CT420 REAL-TIME SYSTEMS

WCET ANALYSIS

Dr. Michael Schukat

Ollscoil na Gaillimhe
University of Galway

Lecture Overview

- This slide deck provides an overview of methodologies to estimate the Worst-Case Execution Time (WCET) of a task or function using
  - empirical evidence (empirical WCET analysis)
  - analytical methods (control flow graph-based WCET analysis)

Recall: CE and Task Execution Times

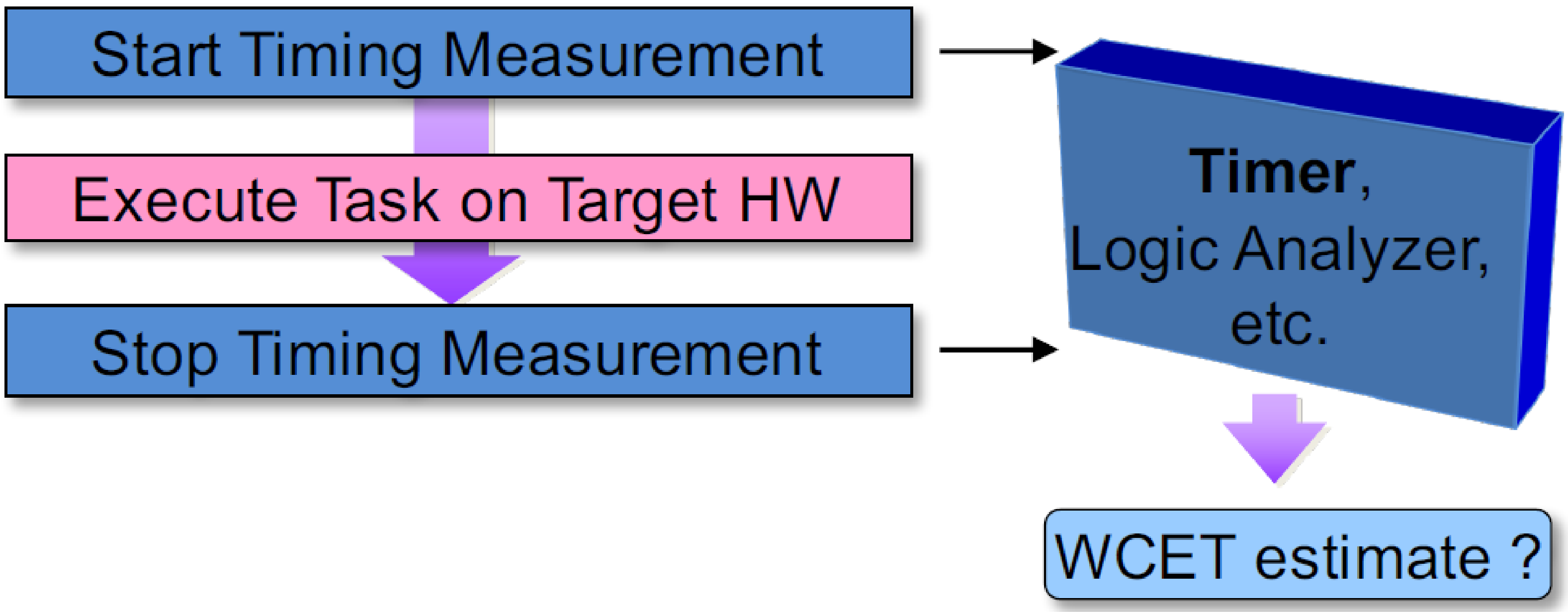
Task	Period p [ms]	Exec Time [ms]
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

- Before we can determine whether or not a scheduling algorithm will allow all periodic / sporadic tasks to satisfy their deadlines, we must be aware of their execution time
- Principal question: How do we determine the (worst case) execution times of tasks?

Estimating Worst-Case Execution Times

- Many tasks exhibit non-uniform run times, e.g.:
  - A task may inspect an environmental condition by simply recording some data; however, occasionally, the task may have to react to a situation that has been observed, that takes up additional CPU time
- Thus, we must estimate for each task the worst-case execution time (WCET) for each task and determine whether or not all deadlines can still be met under such circumstances
- This can be done via
  - an analysis of the source code (CFG-based WCET analysis), or
  - an estimation from empirical evidence (empirical WCET analysis)
- The goal of WCET analysis is to generate a safe (i.e. no underestimation) and tight (i.e. small overestimation) estimate of the worst-case execution time of a program (or program fragment)

Empirical WCET Analysis



- To perform such a WCET analysis, a multitude of measurements with different task inputs and task states are done
- To get meaningful results,
  - the program execution must be uninterrupted (no pre-emptions or interrupts)
  - there must be no interfering background activities, such as garbage collection, blocking, synchronisation, or inter-task communication

Example empirical WCET Analysis

Example 1

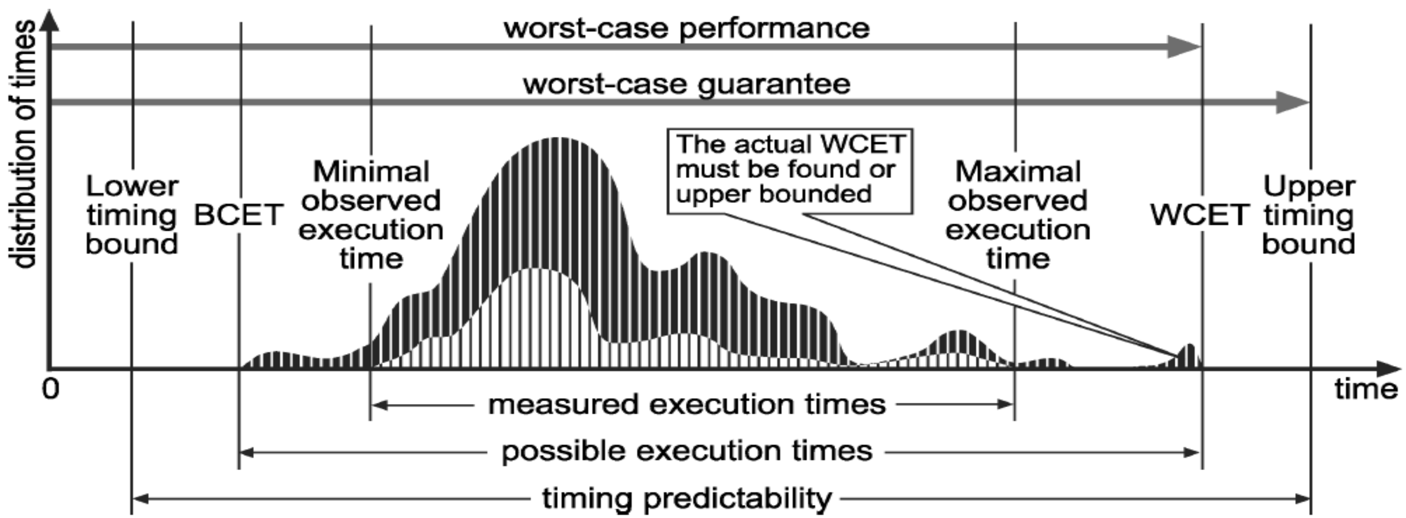
```
int a, b, z, t;
while (1) {
        a = rand();
        b = rand();
        t = 0;
        reset_timer();
        start_timer();
        z = Voter(a, b);
        stop_timer();
        t = read_timer();
        store_timer_content(t);
}
```

Example 2

```
int a, t;
while (1) {
        reset_timer();
        t = 0;
        start_timer();
        a = ReadTempSensorA();
        stop_timer();
        t = read_timer();
        store_timer_content(t);
}
```

Empirical WCET Analysis in Practice

- Execute tests (with different inputs and states), store execution times (store_timer_content() in previous example), quantise determined execution times (e.g., 1ms bin width), plot a histogram for visualisation of results, and determine WCET, possibly also BCET and ACET
- Note: Light bars represent obtained results, black bars represent a (hypothetical) exhaustive test



[Wilhelm+08]

WCET: Worst-Case Execution Time
BCET: Best-Case Execution Time
ACET: Average-Case Execution Time

The WCET/BCET is the longest/shortest execution time possible for a program.
Must consider all possible inputs—including perhaps inputs that violate specification.

Limitations of empirical WCET Analysis

- Measuring all different execution traces of a real size program is intractable in practice
  - e.g., even a mid-size task may have millions of different paths
- Selected task inputs and task states may fail to trigger the longest execution trace
- Rare execution scenarios may be missed (see example on slide 4)

CFG-based WCET Analysis

- For hard RTS we can't effort to miss only a single deadline, so we need to make sure to capture a task's WCET
- Starting point is to implement tasks with a **low complexity**
  - i.e. limit the number of nested loops, if-then-else statements, etc.
  - Software testing tools like Cobertura (a Java tool) allow measuring method complexity
- Subsequently, **flow analysis** techniques using **control flow graphs (CFG)** are used to identify possible ways a program can execute
- These are combined with the execution times of programme blocks
- Both used in tandem allow the calculation of a task's WCET

Steps of a CFG-based WCET Analysis

Create the CFG

- Draw nodes for each basic block of code
- Connect nodes with directed edges to represent control flow (including if statements and loops)

Annotate execution times

- Annotate each node with the execution time of the corresponding basic block

Identify possible paths

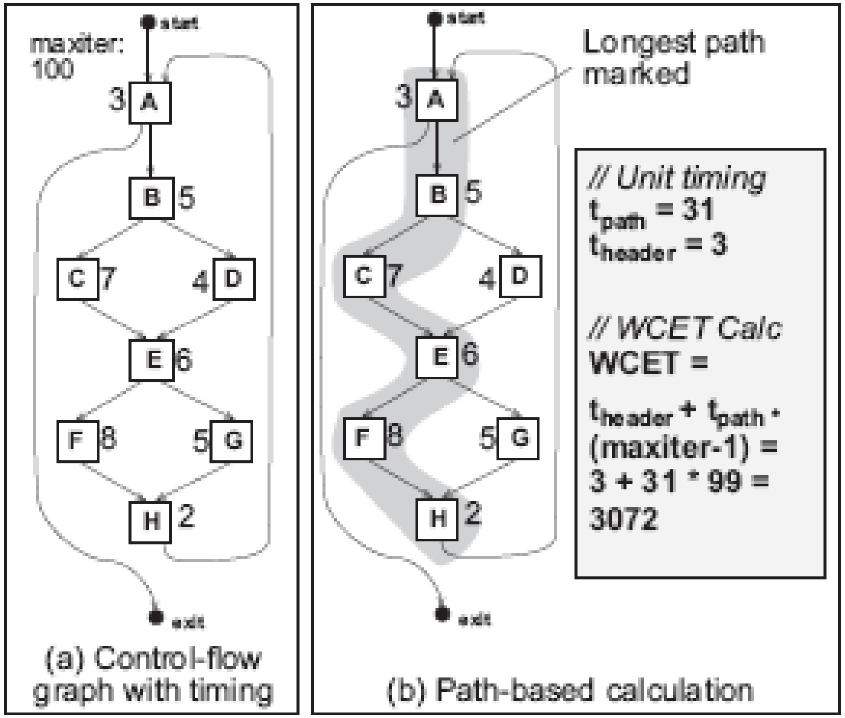
- Traverse the graph to identify all possible paths from the entry node to the exit node; incorporate maximum number of loop iterations
- Calculate the total execution time for each path by summing up the execution times of the nodes along that path

Determine WCET

- The WCET is the maximum execution time among all possible paths in the CFG

Example for a CFG-based WCET Analysis

```
for (…) { // A
    if (…) { // B
        … // C
    }
    else {
        … // D
    }
    if (…) { // E
        … // F
    }
    else {
        … // G
    }
    …   // H
}
```



Acquiring Execution Times of Building Blocks: From C to Assembly Language

```
int arith(int x,
          int y,
          int z)
{
    int t1 = x+y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;

    return t4;
}
```

- Each instruction requires a set amount of CPU cycles for its execution (CPU spec will tell)
- CPU cycle length is derived from a CPU’s clock rate
- E.g.
  - 4 MHz CPU clock → 4 x 10^{-6} [s] cycle length (4 microseconds)
  - An instruction that requires 10 CPU cycles has an execution time of 4 x 10^{-5} [s] (40 microseconds)

```
1   movl 12(%ebp),%eax          Get y
2   movl 16(%ebp),%edx          Get z
3   addl 8(%ebp),%eax           Compute t1 = x+y
4   leal (%edx,%edx,2),%edx     Compute z*3
5   sall $4,%edx                Compute t2 = z*48
6   andl $65535,%eax            Compute t3 = t1&0xFFFF
7   imull %eax,%edx             Compute t4 = t2*t3
8   movl %edx,%eax              Set t4 as return val
```

Pitfalls when calculating Execution Paths


```
   const int max = 100;
   foo ( int x) {
A:  for(i = 1; i <= max; i++) {
B:    if (x > 5)
C:      x = x * 2;
      else
D:      x = x + 2;
E:    if (x < 0)
F:      b[i] = a[i];
G:    bar (i)
     }}
```

- **Loop bounds**: Easy to find in this example; in general, very difficult to determine
- **Infeasible paths**: Can we exclude a path, based on data analysis? A-B-C-E-F-G is infeasible—since if x>5, it is not possible that x * 2 < 0. *Well, really? What about integer overflows? Must be sure that these do not happen in the example...*

Recall: Two's Complement Integer Representation

- C and other programming languages do not check for numeric (signed and unsigned integer) overflows
- E.g., with 4-bit signed int
  "7 + 1" =
  "0111 + 0001" =
  "1000" = -8

Binary Number	Unsigned Value	Signed Value
0000	0	0
0001	1	1
0010	2	2
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

WCET and SOTA CPUs

- Modern processors increase performance by using caches, pipelines, and branch prediction
- These features make WCET computation difficult, as execution times of instructions vary widely
  - Best case - everything goes smoothly: 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
- This makes it very problematic to use such CPUs for empirical WCET analysis
- In CFG-based WCET analysis, performance optimising features are simply ignored

Summary

- The determination of reliable WCET estimates is fundamental for hard, and even soft RTS
- WCET analysis can be done via empirical methods or flow analysis, with both options having their pros, cons, and limitations
- A good starting point, particularly when dealing with hard RTS, is the implementation of tasks with low cyclomatic complexity, that are executed on CPU / hardware with constant instruction execution times, and with no timing accidents