The W-SEPT project\textsuperscript{1}: Towards Semantic-aware WCET Estimation

C. Maiza\textsuperscript{2}, P. Raymond\textsuperscript{2}, C. Parent-Vigouroux\textsuperscript{2}, A. Bonenfant\textsuperscript{3}, F. Carrier\textsuperscript{2}, H. Cassé\textsuperscript{3}, P. Cuenot\textsuperscript{5}, D. Claraz\textsuperscript{5}, N. Halbwachs\textsuperscript{2}, E. Jahier\textsuperscript{2}, H. Li\textsuperscript{4}, Mi. De Michiel\textsuperscript{3}, V. Musso\textsuperscript{3}, I. Puaut\textsuperscript{4}, C. Rochange\textsuperscript{3}, E. Rohou\textsuperscript{4}, J. Ruiz\textsuperscript{3}, P. Sotin\textsuperscript{3}, W-T. Sun\textsuperscript{3}

WCET Workshop, 2017

\textsuperscript{1}This project was founded by ANR
\textsuperscript{2}Univ. Grenoble Alpes-VERIMAG
\textsuperscript{3}Univ. Toulouse-IRIT
\textsuperscript{4}Univ. Rennes 1/INRIA-IRISA
\textsuperscript{5}Continental
General goal

• Increase the precision of WCET estimation...
  ⇔ by focusing on the influence of the software semantics
  ⇔ i.e., not about hardware modeling!

The project

• W-SEPT = WCET: SEmantics, Precision, Traceability

• Find, trace and exploit semantics information:
  ⇔ Main issue: Express infeasible paths given or automatically discovered and
    preserve them through compilation process
  ⇔ Approach:
    ∗ Use a common format: FFX (Flow fact Format Xml-like)
    ∗ Rely/adapt/extend existing timing analysis tool (OTAWA)
Project overview

Application Domain

Critical Real time (transportation, energy)

Hard real time hardware constrained (automotive)

High Level Design

Program analysis

High Level Design (Scade, Simulink etc.)

Annotation Languages and traceability
(compiler optim. aware)

User provided properties

Annotation aware Path Search (IPET or other)

Introduction
Partners: skills/complementarity

- IRIT Toulouse: C/binary level and WCET computation
- Verimag Grenoble: high-level/design, semantic analysis
- Inria/Irisa Rennes: compilation and binary level
- Continental Toulouse: industrial application - Engine Management System (EMS)

This talk

- Focus on 3 topics/experiments
  - Exploiting High Level Properties
  - Tracing flow information through compiler optimization
  - Expressing and exploiting path properties
Exploiting High Level Properties

Model-based design

- Particularly in safety critical domains (Scade/Lustre)
- But not only (Simulink/Stateflow)
- Compilation process: HL to C to bin

Consequences for timing analysis

- Semantic static analysis exist at HL
- HL properties may have strong influence on WCET
- HL properties are “hard” to discover at lower level

Experiment with Lustre

- Representative: Lustre \sim Scade (avionics)
- ... and not so different from Simulink
- What is important: *synchronous paradigm*, i.e., execution = infinite loop, each iteration performs an atomic reaction
Synchronous Programming Workflow

- Design level:
  - Concurrent, Hierarchic design
  - Idealized Concurrency
  - Behavior =
    sequence of reactions
    logical discrete time
  - Several styles/languages
    Here: data-flow/Lustre
Synchronous Programming Workflow

- Synchronous Compiler
  - Target language = C
  - Generates the step procedure (+ the necessary memory/ctx)
  - Basically: no more concurrency (static scheduling)
  - Simple sequential code

```c
#include<...>
struct modes_ctx{
    ...
}
void modes_step{
    ...
    ...
    if (L5){
        ...
    }
    if (L5){
        ...
    } else {
        ...
    }
}
```
Synchronous Programming Workflow

- Example of main code
  - Basically an infinite loop
  - Each loop performs one reaction
  - Depends on system choices periodic/event-driven etc.

```c
#include<...>
#include "modes.h"

void main (){
    while(1){
        wait_period();
        read_inputs();
        modes_step();
        write_outputs();
    }
}
```
Binary code
- via arm-elf-gcc
- WCET estimation should be done here
  for modes_step
  i.e. a step of main infinite loop
High Level Properties (that may help)

- Programming pattern: computation modes, based on clock-enable construct
- Intra-module exclusions: between A₀, A₁, A₂, and between B₀ and B₁
  ➔ may or may not be obvious on the code (i.e. structural)
- Inter-module exclusions: not in mode A₀ implies mode B₁
  ➔ no chance to be obvious on the code
- In all cases, relatively complex properties:
  ➔ infinite loop invariants
  ➔ unlikely to be discovered by analysing the C or bin code of one step
High Level Properties (that may help)

- Programming pattern: computation modes, based on clock-enable construct
- Intra-module exclusions: between A0, A1, A2, and between B0 and B1
  \(\rightarrow\) may or may not be obvious on the code (i.e. structural)
- Inter-module exclusions: not in mode A0 implies mode B1
  \(\rightarrow\) no chance to be obvious on the code
- In all cases, relatively complex properties:
  \(\rightarrow\) infinite loop invariants
  \(\rightarrow\) unlikely to be discovered by analysing the C or bin code of one step
- Can be discovered using, e.g., model-checking techniques
  (here, Lesar = Lustre Model-Checker)
Traceability: form HL property to binary (ILP) constraint

HL Compil (Lustre → C)

Code generation (gcc)

bin CFG

Exploiting High Level Properties
Traceability: form HL property to binary (ILP) constraint

- Relate HL var to C var (compiler patch)

HL Compil (Lustre → C)

C CFG

Code generation (gcc)

bin CFG

Exploiting High Level Properties
Traceability: form HL property to binary (ILP) constraint

- Relate HL var to C var (compiler patch)
- C branches to bin branches?
  - simple heuristic: rely on debugging info

HL Compil (Lustre → C)

Code generation (gcc)

Bin CFG

Exploiting High Level Properties
Traceability: form HL property to binary (ILP) constraint

- Relate HL var to C var (compiler patch)
- C branches to bin branches?
  - simple heuristic: rely on debugging info
- No optimization (−O0)
  - CFG’s strictly match
    - e.g. \( \neg (\text{high} \land \text{nom}) \) becomes in ILP:
      \[ \text{edge}_{29,30} + \text{edge}_{49,50} \leq 1 \]
Traceability: form HL property to binary (ILP) constraint

- Relate HL var to C var (compiler patch)
- Optimization (−O2)
  
- C CFG obfuscated, but still works

- No optimization (−O0)
  CFG’s strictly match
  
  e.g. \( \neg (\text{high} \land \text{nom}) \) becomes in ILP:
  \[
  \text{edge}_{29,30} + \text{edge}_{49,50} \leq 1
  \]

- Optimization (−O2)
  CFG obfuscated, but still works

**HL Compil (Lustre → C)**

- C CFG
- Code generation (gcc)
- bin CFG
Traceability: form HL property to binary (ILP) constraint

- Relate HL var to C var (compiler patch)
- C branches to bin branches ?
- simple heuristic: rely on debugging info
- No optimization (-O0)
  CFG's strictly match
  e.g. \( \neg (\text{high} \land \text{nom}) \) becomes in ILP:
  \[ \text{edge}_{29,30} + \text{edge}_{49,50} \leq 1 \]
- Optimization (-O2)
  CFG obfuscated, but still works
  e.g. \( \neg (\text{high} \land \text{nom}) \) becomes in ILP:
  \[ \text{edge}_{15,14} + \text{edge}_{13,21} \leq 1 \]
  \[ \text{edge}_{11,14} + \text{edge}_{13,21} \leq 1 \]
High Level properties, conclusion

• Fully automatic proof of concept

• Implements 2 strategies:
  ⇝ **Iterative**: computes a WCET candidate, try to refute it with HL model checking,
    and so on until WCET candidate cannot be refuted.
    * reaches a (relative) best solution, but converges very slowly
  ⇝ **Pairwise a priori**: check, once for all, any possible pairwise relation between “well
    chosen” HL variables
    * e.g. clocks are clearly good candidates
    * quadratic number of relations to check, but single WCET analysis

• Experiments: with both strategy, the gain is about 40%, pairwise strategy runs much
  faster (few seconds vs few minutes).

Problem

- Infeasible path properties are generally discovered/given at C level
- Relate infeasible C path to infeasible binary path?
- Radical solution: No optimization: perfect match, no problem...
  But the code is likely to be rather inefficient!
- Impact of optimization on WCET estimation, for 12 classical benchmarks:
  \[ \text{"-O1 code" WCET as a \% of "-O0 code" WCET} \]
Allow optimization in WCET estimation?

- Rely on existing compiler tracing facilities (e.g. dwarf)
  - Accept to lose some properties (cf. previous topic)
- Allow optimization that do not (or slightly) impact the CFG
  - not so bad: data optim. largely speedup code in general
- Modify/adapt compilers to make them trace-property aware.
  - Probably the most satisfactory ...
  - .. but requires a lot of work
  - Not suitable when off-the-shelf, black-box compilers are required

The project approach

- Study the “path-aware” compiler approach
- Experiment/proof-of-concept based on the LLVM compilation platform
General idea

- Flow informations = IPET-like constraints
- CFG transformation = constraint rewriting
  → possible loss in precision
- Example: loop bounds and loop unrolling
  → \( #A \leq X_{\text{max}} \)
  → Becomes:
    * \( #A' \leq X_{\text{max}} / k \)
    * and \( #A'' \leq k - 1 \)
- Proof of concept for ~ 10 classical optim.
- Results for a Lustre program, with and without infeasible path search and tracing:

<table>
<thead>
<tr>
<th>Analysis &amp; tracing</th>
<th>optim. level (-00)</th>
<th>optim. level (-01)</th>
<th>optim. level (-02)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>2896 (100%)</td>
<td>1523 (52.5%)</td>
<td>1542 (53.2%)</td>
</tr>
<tr>
<td>On</td>
<td>2014 (69.5%)</td>
<td>997 (34.4%)</td>
<td>998 (34.5%)</td>
</tr>
</tbody>
</table>

Li H., Puaut I. and Rohou E.

*Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation.* RTNS’14
Expressing and exploiting path properties

Introduction

• How to tell to the WCET analyser that some paths are infeasible?

• Basically two kinds of methods:
  - Make infeasibility explicit, via CFG transformation:
    * can (virtually) handle any property ...
    * ... but beware of graph size explosion!
  - keep infeasibility implicit, via additional IPET constraint
    * “ideally” compact (in fact, complexity is transferred to ILP solver)
    * ... but possible loss in precision
Expressing and exploiting path properties

Introduction

- How to tell to the WCET analyser that some paths are infeasible?
- Basically two kinds of methods:
  - Make infeasibility explicit, via CFG transformation:
    - can (virtually) handle any property ...
    - ... but beware of graph size explosion!
  - Keep infeasibility implicit, via additional IPET constraint
    - “ideally” compact (in fact, complexity is transferred to ILP solver)
    - ... but possible loss in precision
- Or maybe a mix of both?
The project approach

- Design a versatile formalism, mixing explicit and implicit features
- PPA (Path Property Automata):
  - Inherits from formal language theory:
    - a CFG (program) $\iff$ a language whose words are the executions
    - a property $\iff$ an automaton recognizing feasible paths
    - removing infeasible path $\iff$ intersecting the CFG and the property
    - use hierarchic automata (rather than flat ones) for concision
• Program CFG, an execution = a word over alphabet \{E, A, B, X\}
Example (explicit product)

- Program CFG, an execution = a word over alphabet \{E,A,B,X\}
- Informal property: A and B exclusive at each iteration
Example (explicit product)

- Program CFG, an execution = a word over alphabet \{E,A,B,X\}
- Informal property: A and B exclusive at each iteration
- Formalized as a language recognizer (PPA syntax)
Example (explicit product)

- Program CFG, an execution = a word over alphabet \{E,A,B,X\}
- Informal property: A and B exclusive at each iteration
- Formalized as a language recognizer (PPA syntax)
- CFG $\times$ PPA product $\Leftrightarrow$ language intersection
Example (explicit product)

- Program CFG, an execution = a word over alphabet \{E,A,B,X\}
- Informal property: A and B exclusive at each iteration
- Formalized as a language recognizer (PPA syntax)
- CFG × PPA product ⇔ language intersection
- Explicit approach: beware of graph size explosion!

Expressing and exploiting path properties
Example (mixed explicit/implicit product)

- Program CFG, an execution = a word over alphabet \{E, A, B, X\}
- Informal property: A and B exclusive at each iteration
Example (mixed explicit/implicit product)

- Program CFG, an execution = a word over alphabet \{E,A,B,X\}
- Informal property: A and B exclusive at each iteration
- Language recognizer with local counters and constraints (PPA syntax)
Example (mixed explicit/implicit product)

- Program CFG, an execution = a word over alphabet \( \{E,A,B,X\} \)
- Informal property: A and B exclusive at each iteration
- Language recognizer with local counters and constraints (PPA syntax)
- Extended counter-aware product \( \Leftrightarrow \) CFG + ILP constraints
Example (mixed explicit/implicit product)

- Program CFG, an execution = a word over alphabet \{E,A,B,X\}
- Informal property: A and B exclusive at each iteration
- Language recognizer **with local counters and constraints** (PPA syntax)
- Extended counter-aware product $\Leftrightarrow$ CFG + ILP constraints
- Mixed explicit/implicit approach

* Mussot V. and Sotin P.
* *Improving WCET Analysis Precision through Automata Product*. **RTCSA, 2015.**
Conclusion

- Other topics studied/started during the project:
  - Semantic analysis at binary level
  - Limits of IPET/ILP methods
  - Beyond ILP: semantic + timing analysis as a whole
  - Targeting “costly” part of program (branch deltas)
  - User-guided analysis
  - etc. see http://wsept.inria.fr

- General result: a semantic-aware WCET workflow

- Raised interest from industrial partner
Conclusion

- Other topics studied/started during the project:
  - Semantic analysis at binary level
  - Limits of IPET/ILP methods
  - Beyond ILP: semantic + timing analysis as a whole
  - Targeting “costly” part of program (branch deltas)
  - User-guided analysis
  - etc. see http://wsept.inria.fr

- General result: a semantic-aware WCET workflow

- Raised interest from industrial partner

Thanks for your attention!

Questions?