDEA Software Institute, Spain

ENTRA Workshop
Malaga, May 6-7, 2015

Outline

- Verification of energy budgets.
- Modular multi-language resource analysis.
- Improvements in the recurrence relation solver component.

Energy Consumption Verification of Software

- *Conventional* understanding of software *correctness*:
  - Conformance to a *functional* or *behavioral specification* (*what* the program is supposed to compute or do).

- But, in many current applications it is important or essential to ensure conformance wrt. *resource usage specifications*.
  → expressing *energy*, execution time, memory, ... or user-defined resources.

- Examples:
  - An embedded application in a *battery-operated* device – energy budget.
  - An embedded *real-time* program – response time limits.

Energy Consumption Verification of Software (Contd.)

Leverage the CiaoPP general framework for resource usage verification:
→ *energy consumption specifications of XC programs*.

- Specifications can express intervals within which energy usage is to be certified to be within the given bounds.
- The bounds of the intervals can be given as functions on input data sizes.

- Our verification system can infer particular conditions under which the energy usage specifications hold (or do not hold).
- Prototype implementation (also within CiaoPP) for the XC language and (XMOS) XS1-L architecture.

Leveraging the Resource Usage Verification Framework

1. Extending the verification component to verify functions inferred by the new abstract interpretation based resource usage analysis.
2. Assertion transformation from XC to HC IR and vice versa.
   - XC check assertions (specifications) into HC IR for verification component.
   - Resulting assertions by verification component into XC. assertions.
   - Dropping extra function arguments introduced by *xcc* when the HC IR is from LLVM IR (Wrapper function over LLVM IR function).

Example: An XC assertion:

```
#pragma check foo(N) :  (1 <= N) ==> (energy <= 35xN)
```

is translated into the HC IR assertion:

```
:- check comp foo(N) :
            intervals(int(N),[i(1,inf)])+ resource(energy,0,35xN)
```

The CiaoPP Verification/Debugging Framework

- Both program verification and debugging compare the *actual semantics* with the *intended semantics*.
- In CiaoPP, both semantics are (safely) *approximated*:
  - *Actual semantics:* safely approximated by (abstract interpretation-based) *static analyses*.
  - *Intended semantics:* programs include partial specifications (in the form of *assertions*).
- Verification → *compare the analysis approximations with specifications* (to prove them or detect inconsistencies).
- The approximations of the actual and intended semantics are based on *resource usage functions*:

  Monotonic arithmetic functions expressing lower or upper bounds on the resource usage of a procedure depending on input data sizes.

Using the Tool: Example

- **Scenario**: Deciding values for program parameters that meet an energy budget.
- **Example**: Development of an equaliser (XC) program using finite impulse response (FIR) filtering.
  - → The purpose of an equaliser is to take a signal, and to attenuate / amplify different frequency bands.
- The energy consumed directly depends on the number of coefficients, but
- a higher number of coefficients enables more precise frequency response.
- *Objective* → *decide how many coefficients* to use in order to (both):
  - *meet an energy budget*, and
  - *maximize the precision of frequency response curves*.

XC Program (FIR Filter) with Energy Specification

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

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

Demo

Demo

Energy Consumption Verification

Given the "check" assertion (specification):

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

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

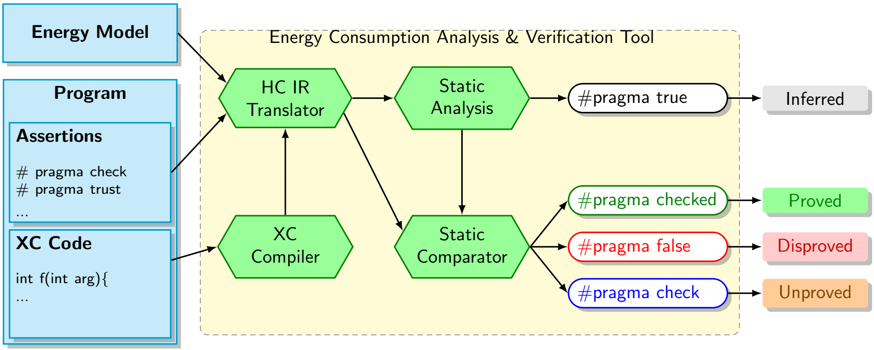
```
#pragma true fir(xn, coeffs, state, N) :
                (3347178*N + 13967829 <= energy &&
                  energy <= 3347178*N + 14417829)
```

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

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

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

Energy Consumption Verification Tool Using CiaoPP



CiaoPP's Modular Analysis

- Instead of analyzing the whole program at the same time, the modular analysis analyzes a part of the program at a time, allowing processing bigger programs.
  - Cost of analysis in terms of memory usage is lower.
  - Module-level incremental analysis.
  - Can deal with parts of the program being in development, just specified; or **Implemented in other languages.**

→ An intermodular resource consumption analysis for programs written in multiple languages.

Multi-language Program Analysis

- Programs written in multiple programming languages is common practice (e.g., foreign interfaces):
  - Utilizing features of a particular language.
  - Building systems from existing components written in different languages.
- Static analysis of such programs not only requires an analysis of the source code but of the foreign code as well.

Modular Multi-language Program Analysis

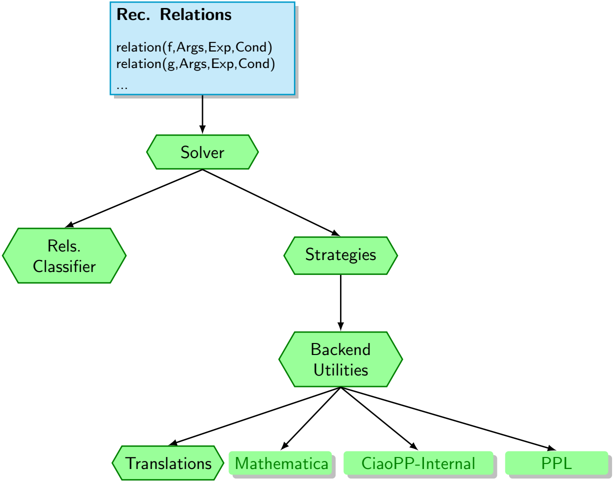
- The foreign interfaces are transformed into *modules/includes* to the source program.
  - Transforming the source code of each foreign module to the *HC IR*, including the resource assertions specifying the cost of each instruction in foreign source code.
- An inter-modular fixpoint is reached over all the modules to infer global resource consumption functions.

Improvements in the Recurrence Relation Solver Component

Improvements in the Recurrence Relation Solver Component

- Refactorization of the recurrence relation solver component:
  - Decoupled back-nend solvers from the analysis (Mathematica, CiaoPP's Internal, PPL).
  - Defined a common expression syntax to combine results from different back-end algebraic systems.
  - Defined different strategies to solve or find bounds of complex recurrence relations (e.g., rec. relations with combinations of incrementing and decrementing arguments).
  - Used techniques from the termination community in order to transform or solve certain kinds of rec. relations (linear ranking functions).

Architecture of the Recurrence Relation Solver



Summary

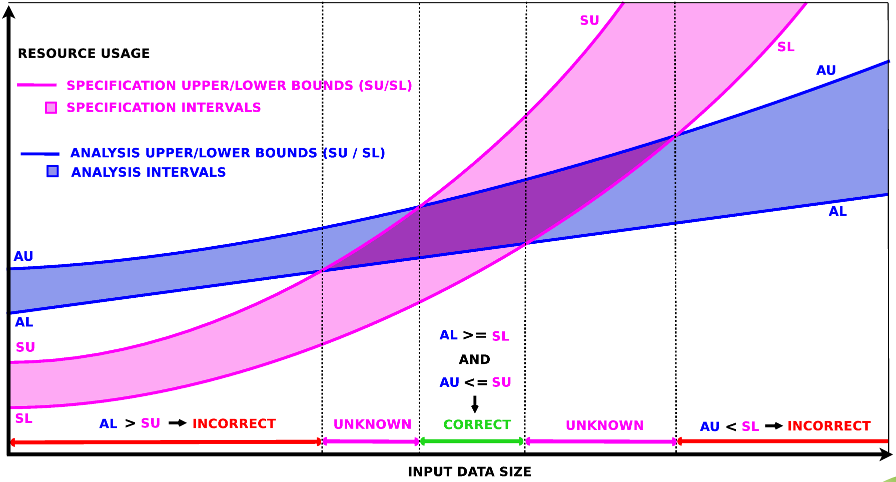
- Specialized existing general framework for resource usage verification in order to verify energy consumption specifications of embedded programs.
- Prototype implementation within the Ciao/CiaoPP system and for the XC language and (XMOS) XS1-L architecture.

- Recent progress on extending the modular analysis of CiaoPP for resource consumption analysis of multi-language applications.

- Recent progress on extending the component for solving cost relations.

Thank you for your attention!

Resource Usage Verification – Function Comparisons (ICLP'10, FOPARA'11)



Resource Verification: Sufficient Conditions

Given a program p, a specification I_α, and an input data size interval S: If $\forall x \in S$,

(1) $\mathbf{AL}(x) \geq \mathbf{SL}(x) \wedge \mathbf{AU}(x) \leq \mathbf{SU}(x) \rightarrow p$ is partially *correct* w.r.t. I_α

(2) $\mathbf{AU}(x) < \mathbf{SL}(x) \vee \mathbf{AL}(x) > \mathbf{SU}(x) \rightarrow p$ is *incorrect* w.r.t. I_α