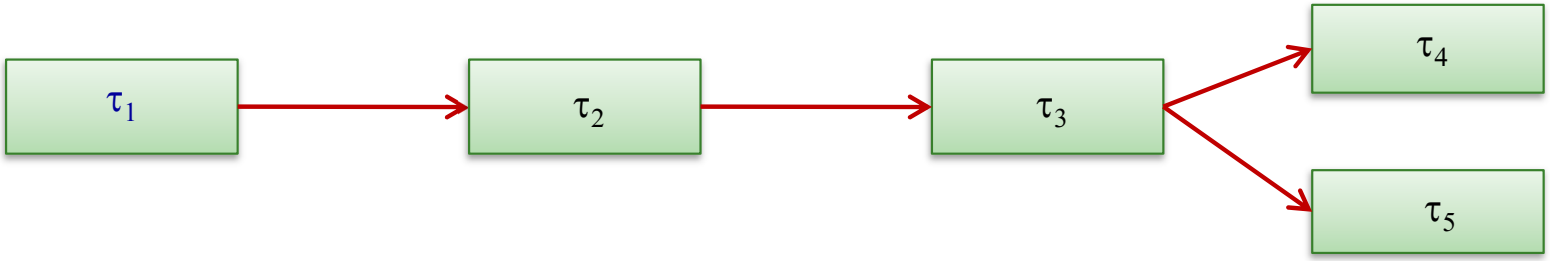
6.a End-to-end analysis

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

Where we appreciate how worst-case offset-based analysis serves very well the purpose of performing end-to-end response-time analysis of concatenation of tasks deployed on single-CPU systems (in addition to distributed systems as seen in §5)

Transactions (precedence chains) /1

- A term that denotes causal relations between tasks
  - "Transactions" express dependency relations that cannot be captured with classic workload models
    - Chains of dependencies in job releases

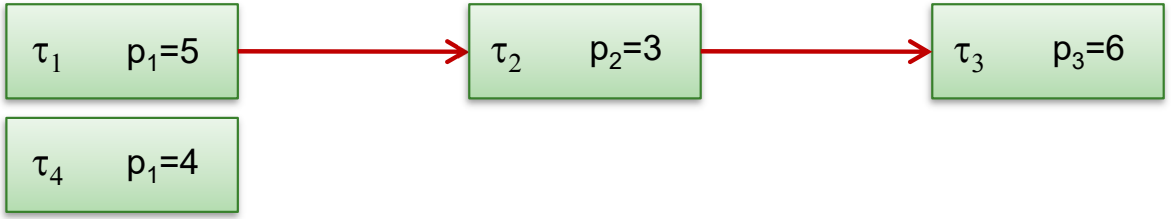


  - Originated in the analysis of distributed systems, where the use of offsets helps contain the pessimism of release jitter
  - They become useful also for the analysis of "*collaboration (release) patterns*" employed for single-CPU systems

Transactions /2

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

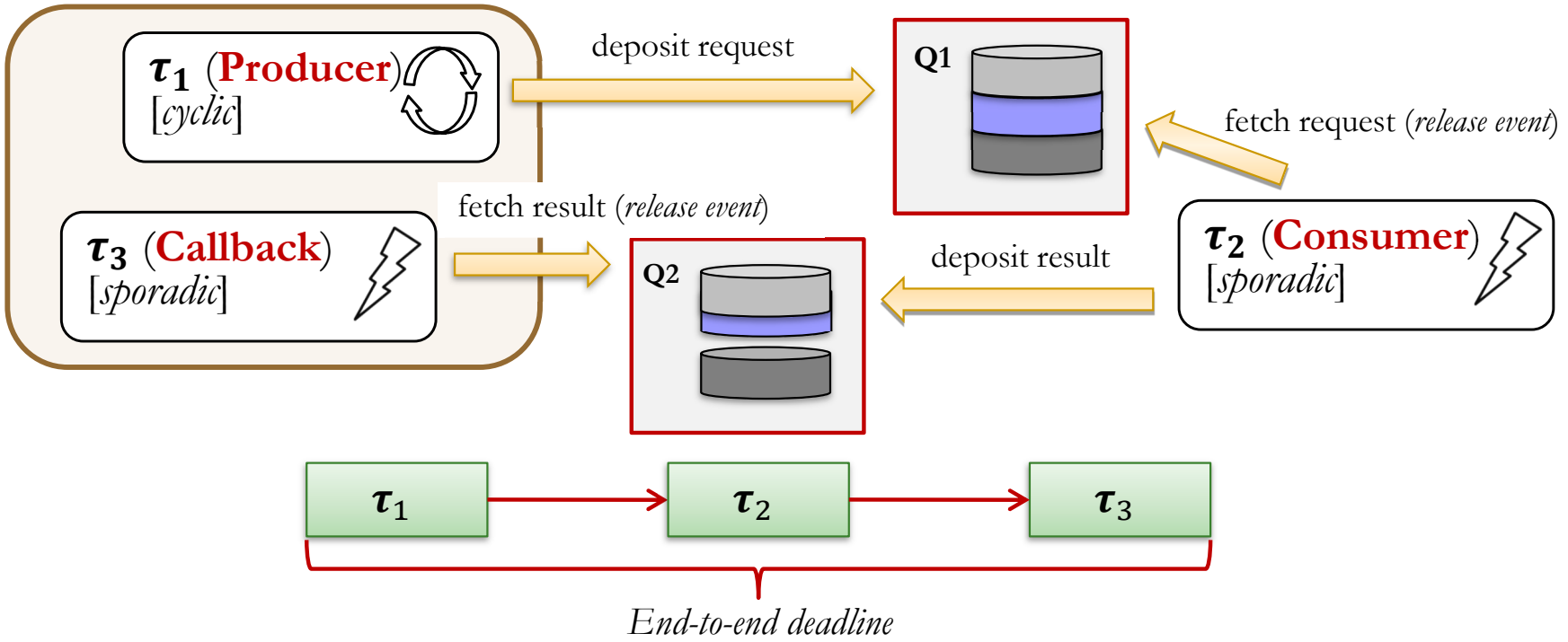
  - *Indirect-priority* relation
    - τ_4 does *not* suffer interference from τ_3 (under FPS and synchronous release of τ_1 and τ_4 for priorities increasing with values)



The call-back problem /1

- In §3.b we have seen that real-time tasks *cannot* make direct `in-out` calls to one another
  - If they did, the worst-case wait time of their synchronization would be *unbounded* in the general case
- Direct synchronization is therefore banned
  - But in the real world we may need the corresponding semantic effects
- The Ada Ravenscar profile *call-back pattern* shows us how to achieve those effects indirectly

The call-back problem /2



- This pattern creates a *concatenation* of tasks $\{\tau_1, \tau_2, \tau_3\}$, which has an *end-to-end* deadline and an *end-to-end* feasibility concern
  - τ_3's deadline in relation to τ_1's release

The call-back problem /3

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 priority $= max(P_1, P_2) = 4$
Q2 Ceiling priority $= max(P_2, P_3) = 5$

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

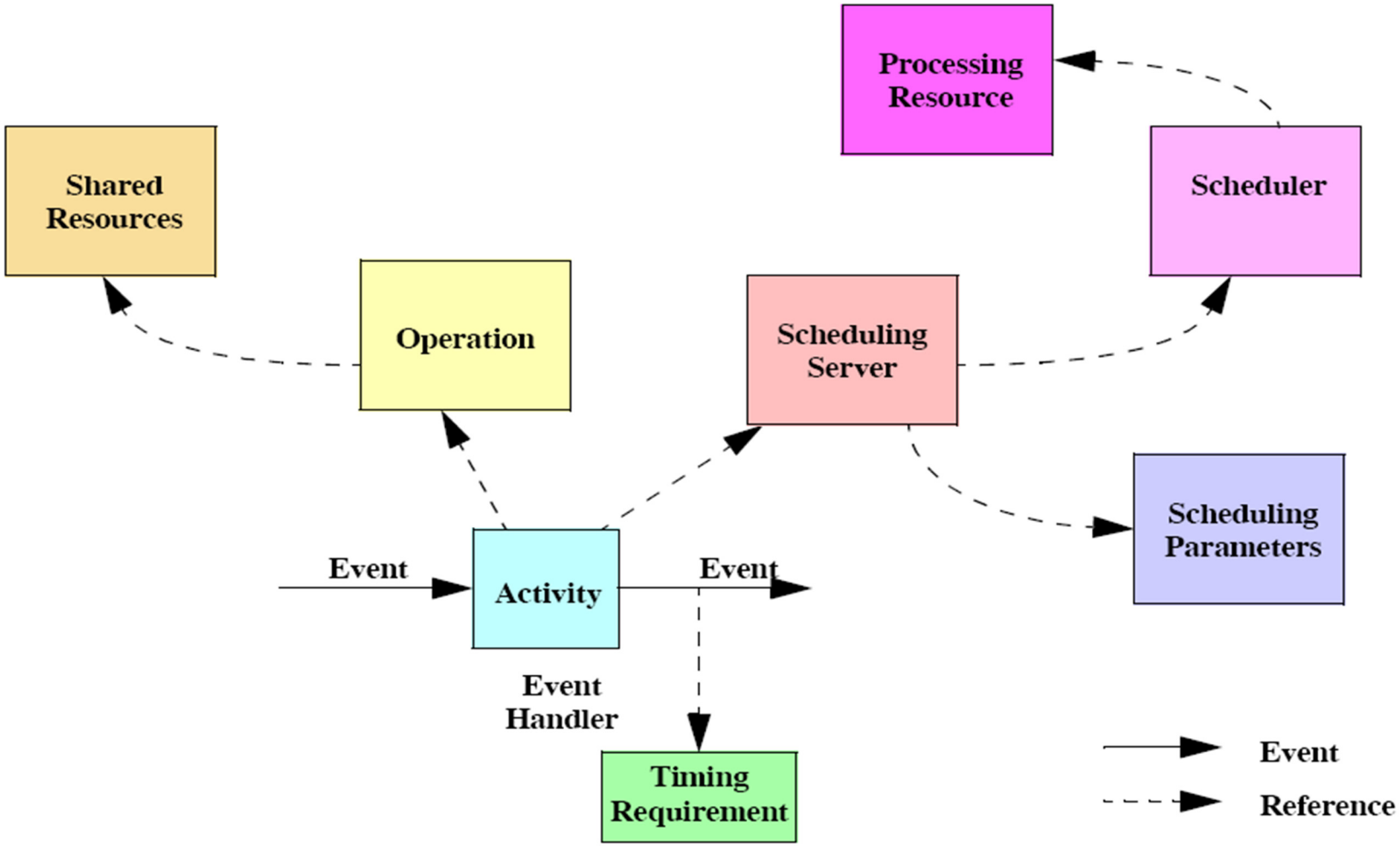
$R_1 = 17$
$R_2 = 25$
$R_3 = 7$

This analysis assumes a critical instant in which all tasks become ready simultaneously, and *misses out completely* that τ_3's release is to be preceded by the completion of τ_2 and τ_1

MAST "understands" transactions

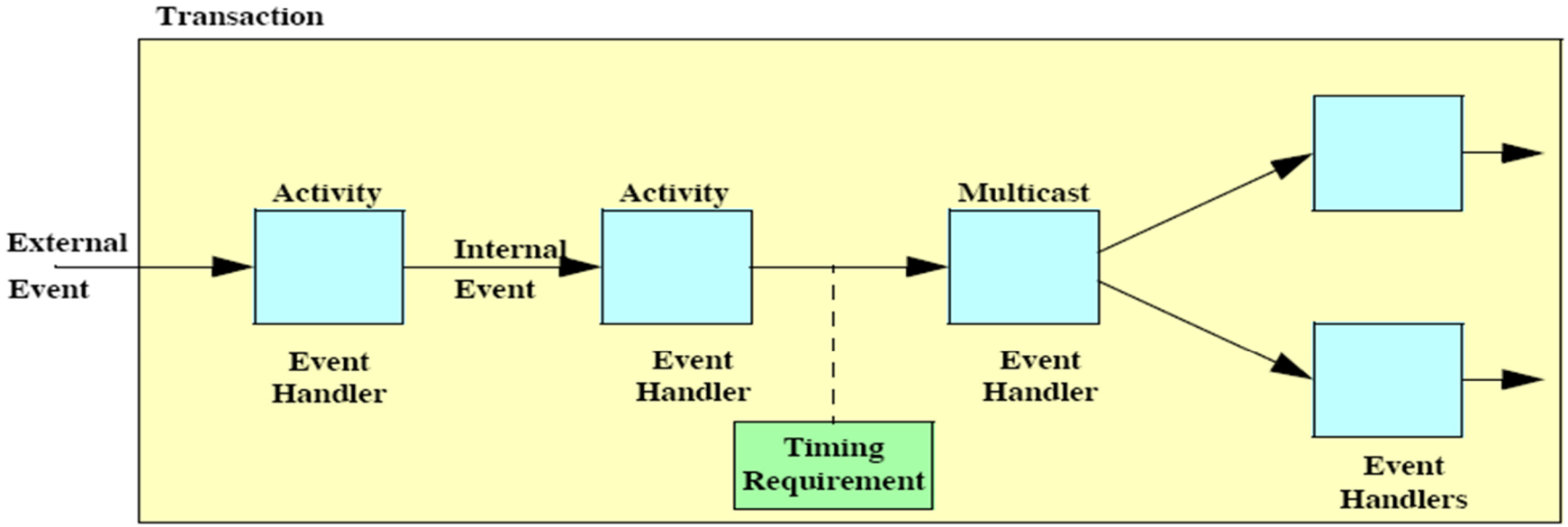
- Modeling and Analysis Suite for Real-Time Systems (MAST, http://mast.unican.es)
  - Developed at University of Cantabria, Spain
  - Open source
  - Implements several analysis techniques
    - For uniprocessor and distributed (no-shared memory) processor systems
    - Under FPS or EDF, alone or in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

MAST: real-time model



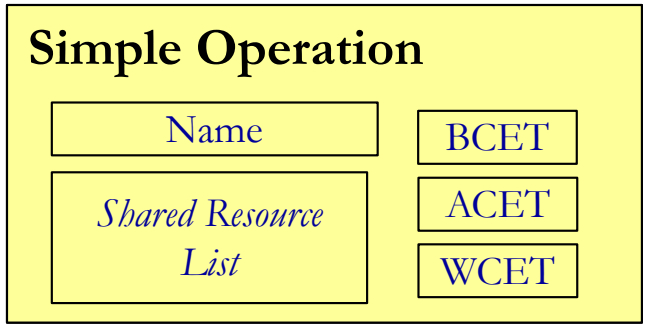
MAST: transaction

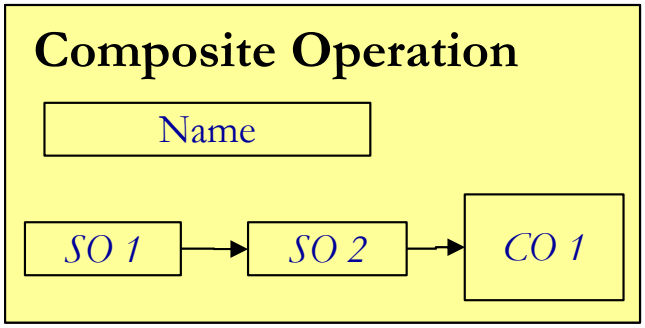
- To model causal relations between activities and form concatenations with them
  - Triggered by external events with various release patterns: periodic, sporadic, aperiodic, in burst, …

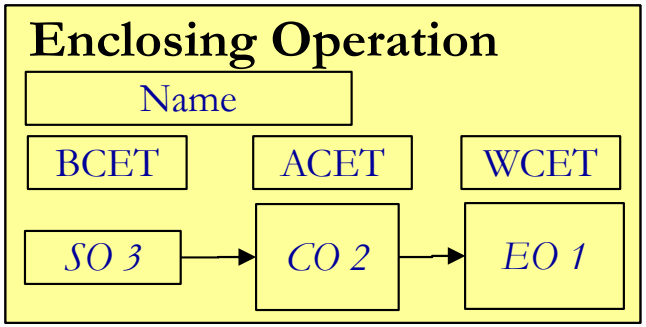


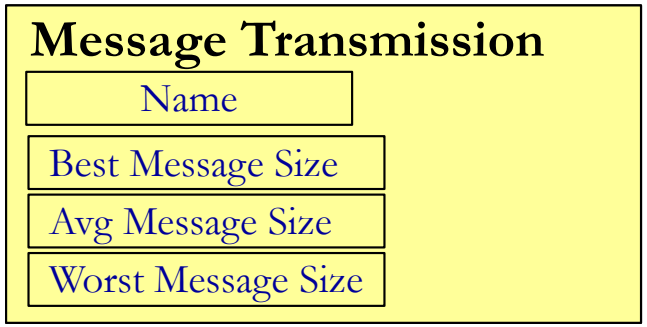
MAST: operations

- The MAST model of a system includes the description of all the operations that it comprises

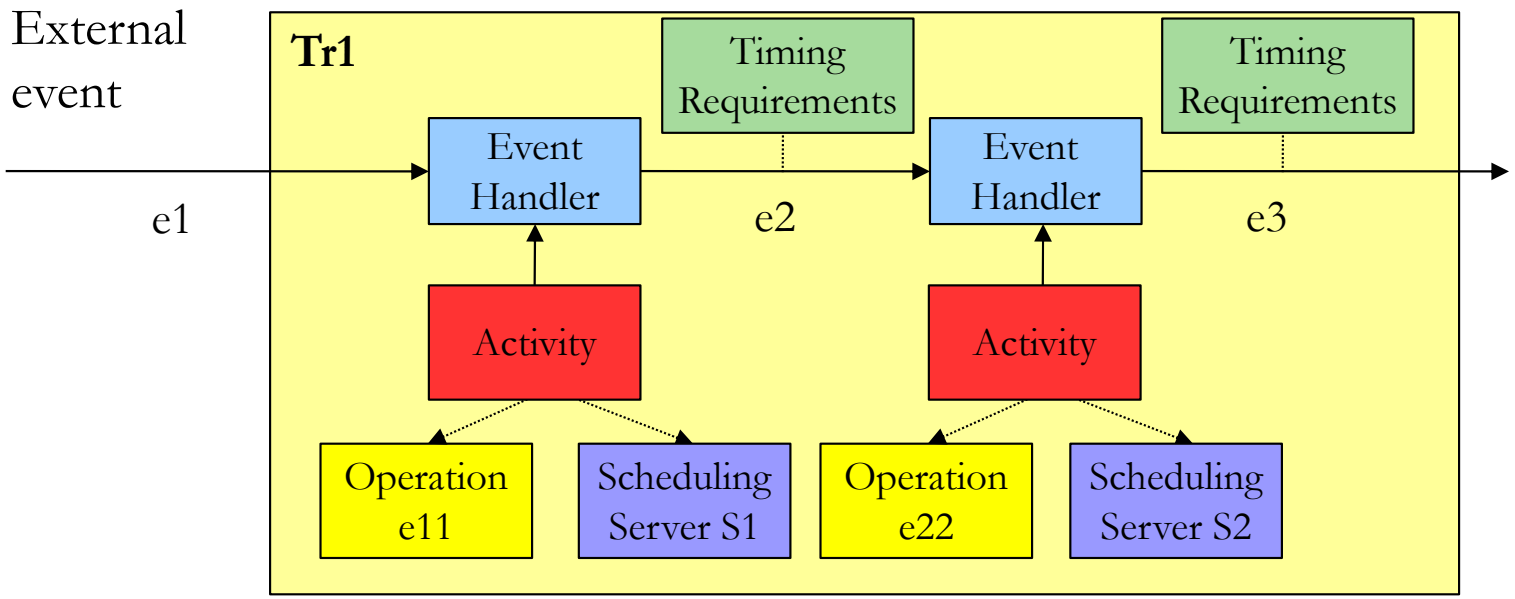




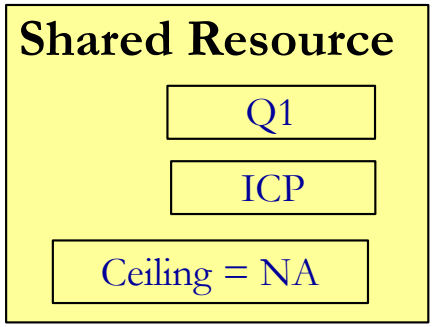


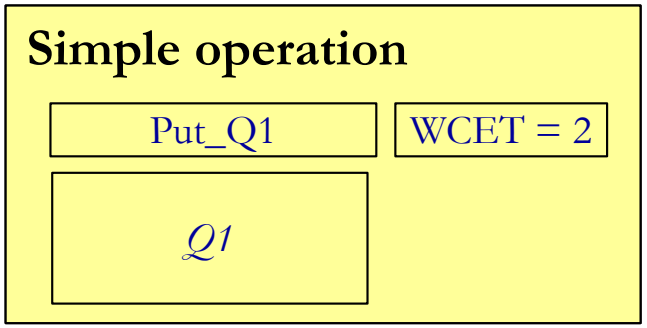


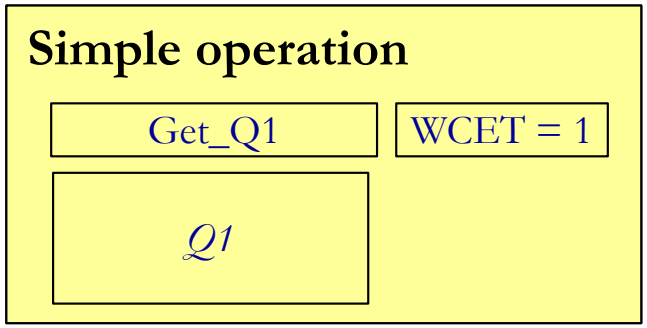
MAST: an example transaction



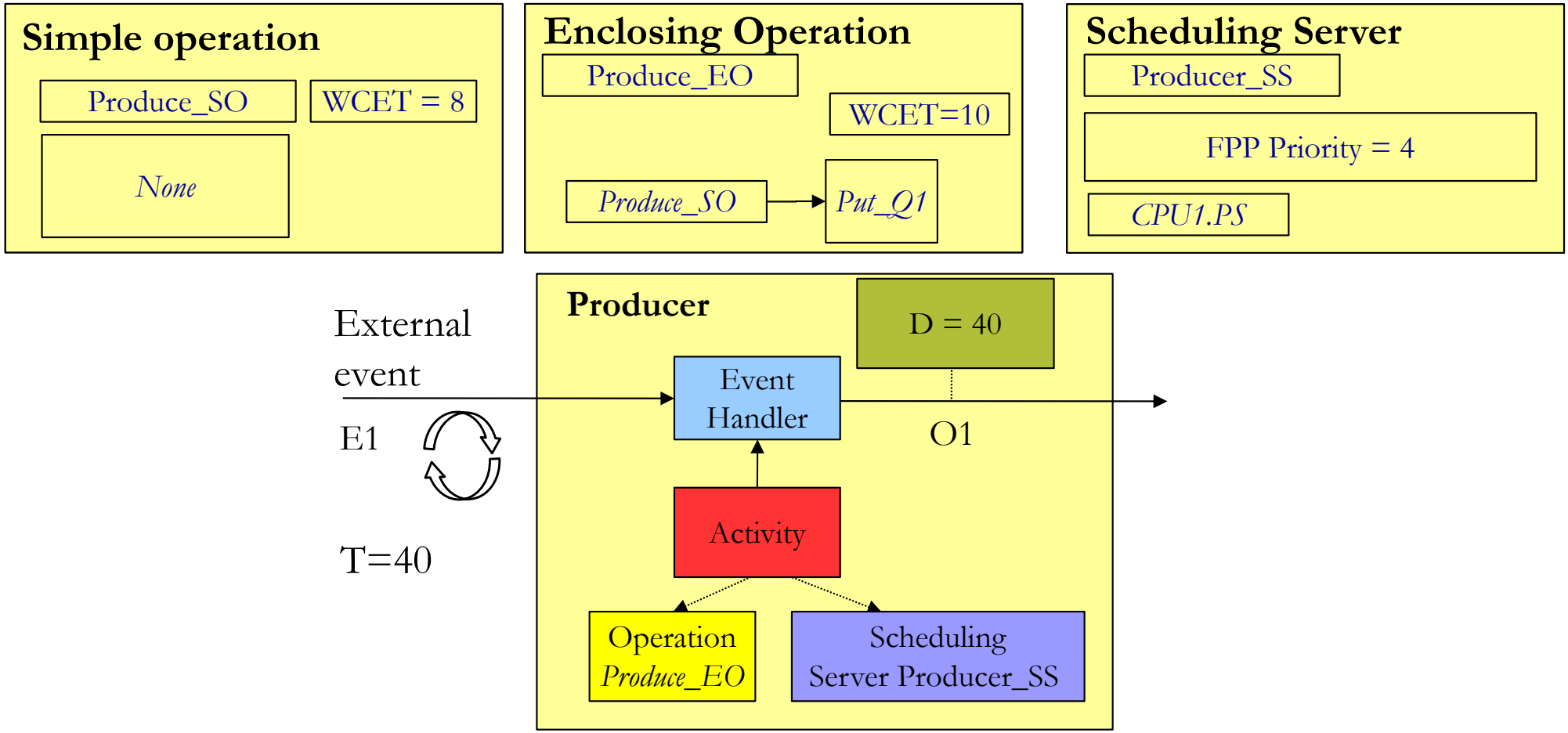
MAST: example shared resources



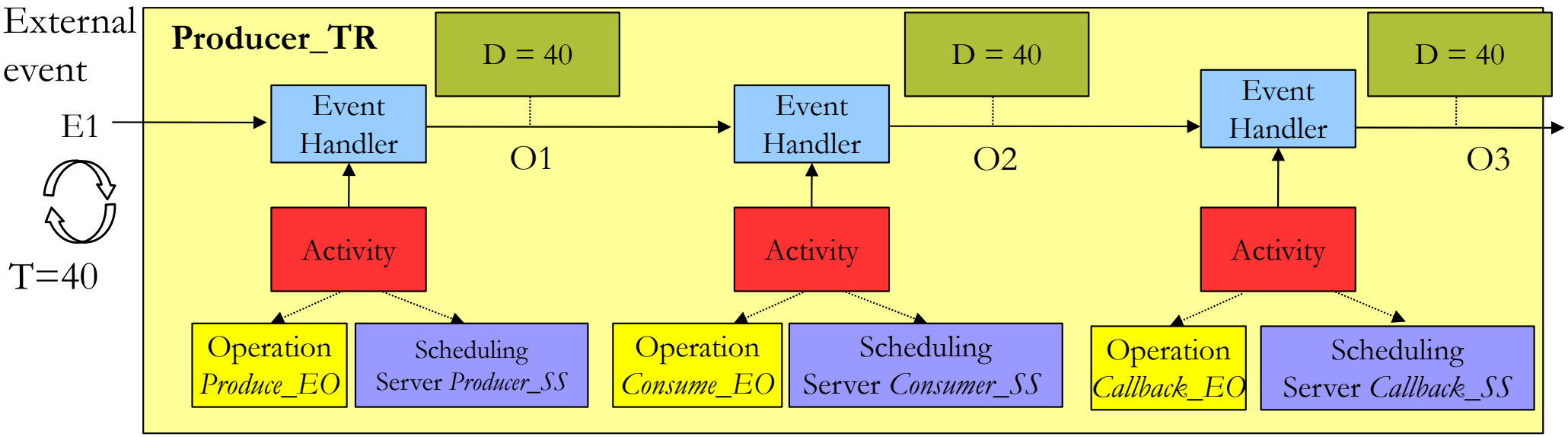




MAST: example tasks



MAST model of the call-back pattern



The call-back problem /4

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 priority $= max(P_1, P_2) = 4$

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

Classic RTA

$R_1 = 17$

$R_2 = 25$

$R_3 = 7$

Worst-case offset-based analysis

$R_1 = 12, O_1 = 0, J_1 = 0$

$R_2 = 20, O_2 = R_1^{best}, J_2 = R_1 - R_1^{best}$

$R_3 = 27$ ⟸ **Relative to the beginning of the transaction, $O_3 = R_2$, 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

- We have taken a deeper look into how "transactions" can help reason about end-to-end response time of concatenations
- We have seen how MAST can be used to model systems with transactions and analyze them with the worst-case dynamic offset (WCDO) technique developed by the University of Cantabria

6.b WCET analysis

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

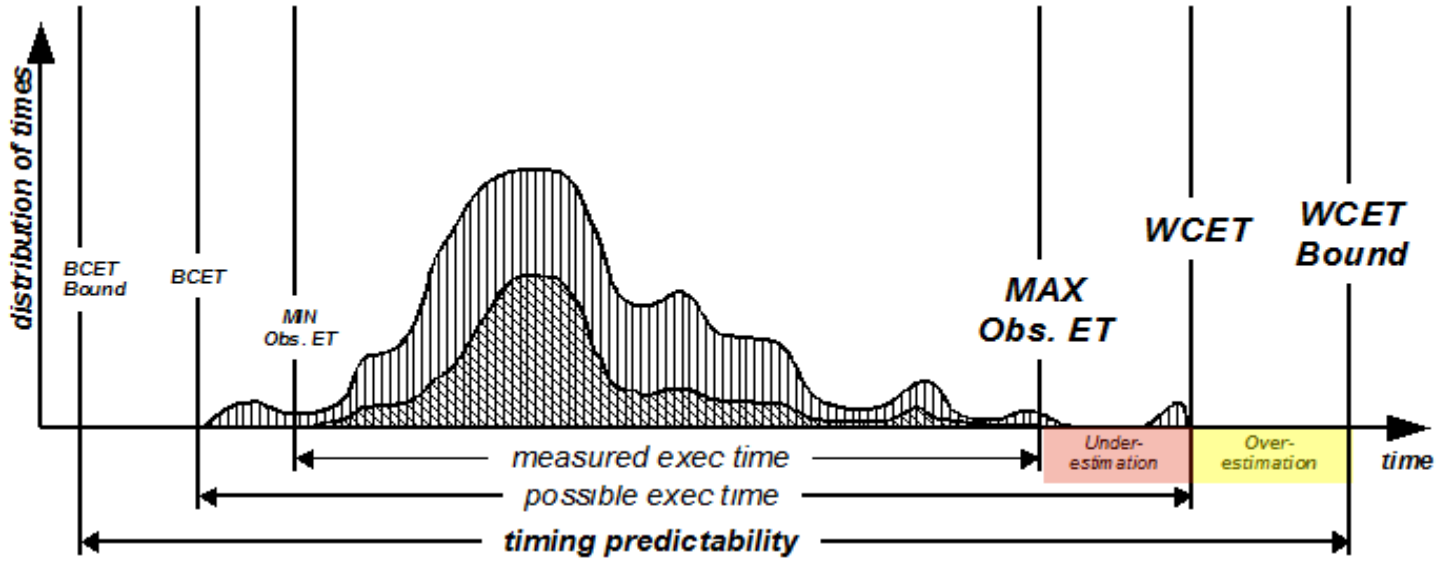
Where we learn how the worst-case execution-time (WCET) value used in response-time analysis can be determined, and explore the taxonomy of WCET analysis techniques

Worst-case execution time (WCET)

- Across all input data and all initial logical states
  - So that all *execution paths* of the program can be traversed
  - May be prohibitively difficult to determine and provide
- For any hardware state
  - So that the worst-case *execution conditions* are in effect on the processor
  - Increasingly hard on modern processors, full of access restrictions and of state-perturbing features
    - Out-of-order execution pipelines, caches, branch predictors, speculative execution
- Measurement-based WCET analysis
  - On either the real HW or a cycle-accurate simulator of it
  - The *high-watermark* value can be much smaller than the WCET !
- Static WCET analysis
  - Uses an abstract model of the HW and of the program

Computing the WCET

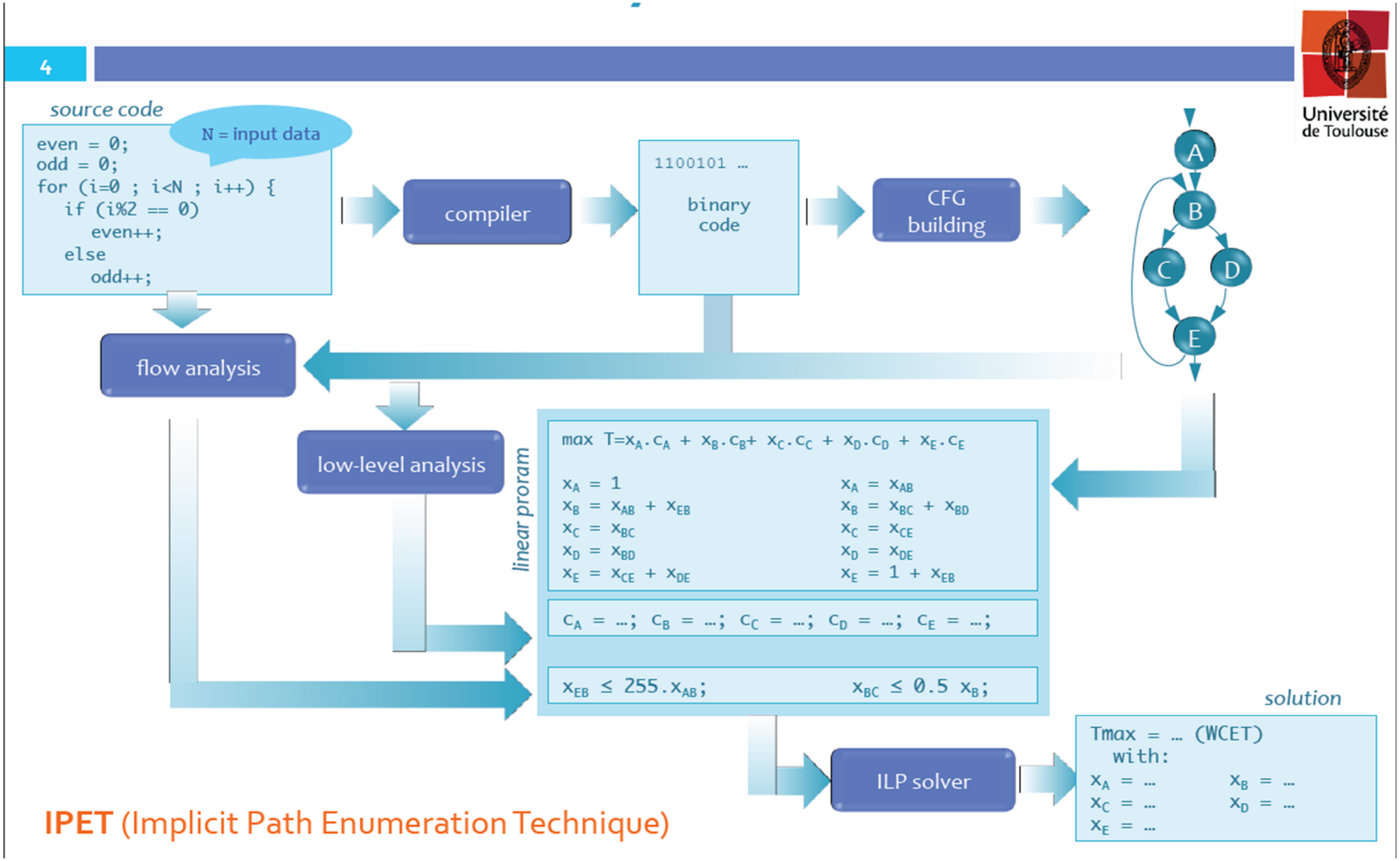
- The exact WCET is *not* generally computable
  - Another kind of halting problem
- Yet, WCET bounds are essential to feasibility analysis
  - Must be *safe*, to upper bound all possible executions
  - Must be *tight*, to avoid costly over-dimensioning



Static WCET analysis /1

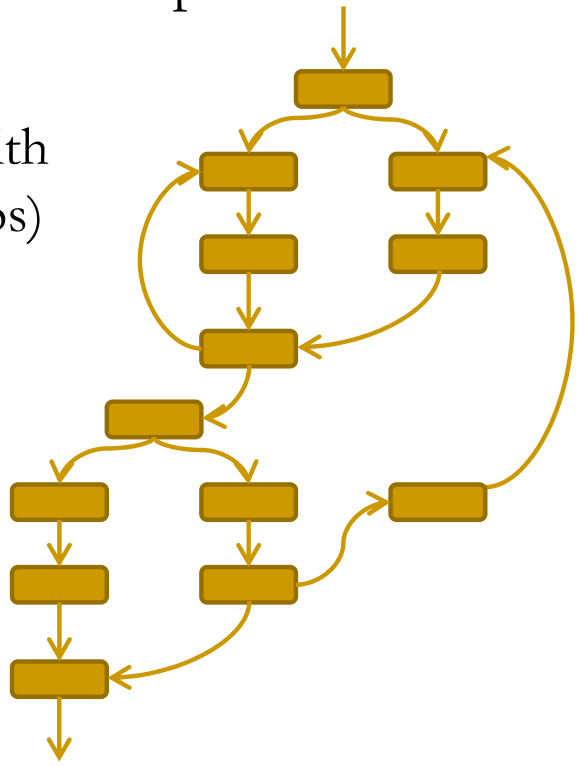
- To analyze a program without executing it
  - Needs an *abstract model* of the target HW
  - As well as the binary executable of the program
- Execution time depends on the program's control flow **and** on the fine-grained behavior of the HW
  - ***High-level analysis*** addresses program execution
    - *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



Static WCET analysis /3

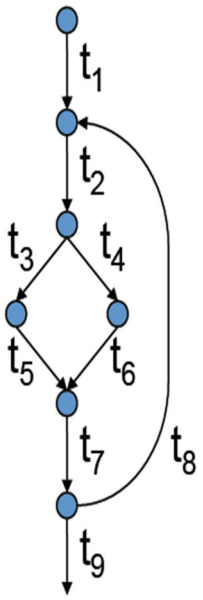
- ***High-level analysis*** /1
  - Must analyze all possible execution paths of the program
    - Builds the CFG as a superset of all possible execution paths
    - The unit of that analysis is the *basic block*
      - The longest sequence of program instructions with single entry and single exit (no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



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 : *execution frequency* of edge i
  - t_i : *execution time* of edge i
- While respecting the given flow-graph constraints

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 /4

- ***High-level analysis*** /2
  - Using the IPET requires employing several techniques
    - *Control-flow analysis* to construct the CFG
    - *Value analysis* to resolve memory accesses
    - *Data-flow analysis* to find loop bounds for graph constraints
  - 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

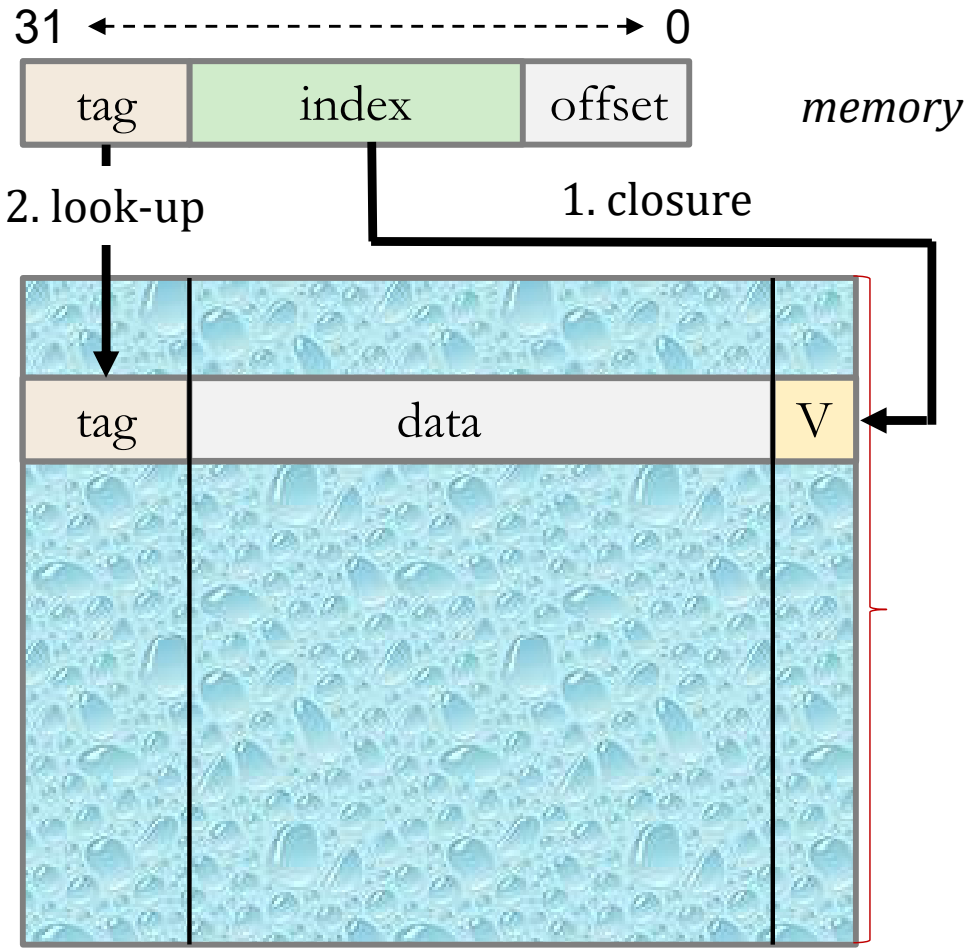
Static WCET analysis /6

- ***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

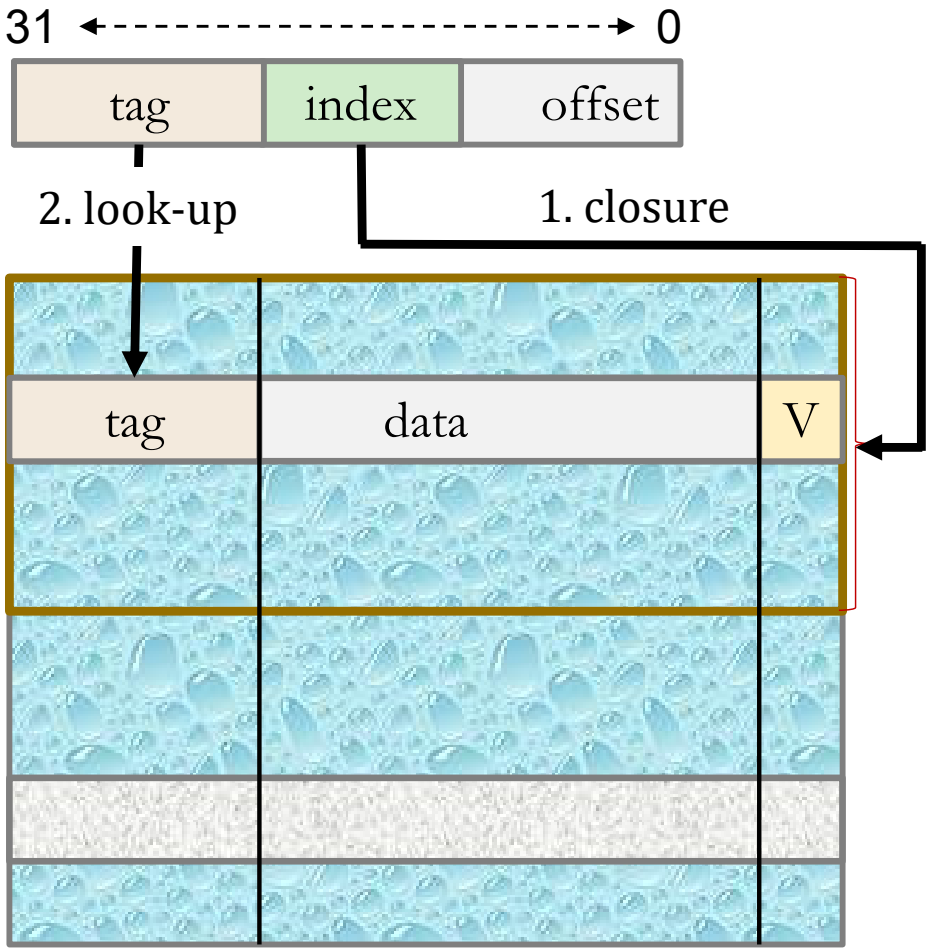
Understanding the cache

- Cache holds 2^k **lines** of $2^{n=6} = 64$ bytes each: this yields space locality
- RAM has 2^m addresses, for $m \gg n$
  - RAM address is read as [**tag** ($m - k - n$ bits): **index** (k bits) : **offset** (n bits)]
- Three strategies map RAM addresses to cache lines
  - Chance of access conflict as high as size of tag field
- ***Direct mapping***: each memory address maps to a *single cache line*
  - Index tells cache line, offset tells position in it, tag tells hit or miss
- ***N-way set associative***: each memory address maps to a *single cache set*
  - The cache is divided into **sets**, each holding $s = 2$ or 4 lines
  - Index tells cache set, offset tells position in cache line, tag tells hit or miss across the set: lower chance of access conflicts
- ***Fully associative***: each memory address maps to *any* cache line
  - *No* index field: tag-based matching search across the entire cache (very complex!)

Understanding the cache



memory

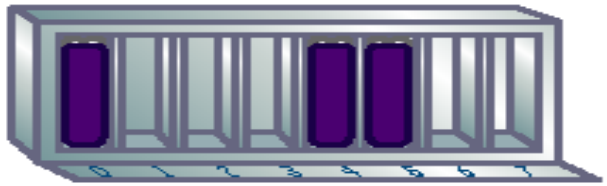


Direct mapping (by index)
Each memory address maps to a *unique* cache line:
the index field gives its placement;
the tag field tells match or miss

Set-associative mapping (by set)
Each memory address maps to a *set* of cache lines:
the index field tells the set;
the tag field tells match or miss across the set

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.

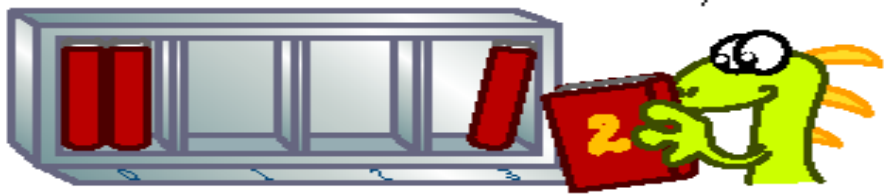
Direct Mapped



Tag	Index	Offset

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.

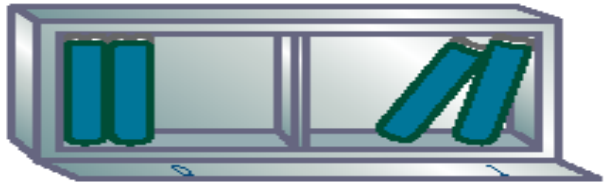
2-Way Set Associative



Tag	Index	Offset

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



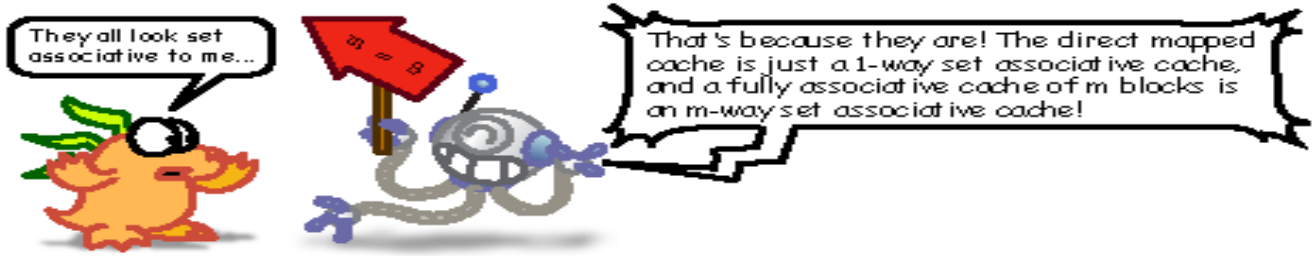
Tag	Index	Offset

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

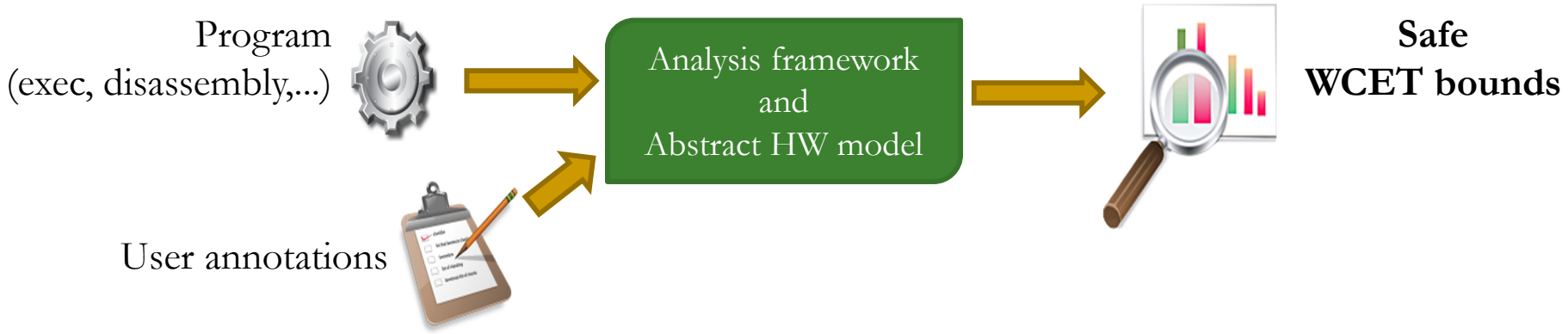
Fully Associative

Tag	Offset

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!



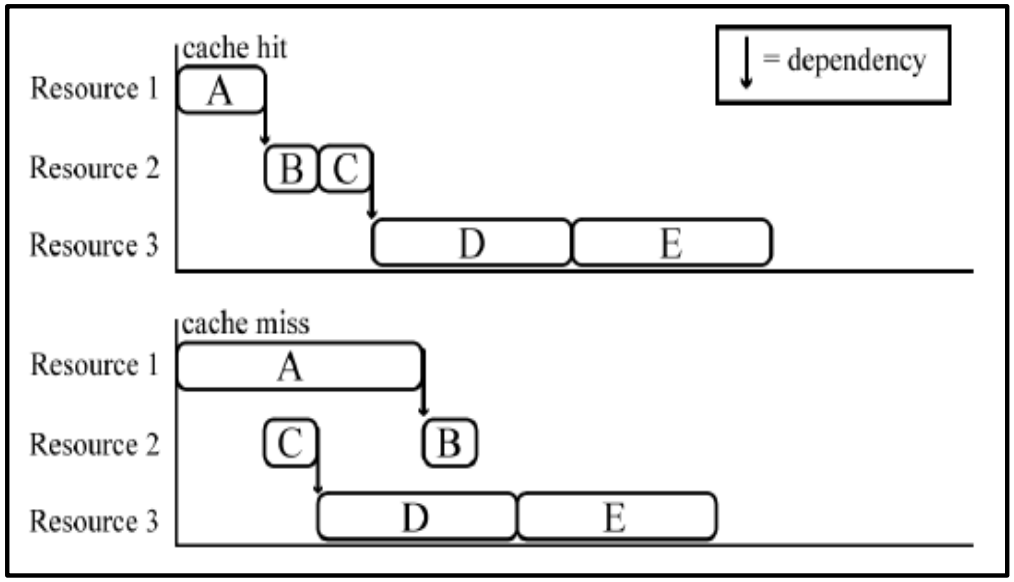
Static WCET analysis: the big picture



- Open problems
  - Can we trust the abstract model of the processor hardware?
  - 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
  - Are we free from *timing anomalies*?
    - When the *local* worst case does not lead to *global* worst case

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

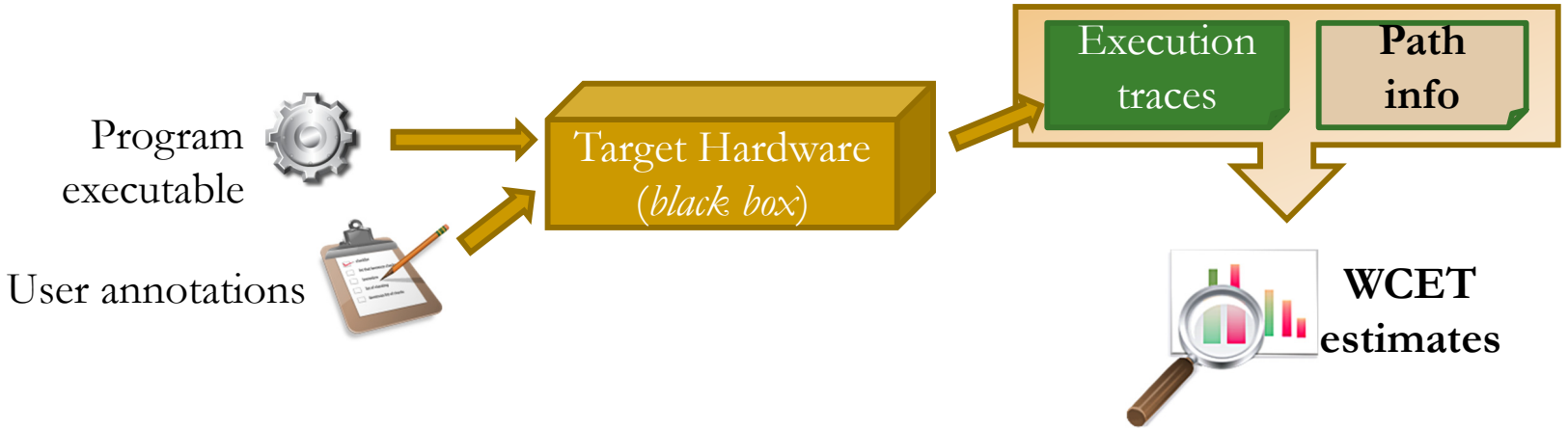
Hybrid analysis /1

- 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 observations and the consequent 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 be costly to have, subtly different, or too late to arrive

Summary

- We have reckoned with the challenge WCET analysis
- 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 *measurement-based* analysis is more pragmatic, but riskier: *hybrid* methods work are safer
- For HW with *deterministic* behavior, the uncertainty on the WCET bound is *epistemic*: we may not know enough
- For HW with *randomized* behavior, knowledge is *aleatory*

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