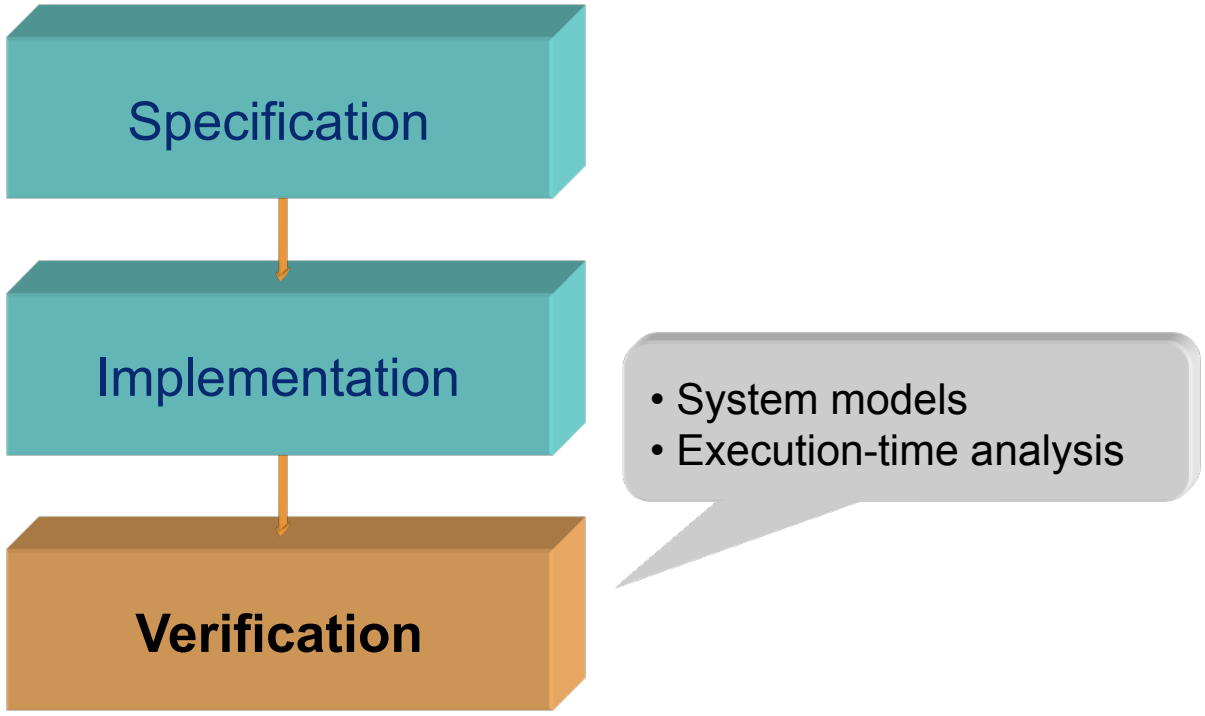
Real-Time Systems

Lecture #7

Professor Jan Jonsson

Department of Computer Science and Engineering
Chalmers University of Technology

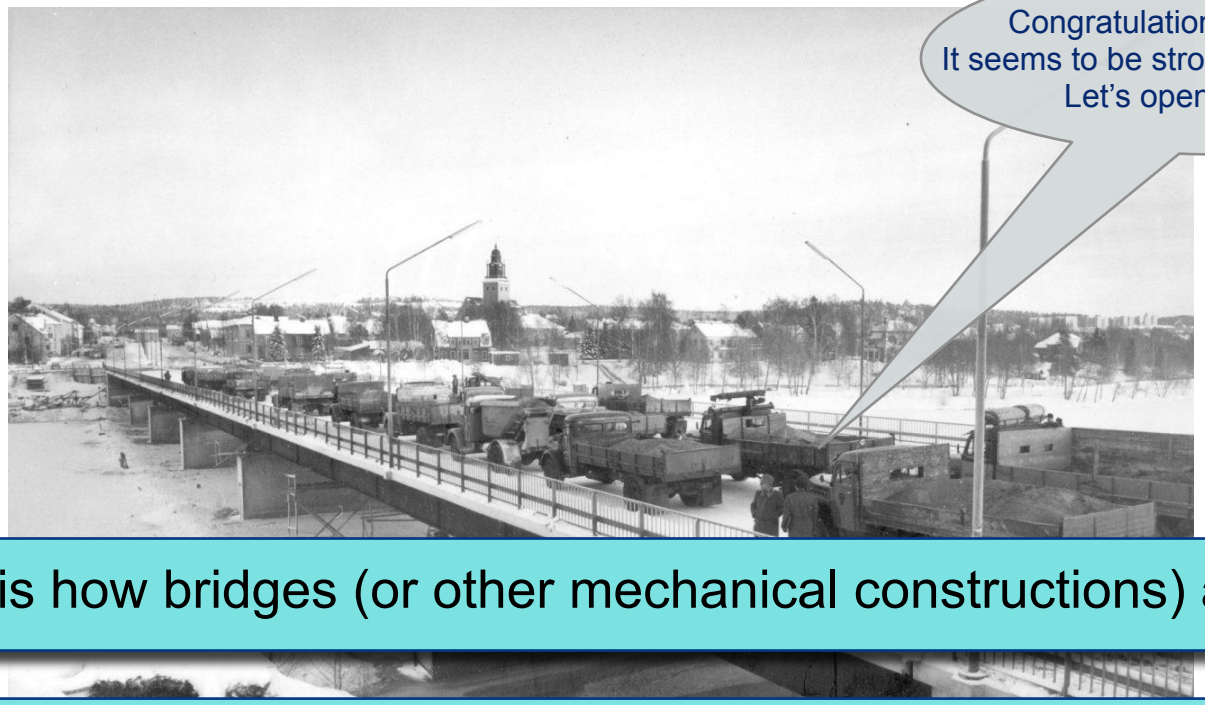
Real-Time Systems



Verification by testing

Free translation from Swedish by J. Jonsson

Verification by testing

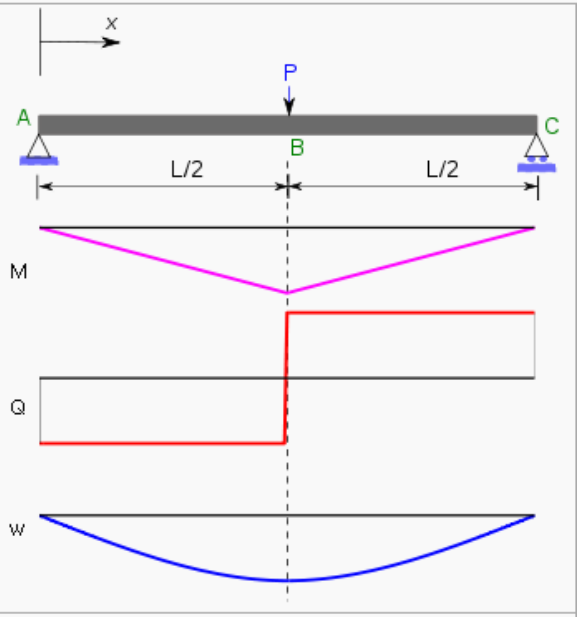


So, is this how bridges (or other mechanical constructions) are built?

Of course not! There are models (properties of materials) and theories (laws of mechanics) involved to determine in advance that a construction will withstand the predicted load.

Verification by models & theory

Distribution	Max. value
Simply supported beam with central load	
$M(x) = \begin{cases} \frac{Px}{2}, & \text{for } 0 \le x \le \frac{L}{2} \\ \frac{P(L-x)}{2}, & \text{for } \frac{L}{2} < x \le L \end{cases}$	$M_{L/2} = \frac{PL}{4}$
$Q(x) = \begin{cases} \frac{P}{2}, & \text{for } 0 \le x \le \frac{L}{2} \\ \frac{-P}{2}, & \text{for } \frac{L}{2} < x \le L \end{cases}$	$\lvert Q_0 \rvert = \lvert Q_L \rvert = \frac{P}{2}$
$w(x) = \begin{cases} -\frac{Px(4x^2-3L^2)}{48EI}, & \text{for } 0 \le x \le \frac{L}{2} \\ \frac{P(x-L)(L^2-8Lx+4x^2)}{48EI}, & \text{for } \frac{L}{2} < x \le L \end{cases}$	$w_{L/2} = \frac{PL^3}{48EI}$



So, why cannot computer systems be built and verified in advance using models and theories?

Well, they can … using system models and schedulability analysis

Verification

How do we perform schedulability analysis?

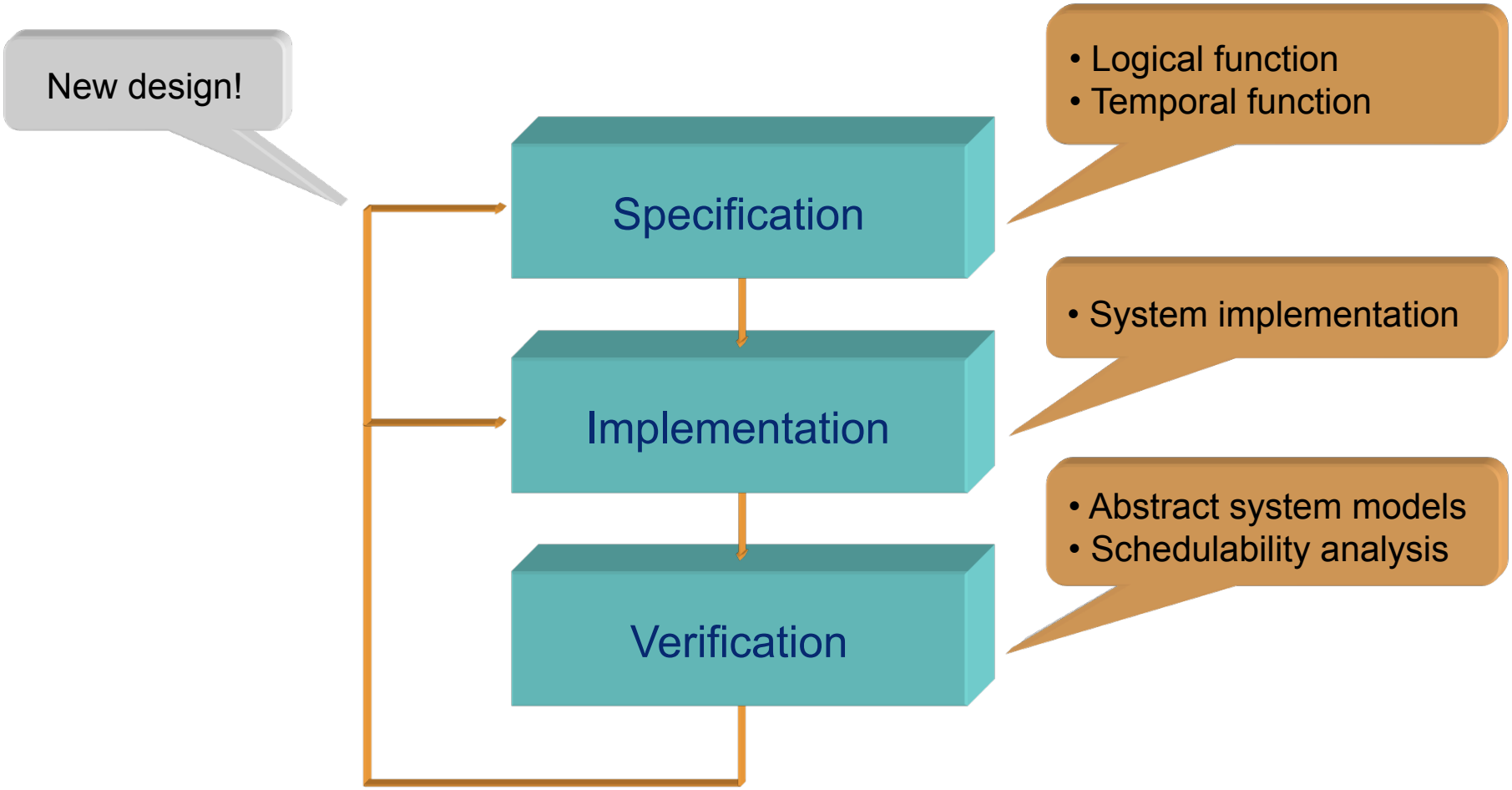
- Introduce abstract models of system components:
  - Task model (computation requirements, timing constraints)
  - Processor model (resource capacities)
  - Run-time model (task states, dispatching)
- Predict whether task executions will meet constraints
  - Use timing-correct abstract system models
  - Make sure that computation requirements never exceed resource capacities
  - Generate a (partial or complete) run-time schedule resulting from task executions and detect worst-case scenarios

Verification

How do we simplify schedulability analysis?

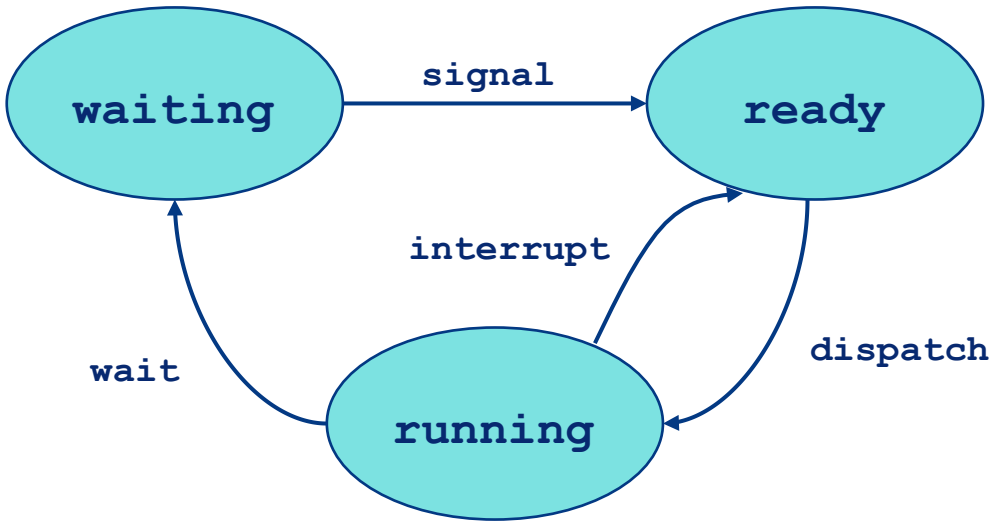
- Concurrent and reactive programming paradigm
  - Suitable schedulable entity (thread, method, …)
  - Language constructs for expressing application constraints for schedulable entities (data types, annotations, macros, …)
  - Estimated WCET for schedulable entities
- Deterministic task execution
  - Time tables or static/dynamic task priorities
  - Preemptive task execution
  - Run-time protocols for access to shared resources (dynamic priority adjustment and non-preemptable code sections)

Designing a real-time system



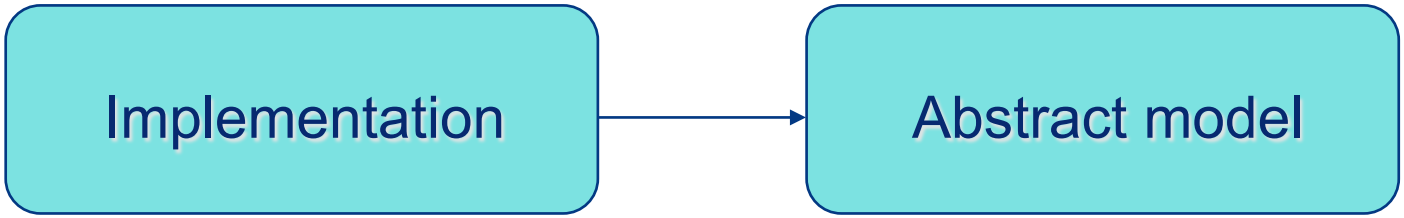
Run-time model

The run-time model expresses the state of a task:



Running: Currently executing task
Ready: Task that is available for execution
Waiting: Task that cannot execute because it is needs access to a resource other than the processor

Task model

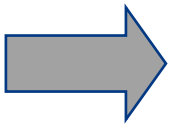



```
void task1(Object *self, int p) {
    Action1();
    SEND(Period1, Deadline1, self, task1, p);
}

void task2(Object *self, int p) {
    Action2();
    SEND(Period2, Deadline2, self, task2, p);
}

void kickoff(Object *self, int p) {
    AFTER(Offset1, &app1, p);
    AFTER(Offset2, &app2, p);
}

main() {
    TINYTIMBER(&app_main, kickoff, 0);
}
```



τ_1 $\quad \tau_1 = \{ C_1, T_1, D_1, O_1 \}$

τ_2 $\quad \tau_2 = \{ C_2, T_2, D_2, O_2 \}$

Task model

The task model expresses the timing behavior of a task:

- The static parameters describe characteristics of a task that apply independent of other tasks.
  - These parameters are derived from the specification or the implementation of the system
  - For example: period, deadline, WCET
- The dynamic parameters describe effects that occur during the execution of a task.
  - These parameters are a function of the run-time system and the characteristics of other tasks
  - For example: start time, completion time, response time

Task model

Static task parameters:

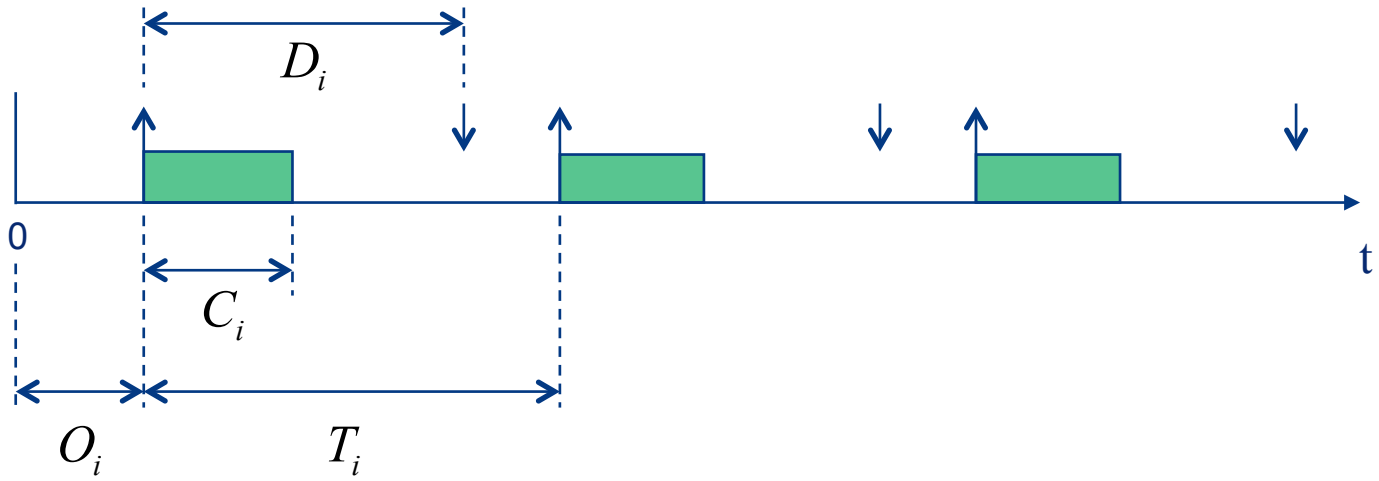
τ_i $\quad \tau_i = \{ C_i, T_i, D_i, O_i \}$

C_i : (undisturbed) WCET

T_i : period

D_i : (relative) deadline

O_i : (absolute) time offset



Task model

Dynamic task parameters:

τ_i $\quad \tau_i = \{ C_i, T_i, D_i, O_i \}$

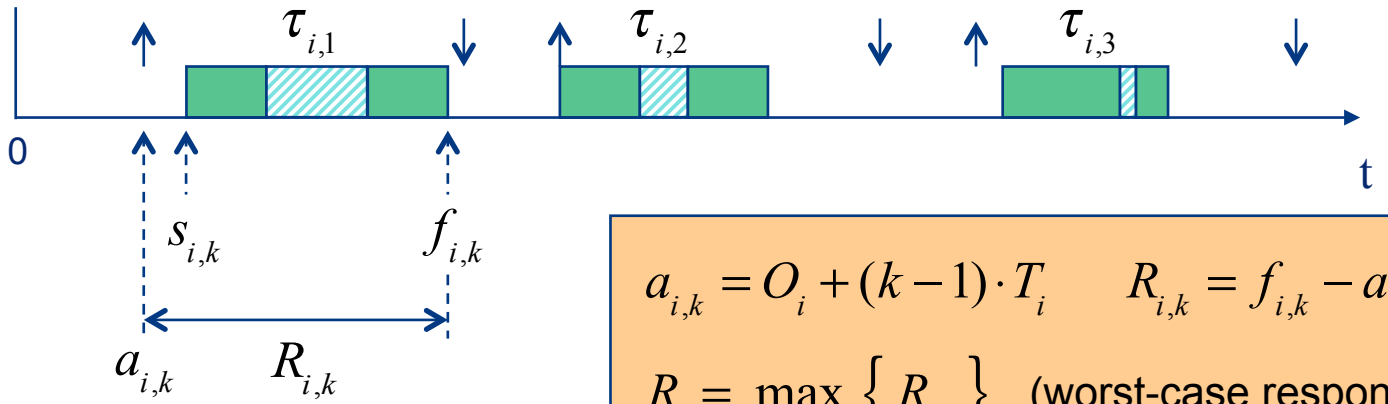
$a_{i,k}$: arrival time of k^{th} instance

$s_{i,k}$: start time of k^{th} instance

$f_{i,k}$: completion time of k^{th} instance

$R_{i,k}$: response time of k^{th} instance

$\tau_{i,k}$: the k^{th} instance of τ_i



$$a_{i,k} = O_i + (k-1) \cdot T_i \qquad R_{i,k} = f_{i,k} - a_{i,k}$$

$$R_i = \max_{\tau_i \in \mathbf{T}, k \geq 1} \{ R_{i,k} \}$$ (worst-case response time)

Task model

Synchronous and asynchronous task sets:

- In a *synchronous* task set the offsets of tasks are identical, that is: $\forall i, j : O_i = O_j$
- In an *asynchronous* task set the offsets of at least one pair of tasks are not identical, that is: $\exists i, j : i \neq j, O_i \neq O_j$

Asynchronous task sets are typically used to reduce local skew (jitter) or to remove the need for resource access protocols.

Note: Two tasks with identical periods, but different offsets, will never arrive simultaneously during the lifetime of the system. This means that the worst-case response times of the tasks will be lower than if the offsets of the tasks were equal.

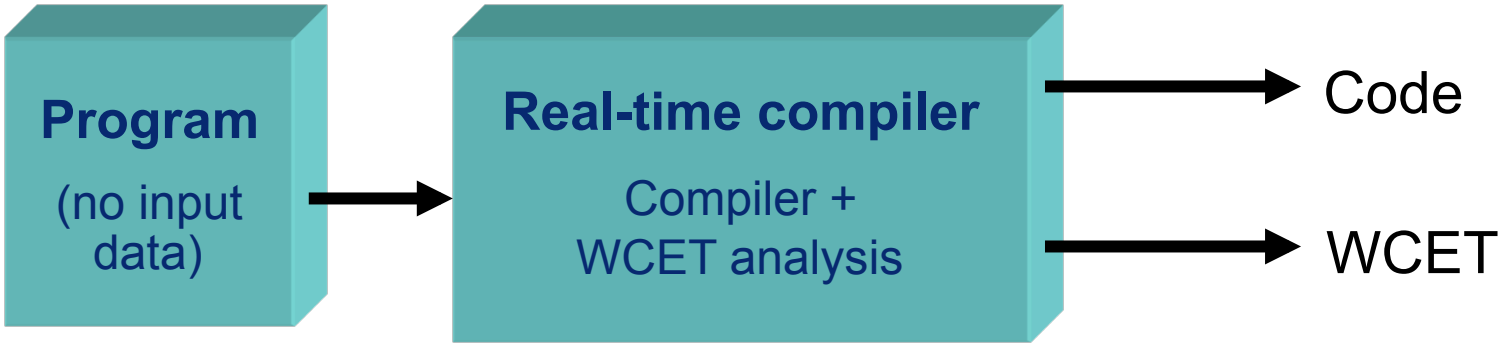
Task model

Task arrival patterns:

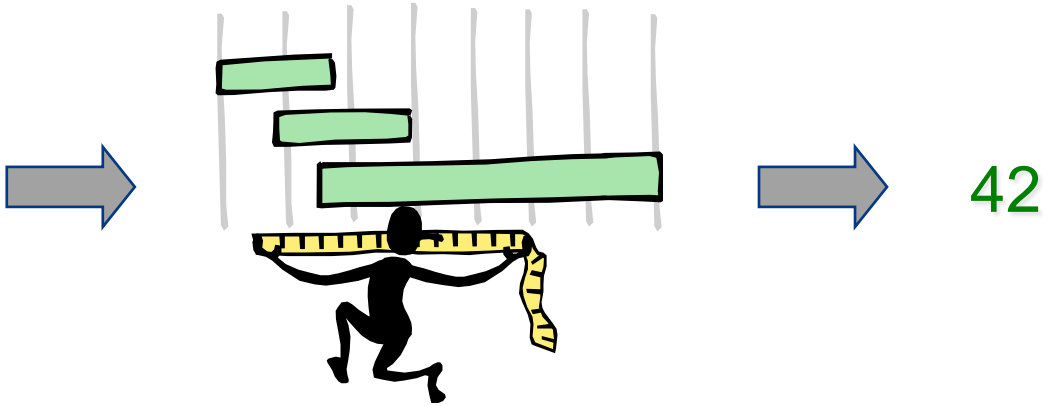
- Periodic tasks
  - A periodic task arrives with a time interval T_i
- Sporadic tasks
  - A sporadic task arrives with a time interval $\geq T_i$
- Aperiodic tasks
  - An aperiodic task has no guaranteed minimum time between two subsequent arrivals

⇒ *A priori* schedulable (hard) real-time systems can only contain periodic and sporadic tasks.

Execution-time analysis




```
for (i=1; i<=N; i++) {
  if (A > K)
    A = K-1;
  else
    A = K+1;
  if (A < K)
    A = K;
  else
    A = K-1;
}
```



Execution-time analysis

Background:

- Worst-case execution time (WCET) is needed to
  - perform (hard) schedulability analysis
  - identify resource needs early in the design phase
  - perform program tuning (critical loops and interrupt handlers)
- The WCET of a task depends on
  - program structure + initial system state + input data
  - temporal properties of the system (OS + hardware)
  - internal and external system events
- WCET estimates can be obtained via
  - measurements
  - static analysis

Execution-time analysis

Requirements:

- A WCET estimate must be pessimistic but tight

  0 ≤ ”Estimated WCET” – “Real WCET” < ε

  (ε small compared to real WCET)

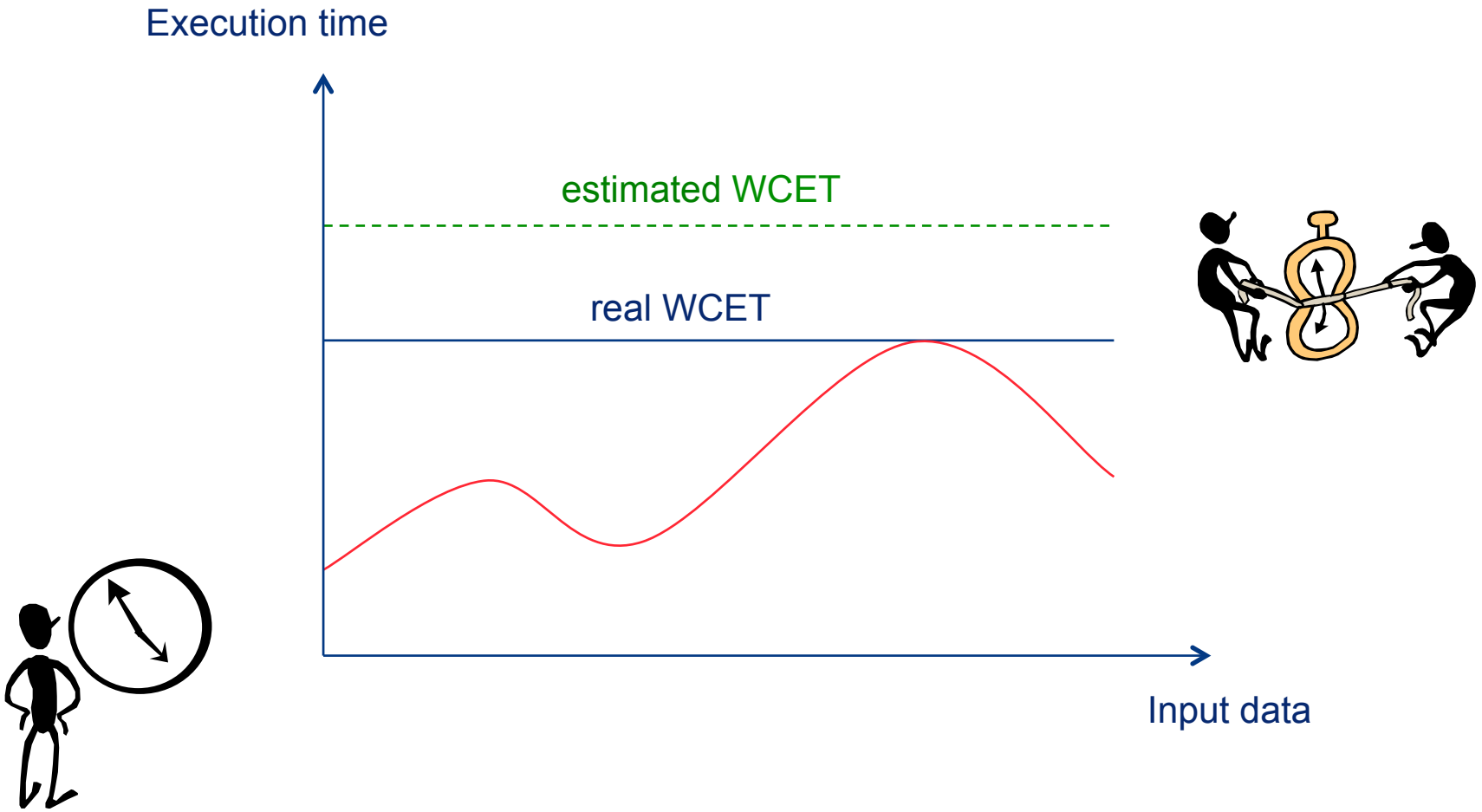
Pessimistic:

to make sure assumptions made in the schedulability analysis of hard real-time tasks also apply at run time

Tight:

to avoid unnecessary waste of resources during scheduling of hard real-time tasks

Execution-time analysis



Execution-time analysis

Estimating WCET via measurements:

- Methodology:
  - identify potential worst-case scenario
  - run program code on hardware using worst-case scenario
  - measure the execution time
  - add a safety margin
- Measuring techniques:
  - system clocks, cycle-level simulators, in-circuit emulators
  - observe hardware signals with oscilloscope or logic analyzer
- Reflection:
  - measured execution time will never exceed real WCET
  - how large must safety margin be to get a pessimistic estimate?

Execution-time analysis

Estimating WCET via static analysis:

- Methodology:
  - determine the longest execution time of the program code without actually running it
  - uses models based on properties of software and hardware
  - typically integrated with the compiler tools
- Analysis techniques:
  - Path analysis: bound the number of times that different program parts may be executed
  - Timing analysis: bound the execution time of program parts
- Reflection:
  - real WCET will never exceed estimated execution time
  - how accurate must the models be to get a tight estimate?

A simple (yet challenging) example

Derive WCET for the following program:

```
for (i=1; i<=N; i++) {
  if (A > K)
    A = K-1; (T1)
  else
    A = K+1; (E1)
  if (A < K)
    A = K;   (T2)
  else
    A = K-1; (E2)
}
```

Issues to consider:

- Input data is unknown
  - Iteration bounds must be known to facilitate analysis
- Path explosion
  - 4^N paths in this example
- Exclusion of non-executable (false) paths
  - T1 + E2 is a false path in the example

A simpler (but non-trivial) example

Derive WCET for the following statement:

```
A = A / B;
```

Issues to consider:

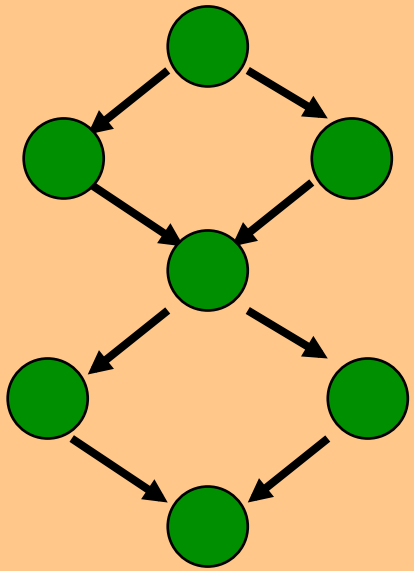
- Execution time:
  - affected by cache misses, pipeline conflicts, exceptions ...
  - depends on previous and (!) subsequent instructions
  - also depends on (unknown) input data
- Observations:
  - accurate estimation of WCET must be based on a detailed timing model of the system architecture
  - uncertainties are handled by making worst-case assumptions

Fundamental issues

- In the path analysis:
  - how to bound the number of iterations in a loop / recursion
  - how to eliminate false (non-executable) paths cause by e.g. if-then-else statements
- In the timing analysis:

  Everything that takes time must be modeled in a realistic fashion (or at least not optimistically)
  - must accurately model the temporal behavior of hardware (influence of, e.g., cache memories, pipelining, …)
  - must account for consequences of run-time events (e.g.: exceptions, interrupts, context switches)

Path analysis



A control flow graph (CFG) describes the structure of the program

Path analysis problem:

Find the longest executable path in the program's CFG

Path analysis

Shaw's Timing Schema (1989):

```
for (i=1; i<=N; i++) {
  if (A > K)
    A = K-1; (T1)
  else
    A = K+1; (E1)
  if (A < K)
    A = K;   (T2)
  else
    A = K-1; (E2)
}
```

The estimated WCET (WCETe) is the execution time of the longest structural path through the program

```
WCETe =
 N*(WCET(loop) +
 WCET(I1) +
 max(WCET(T1), WCET(E1)) +
 WCET(I2) +
 max(WCET(T2), WCET(E2)))
```

Methods for path analysis

Manual method:

Programmer must provide information

Annotation of loop bounds:

- Provide upper bounds on loop indices and catch potential exceptions at run time

Elimination of false paths:

- Enumerate all possible paths and list the set of false paths so that these can be avoided in the analysis

Requires very detailed knowledge of the program's function, and is therefore also very prone to errors!

Methods for path analysis

Automated method:

Support from the compiler

Derive upper bounds on loop indices:

- Requires an explicit loop index
- May not work for complicated termination conditions

Elimination of false paths:

- Symbolically execute the program to detect non-executable program statements

Current methods are promising but only for fairly simple programs where the analysis is trivial!

Methods for path analysis

The reality?

Shaw's timing schema implicitly assume that the execution time of each language statement is constant and known

This is a realistic assumption for older types of processors, that:

- lack execution pipelines
- lack cache memories
- do not generate exceptions

However, for the RISC type processor architectures, these methods yield very pessimistic results!

Timing analysis for RISC processors

RISC processors have several advanced mechanisms (pipelining, caching, branch prediction, out-of-order execution, …) that cause significant variation in the execution time of a processor instruction.

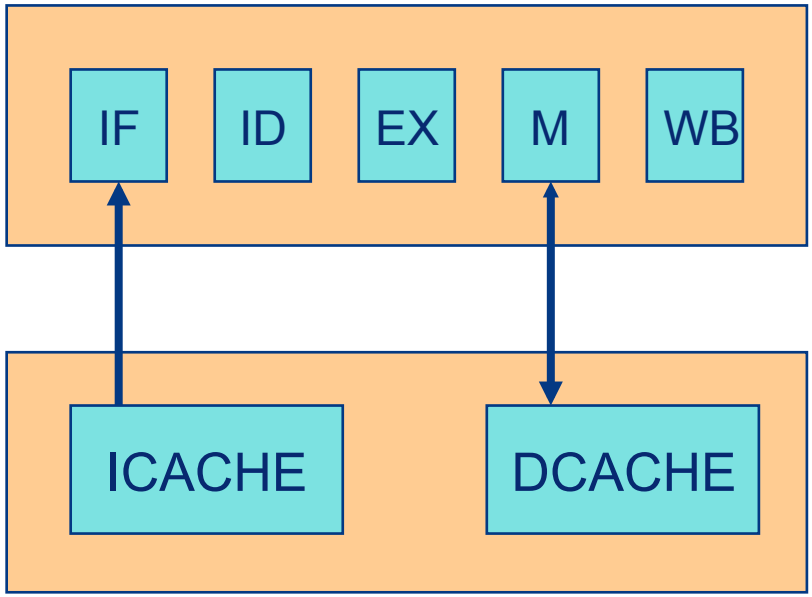
We must therefore estimate the execution time for each executable path through the program and at the same time account for these mechanisms.

This can be solved by partitioning the program code into code blocks and analyze each block separately.

Today, mature methods for timing analysis only exist for pipelining and caching.

Timing analysis for RISC processors

Processor with pipeline:



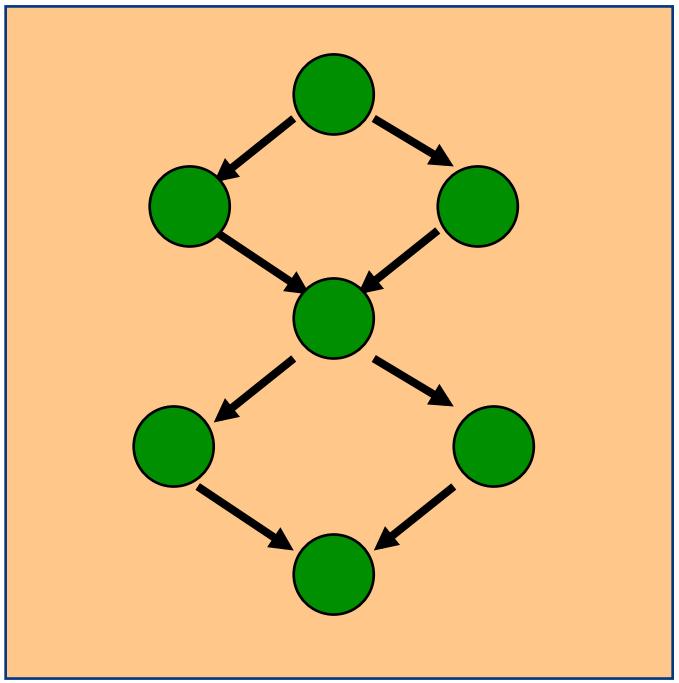
Sources of time variations:

- data conflicts
- branch conflicts

Sources of time variations:

- cache misses
  (have order-of-magnitude higher access times than cache hits)

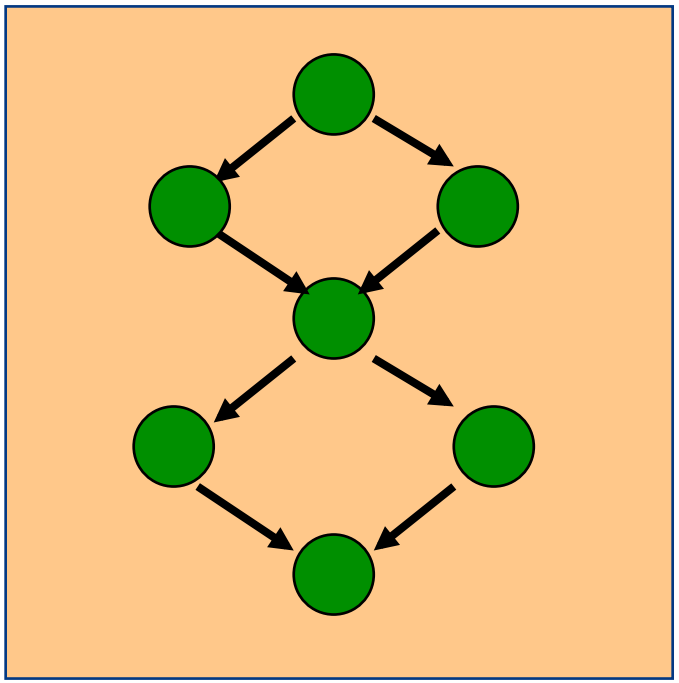
Timing analysis of cache memory



Issues:

- Not enough to investigate an isolated code block
  - miss/hit depends on previous executions of the code
- Instruction cache behavior is predictable for each path
  - known sequence of code
- Data cache behavior is more difficult to analyze
  - data addresses can depend on the program's input data

Timing analysis of pipeline



Issues:

- Not enough to investigate an isolated code block
  - conflicts may occur on the boundary between code blocks
- Pipeline behavior is predictable for each path
  - known sequence of code

Methods for timing analysis

Extension of Shaw's Timing Schema

- Analysis is performed at code block level
- Merging of paths at certain code locations by estimating the effects of worst-case situations (reduces path explosion)

Data flow analysis:

- Analysis performed at code block level
- Propagation of pipeline and cache states between blocks

Integer Linear Programming

- Formulate an ILP problem as a function of execution time and number of executions at code block level

Challenges

So far, non-preemptive execution of program code on a single processor has been assumed.

In reality, pseudo-parallel execution is typically used, something which requires preemptive execution.

- Preemptions will affect system state (i.e., cache contents will change and pipeline will be flushed) and must therefore be accounted for in the analysis.
- However, it is difficult to account for these effects in the analysis of WCET, which means that it must be handled at a higher level (i.e., in the schedulability analysis).

Challenges

So far, non-preemptive scheduling of program code on a single processor has been assumed.

In reality, multicore processors are used in real-time systems, something which presents new problems:

- Several processors may have copies of the same code and data in their local cache memories, and any updates will invalidate the other copies. This must be accounted for in the analysis.
- ...