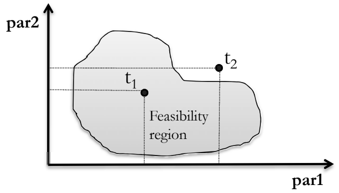
6.a Ramifications of schedulability analysis

Credits to Marco Panunzio, PhD
(marco.panunzio@thalesaleniaspace.com)

Feasibility region

- A topological space that represents the set of systems deemed feasible with given workload model parameters
  - A N-dimensional space for a N-parameter analysis
  - Specific to the schedulability tests in use
  - Helps visualize the geometric meaning of feasibility



t_1 is feasible
t_2 is not feasible

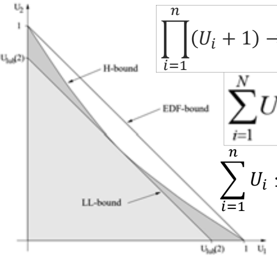
Example

- *Hyperbolic bound* [Bini & Buttazzo, ECRTS, 2001] improves the Liu & Layland utilization test for RM
  - It helps prove that RM achieves 100% utilization when *all pairs* of periods in the task set are in harmonic relation

$$\prod_{i=1}^{n}(U_i + 1) \rightarrow \sqrt{2} + O(n^{-1}) \leq 2$$

$$\sum_{i=1}^{N} U_i \leq 1$$

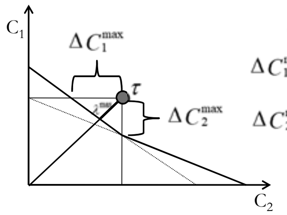
$$\sum_{i=1}^{n} U_i \leq n\left(2^{\frac{1}{n}} - 1\right) \rightarrow \leq ln(2)$$



Examples of feasibility regions
Plot in an $n = 2$ U-space, where each point $U = \{U_1, U_2, \ldots, U_n\}$ represents a periodic task set with utilization U_i

Sensitivity analysis

- Investigates the parameter changes in a real-time system that may possibly
  - Improve the goodness of fit of an already feasible system
  - Turn an infeasible system into a feasible one



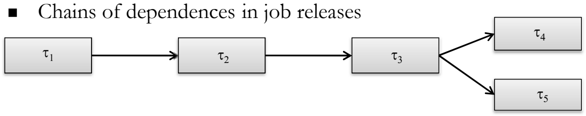
τ Position of the system in the feasibility region

ΔC_1^{max} Maximum feasible variation for the WCET of t_1 (negative in the example)

ΔC_2^{max} Maximum feasible variation for the WCET of t_2 (negative in the example)

Transactions (precedence chains) /1

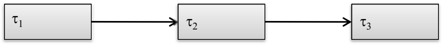
- Causal relations between activities
  - They allow considering relevant information not captured by classic workload models
    - Chains of dependences in job releases



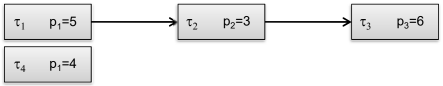
  - Originated in the analysis of distributed systems, where the use of offsets helps contain the pessimism of release jitter
    - Also useful for the analysis of "*collaboration (release) patterns*" employed for single-CPU systems

Transactions /2

- Two main kinds of dependence are of interest here
  - *Direct precedence* relation (e.g., producer-consumer)
    - τ_2 cannot proceed until τ_1 completes

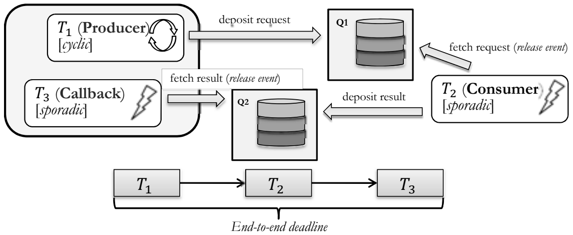


  - *Indirect priority* relation
    - τ_4 does not suffer interference from τ_3 (under FPS and synchronous release of τ_1 and τ_4 for priorities increasing with values)



Example: Ravenscar call-back

- The "*call-back pattern*" helps realize indirect **in-out** interactions between tasks in Ravenscar systems



- The feasibility of the *end-to-end response time* against the corresponding deadline is the matter of interest here (**!**)

Example: classic RTA results

Id	Task	T_i	C_i	Priority	Blocking
τ_1	Producer (periodic)	40	10	4	$B_1 = 2$
τ_2	Consumer (sporadic)	40	10	2 (L)	$B_2 = 0$
τ_3	Call-back (sporadic)	40	5	5 (H)	$B_3 = 2$

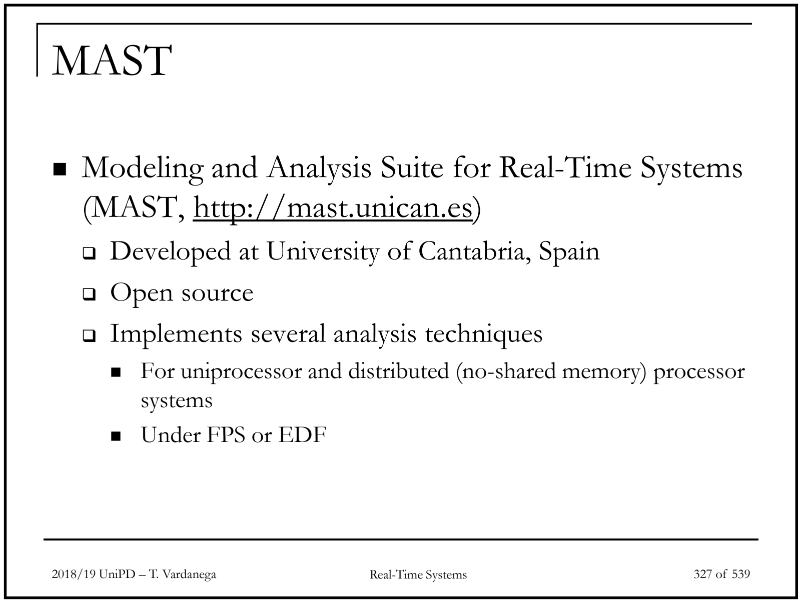
Q1 Ceiling $= max(P_1, P_2) = 4$

Q2 Ceiling $= max(P_2, P_3) = 5$

Classic RTA

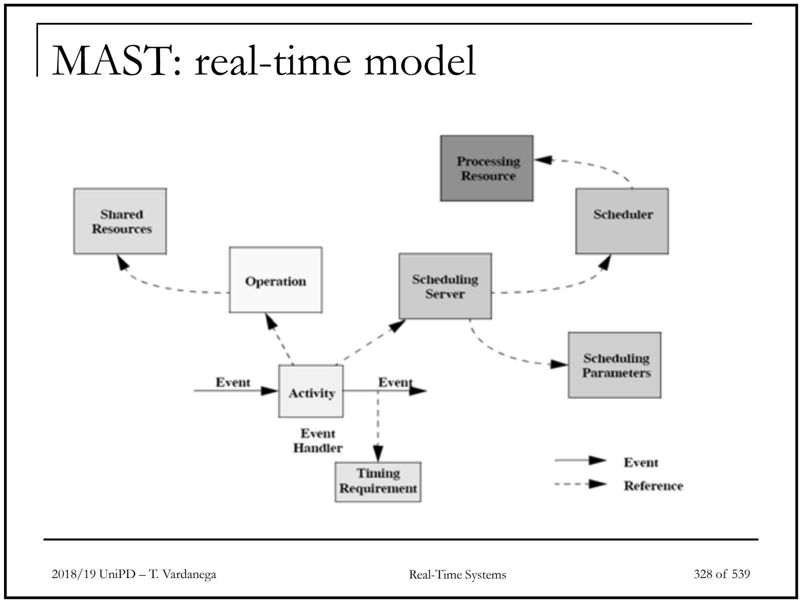
$$\left.\begin{array}{l} R_1 = 17 \\ R_2 = 25 \\ R_3 = \ \ 7 \end{array}\right\}$$ This misses out completely that τ_3 is to be *preceded* by τ_2 and τ_1 (!)

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

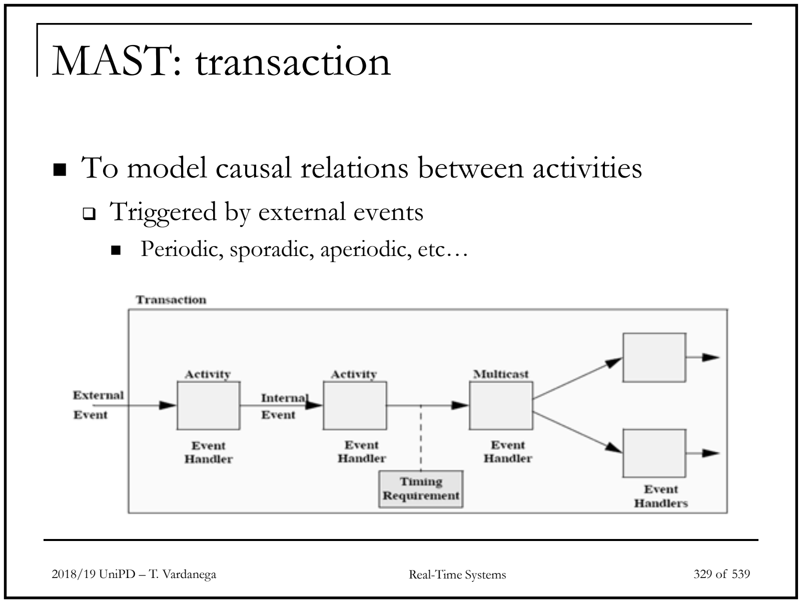


MAST

- Modeling and Analysis Suite for Real-Time Systems (MAST, http://mast.unican.es)
  - ❑ Developed at University of Cantabria, Spain
  - ❑ Open source
  - ❑ Implements several analysis techniques
    - For uniprocessor and distributed (no-shared memory) processor systems
    - Under FPS or EDF

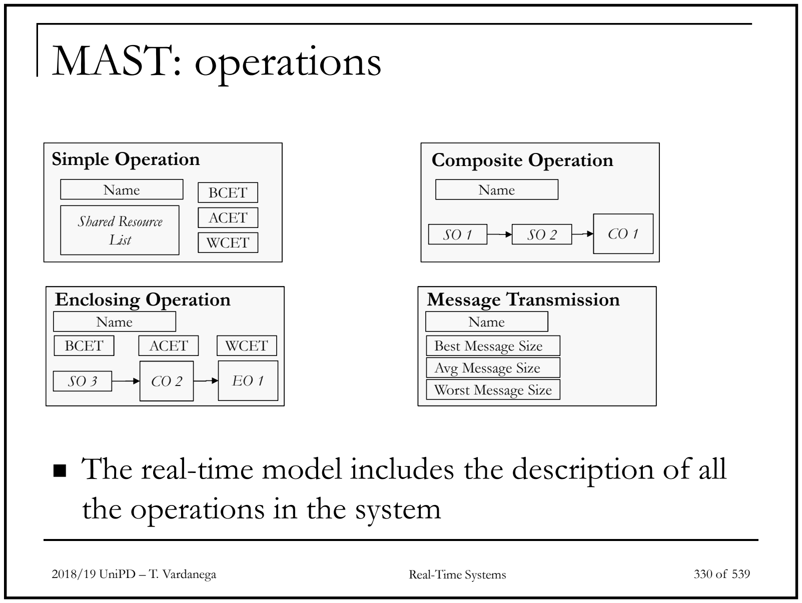


MAST: real-time model



MAST: transaction

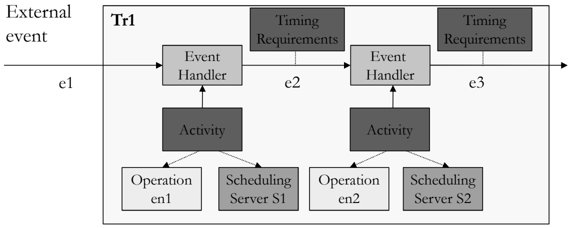
- To model causal relations between activities
  - ❑ Triggered by external events
    - Periodic, sporadic, aperiodic, etc…



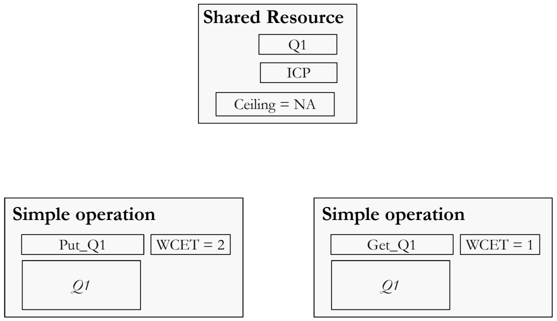
MAST: operations

- The real-time model includes the description of all the operations in the system

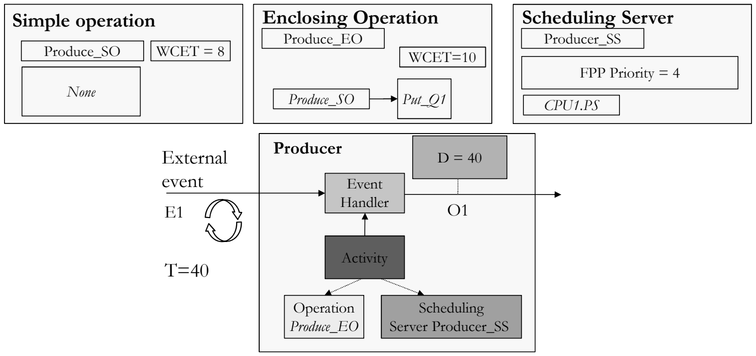
MAST: an example transaction



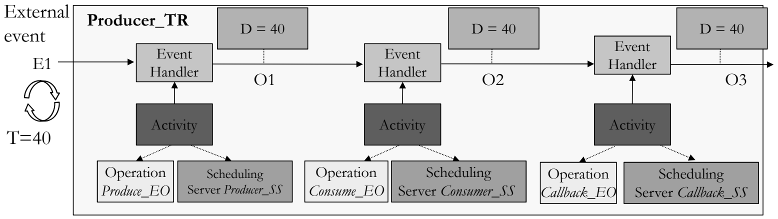
MAST: example shared resources

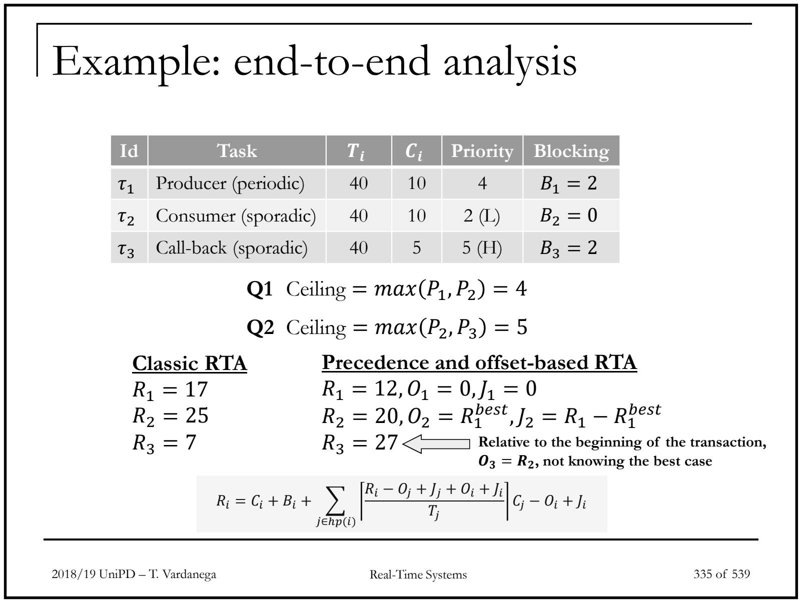


MAST: example tasks



Example: introducing transactions





Example: end-to-end analysis

Id	Task	T_i	C_i	Priority	Blocking
τ_1	Producer (periodic)	40	10	4	$B_1 = 2$
τ_2	Consumer (sporadic)	40	10	2 (L)	$B_2 = 0$
τ_3	Call-back (sporadic)	40	5	5 (H)	$B_3 = 2$

Q1 Ceiling $= max(P_1, P_2) = 4$

Q2 Ceiling $= max(P_2, P_3) = 5$

Classic RTA

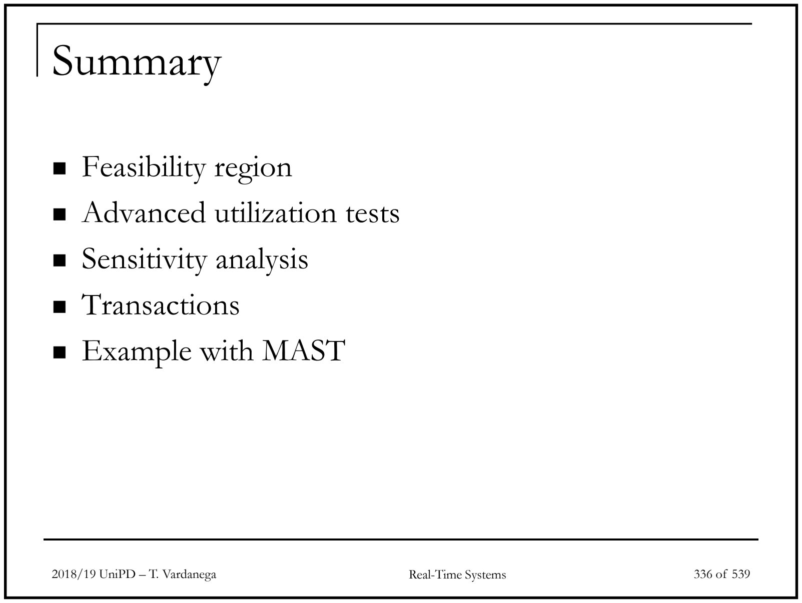
$R_1 = 17$
$R_2 = 25$
$R_3 = 7$

Precedence and offset-based RTA

$R_1 = 12, O_1 = 0, J_1 = 0$
$R_2 = 20, O_2 = R_1^{best}, J_2 = R_1 - R_1^{best}$
$R_3 = 27$

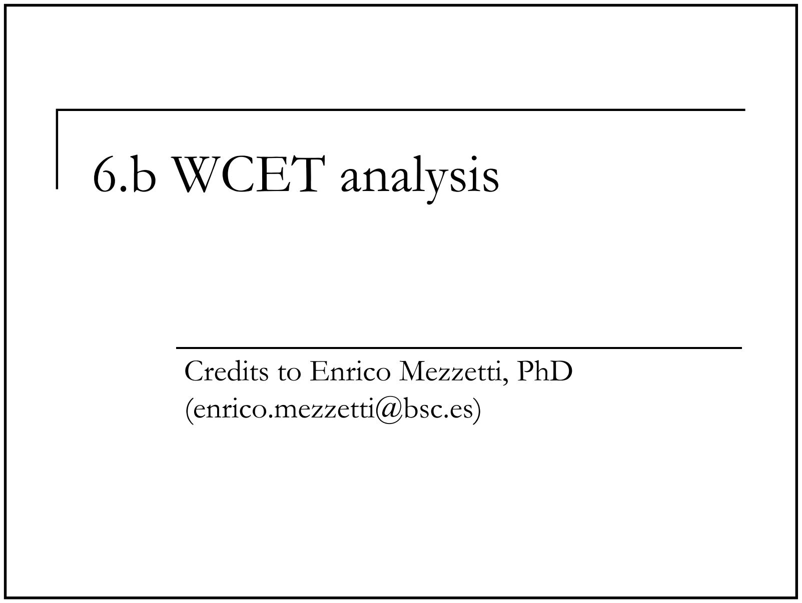
Relative to the beginning of the transaction, $\boldsymbol{O_3 = R_2}$, not knowing the best case

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i - O_j + J_j + O_i + J_i}{T_j} \right\rceil C_j - O_i + J_i$$



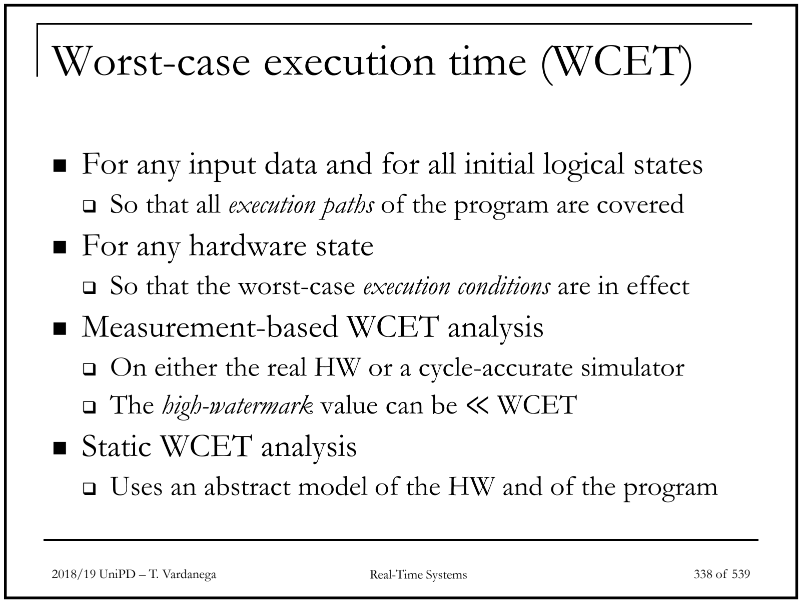
Summary

- Feasibility region
- Advanced utilization tests
- Sensitivity analysis
- Transactions
- Example with MAST



6.b WCET analysis

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

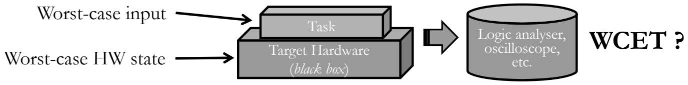


Worst-case execution time (WCET)

- 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
  - 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

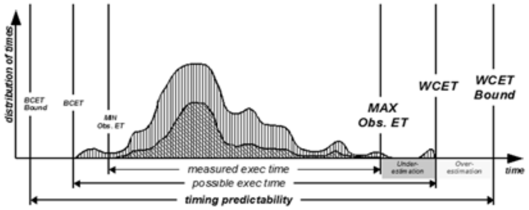
- Why not measure the task's WCET on its real target HW?



- 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 difficult to determine
    - Complex pipelines (out-of-order execution)
    - Caches
    - Branch predictors and speculative execution

Computing the WCET /2

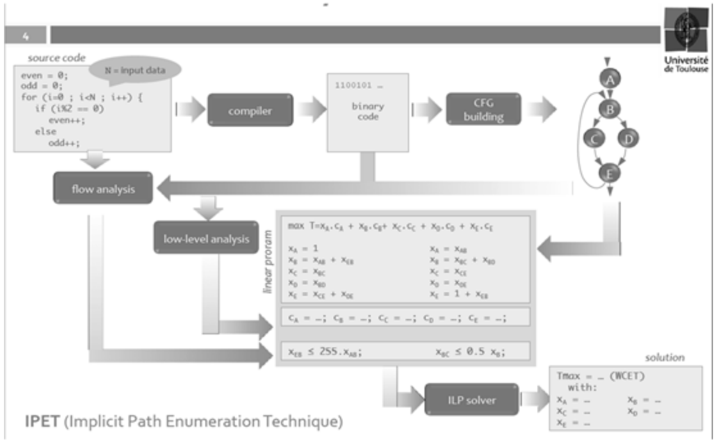
- Exact WCET not generally computable (~ the *halting problem*)
- Yet, WCET *bounds* are essential to feasibility analysis
  - ❑ Which must be *safe* to upper bound all possible executions
  - ❑ Which must be *tight* to avoid costly over-dimensioning



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 the HW behavior
  - ❑ ***High-level analysis*** addresses the program behavior
    - *Control flow analysis* builds a control flow graph (CFG)
  - ❑ ***Low-level analysis*** determines the timing cost of individual instructions on the abstract model of the HW
    - Not constant for modern HW
    - Must be aware of the HW inner workings (pipeline, caches, etc.)

Static WCET analysis /2



```
even = 0;
odd = 0;
for (i=0 ; i<N ; i++) {
  if (i%2 == 0)
    even++;
  else
    odd++;
```

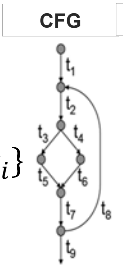
```
max T=xA.cA + xB.cB+ xC.cC + xD.cD + xE.cE

xA = 1                    xA = xAB
xB = xAB + xEB            xB = xBC + xBD
xC = xBC                  xC = xCE
xD = xBD                  xD = xDE
xE = xCE + xDE            xE = 1 + xEB

cA = …; cB = …; cC = …; cD = …; cE = …;

xEB ≤ 255.xAB;            xBC ≤ 0.5 xB;
```

```
Tmax = … (WCET)
  with:
xA = …          xB = …
xC = …          xD = …
xE = …
```

IPET (Implicit Path Enumeration Technique)

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

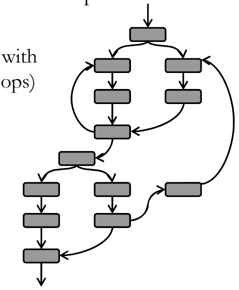


Flow graph constraints

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
      - First finding the basic blocks and then building the call graph among them
    - *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 facilitate detection of infeasible paths
      - To refine loop bounds
      - To define frequency relations between basic blocks
      - To specify the target of dynamic calls and referenced memory addresses

Static WCET analysis /5

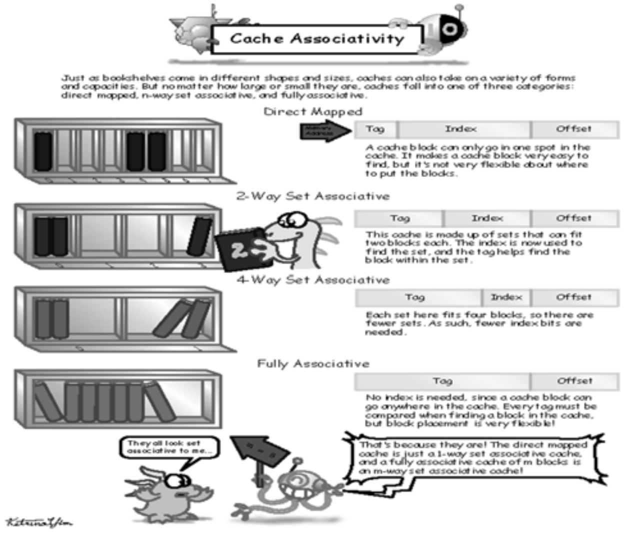
- ***Low-level analysis*** /1
  - Requires abstract modeling of all HW features
    - Processor, memory subsystem, buses, peripherals, …
    - It is *conservative* : it must never underestimate actual costs
    - All possible HW states should be accounted for
  - HW modeling faces multiple challenges
    - *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 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 ($\frac{M}{K}$ bytes each: access conflicts occur *within* blocks for all addresses spaced more 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 $\text{S} = \frac{K}{N}$ sets holding $N = 2$ or 4 lines each
    - 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



Just as bookshelves come in different shapes and sizes, caches can also take on a variety of forms and capacities. But no matter how large or small they are, caches fall into one of three categories: direct mapped, n-way set associative, and fully associative.

Direct Mapped

A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

2-Way Set Associative

This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

4-Way Set Associative

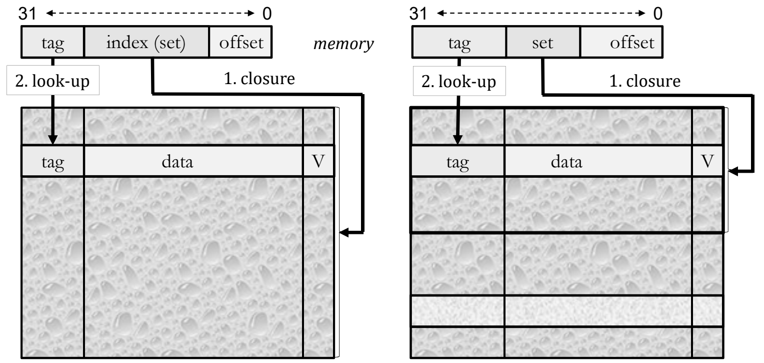
Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.

Fully Associative

No index is needed, since a cache block can go anywhere in the cache. Every tag must be compared when finding a block in the cache, but block placement is very flexible!

That's because they are! The direct mapped cache is just a 1-way set associative cache, and a fully associative cache of m blocks is an m-way set associative cache!

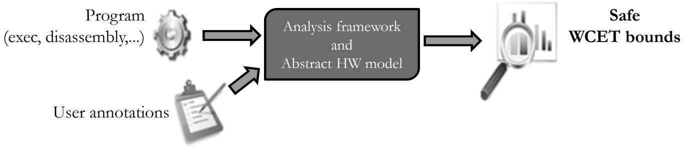
Understanding the cache



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



Program
(exec, disassembly,...)

User annotations

Analysis framework
and
Abstract HW model

**Safe
WCET bounds**

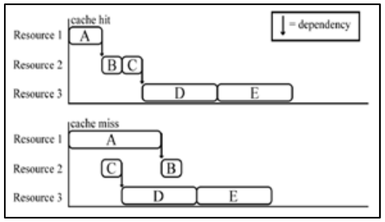
- Open problems
  - Can we always trust the abstract model of the HW?
  - How much overestimation do we incur?
    - Inclusion of infeasible paths
    - Overestimation is inevitable in abstract state computation
  - Intrinsic weakness of user annotations
    - Labor intensive and error prone

Static WCET analysis /7

- Safeness is at risk
  - When *local* worst case does not always lead to *global* worst case
  - Which is the case when ***timing anomalies*** occur
    - Complex hardware architectures (e.g., out-of-order pipelines)
    - Even improper design choices (e.g., inept cache replacement policies)
    - *Counter-intuitive* timing behavior
    - Faster execution of a single instruction causes *long-term* negative effects
  - Both are very difficult to account for in static analysis

Timing anomaly: example

- Assume dependency between (some) instructions
- Shared HW resources (e.g. pipeline stages) and opportunistic scheduling of request service



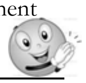
- Faster execution of A leads to worse overall execution, owing to the order in which the instructions are executed

Hybrid analysis /1

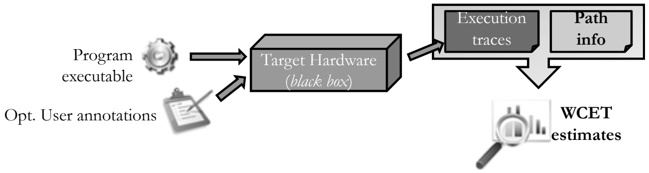
- To obtain *realistic* (less pessimistic) WCET estimates
  - On the real target processor
  - On the final executable
  - Knowing that safeness not guaranteed (**!**)
- Hybrid approaches leverage
  - The measurement of *basic blocks* on the real HW
    - To avoid pessimism from abstract modeling
  - 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 effects 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 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

- The challenge of computing the WCET
- Static analysis
  - High-level analysis
  - Low-level analysis
- Hybrid analysis (measurement-based)

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