Real-time Systems – Design and analysis

Chapter 2: Analysis and prediction for real-time systems performances

Execution Time Prediction

- Motivations
  - The predicted execution times can be used:
    - to determine an appropriate scheduling scheme for the tasks
    - to perform an overall schedulability analysis in order to guarantee that all timing constraints will be met (also called *timing validation*).
  - Some real-time operating systems offer tools for schedulability analysis, but all these tools require as input the estimated execution times of tasks.

General Approaches for Execution Time Prediction

- Measurement
  - Run the program of interest and measure its execution times
- Simulation
  - The program and its environment are modelled, and the model is executed or interpreted to produce performance estimates
- Analysis
  - Execution times are obtained by mathematical analysis of a model of the system

General Approaches for Execution Time Prediction

- Measurement
  - Attractive and important technique
  - The method can not always be applied in practice
  - The best and worst case path through a program are often difficult to find
  - Interfaces from an operating system, from the measuring procedure itself, or from hardware, such as interruptions and bus contentions, can be unpredictable and change from run to run
  - For truly accurate measurements, the system must be run while performing its real-time mission → impractical because of the high costs

General Approaches for Execution Time Prediction

- Simulation
  - Allows the design to abstract the system and focus on the performance of the particular parts
  - It is possible to perform many experiments without incurring the failure cost of a real system
  - Typical approach is to compile the real-time software into object code and run it on a simulated architecture and physical environment
  - Problems
    - The correct shortest and longest path detection
    - It is difficult to simulate accurately architectures

General Approaches for Execution Time Prediction

- Analysis
  - Permits source program reasoning about execution times
  - The most abstract of the three approaches
- Problem
  - Care must be taken so that important practical system properties are not excluded from the models
    - The influence of hardware and software interfaces have to be incorporated into execution time prediction

Direct measurements using software probes

- Elements that may need to be timed
  - Complete applications
  - Individual programs
  - Procedures and functions
  - Code blocks
  - Instructions
  - System components
    - Context-switches mechanisms
    - Schedulers mechanisms
    - Synchronizations
    - …

Direct measurements using software probes

- Suppose S a self-contained body of code, containing no input/output or other system calls
- To obtain execution time of S, we need a timer or a clock. Let' s *clock* be a function that returns the current value of time
- Measurement overhead

```
-- Test program 1
t_start := Clock;
S;
t_finish :=Clock;
Test_time := t_finish - t_start ;
```

```
-- Control program 1
t_start := Clock;
-- this line is empty
t_finish :=Clock;
Control_time := t_finish - t_start ;
```

S_time = Test_time – Control_time

Direct measurements using software probes

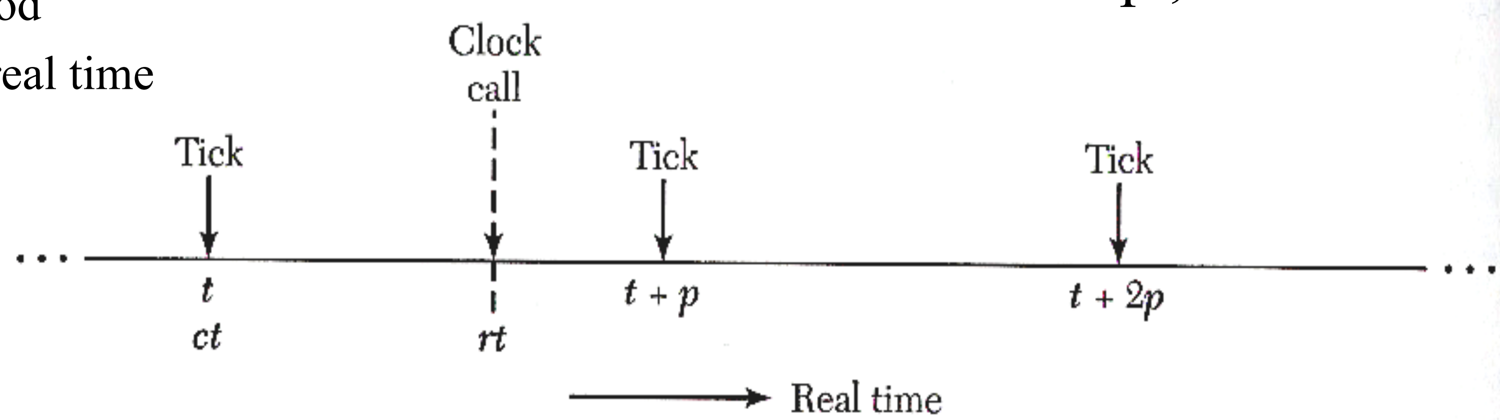
- Errors due to the tick granularity and accuracy of the clock
- Let consider

ct – time returned from the *Clock* function

p – tick period

rt – perfect real time

$$ct = rt - \delta p \text{ , } 0 \leq \delta < 1$$



The difference Δct obtained from two calls of *Clock*

$$\Delta ct = ct_2 - ct_1 = rt_2 - \delta_2 p - rt_1 + \delta_1 p = \Delta ct \pm \varepsilon$$

=>

S_time will than have a max. error of 2p

Direct measurements using software probes

- To improve approximation it is common to time the execution of many instances of S in a loop

```
-- Test program2
t_start := clock
for I in 1 … n loop
        S;
end loop;
t_finish := Clock;
test_time := t_finish – t_start;
```

test_time – control_time = n **x** S_time + error
where error $\leq$ 2p

$\Downarrow$

S_time = (test_time – control_time)/n + error/n

If the loop count n is sufficiently large, the error may be ignored

Program Analysis with Timing Schema

- Predicting best and worst case execution time
- Reasoning at the source program level
- A *timing schema* is simply an expression or formula that can be instantiated with particular statements elements
- The basic requirements is that the programming language have *compositional timing semantics*
  - The time bounds for a statement S composed of two statements S_1 and S_2, can be expressed in terms of the bounds of its constituents

$$T(S) = f(T(S_1);T(S_2))$$

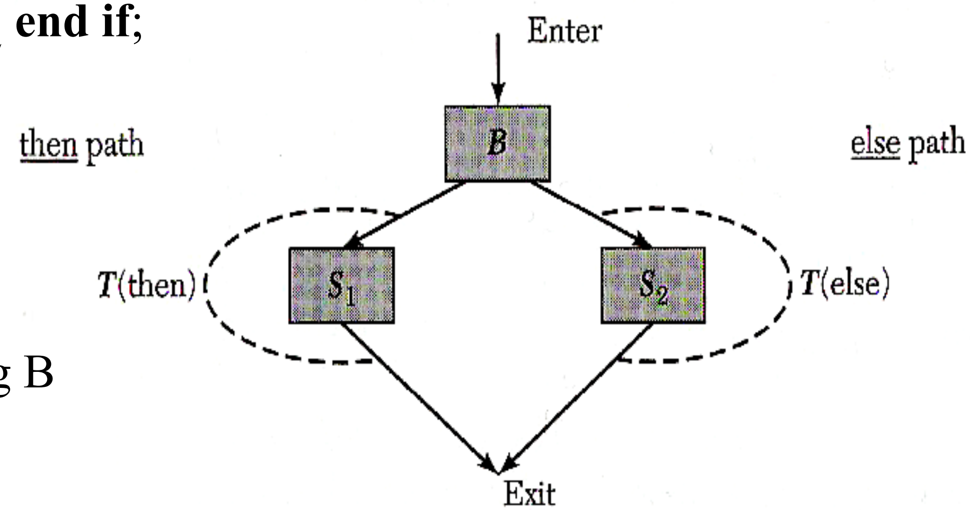
e.g. S = S1;S2

$$T(S) = T(S_1) + T(S_2) + T(;)$$

Program Analysis with Timing Schema

- Example of conventional construct

S = **if** B **then** S_1 **else** S_2 **end if**;



$[t_{1b}, t_{1w}] = T(B) + T(S_1) + T(\text{then})$

$[t_{2b}, t_{2w}] = T(B) + T(S_2) + T(\text{else})$

where

T(B) – execution bounds for evaluating B

T (S_1) – execution bound for S_1

T (S_2) – execution bound for S_2

T(then) – the control time associated with executing the "then" path T(else) – the control time associated with executing the "else" path

$$T(S) = [\min(t_{1b}, t_{2b}), \max(t_{1w}, t_{2w})]$$

Program Analysis with Timing Schema

- The atomic blocks and their execution times may be obtained from a separate analysis of the compiler and the target machine
- Example

$$\text{Exp} ::= v \mid c \mid \text{PF\_n}\ (E_1, \ldots E_n) \mid (\text{Exp}_1 \sigma\ \text{Exp}_2)$$

Typical atomic blocks

$$T(x),\ T(.v),\ T(:=),\ T(\sigma),\ T(c/r),\ T(par)$$

Timing schema

$$T(v{:=}\text{Exp}) = T(.v) + T(:=) + T\ (exp)$$

$$T(\text{Exp}) = \begin{cases} T(v), \text{ if Exp} = v \\ T(c), \text{ if Exp}=c \\ T(\text{Exp}_1) + T(\text{Exp}_2) + T(\sigma), \text{ if Exp} = (\text{Exp}_1\ \sigma\ \text{Exp}_1) \\ T(c/r) + n \text{ x } T(par) + T(\text{PF\_body}) + T(E_1) + \ldots + T(E_n), \text{ if Exp} = \text{PF\_n}(E1, \ldots \text{En}) \end{cases}$$

Program Analysis with Timing Schema

- Timing schema for conditional statements

if B_1 **then** S_1 **elsif** B_2 **then** S_2 **elsif** …. **elsif** B_n **then** S_n **end if**;

```
            < B1 >
      Transfer on false to 2
            < S1 >
      Transfer to next
2 :         < B2 >
      Transfer on false to 3
            < S2 >
      Transfer to next
3 :           .
              .
              .
n :          < Bn >
      Transfer on false to next
            < Sn >
next : ...
```

$[t_{kb}, t_{kw}] = T(S_k) + \min(k+1, n) \times T(if) + \sum_{i=1}^{k} T(B_i)$

$T(\text{if } B_1 \text{ then } S_1 \ldots \text{ then } S_n \text{ end if}) = [t_b, t_w]$

With $t_b = \min_{k \le n}(\min(t_{bk}), n \times t_{\min}(if) + \sum_{k=1}^{n} t_{\min}(B_i))$

and $t_w = \max(t_{kw})$ over all k

Program Analysis with Timing Schema

- Timing schema for loop
  while B loop S end loop;

```
start:  < B >
        Transfer on false to next
             < S >
        Transfer to start
next: ...
```

T(whie B loop S end loop) =
(n+1) x T(B) + n x T(S) +
(2n+1) x T(while)

Assume for simplicity that
T(transfer_on_flase_to_x) = T(transfer_to_x) = T(while)

Program Analysis with Timing Schema

- The timing schema is based primarily on a static analysis of the program text – an exception is the count on loops bounds
- This pure static method produce very satisfactory results in some cases, but in general can yield loose predictions because the path traced through the program text include many *infesabile paths*
- An infeasible path is an impossible execution sequence; it can not occur because of the program logic
  - A typical case is where a short (long) path through one path of a program implies a long (short) path through a subsequent portion
  - Static analysis pairs the two short (long) pairs together, even though it is semantically impossible to them to be part of the same execution sequence

Program Analysis with Timing Schema

- Infesabile paths – example 1
  - Multi-processor scheduler algorithm

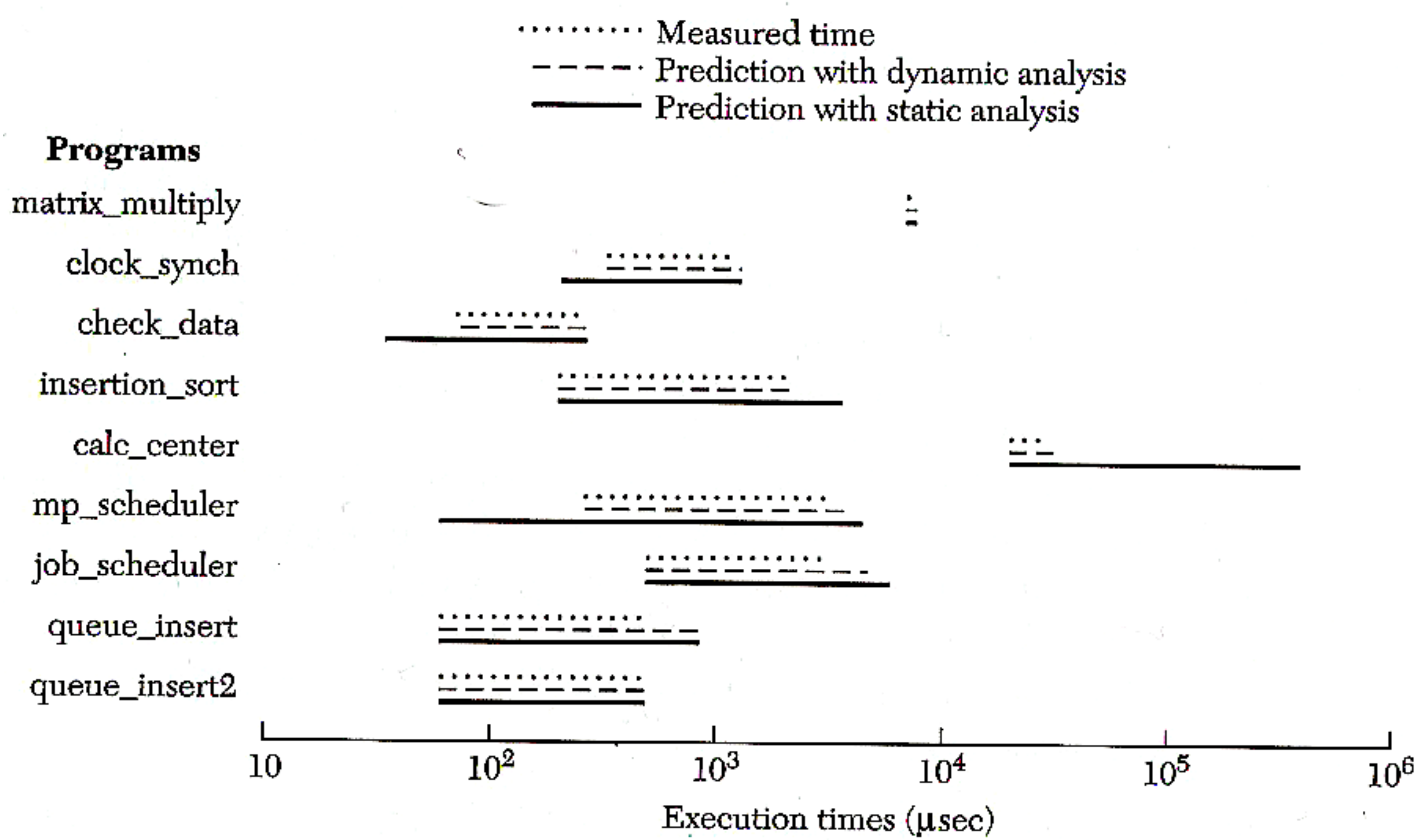
```
mp_scheduler::
     loop
          Search_Process_List ;     --P1
          if ready_process then Allocate_Idle ; end if; --P2
          if ready_process and no_idle_processor then Preempt ; end if;
          exit when no_ready_process or not_preemptible ;
     end loop;
```

  - Worst case of phase P1 – there are no ready process on the process list
  - The second phase P2 then has its shortest execution time because it immediately exits the loop
  - Static analysis includes the impossible sequences of the best path of P1 with the best path of P2, and the worst path of P1 with the worst path of P2.

Program Analysis with Timing Schema

- Dynamic analysis
  - Eliminate many infeasible program path
  - Path through a program are described
    - Example
      - One can state that whenever one part of (or partial path through) a program is taken or executed, another part is not taken
      - Each such path description covers or includes all feasible path through the program

Dynamic Analysis vs. Static Analysis



Execution Time Prediction

- Timing analysis in practice must also incorporate the effects of a variety of hardware and software interfaces
- One major classes of such interfaces is caused by *interrupts*
  - From timers
  - From I/O devices (e.g. sensors)
  - …
- These interrupts preempt the running program and transfer the control to some operating system software that "handles" the interrupt
- When bounds on interrupt handling times and on interrupt frequency are available, execution times can be adjusted to include the processor sharing between a program and the interleaved interrupt handling that occurs during its execution

Execution Time Prediction

- Assume that there is only one kind of interrupt and an associated interrupt handler IH

$$t'(S) = t(S) + t'(S) \times f \times t(IH) \Rightarrow t'(S) = t(S)/(1-f \times t(IH))$$

where

t(S) – execution time of program S without the interrupt

t'(S) – execution time of program S in the presence of the interrupt

t(IH) – the interrupt handeling time

f – the interrupt frequency

- The effect of more then one interrupt can be included

$$t'(S) = t(S)/(1-\sum_i (f_i \times t(IH_i)))$$

Execution Time Prediction

- Modern processors increase performance by using: Caches, Pipelines, Branch Prediction
- These features make WCET computation difficult: Execution times of instructions vary widely
  - Best case - everything goes smoothely: no cache miss, operands ready, needed resources free, branch correctly predicted
  - Worst case - everything goes wrong: all loads miss the cache, resources needed are occupied, operands are not ready
  - Span may be several hundred cycles

Execution Time Prediction

- Timing Accident – cause for an increase of the execution time of an instruction
- Timing Penalty – the associated increase
- Types of timing accidents
  - Cache misses
  - Pipeline stalls
  - Branch mispredictions
  - Bus collisions
  - Memory refresh of DRAM
  - TLB misses
  - …

Execution Time Prediction

- Finding safe and tight execution bounds for nontrivial programs is still a research problem
- Many of the delays that do occur and affect deadlines are due to the interaction and communication between machines and processes
- To learn more on existing products analysing worst case execution times, see
  - http://www.absint.com/ait/ - WCET Analyser
  - http://www.symta.org/ - SymTA/S