6.b WCET analysis

Credits to Enrico Mezzetti, PhD
(enrico.mezzetti@bsc.es)

Where we learn how the worst-case execution-time (WCET) value used in response-time analysis can be determined, and explore the taxonomy of WCET analysis techniques

Worst-case execution time (WCET)

- For any input data and for all initial logical states
  - So that all *execution paths* of the program are covered
- For any hardware state
  - So that the worst-case *execution conditions* are in effect
- Measurement-based WCET analysis
  - On either the real HW or a cycle-accurate simulator of it
  - Caution: the *high-watermark* value can be ≪ WCET
- Static WCET analysis
  - Uses an abstract model of the HW and of the program

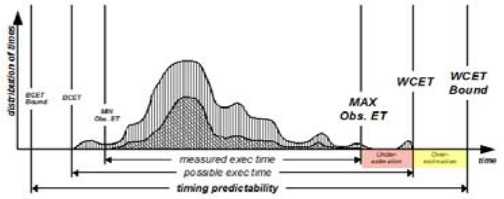
Computing the WCET /1

- Why not measure the task's WCET on its real target HW?

- Triggering the WCET by test is exceedingly difficult
  - Supplying input data that cover all possible program executions is intractable in practice
  - Worst-case initial state on modern HW is very hard to determine
    - Complex pipelines (out-of-order execution)
    - Caches
    - Branch predictors and speculative execution

Computing the WCET /2

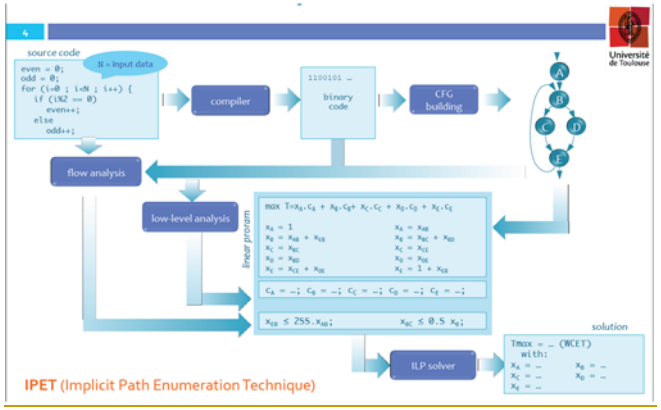
- Exact WCET not generally computable (~ the *halting problem*)
- Yet, WCET *bounds* are essential to feasibility analysis
  - Which must be *safe*, to upper bound all possible executions
  - Which must be *tight*, to avoid costly over-dimensioning



Static WCET analysis /1

- To analyze a program without executing it
  - Needs an *abstract model* of the target HW
  - As well as the binary executable of the program
- Execution time depends on the program's control flow and on the HW fine-grained behavior
  - ***High-level analysis*** addresses program execution
    - *Control flow analysis* builds a control flow graph (**CFG**) for it
  - ***Low-level analysis*** determines the timing cost of individual processor instructions on the abstract model of the HW
    - Not constant in modern HW
    - Must be aware of the HW inner workings (pipeline, caches, etc.)

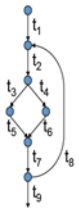
Static WCET analysis /2



Implicit path enumeration technique

- The program's CFG is augmented with flow graph constraints
- The WCET is computed with integer linear programming or constraint programming
- $WCET = max\{\sum_i x_i \times t_i\}$
  - x_i is the *execution frequency* of CFG edge i
  - t_i the *execution time* of CFG edge i

CFG

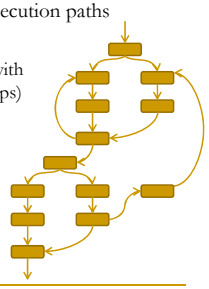


Flow graph constraints

```
x1       = 1
x1 + x8 = x2
x2       = x3 + x4
x3       = x5
x4       = x6
x5 + x6 = x7
x7       = x8 + x9
x2       <= LB * x1
```


Static WCET analysis /3

- ***High-level analysis*** /1
  - Must analyze all possible execution paths of the program
    - Builds the CFG as a superset of all possible execution paths
    - The unit of that analysis is the *basic block*
      - The longest sequence of program instructions with single entry and single exit (no branches, no loops)
  - Path analysis faces multiple challenges
    - *Input-data dependency*
    - *Infeasible paths*
    - *Loop bounds* and recursion depth
    - *Dynamic calls* through pointers



Static WCET analysis /4

- ***High-level analysis*** /2
  - Several techniques are employed to enable the use of IPET
    - *Control-flow analysis* to construct the CFG
    - *Data-flow analysis* to find loop bounds
    - *Value analysis* to resolve memory accesses
  - Automated information extraction is insufficient 
    - User annotation of *flow facts* is needed
      - To help detect infeasible paths
      - To refine loop bounds
      - To define frequency relations between basic blocks
      - To specify the target of dynamic calls and memory references

Static WCET analysis /5

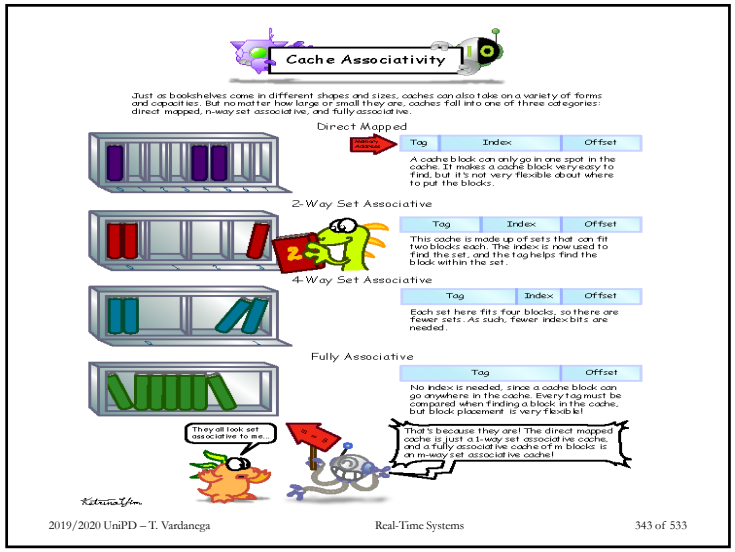
- ***Low-level analysis*** /1
  - Requires abstract modeling of all HW features
    - Processor, memory subsystem, buses, peripherals, …
    - It is *conservative* : it must never underestimate actual costs
    - All possible HW states should be accounted for
  - HW modeling faces multiple challenges
    - *Precise modeling* of complex hardware is difficult
      - Inherent complexity (e.g., out-of-order pipelines)
      - Lack of comprehensive information (intellectual property, patents, …)
      - Differences between specification and implementation (!)
    - *Exhaustive representation* of all HW states is computationally infeasible

Static WCET analysis /6

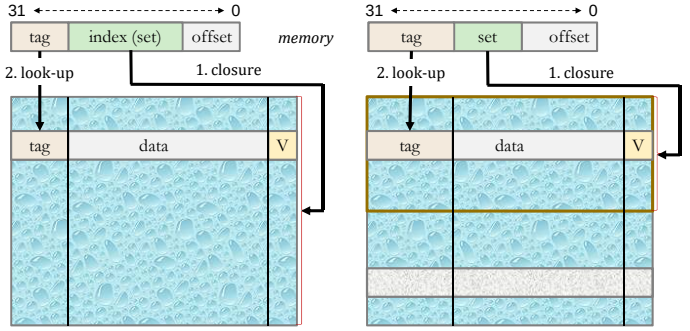
- ***Low-level analysis*** /2
  - Concrete HW states
    - Determined by the history of execution
    - Cannot compute all HW states for all possible executions
      - Invariant HW states are grouped into execution contexts
      - *Conservative overestimations* are made to reduce the research space
  - *Abstract interpretation*
    - Computes abstract states and specific operators in the abstract domain
      - *Update function* to keep the abstract state current along the exec path
      - *Join function* to merge control flows after a branch
  - Some techniques are specific to each HW feature

Understanding the cache

- The cache memory is much smaller than the RAM
  - Chunks of the latter map to individual units of the former
- Three such mappings (called cache *associativity*) are used
  - *Direct mapping*
    - The cache holds K lines (16-128 bytes each, to leverage locality)
    - The RAM of size M bytes is divided in K blocks sized $\frac{M}{K}$ bytes each: access conflicts occur *within* blocks for addresses more distant than a single cache line
  - *Fully associative*
    - Any RAM address can map to *any* cache line
    - Much reduced chance of conflict but massively more complex mapping
  - *N-way set associative*
    - The cache is divided into $S = \frac{K}{N}$ sets, each holding $N = 2$ or 4 lines
    - The RAM of size M bytes is divided in S blocks: as each cache set holds N lines, the chance of access conflict is reduced accordingly



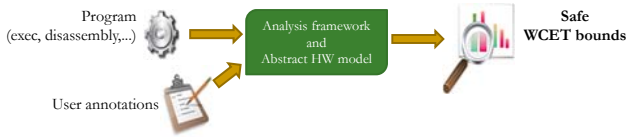
Understanding the cache



Direct mapping (by index)
Each memory address maps to a *single* cache block: the (hash of the) tag field gives it placement

Set-associative mapping (by set)
Each memory address maps to a *set* of cache blocks: the index field tells the set and the tag the placement in it

Static WCET analysis: the big picture



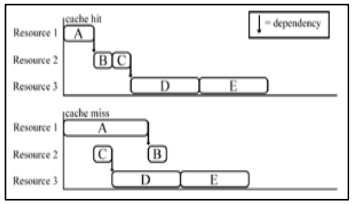
- Open problems
  - Can we always trust the abstract model of the HW?
  - How much overestimation do we incur?
    - Inclusion of infeasible paths
    - Overestimation is inevitable in abstract state computation
  - Intrinsic weakness of user annotations
    - Labor intensive and error prone

Static WCET analysis /7

- Safeness is at risk
  - When *local* worst case in our processing resource may not always lead to *global* worst case at program level
  - This reflects ***timing anomalies*** that originate from
    - Complex hardware architectures (e.g., out-of-order pipelines)
    - Improper design choices (e.g., inept cache replacement policies)
    - *Counter-intuitive* timing behavior
    - Faster execution of a single instruction with *long-term* negative effects
  - Very difficult to account for in static analysis

Timing anomaly: example

- Assume there is dependency between (some) instructions because of shared HW resources (as in pipeline stages)
- And opportunistic scheduling is made of individual requests



- Faster execution of A leads to worse overall execution, owing to the order in which the instructions are executed

Hybrid analysis /1

- To obtain realistic (less pessimistic) WCET estimates
  - On the *real* target processor and on the *final* executable
    - WCET analysis helps software design before coding: analysis loses value if the program is modified (!)
  - Yet, understanding that safeness is not guaranteed (!)
- Hybrid approaches leverage
  - The measurement of basic blocks on the real HW
    - To avoid pessimism from abstract modeling
  - Static analysis techniques to combine the obtained measures
    - Knowledge of the program execution paths
- Newer approaches explore *probabilistic properties* (!)

Hybrid analysis /2

- Approaches to collect timing information
  - *Software instrumentation*
    - The program is augmented with instrumentation code
    - Instrumentation affects the timing behavior of the program (aka the *probe effect*) and causes problems to deciding what's the final system
  - *Hardware instrumentation*
    - Depends on specialized HW features (e.g., debug interface)
- Confidence in the results is contingent on the coverage of the executions and on the exploration of worst-case states
  - Exposed to the same problems as static analysis and measurement
  - *Worst-case state dependence is gone if HW response time is randomized*

Hybrid analysis: the big picture

- Open problems
  - Can we trust the resulting estimates?
    - Contingent on worst-case input and worst-case HW state
    - Consideration of infeasible paths
  - Needs the real execution environment or an identical copy of it
    - May cause serious cost impact and inherent difficulty of exactness

Summary

- We have reckoned with the challenge of computing the WCET
- We have seen how *static* WCET analysis works and where its weaknesses are
  - We have learned what *high-level analysis* is
  - And what *low-level analysis* does
- We have seen how *hybrid* analysis (measurement-based) is more pragmatic, but also riskier

Selected readings

- R. Wilhelm et al. (**2008**), *The worst-case execution-time problem—overview of methods and survey of tools*
  DOI: 10.1145/1347375.1347389
- F.J. Cazorla *et al.* (**2019**), *Probabilistic worst-case timing analysis: taxonomy and comprehensive survey*
  DOI: 10.1145/3301283