Worst Case Execution Time Estimation

Pascal Raymond

Verimag-CNRS

MOSIG - Embedded Systems

Introduction

Program correction

- A reactive system is correct if:
  - ↪ it computes the right outputs (functionality)
  - ↪ it reacts fast enough (real-time)
- Synchronous approach addresses mainly the 1st problem (functionality)
  while guarantying that the 2nd will be solvable

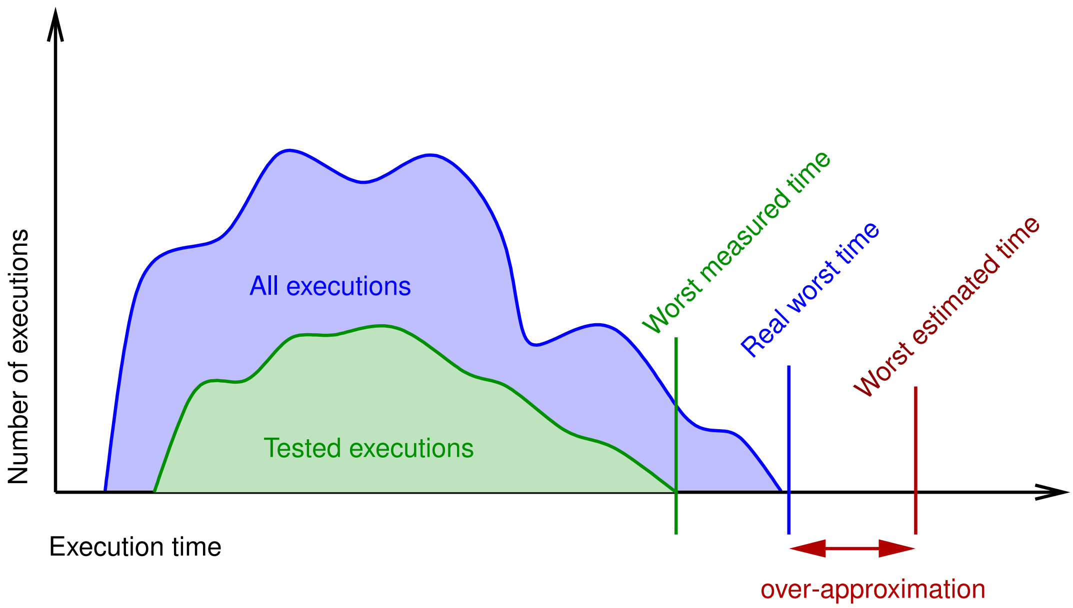
Goal of this course

- Brief state of the art in timing analysis
  focusing in particular on the influence of software (semantics)
  .., less on the hardware influence.
- Focus on the particular case of Synchronous Programs,
  trying to *exploit* their specificities

Timing analysis

- The whole reaction of the program must respect the real-time constraint i.e. must be faster than any significant modification of the environment
- A reaction includes not only computation but also:
  - ↪ inputs acquisition and outputs transfer,
  - ↪ depends on physical and electronic devices (sensors, actuators, buses ...)
  - ↪ The full problem is called: *Worst Case Reaction Time estimation* (WCRT)
- Moreover, computation may not be sequential:
  - ↪ multi thread implementation, on single or multi core
  - ↪ The general problem is referred as *Schedulability Analysis*
- However, there is (mandatory) basic problem:
  - ↪ Estimate the Worst Case Execution Time (WCET) of a (piece of) purely sequential code, running on a particular hardware architecture

Execution Time Distribution



- Dynamic methods (test) give realistic, feasible exec. times , but are not safe
- Static methods (WCET analysis) give guaranteed upper bound to exec. time, but necessarily over estimated

State of the Art WCET analyser

- Requires:
  - ↪ the binary code,
  - ↪ a (precise) model of the hardware (processor, memory)
  - ↪ the result is given in cpu cycles

Note on the (good) old times

- until the mid 90's , processors where intrinsically *time predictable*:
  - ↪ e.g. Motorola 68000:
    WCET(instr1 ; instr2) = WCET(instr1) + WCET(instr2)
- Nowadays: false even for very "simple" architecture:
  - ↪ memory caches, (micro)-instruction pipeline, or even branch prediction make Exec Time depending on the precise state of the architecture
  - ↪ WCET(HWS, instr1 ; instr2) = WCET(HWS, instr1) + WCET(HWS', instr2) where HWS' = Post(HSW, instr1)

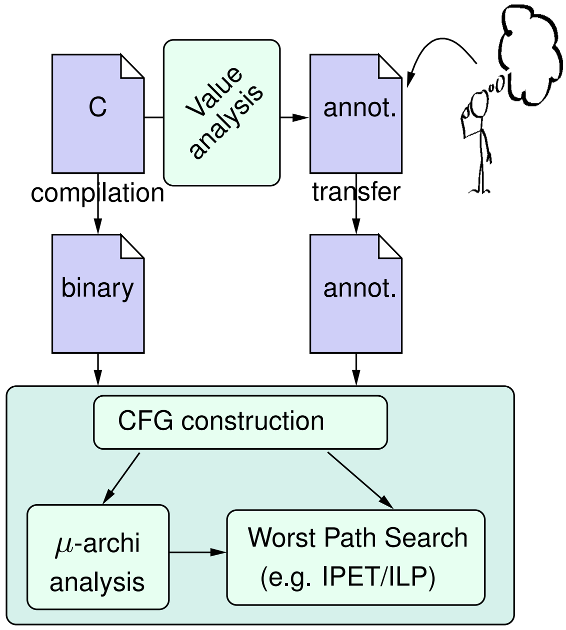
Main sources of over-approximation

- Hardware:
  - ↪ precise modeling of hardware state is impossible in practice
  - ↪ abstractions (simplifications) are necessary
  - ↪ these abstractions MUST be pessimistic, in order to get a safe upper bound
- But also Software:
  - ↪ Some execution of the code are infeasible, because of the program semantics (and/or also some assumptions we have on the inputs)
  - ↪ Considering infeasible executions may lead to a false WCET
- Here: we will focus mainly on software (semantics)

Classical WCET tool organization

Micro-architecture analysis

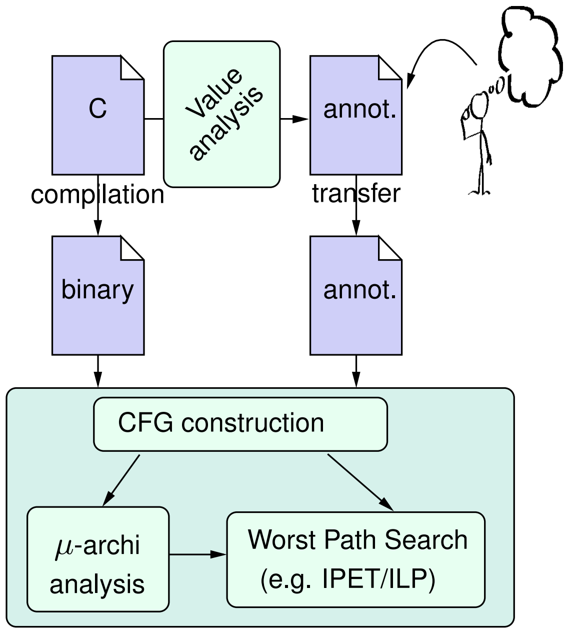
- Control Flow Graph (CFG) construction
  - ↪ Basic Blocks of sequential instructions (one entry, one exit)
  - ↪ Connected by edges (contol flow)
- Assign a local WCET to each BB/edge requires model of the processor/hardware
  - ↪ instruction specification
  - ↪ hardware state (pipeline)
  - ↪ flow history (caches) etc.
  - ↪ N.B. given in cpu cycles



Classical WCET tool organization

Value analysis

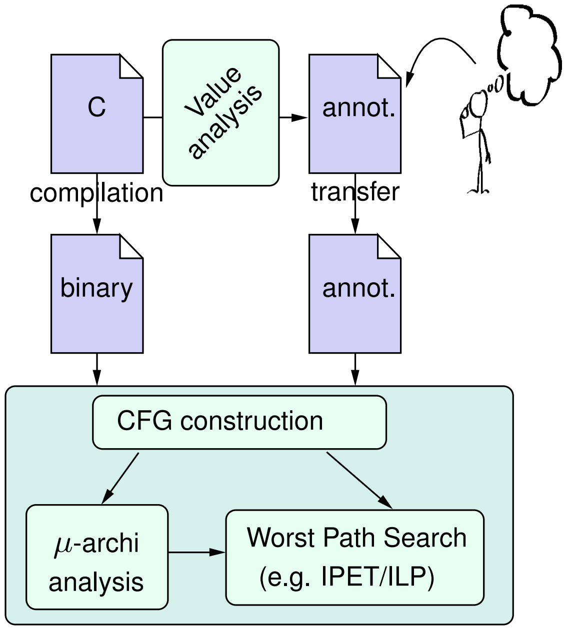
- i.e. Data-Flow Analysis
- focus on program semantics: which execution paths are feasible ?
- Must at least provide loop bounds
- In general performed at source level (C):
  - ↪ May take into account user informations (e.g. input ranges, input exclusions etc.)
  - ↪ Raise a transfer problem between C and bin (traceability)
  - ↪ Strongly depends on the compilation



Classical WCET tool organization

Path analysis

- Search Worst Execution Path (WEP) in the CFG according to:
  - ↪ Local weights provided by μ-archi analysis
  - ↪ Flow facts provided by Value analysis
- Algorithms: graph traversal possible...
- Most widely used:
  Implicit Path Enumeration Technique (IPET)
  - ↪ Encode the WP as an optimization problem: an Integer Linear Program (IPL)
  - ↪ Let's focus on this topic...

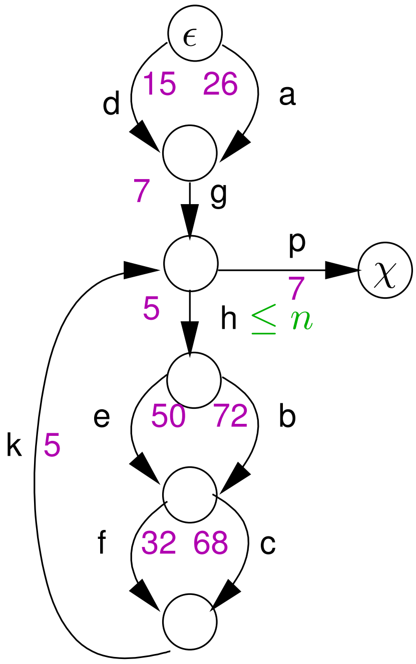


Implicit Path Enumeration Technique

Integer Linear Programming

- LP (Linear Programming) is a branch of Operational Research field
- Input:
  - ↪ a set of linear constraints over rational variables, i.e. $AX \leq B$
  - ↪ a linear objective function to maximize (or minimize), i.e. MAX $f(X)$
- Output:
  - ↪ an optimal valuation $\vec{v}$, such that $A\vec{v} \leq B$ and $f(\vec{v})$ is maximal (resp. minimal)
- State of the art (family of) algorithm: the *simplex*
- ILP is similar, but variables are integers
  - ↪ Theoretically strictly more complex
  - ↪ However works well in many cases

ILP encoding on an example



- μ-archi analysis has assigned weights e.g. $w_a = 26$, $w_b = 72$ etc.
- data-flow analysis has found loop bounds 'h' taken at most $n = 10$ times
- ILP encoding:
  - $\hookrightarrow$ Structural constraints
    $a + d = 1$
    $g = a + d$
    $g + k = p + h$
    $h = e + b$
    $e + b = f + c$
    $f + c = k$
    $p = 1$
  - $\hookrightarrow$ Semantic constraints
    $h \leq n = 10$
  - $\hookrightarrow$ Objective function: MAX$(\sum_{x \in \mathcal{E}} w_x x)$
    MAX$(\sum_{x \in \mathcal{E}} w_x x)$

    Optimal for: $a = g = p = 1, h = b = c = k = 10, d = e = f = 0$
    with: $26 + 7 + 7 + 10 * (5 + 72 + 68 + 5) = 1540$

Interest of ILP

- It handles "naturally" the problem of loops ...
- however, a "simple" graph-based traversal algorithm can do the same !

A simple graph-based algo

- Trivial for well-nested loops (MAX/PLUS),
- Less trivial otherwise, but possible.
- Well-nested program: $prg ::= e \mid prg; prg \mid prg + prg \mid (prg)^n$
- Algo:

$$\begin{aligned}
\mathcal{W}(e) &= w_e \\
\mathcal{W}(p1; p2) &= \mathcal{W}(p1) + \mathcal{W}(p2) \\
\mathcal{W}(p1 + p2) &= \text{MAX}(\mathcal{W}(p1), \mathcal{W}(p2)) \\
\mathcal{W}(p^n) &= n * \mathcal{W}(p)
\end{aligned}$$

Adding extra constraints

- ILP becomes (really) useful when *extra constraints* can be added, that reflect *known properties* on feasible paths
- Example (C-code for simplicity):

```
if (init) { /*a:26*/ }
else { /*d:15*/ }
/*g:7*/
for(i=0;i<n;i++){
  /*h:5*/
  if (i < n/2) {
    /* b:72*/
    cond = false;
  } else {
    /*e:50*/
  }
  if (cond){
    /*c:68*/
  } else {
    /*f:32*/
  }
  /*k:5*/
}
/*p:7*/
```

- branch b cannot be taken more than $n/2$ times:
  $\hookrightarrow$ easy to express in ILP: $b \leq n/2$, i.e. $b \leq 5$
- if b is taken, c cannot be taken
  $\hookrightarrow$ less obvious, but: $b + c \leq n$, i.e. $b + c \leq 10$
- ILP system + extra constraint reach optimal solution for:
  $\hookrightarrow$ $a = g = p = 1, d = 0, h = k = 10,$
  $b = c = e = f = 5$
  $\hookrightarrow$ $26+7+7+10*5+5*(72+50+68+32) = 1200$
  $\hookrightarrow$ enhancement (from 1540): 22%

Infeasibility properties: many problems...

- May or may not enhance the WCET estimate
  $\hookrightarrow$ does they concern “heavy” or “light” paths ?
- How to find them ?
- Is it possible and how to express them in ILP ?

Find infeasible path

- Hard problem, c.f. program analysis (NP-hard/even undecidable)
- Target (as far as possible) “heavy” paths
- Restrict to some patterns, e.g. pairwise condition exclusion

Express infeasibility in ILP (examples)

```
if (init) {
  /* a */
} else {
  /* d */
}
for(i=0;i<n;i++){
   if (Y[i]) {
    cond = not init
           and Z[i];
     /* b */
   } else {
    cond = true;
     /* e */
   }
  /* ... */
   if (cond){
     /* c */
   } else {
     /* f */
   }
}
```

- at each iteration, if e is taken, f cannot be taken:
  - $\hookrightarrow$ similar to the previous example: $e + f \leq n$
- More subtle: if a is taken, then at each iteration, if b is taken, then c cannot be taken
  - $\hookrightarrow$ less obvious, but: $n \cdot a + b + c \leq 2n$, works
  - $\hookrightarrow$ suppose a is NOT taken, then $a = 0$ and the constraint becomes:
    $b + c \leq 2n$ which is trivially satisfied
  - $\hookrightarrow$ suppose a is taken,then $a = 1$ and the constraint becomes:
    $b + c \leq n$ which express the exclusion

Express infeasibility in ILP (examples)

```
for(i=0;i < n;i++){
   if (X[i]) {
      /*a*/
      cond = false;
   }
   for(j=0;j < m;j++){
      if (cond){
        /*b*/
      }
   }
}
```

- conflict between a and b: each time a is taken ... b is forbidden all along the forthcoming “m”-loop
  - $\hookrightarrow$ how to express in ILP ?
    $m * a + b \leq n * m$

```
cond = read();
for(i=0;i < n;i++){
   if (cond) {
      /*a*/
   }
   if (Y[i]) {
      /*b*/
      cond = false;
   } else {
      cond = X[i];
   }
}
```

- conflict across iteration: if b is taken, a cannot be taken in the next loop
  - $\hookrightarrow$ how to express in ILP ?
    $a + b \leq n + 1$

WCET and synchronous programming

Complementarity

- Synchronous approach guarantees that programs are intrinsically real-time
  ↪ execution time is bounded by construction,
  for any particular implementation on any particular architecture
- WCET estimation checks that the program implementation is actually real-time
  ↪ tries to compute accurate and precise bound for the actual implementation
  ↪ checks whether this bound is small enough to fulfill the real-time requirements

Synchronous program vs micro-architecture analysis

Micro-architecture analysis simple (and hopefully precise):

- no recursion, no dynamic allocation:
  ↪ no heap, no (or very simple) stack...
  ↪ makes memory access analysis simple (e.g. cache analysis)
- no (or very simple) loops, simple control structure (nested if-then-else):
  ↪ makes control analysis simple (e.g. pipeline, branch prediction)

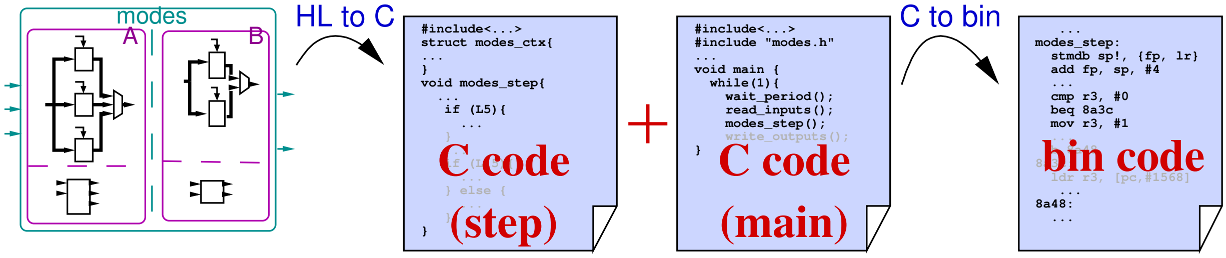
Synchronous program vs data analysis

- The simplest is the code, the simplest (and precise) is the analysis
- Features that make data (semantics) analysis difficult are absent:
  ↪ no aliasing (pointers)
  ↪ no complex loops (while)

Go further?

- A synchronous program has a global “infinite” behavior:
  ↪ Explicit at the high-level (Lustre, Esterel)
  ↪ Hard to (re)-discover at the step procedure level (C, binary)
  ↪ Is it pssible to exploit global properties of S.P. to enhance WCET estimation ?
  ↪ Indeed: it strongly depends on the compilation scheme:
    * high-level properties may or may not have influence on the generated code!
- Let see a typical example ...

Synchronous Program Example: compilation



- Binary code
  - ↪ via arm-elf-gcc
  - ↪ WCET estimation should be done here

  for **modes_step**
  i.e. a step of main infinite loop

```
  ...
modes_step:
   stmdb sp!, {fp, lr}
   add fp, sp, #4
   ...
   cmp r3, #0
   beq 8a3c
   mov r3, #1
   ...
   b 8a48
8a3c:
   ldr r3, [pc,#1568]
   ...
8a48:
   ...
```


Example (cntd): WCET estimation

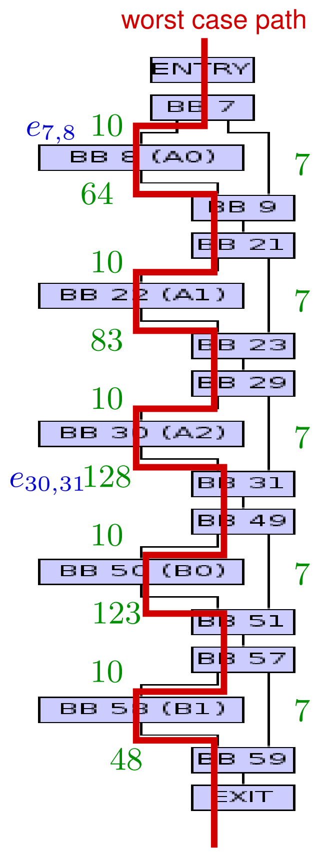
- Works at binary level
- Control Flow Graph (CFG) reconstruction
  - ↪ Basic Blocks + edges (small part here)
- μ-archi analysis
  - ↪ local costs, $c_{i,j}$, in cpu cycles
- *Data-flow analysis*
  - ↪ *loop bounds + others (not here)*
- Implicit Path Enumeration Technique (IPET)

  Integer Linear Programming encoding
  - ↪ one counter variable per edge ($e_{i,j}$)

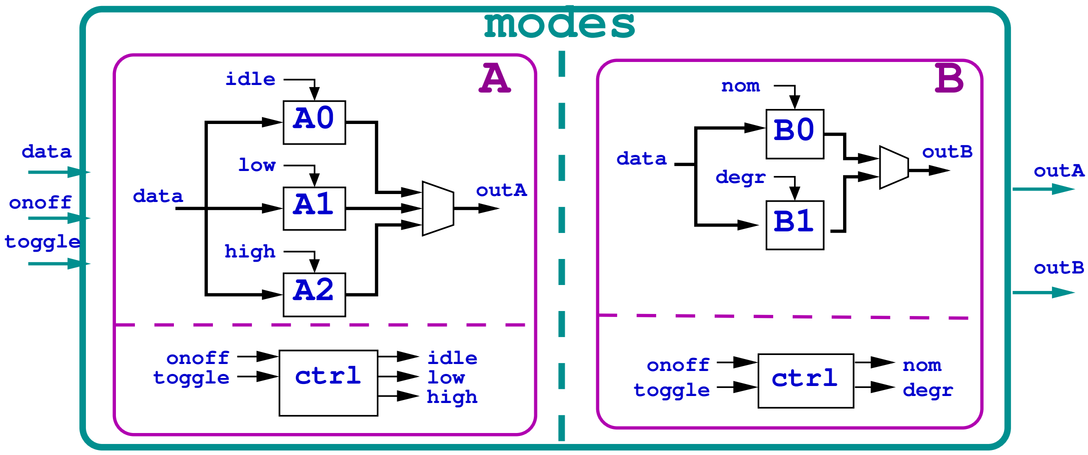
    (n.b. here, $e_{i,j}$ = 0 or 1)
  - ↪ Structural Constraints: $\Sigma e_{i,j} = \Sigma e_{j,k}$

    (and indeed: entry = exit = 1)
  - ↪ Semantics Constraints

    loop bounds (not here), others ?
  - ↪ Objective: MAX $\Sigma c_{i,j} \times e_{i,j}$
- Call an ILP Solver (here LPSolve)
  - ↪ get 496 + the left-most path



Example (cntd): High Level properties (that may help estimation)



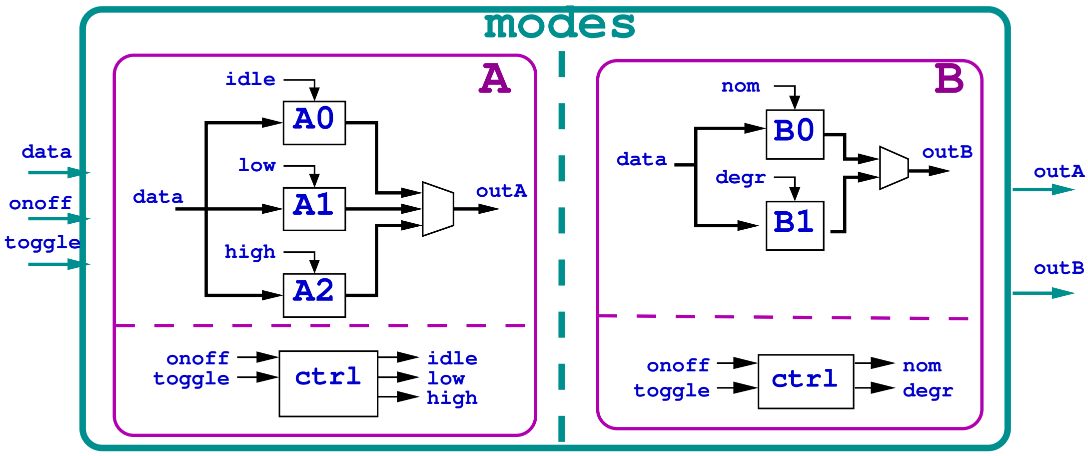
- Typical embedded application: several sub-modules running (logically) in parallel
- Programming pattern: computation modes

$\hookrightarrow$ Implemented with the notion of "clock-enabled" (e.g. **when**/**current** in Lustre)

- Compiler correct $\Rightarrow$ codes of the modes *must be exclusive*

$\hookrightarrow$ Interesting property for enhancing WCET

Example (cntd): High Level properties (that may help estimation)



- Intra-module exclusions: between **A0**, **A1**, **A2**, and between **B0** and **B1**

$\hookrightarrow$ may or may not be "obvious" on the generated code (i.e. structural)

- Inter-module exclusions: not in mode **A0** implies mode **B1**

$\hookrightarrow$ no chance to be obvious on the generated code

- In all cases, relatively "complex" properties:

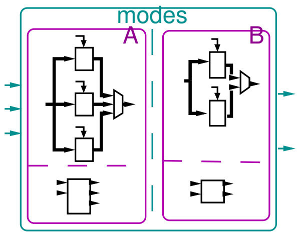
$\hookrightarrow$ infinite loop invariants, unlikely to be discovered by analysing C or bin code

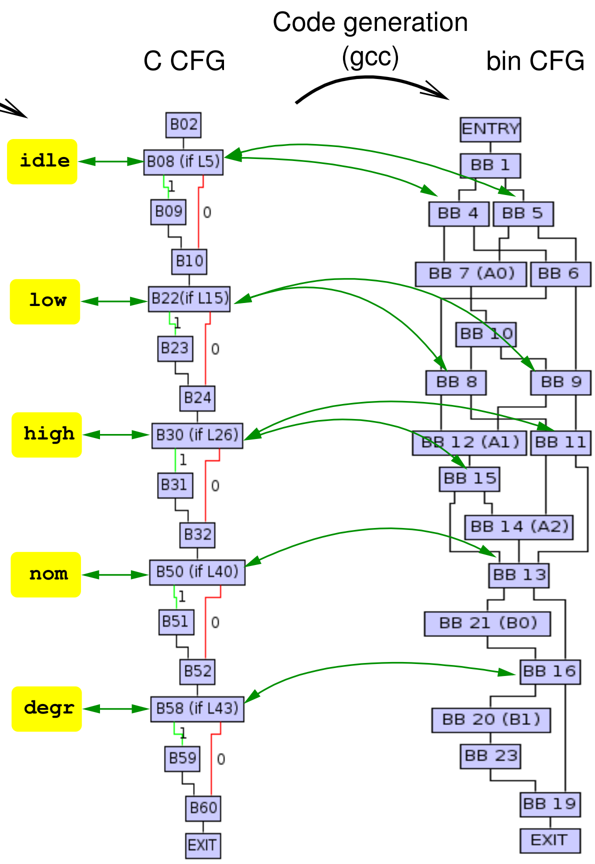
Exploiting high-level properties

Several problems:

- How to relate HL properties and binary code ? (traceability)
- How to express properties in the (classical) IPET/ILP method ?
- How to *automatically* find the “interesting” properties ?

Traceability



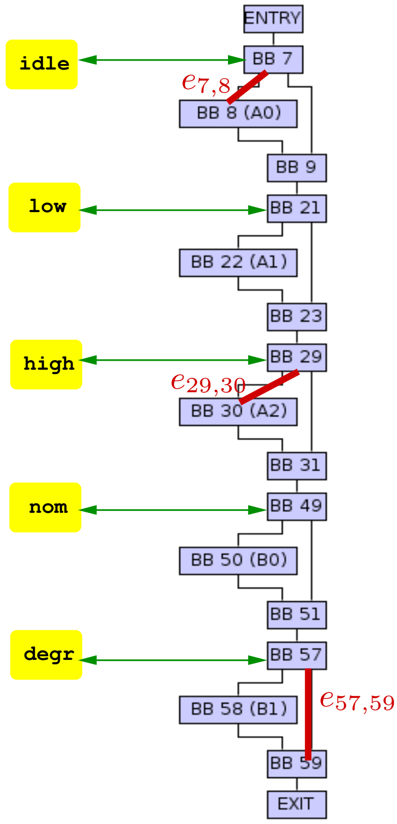


- problem: relate branches in bin CFG to branches in C-CFG, and then predicates (variables) at the HL level
- between HL and C: not a problem (compiler annotations)
- between C and bin: more difficult (simple heuristic: rely on debugging info)
  - ↪ No optimization (−O0) CFG’s strictly match
  - ↪ Optimization (−O2) CFG relatively obfuscated but debug info still works

HL Properties vs ILP constraints

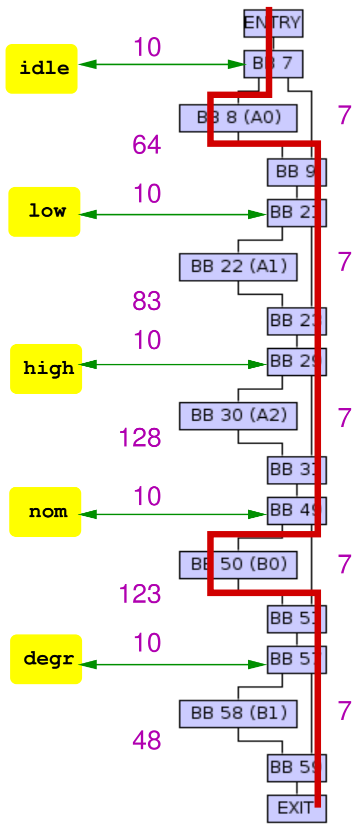
- Traceability has been achieved
  - ↪ Some binary edges are associated to HL variables
  - ↪ N.B. Same HL variable nay control several bin edges (not here)
- Feasibility of binary paths ?
  e.g. $e_{7,8}$ & $e_{29,30}$ & $e_{57,59}$
- Feasibility as HL predicate:
  $\Phi = (\texttt{idle} \wedge \texttt{high} \wedge \neg\texttt{degr})$
- Ask some HL verification tool:
  Is $\neg\Phi$ an invariant of the HL program ?
  (here: Lesar = Lustre model-checker)
  - ↪ Not proven, some path may be feasible...
  - ↪ Proven. Infeasibility as ILP constraint:

$$e_{7,8} + e_{29,30} + e_{57,59} < 3$$



Putting it all together: an iterative algorithm

- Call IPET/ILP solver
  - ↪ Find worst case path (496 cycles)
- Is this path infeasible ?
  - ↪ Call model-checker to prove:
    $\neg(\texttt{idle} \wedge \texttt{low} \wedge \texttt{high} \wedge \texttt{nom} \wedge \texttt{degr})$
  - ↪ Result is “TRUE PROPERTY”, thus infeasible
  - ↪ Add the corresponding ILP constraint:
    $e_{7,8} + e_{21,22} + e_{29,30} + e_{50,51} + e_{58,59} \leq 4$
- Call IPET/ILP solver
  - ↪ Find worst case path (455 cycles)
  - ↪ Check infeasibility, ... YES, and so on
- Eventually reach the WORST (feasible) path:
  - ↪ reached for $\texttt{idle} \wedge \texttt{nom}$ (258 cycles)
- Likely to VERY inefficient: converge VERY slowly
  - ↪ 16 iterations for this simple example ...



An alternative top-down algorithm

- Identify in the HL code the variables that are likely to influence the WCET
  - ↪ Simple heuristics: those that are associated to bin edges,
  - ↪ Here clearly: `idle`, `low`, `high`, `nom`, `degr`
- Try to find *a priori*, exclusive relations between these variables
  - ↪ Warning: there are a combinatorial number of such relations!
  - ↪ Heuristics: limit the search to pairwise relations,
    - * e.g. is $\neg(\texttt{idle} \wedge \texttt{low}) = (\neg\texttt{idle} \vee \neg\texttt{low})$ an invariant ?
    - * e.g. is $\neg(\texttt{idle} \wedge \neg\texttt{low}) = (\neg\texttt{idle} \vee \texttt{low})$ an invariant ?
    - * etc. there are $2 * C_5^2 = 20$ such potential relations to check
  - ↪ seems a lot, but polynomial: quadratic: $C_n^2 = n(n-1)/2$

An alternative top-down algorithm (cnt'd)

- Example: checks the $2 * C_5^2 = 20$ pairwise disjunctive relations
- six of them are proved invariant:
  $\neg\texttt{idle} \vee \neg\texttt{low}$ and $\neg\texttt{idle} \vee \neg\texttt{high}$ and $\neg\texttt{low} \vee \neg\texttt{high}$ and
  $\neg\texttt{nom} \vee \neg\texttt{degr}$ and $\neg\texttt{low} \vee \neg\texttt{nom}$ and $\neg\texttt{high} \vee \neg\texttt{nom}$
- that are translated into 6 ILP constraints (N.B. it can be more in general):
  $e_{7,8} + e_{21,22} \leq 1$ and $e_{7,8} + e_{29,30} \leq 1$ and $e_{21,22} + e_{29,30} \leq 1$ and
  $e_{49,50} + e_{57,58} \leq 1$ and $e_{21,22} + e_{49,50} \leq 1$ and $e_{29,30} + e_{49,50} \leq 1$
- Call IPET/ILP solver once: get the optimal solution (258 cycles)
- Remarks:
  - ↪ Checking relations is costly: heuristics ! (choice of variables, restriction to pairwise)
  - ↪ The obtained solution is not guaranteed to be optimal:
    a path can be infeasible because of more than 2 variables
- However: this algo is empirically (and relatively) efficient