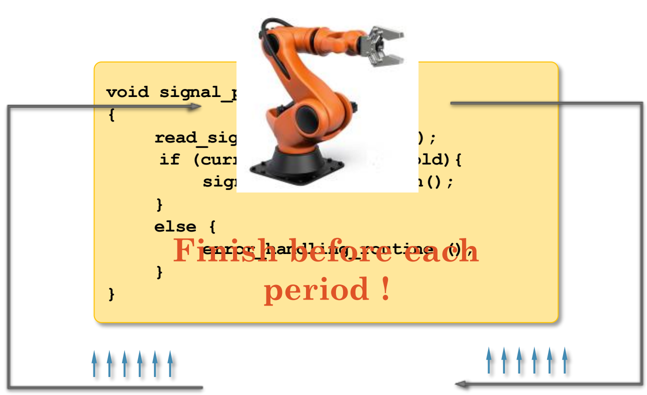
Estimation of Worst-Case Execution Time (WCET)

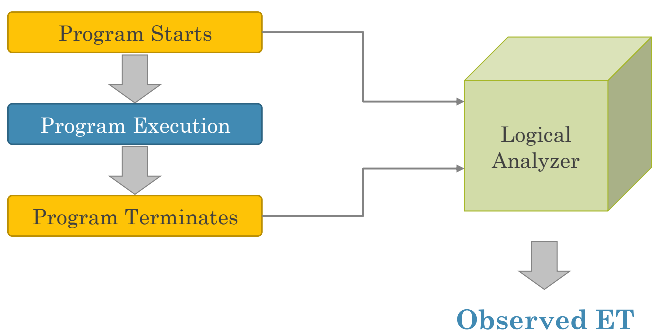
1

Execution Time



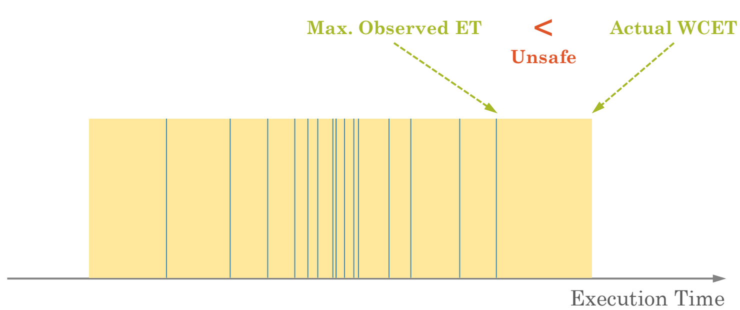
2

Measuring Execution Time



3

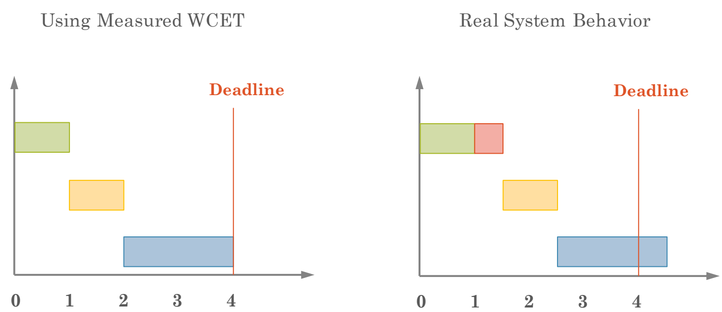
Measuring is Unsafe



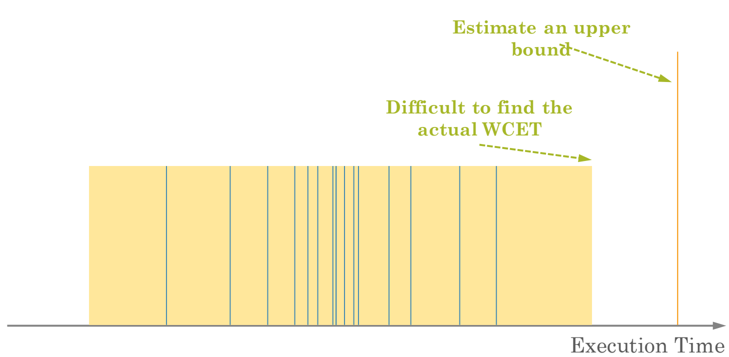
4

The Consequence

- Unsafe estimation → incorrect timing prediction

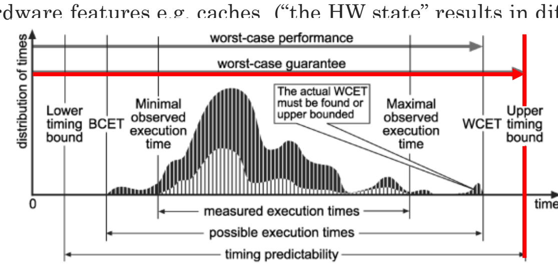


Then, What Do We Need?



The WCET Analysis Problem

- Hard real-time systems subjected to strict timing constraints
  - E.g. Automotive, Avionics
- A fundamental problem: Worst-Case Execution Time (WCET) analysis
  - Given a program (sequence of instructions) – that always terminates (no recursion, bounded loops etc),
  - Question: what is its Worst-Case Execution Time (WCET)?
- Challenges ("termination" doesn't make the problem easy)
  - "too many input" ➔ too many execution paths (difficult to find the worst-case)
  - hardware features e.g. caches ("the HW state" results in different execution times)



7

How?

-- Static Analysis

8

```
void main() {
    int b;
    int i = 0, j = 0;
    while (i < 100) {
        if (b)
            j++;
        else
            j--;
        i++;
    }
}
```

```
main():
simple.c:2
004001f0 <main>          addiu
$29,$29,-24
004001f8 <main+0x8>     sw
$30,16($29)
00400200 <main+0x10> addu      $30,$0,$29
simple.c:4
00400208 <main+0x18> sw        $0,4($30)
00400210 <main+0x20> sw        $0,8($30)
simple.c:5
00400218 <main+0x28> lw        $2,4($30)
00400220 <main+0x30> slti       $3,$2,100
00400228 <main+0x38> bne
$3,$0,00400238
00400230 <main+0x40> j          004002b8
simple.c:6
00400238 <main+0x48> lw        $2,0($30)
00400240 <main+0x50> beq
$2,$0,00400270
simple.c:7
00400248 <main+0x58> lw        $3,8($30)
00400250 <main+0x60> addiu    $2,$3,1
00400258 <main+0x68> addu     $3,$0,$2
00400260 <main+0x70> sw        $3,8($30)
00400268 <main+0x78> j          00400290
simple.c:9
00400270 <main+0x80> lw         $3,8($30)
00400278 <main+0x88> addiu     $2,$3,-1
00400280 <main+0x90> addu      $3,$0,$2
00400288 <main+0x98> sw        $3,8($30)
simple.c:10
00400290 <main+0xa0> lw         $3,4($30)
00400298 <main+0xa8> addiu     $2,$3,1
004002a0 <main+0xb0> addu      $3,$0,$2
004002a8 <main+0xb8> sw        $3,4($30)
simple.c:11
004002b0 <main+0xc0> j          00400218
simple.c:12
004002b8 <main+0xc8> addu      $29,$0,$30
004002c0 <main+0xd0> lw
$30,16($29)
```

Entering Loop

Exiting Loop

Next Iteration

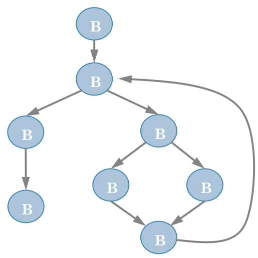
9

Control Flow Graph

10

Enumerating all possible executions

- All possible initial states
- All possible program paths

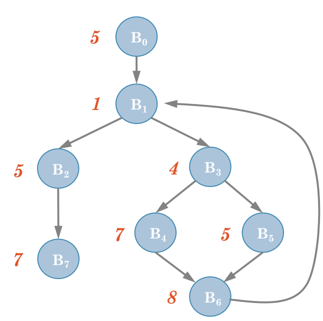


If this loop iterates 100 times,
There will be 2^{100} different paths

It would never work
even with bounded loops!

A better solution

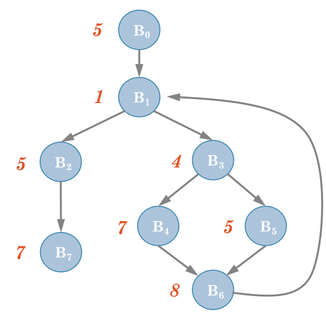
- STEP 1: estimate the WCET for each basic block
- STEP 2: enumerate all possible execution paths and find the worst path



It may work but it is still too many path to enumerate

A (fairly good) solution

- Separate path and micro-architecture analysis
  - STEP 1: estimate the WCET for each basic block under given hardware features
  - STEP 2: Find an upper bound on the "maximal" execution time (no enumeration)



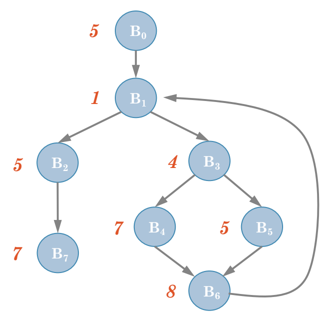
Implicit Path Enumeration

- Main idea of path anlaysis

ORIGINAL GOAL	NEW GOAL
Finding the actual path with the maximal execution time	Finding the execution count of each block, implying the longest path

Implicit Path Enumeration

- Some variables
  - X_i : the execution count of basic block B_i
  - C_i : the WCET of basic block B_i (assuming known for now)



Execution Time

$$\sum_{i=0}^{7} X_i \times C_i$$

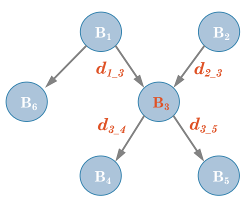
Implicit Path Enumeration

- Now, the path analysis problem becomes
- Finding a valuation of $< X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7 >$
- Such that the execution time is maximized

$$WCET = \max \sum_{i=0}^{7} X_i \times C_i$$

Implicit Path Enumeration

- X_i cannot have an arbitrary value → The constraints
- Structural constraints
  - d_{i_j} : the execution count of the edge from B_i to B_j



$X_3 = d_{1\_3} + d_{2\_3}$
$X_3 = d_{3\_4} + d_{3\_5}$

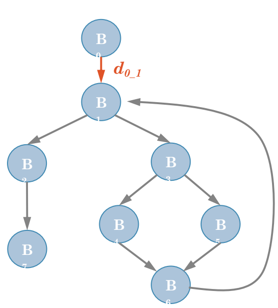
For each basic block, we have

$$X_i = \sum_{\text{all } B_j \to B_i} d_{j\_i} = \sum_{\text{all } B_i \to B_k} d_{i\_k}$$

17

Implicit Path Enumeration

- Constraints for the start/end nodes
  - $X_0 = 1$
  - $X_7 = 1$
- Bounding loop iterations

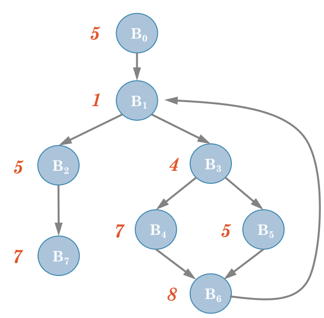


Loop bound = 10

$X_1 \leq 11 * d_{0\_1}$
$X_3 \leq 10 * d_{0\_1}$
$X_4 \leq 10 * d_{0\_1}$
$X_5 \leq 10 * d_{0\_1}$
$X_6 \leq 10 * d_{0\_1}$

18

Implicit Path Enumeration



Maximize

5 X0 + 1 X1 + 4 X3 + 7 X4 + 5 X5 + 8 X6 + 5 X2 + 7 X7

Subject to

X0 = 1;
X0 – d_0_1 = 0;
X1 – d_0_1 – d_6_1 = 0;
X1 – d_1_2 – d_1_3 = 0;
X2 – d_1_2 = 0;
X2 – d_2_7 = 0;
X3 – d_1_3 = 0;
X3 – d_3_4 – d_3_5 = 0;
X4 – d_3_4 = 0;
X4 – d_4_6 = 0;
X5 – d_3_5 – 0;
X5 – d_5_6 = 0;
X6 – d_4_6 – d_5_6 = 0;
X6 – d_6_1 = 0;
X7 =1;
X6 – 10 d_0_1 <= 0;   // loop bound

How to estimate the WCET for each basic block?

Micro-Architecture Analysis

- Goal
  - Given the hardware features, estimate an upper bound for each instruction (then, basic block)

- Why is it hard?
  - Caches: instruction/data, multi-level, shared, replacement
  - Pipelines (not so often in embedded processors)
  - Branch predictor (not so often in embedded processors)
  - Memory controller, main memory
  - Etc.

21

Cache Analysis

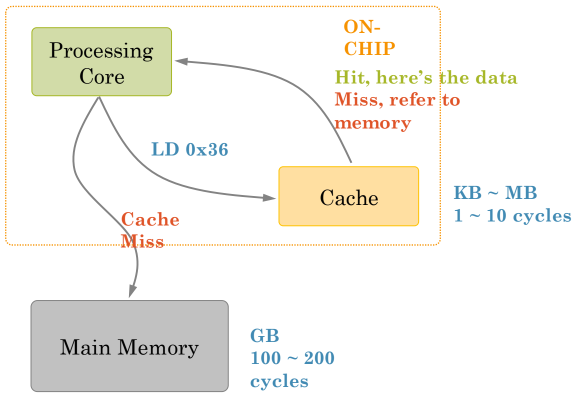
- A program
  - 100 instructions, 50% cache hit in real execution
  - Hit latency = 2; miss latency = 100

Analysis	Result	WCET
No Analysis	Assuming all accesses are cache miss for safety	10,000
With Analysis	90% of the cache hits are successfully identified	5,590

44.1% reduction in the estimated WCET!

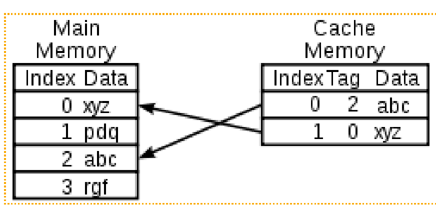
22

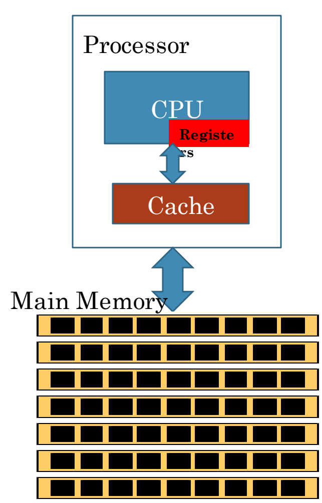
Cache in a Nutshell

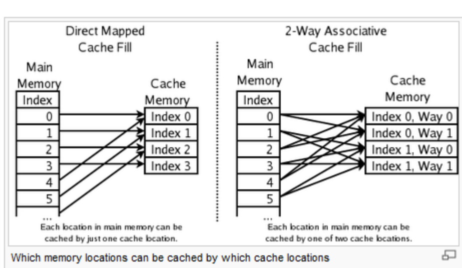


23

Processor, Caches & Off-chip Memory







Which memory locations can be cached by which cache locations

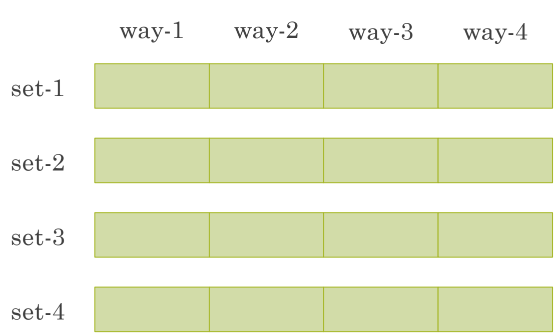
Cache in a Nutshell

Why caches work?

- Memory reuse (think of a loop)
- The principle of locality
  - Temporal locality: the reuse of specific data within a relatively small time duration
  - Spatial locality: the use of data elements within relatively close storage locations

Cache in a Nutshell

- Set-associative caches



Cache in a Nutshell

- Set-associative caches

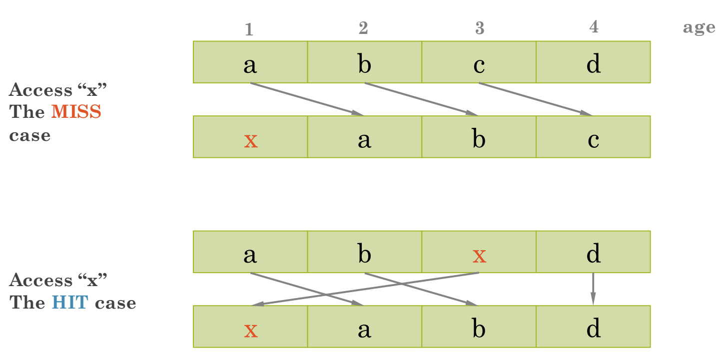
	way-1	way-2	way-3	way-4	Replacement policy	
set-1	adr1	adr5	adr9	adr13	adr17	adr21
set-2	adr2	adr6	adr10	adr14	adr18	…
set-3	adr3	adr7	adr11	adr15	adr19	…
set-4	adr4	adr8	adr12	adr16	adr20	…

(set nr = adr mod #set)

27

Cache in a Nutshell

- Cache Replacement
  - E.g. Least-Recently-Used (LRU)



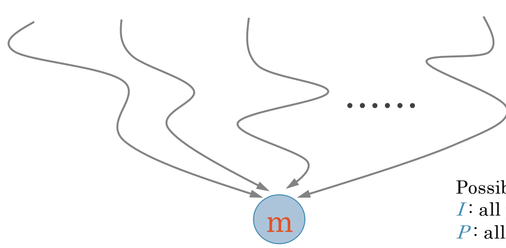
28

Cache Analysis

- The purpose of cache analysis for WCET analysis is to statically determine whether each memory reference is hit or miss, regarding the worst-case execution.
- In case precise estimations are hard to get, you are allowed to make mistakes in your prediction, as long as they do not underestimate the WCET. (Safety Requirement)
- But, try to make less mistakes. (Precision Requirement)

29

The Fundamental Challenge

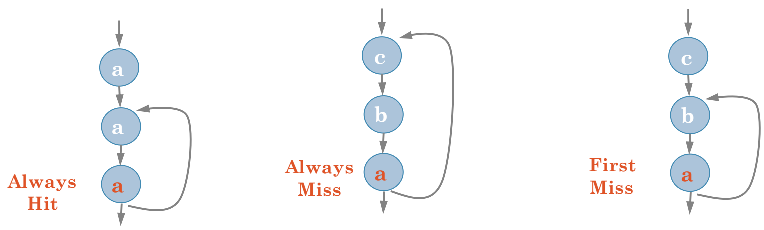


Possible incoming states : $S = I \times P$
I : all possible initial HW states
P : all possible program paths

How can we ensure all the possibilities are considered?
How to efficiently manage so many states?

30

Example cache states

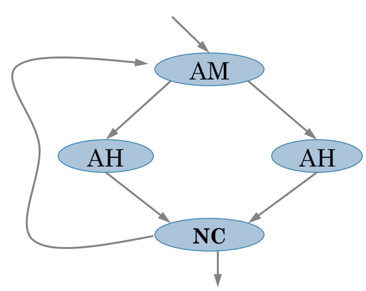


(cache size = 2)

31

4 possible outcomes in accessing a basic block

1. Always hit (AH)
2. Always miss (AM)
3. First miss (FM)
4. Not Classified (NC)



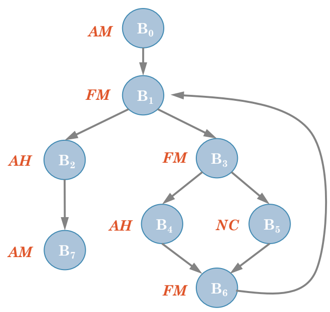
- Access times e.g. AH → 2, AM→100, FM→ (100,2), NC→100

This can be predicted by Static Analysis (Abstract Interpretation)
There are commercial tools e.g. **aiT** from Absint

32

WCET Calculation

- Integration cache analysis results into IPET



// hit latency =2; miss latency = 10

Maximize

$(2\ x_{0h} + 10\ x_{0m}) + (2\ x_{1h} + 10\ x_{1m}) + (2\ x_{2h} + 10\ x_{2m}) + (2\ x_{3h} + 10\ x_{3m}) + (2\ x_{4h} + 10\ x_{4m}) + (2\ x_{5h} + 10\ x_{5m}) + (2\ x_{6h} + 10\ x_{6m}) + (2\ x_{7h} + 10\ x_{7m})$

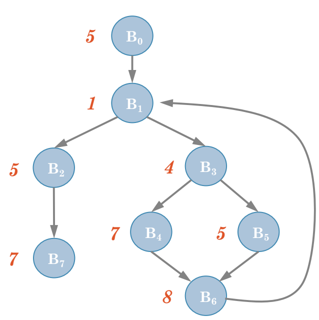
// cache constraints
$X0 = x_{0h} + x_{0m}$
$0 <= x_{0m} <= X0$
$x_{0h} = 0$

$X1 = x_{1h} + x_{1m}$
$0 <= x_{1h} <= X1$
$x_{1m} <= 1$

$X2 = x_{2h} + x_{2m}$
$0 <= x_{2h} <= X2$
$x_{2m} = 0$

……

WCET Calculation



Maximize

5 X0 + 1 X1 + 4 X3 + 7 X4 + 5 X5 + 8 X6 + 5 X2 + 7 X7

Subject to

X0 = 1;
X0 – d_0_1 = 0;
X1 – d_0_1 – d_6_1 = 0;
X1 – d_1_2 – d_1_3 = 0;
X2 – d_1_2 = 0;
X2 – d_2_7 = 0;
X3 – d_1_3 = 0;
X3 – d_3_4 – d_3_5 = 0;
X4 – d_3_4 = 0;
X4 – d_4_6 = 0;
X5 – d_3_5 – 0;
X5 – d_5_6 = 0;
X6 – d_4_6 – d_5_6 = 0;
X6 – d_6_1 = 0;
X7 =1;
X6 – 10 d_0_1 <= 0; // loop bound