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, according two topics:
  - hardware analysis (overview, deserve a whole course !)
  - software analysis (feasibility)

- Then 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
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

WCET estimation: overview

The timing analysis problem

• given a binary code,
• and a (more or less) precise model of the hardware (processor, memory)
• found an upper bound of its execution time (given in cpu cycles)

The “right” structure to start with: Control Flow Graphs (CFG)

• Identify Basic Blocks (BB):
  ↩ purely sequential piece of code
• Represent the control flow with transitions connecting the BB
Example of Control Flow Graph

```assembly
foo:
    str fp, [sp, #-4]!
    add fp, sp, #0
    ...  
    cmp r3, #0
    beq .L2
    mov r3, #0
    str r3, [fp, #-4]
    b .L3
.L2:
    ldr r3, [fp, #-20]
    ldr r3, [r3, #0]
    str r3, [fp, #-4]
    mov r3, #0
    str r3, [fp, #-12]
    bne .L9
    ldr r3, [fp, #-24]
    mov r2, r3, lsr #31
    cmp r2, r3
    ...  
    ldr r3, [fp, #-4]
    mov r0, r3
    add sp, fp, #0
    b .L3
.L9:
    ldmfd sp!, fp
    bx lr
```

Problems to solve:
- Assign (local) WCET to each BB ...
- ... and penalties to transitions (jump vs sequence)
- Find loop bounds (B8 to B4 !)
- Find the Worst Case execution path

WCET estimation: overview

Classical WCET tool organization

Micro-architecture analysis
- Control Flow Graph (CFG) construction
  - Basic Blocks of sequential instructions (one entry, one exit)
  - Connected by edges (control 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

Path analysis

• Search Worst Execution Path (WEP) in the CFG
  according to:
  − Local weights provided by $\mu$-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)
Micro-Architecture Analysis

Goal

- find an upper bound to the execution time of a Basic Block (purely sequential piece of binary code)
- idem for a transition

Analysing the binary instructions, the good old time...

- until the 80’s, processors where (mostly) time predictable, e.g. MC68000:
  - instruction (according to the user manual):
    \[ \text{WCET}(\text{ADD}, \ L \ #5, \ D0) = 10 \text{ cpu cycles} \]
  - sequence:
    \[ \text{WCET}(\text{instr1} ; \text{instr2}) = \text{WCET}(\text{instr1}) + \text{WCET}(\text{instr2}) \]
  - branching penalty, e.g. \text{bne 0x00EF42}:
    taken: +4 penalty  
    not taken: −2 penalty
  - finally: not “exact” (e.g. instruction fetch pipeline), but fairly precise...
Analysing the binary instructions, nowadays

• Nowadays: the “additive” principle is false even for very “simple” architecture:
  ➔ complex (micro)-instruction pipeline (3/4 instructions in parallel)
  ➔ branch prediction in pipeline: big penalty when the “guess” is wrong!
  ➔ memory caches: LOAD/STORE may be 10 times faster if the address is in cache (hit) or not (miss)
  ➔ even more complicated with several cache layers!

• Exec Time depends on the precise state of the architecture
  ➔ WCET($HWS$, instr1 ; instr2) = WCET($HWS$, instr1) + WCET($HWS'$, instr2)
  ➔ where $HWS'$ = Post($HWS$, instr1)

• In practice:
  ➔ The number of actual $HWS$ is untractable
  ➔ Need to abstract (simplify) ... while keeping safe (over-approximation)

Analysing the binary instructions, nowadays (cntd)

• “monotonicity principle”
  ➔ $AHS$ = abstract = set of (concrete) $HWS$
  ➔ WCET($AHS$, instr1 ; instr2) ≤ WCET($AHS$, instr1) + WCET($AHS'$, instr2)
  ➔ where $AHS'$ ⊇ ⋃ Post($HWS$, instr1) s.t. $HWS \in AHS$

• A BIG problem: timing anomalies
  ➔ there exist machines s.t. MONOTONICITY DOES NOT HOLD
    * i.e. local WCET does not lead to global WCET
    * Example: speculation anomaly read x; if cond then B else C(x)

  cache hit
  read x
  eval cond
  prefetch B
  pred. miss
  B canceled
  C

  cache miss
  read x
  eval cond
  C

  ➔ and plenty of anomalies as soon as multi threading and concurrency is involved!
Analysing the binary instructions, nowadays (cntd)

- Classification of architectures:
  - Timing Compositional
    - No timing anomalies, e.g. ARM7
  - Compositional with bounded effects
    - Timing anomalies limited (i.e anomalies do not cross branches)
    - e.g. (probably) TriCore
  - Non-compositional
    - Timing anomalies with observed *domino effect* (i.e anomalies cross branches)
    - e.g. PPC 755

Modern archi vs (hard) Real-time

- Most of advanced features improve *average* execution time
  but make *worst case* highly unpredictable

- Hard-real time domains try to use only *Timing Compositional* architecture (perhaps with bounded effects)
  n.b. It is often possible to *disable* unpredictable features (e.g. branch prediction)

- However, analysing features like pipeline and memory caches is mandatory to get realistic (not too pessimistic) estimation.
Micro-archi analysis: memory cache example

Principle of a simple, one-layer cache

- Memory divided in pieces called lines: \( @ = \text{line number} + \text{offset} \)
- Cache contains a fixed number of copies of memory lines
- Processor need to access \( @ = \text{line} \_i + \text{offset} \):
  - \( l@ \) is in the cache: HIT, costs few cycles
  - \( l@ \) is NOT in the cache: MISS,
    - a cache entry is chosen, its content replaced by (the whole) \( l_i \),
    - costs tens to hundreds cycles

Cache and WCET analysis

- HIT costs much less than MISS
- Supposing MISS all the time is safe but far too pessimistic
- For any memory access in the program:
  - if one can prove that it is necessarily a HIT, count a HIT
  - otherwise count a MISS (even if it may be a HIT: over-approximation)
- is it possible to predict HIT/MISS?
Predictability of caches

- Characteristics of a simple cache:
  - **Fully Associative caches**: any line of the memory can be stored in any line of the cache
  - **Least Recently Used replacement policy**: in case of miss, the evicted line is the least recently accessed one
- With these properties, the cache behavior is highly predictable:
  - Suppose that the cache has 4 lines,
  - and that the program has just accessed 4 different memory lines \( a \), then \( b \), then \( c \), then \( d \), then whatever is the initial state of the cache, we know that:
    * the cache contains \( a, b, c, d \),
    * the LRU line, that will be replaced in case of miss, is \( a \) (and then \( b \), \( c \) etc).

Concrete State of a LRU cache

- A state is a function \( C : C \rightarrow L \cup \emptyset \)
  - \( C = 1 \cdots k \) is the set of cache line indices
  - \( L = 1 \cdots n \) is the set of memory line indices
  - \( \emptyset \) denotes an empty cache line (very initial state only)
- Age of cache line:
  - cache lines are sorted from most recently used (1) to least recently used (\( k \))
  - in case of **MISS**, lines are shifted:
    
    | 1 | 2 | 3 | 4 |
    |---|---|---|---|
    | a | b | c | d |
    
    Access \( c \) ⇒
    
    | 1 | 2 | 3 | 4 |
    |---|---|---|---|
    | e | a | b | c |

  - i.e. in case of **HIT**, order is updated:
    
    | 1 | 2 | 3 | 4 |
    |---|---|---|---|
    | a | b | c | d |
    
    Access \( c \) ⇒
    
    | 1 | 2 | 3 | 4 |
    |---|---|---|---|
    | c | a | b | d |

- Notation: \( C' = up(C, a) \) (the update of cache \( C \) after access \( a \))
- n.b. “behavioural” modeling, in real hardware lines don’t move but are re-numbered!
Uncertainty in cache analysis

- when analysing a piece of code:
  - the starting state is (in general) not precisely known
  - even if it is known, the code may result in several possible states
- example (with a 4-lines cache):

```c
if (access(a)) {
    access(b);
    access(c);
} else {
    access(b);
    access(a);
    access(c);
} access(d);
access(e);
access(a); // HIT or MISS?
```

- beginning, cache is

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

- end of “then” branch, cache is

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>b</td>
<td>a</td>
<td>?</td>
</tr>
</tbody>
</table>

- end of “else” branch, cache is

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a</td>
<td>b</td>
<td>?</td>
</tr>
</tbody>
</table>

- last access(a) may be a HIT or a MISS...

- safe approximation: count a MISS

Micro-archi analysis: memory cache example

Abstract State of a LRU cache

- What a (safe) abstraction must satisfy:
  - abstract state = a set of concrete state ($A = \{C\}$)
  - abstract union, when merging abstract states:
    
    $A \cup A' = \{C\} \cup \{C'\}$
  - abstract update:
    
    $A' = up(A, a) \Rightarrow A' \supseteq \bigcup_{C \in A} up(A, a)$
  - HIT-preserving:
    
    access(a) is HIT in $A \Rightarrow \forall C \in A$ access(a) is HIT in $C$

Micro-archi analysis: memory cache example
Abstract State of a LRU cache (cntd)

- Classical abstraction, “max age”: \( \mathcal{A} : C \rightarrow L \cup \infty \)
  \( \mathcal{A}(a) = j \) means
  “in all concrete state, line \( a \) is present and its age (position) is \( \leq j \)”
  \( \mathcal{A}(a) = \infty \) means
  “in all concrete state, line \( a \) is NOT present”

- \( a \) is HIT in \( \mathcal{A} \) \( \Rightarrow \) \( \mathcal{A}(a) \neq \infty \)

- merge:
  \( \mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2 \) \( \iff \) \( \forall a \in C \) \( \mathcal{A}(a) = \text{MAX}(\mathcal{A}_1(a), \mathcal{A}_2(a)) \)

- Abstract update \( \mathcal{A}' = \text{Aup}(\mathcal{A}, a) \) \( \iff \)
  \( \mathcal{A}'(a) = 1 \)
  \( \forall b \neq a \) \( \mathcal{A}'(b) = \text{if } \mathcal{A}(b) < k \) then \( \mathcal{A}(b) + 1 \) else \( \infty \)

Back to the example

- Notation: \( \mathcal{A} = \{ x / \mathcal{A}(x) = 1 \} \{ x / \mathcal{A}(x) = 2 \} \{ x / \mathcal{A}(x) = 3 \} \{ x / \mathcal{A}(x) = 4 \} \)
- \( \{ x / \mathcal{A}(x) = \infty \} \) are not represented
Cache analysis and loops

- Loop bounds Analysis

Goal

- find an upper bound for the number of times each back-edge in the CFG can be taken
- strongly related to the **HALTING problem**, and thus **undecidable** (in general)

The classical Collatz problem

```c
void collatz(int n){
    assert(n > 0);
    while (n != 1) {
        if (n & 1)
            n = 3 * n + 1;
        else
            n = n / 2;
    }
}
```

- It is widely believed that this program **halts** for any n
- But nobody knows how to prove it (for now, and probably for a long time ...)

Note: convergence is trivial, monotonicity in finite lattice

Micro-archi analysis: memory cache example

Loop bounds Analysis

---

24/47

25/47
The general approach: termination analysis

- Handles any kind of “loops” (recursion, for, while ...)
- Tries to find a decreasing measure of the loop
- Hardly (fully) automatic

Loop bounds in Real-time applications

- Pragmatic approach: the program is supposed to be real-time, thus the loops must be bounded by some simple decreasing measure.
- A classical solution:
  - let $i_1, i_2, \ldots$ be the numerical local variables
  - i.e. appearing in the loop condition and the loop body
  - search for a linear combination $\sum \alpha_i k_i$ that decrease at each iteration of the loop
- Works well for simple for and while loops

Examples of simple decreasing sequences

- basic for (or equivalent while)
  - `int i; for(i = 0; i < n; i++) { foo(); }`
  - `int i = 0; while (i < n) { foo(); i++; }
  - decreasing sequence $n - i$,
  - max value = $n - 1$,
  - min value 0,
  - decreasing step = 1,
  - thus bound = $max - min)/step = n - 1$
- Warning: the min decreasing step must be taken into account:
  - `int i = n;
    while(i > 0){ if (...) {i -= 4;} else {i -= 2;} }
  - bound = $(n - 1)/2$
Conclusion: loop (and value) analysis in general

- Involves/uses all the techniques of static program analysis, in particular abstract interpretation
- Deserves a whole course!
- Note: these techniques are also used in micro-architecture analysis (cf. cache analysis)

Path Analysis: the 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. $\text{MAX } f(X)$

  - Output:
    - An optimal valuation $\bar{v}$, such that $A\bar{v} \leq B$ and $f(\bar{v})$ 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

- $\mu$-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:
  $\leftrightarrow$ Structural constraints
  $a + d = 1$
  $g = a + d$
  $g + k = p + h$
  $h = e + b$
  $e + b = f + c$
  $f + c = k$
  $p = 1$
  $\leftrightarrow$ Semantic constraints
  $h \leq n = 10$
  $\leftrightarrow$ Objective function: $\text{MAX}(\sum_{x \in E} w_x x)$

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

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:

  \[
  \begin{align*}
  W(e) &= w_e \\
  W(p_1 ; p_2) &= W(p_1) + W(p_2) \\
  W(p_1 + p_2) &= \text{MAX}(W(p_1), W(p_2)) \\
  W(p^n) &= n \times W(p)
  \end{align*}
  \]
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):

```c
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*/
    }
} /*p:7*/
```

- branch b cannot be taken more than n/2 times:
  \[ b \leq \frac{n}{2} \text{, i.e. } b \leq 5 \]

- if b is taken, c cannot be taken
  \[ b + c \leq n \text{, i.e. } b + c \leq 10 \]

- ILP system + extra constraint reach optimal solution for:

  \[ a = g = p = 1, d = 0, h = k = 10, b = c = e = f = 5 \]

  \[ 26 + 7 + 7 + 10 * (5 + 5) + 5 * (72 + 50 + 68 + 32) = 1250 \]

  \[ \text{enhancement (from 1540): 19%} \]

Infeasibility properties: many problems...

- May or may not enhance the WCET estimate
  \[ \leftrightarrow \text{ do they concern “heavy” or “light” paths?} \]

- 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)

```c
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:
  \[ e + f \leq n \]

- More subtle: if \(a\) is taken, then at each iteration, if \(b\) is taken, then \(c\) cannot be taken:
  \[ n \cdot a + b + c \leq 2n, \text{ works} \]

- Suppose \(a\) is NOT taken, then \(a = 0\) and the constraint becomes:
  \[ b + c \leq 2n \] which is trivially satisfied

- Suppose \(a\) is taken, then \(a = 1\) and the constraint becomes:
  \[ b + c \leq n \] which express the exclusion

---

**Path Analysis: the Implicit Path Enumeration Technique**

Express infeasibility in ILP (examples)

```c
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
  \[ m \cdot a + b \leq n \cdot m \]

- Conflict across iteration: if \(b\) is taken, \(a\) cannot be taken in the next loop:
  \[ a + b \leq n + 1 \]

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

Example (cntd): WCET estimation

- Binary code
  - via arm-elf-gcc
  - WCET estimation should be done here for `modes_step`
  - i.e. a step of main infinite loop

- Call an ILP Solver (here LPSolve)
  - get 496 + the left-most path

- Control Flow Graph (CFG) reconstruction
  - Basic Blocks + edges (small part here)

- \( \mu \)-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: \( \sum e_{i,j} = \sum e_{j,k} \)
      (and indeed: entry = exit = 1)
    - Semantics Constraints
      loop bounds (not here), others?
    - Objective: \( \text{MAX} \sum c_{i,j} \times e_{i,j} \)

- Control Flow Graph (CFG) reconstruction
  - Works at binary level
  - \( \mu \)-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: \( \sum e_{i,j} = \sum e_{j,k} \)
        (and indeed: entry = exit = 1)
      - Semantics Constraints
        loop bounds (not here), others?
      - Objective: \( \text{MAX} \sum c_{i,j} \times e_{i,j} \)
  - 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
  → Implemented with the notion of “clock-enabled” (e.g. when/current in Lustre)
- Compiler correct ⇒ codes of the modes must be exclusive
  → Interesting property for enhancing WCET

WCET and synchronous programming

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

- Intra-module exclusions: between A0, A1, A2, and between B0 and B1
  → may or may not be “obvious” on the generated code (i.e. structural)
- Inter-module exclusions: not in mode A0 implies mode B1
  → no chance to be obvious on the generated code
- In all cases, relatively “complex” properties:
  → 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)
  ➔ Optimization (-O2)
  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 may 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:
  $\Phi = (\text{idle} \land \text{high} \land \neg \text{degr})$
- 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:
    $\neg (\text{idle} \land \text{low} \land \text{high} \land \text{nom} \land \text{degr})$
  - 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 $\text{idle} \land \text{nom}$ (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 \( \neg(idle \land low) = (\neg idle \lor \neg low) \) an invariant ?
    * e.g. is \( \neg(idle \land \neg low) = (\neg idle \lor low) \) an invariant ?
    * etc. there are \( 2 \times C^2_5 = 20 \) such potential relations to check
  - seems a lot, but polynomial: quadratic: \( C^2_n = n(n - 1)/2 \)

Example: checks the \( 2 \times C^2_5 = 20 \) pairwise disjunctive relations

- six of them are proved invariant:
  - \( \neg idle \lor \neg low \) and \( \neg idle \lor \neg high \) and \( \neg low \lor \neg high \) and
  - \( \neg nom \lor \neg degr \) and \( \neg low \lor \neg nom \) and \( \neg high \lor \neg nom \)

- 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

An alternative top-down algorithm (cnt’d)