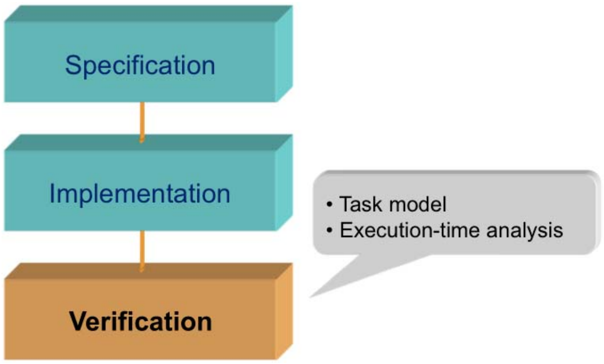
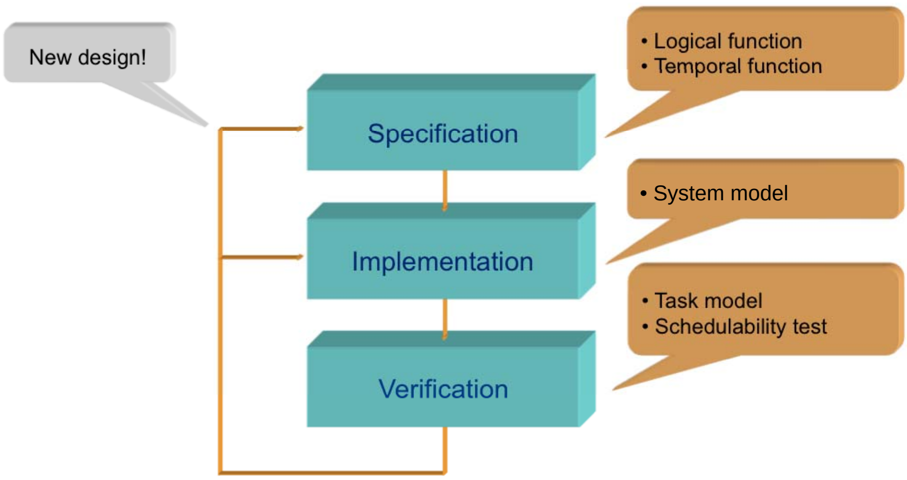
CHALMERS

Real-Time Systems



CHALMERS

Designing a real-time system



CHALMERS

Task model

Implementation → Abstract model

```
task body P1 is
   Interval : constant Duration := 5.0;
   Next_Time : Time;
begin
   Next_Time := Clock + Interval;
   loop
      Action;
      delay until Next_Time;
      Next_Time := Next_Time + Interval;
   end loop;
end P1;

task body P2 is
   Interval : constant Duration := 7.0;
   Next_Time : Time;
begin
   Next_Time := Clock + Interval;
   loop
      Action;
      delay until Next_Time;
      Next_Time := Next_Time + Interval;
   end loop;
end P2;
```

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

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

CHALMERS

Task model

A task model must be defined to be able to analyze the temporal behavior of a set of tasks.

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

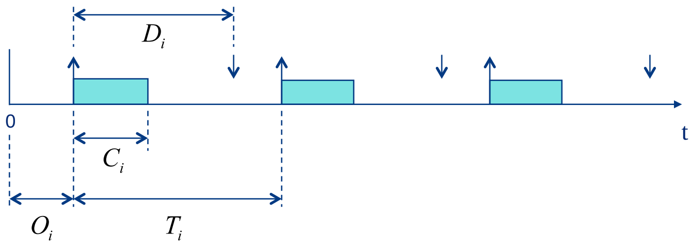
CHALMERS

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



CHALMERS

Task model

Static task parameters:

C_i Task's worst-case execution time (WCET)

- Represents the longest undisturbed execution time for one iteration of the task
- Derived as a function of the task's program code

D_i Task's relative deadline (responsiveness constraint)

- Represents the maximum allowed time within which the task must complete its execution
- Applies relative to the time when the task becomes executable
- Derived as a function of the environment (e.g., laws of nature, control theory, ...)

CHALMERS

Task model

Static task parameters:

T_i Task's periodicity

- Represents how often the task should be repeated
- Each iteration of the task has the same WCET

O_i Task's time offset

- Represents the <u>first</u> arrival time of the task, e.g., the earliest time instant at which the task becomes executable
- Applies relative to a given "origin" of the system

The <u>arrival time</u> of the n:th iteration of a task then becomes

$$A_i^n = O_i + (n-1) \cdot T_i$$

CHALMERS

Task model

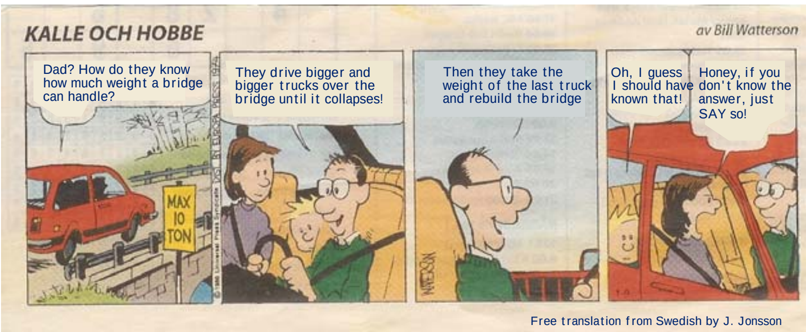
Different types of tasks:

- 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

⇒ **Hard real-time systems can only contain periodic and sporadic tasks.**

CHALMERS

The importance of models

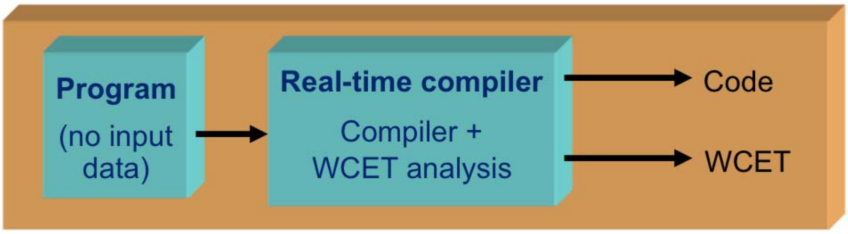


CHALMERS

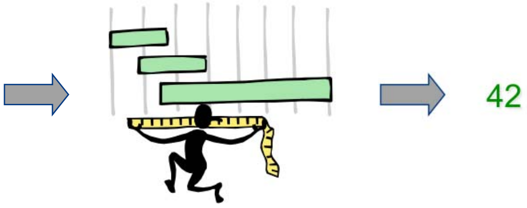
(this page deliberately left blank)

CHALMERS

Execution-time analysis




```
for I:=1 to N loop
begin
  if A > K
    then A:=K-1;
    else A:=K+1;
  if A < K
    then A:=K;
    else A:=K-1;
end;
```



CHALMERS

Execution-time analysis

Motivation:

- Worst-case execution time (WCET) is important since
  - it is a prerequisite for (hard) schedulability analysis
  - resource needs should be estimated early in the design phase
- The execution time of a task depends on
  - program structure + input data
  - initial system state
  - temporal properties of the system (OS + hardware)
  - internal and external system events

Estimation of WCET should consequently be made while the program is compiled!

CHALMERS

Execution-time analysis

Requirements:

- WCET must be pessimistic but tight

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

  (ε small compared to real WCET)

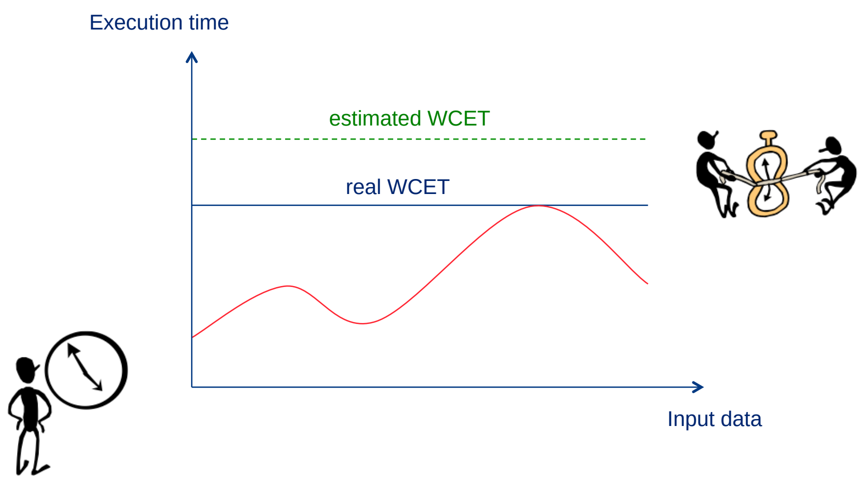
  pessimistic:
  to make sure assumptions made in the schedulability analysis of hard real-time tasks also apply at run time

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

- The computational complexity of the analysis method must be tractable

CHALMERS

Execution-time analysis



CHALMERS

A simple (yet challenging) example

Derive WCET for the following program:

```
for I:=1 to N loop
begin
  if A > K
    then A:=K-1;  (T1)
    else A:=K+1;  (E1)
  if A < K
    then A:=K;    (T2)
    else A:=K-1;  (E2)
end;
```

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

CHALMERS

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

CHALMERS

Formulation of the WCET problem

Given a system
(= program structure + system platform)
find the program's "worst-case" execution time
for all possible input data, initial system states
and (internal and external) system events

CHALMERS

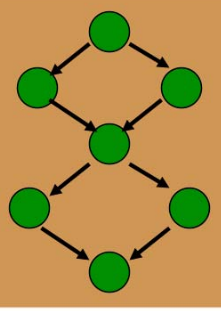
Fundamental issues

- Issues in the analysis of program paths
  - how to limit WCET (if necessary, pessimistically)
  - how to eliminate false paths (in order to derive a tight WCET estimate)
- Issues in the analysis of temporal behavior

  "everything that takes time must be modeled in a realistic fashion (or at least not optimistically)"
  - accurate and effective timing model of the system platform
    (influence of, e.g., cache memories, pipelining, …)
  - consequences of system events at run time
    (e.g.: exceptions, interrupts, context switches)

CHALMERS

Path analysis



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

Timing analysis problem:
Find the longest executable path in the program's CFG

- CFG may not contain cycles
- Non-executable paths must be eliminated

CHALMERS

Path analysis

Shaw's Timing Schema (1989):

```
for I:=1 to N loop
begin
  if A > K          (I1)
    then A:=K-1;  (T1)
    else A:=K+1;  (E1)
  if A < K          (I2)
    then A:=K;    (T2)
    else A:=K-1;  (E2)
end;
```

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

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

CHALMERS

Methods for path analysis

Branches (alternative paths) introduces the following set of problems:

1. Iterations (loops, recursions …)
2. Alternative (if-then-else, case …)

Goal:

- Bound the number of iterations in a loop or recursion
- Eliminate non-executable (false) program paths

CHALMERS

Methods for path analysis

The user annotates the program so that its CFG only contains a limited number of executable paths:

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, but is therefore also very prone to errors!

CHALMERS

Methods for path analysis

Automated method:

Static analysis (embedded in compiler):

- Derive upper bounds on loop indices
  - requires an explicit loop index
  - does not always work for complicated termination conditions
- Eliminate false paths
  - symbolically execute the program and do ”assert” with respect to the possible values that variables are able to assume

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

CHALMERS

Methods for path analysis

The reality?

Existing methods implicitly assume that the execution time of each language statement is constant and known

- This is a quite realistic assumption for a micro-controller that
  - lacks pipelined execution
  - lacks cache memories
  - does not generate exceptions

However, for modern processor architectures (= RISC), these methods yield very pessimistic results!

CHALMERS

Timing analysis for modern processors

Modern processors have several advanced mechanisms (e.g., 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.

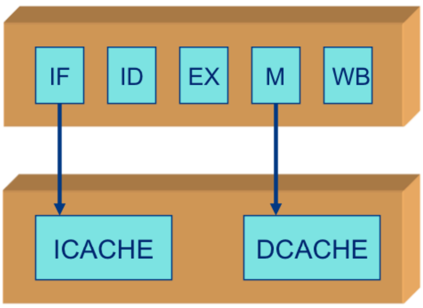
This can be solved by partitioning the program code into code blocks and analyze each block separately.

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

CHALMERS

Timing analysis for modern processors

Processor with pipeline:



Sources of time variations:

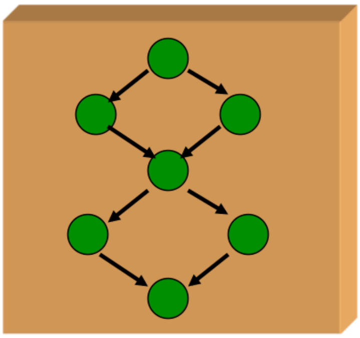
- structural conflicts
- data conflicts
- branch conflicts

Sources of time variations:

- cache misses

CHALMERS

Timing analysis of cache memory

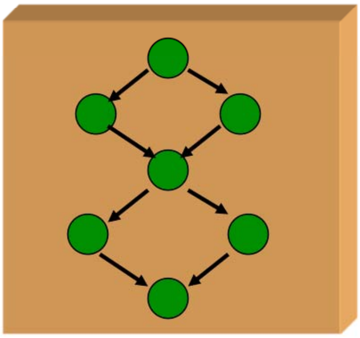


Issues:

- 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

CHALMERS

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

CHALMERS

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

CHALMERS

Challenges

So far, non-preemptive scheduling of program code has been assumed (which is not always realistic).

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 test).