Estimation of Worst-Case Execution Time (WCET)
void signal_processing()
{
    read_signal(&curr_signal);
    if (curr_signal < threshold){
        signal_transformation();
    }
    else {
        error_handling_routine();
    }
}

Finish before each period!
Measuring Execution Time

Program Starts → Program Execution → Program Terminates → Logical Analyzer → Observed ET
Measuring is Unsafe

Max. Observed ET < Actual WCET

Unsafe
The Consequence

- Unsafe estimation $\rightarrow$ incorrect timing prediction

Using Measured WCET

Real System Behavior
Then, What Do We Need?

- Difficult to find the actual WCET
- Estimate an upper bound
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)
How?

-- Static Analysis
```c
void main() {
    int b;
    int i = 0, j = 0;
    while (i < 100) {
        if (b)
            j++;
        else
            j--;
        i++;
    }
}
```
Control Flow Graph
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 analysis

**ORIGINAL GOAL**

Finding the actual path with the maximal execution time

**NEW GOAL**

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)

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

\( d_{i,j} \): 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

\[
X_i = \sum_{\text{all } B_j \rightarrow B_i} d_{j,i} = \sum_{\text{all } B_i \rightarrow B_k} d_{i,k}
\]
Implicit Path Enumeration

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

- Bounding loop iterations

\[ X_1 \leq 10 \times (d_{0,1} + d_{0,1}) \]
\[ X_3 \leq 10 \times d_{0,1} \]
\[ X_4 \leq 10 \times d_{0,1} \]
\[ X_5 \leq 10 \times d_{0,1} \]
\[ X_6 \leq 10 \times d_{0,1} \]
Implicit Path Enumeration

Maximize

\[ 5 \times X_0 + 1 \times X_1 + 4 \times X_3 + 7 \times X_4 + 5 \times X_5 + 8 \times X_6 + 5 \times X_2 + 7 \times X_7 \]

Subject to

\[ X_0 = 1; \]
\[ X_0 - d_{0.1} = 0; \]
\[ X_1 - d_{0.1} - d_{6.1} = 0; \]
\[ X_1 - d_{1.2} - d_{1.3} = 0; \]
\[ X_2 - d_{1.2} = 0; \]
\[ X_2 - d_{2.7} = 0; \]
\[ X_3 - d_{1.3} = 0; \]
\[ X_3 - d_{3.4} - d_{3.5} = 0; \]
\[ X_4 - d_{3.4} = 0; \]
\[ X_4 - d_{4.6} = 0; \]
\[ X_5 - d_{3.5} = 0; \]
\[ X_5 - d_{5.6} = 0; \]
\[ X_6 - d_{4.6} - d_{5.6} = 0; \]
\[ X_6 - d_{6.1} = 0; \]
\[ X_7 = 1; \]
\[ X_6 - 10 \times d_{0.1} \leq 0; \quad // \text{loop bound} \]
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.
Cache in a Nutshell

- **Processing Core**
  - On-Chip
    - Hit, here's the data
    - Miss, refer to memory
  - LD 0x36
- **Cache**
  - KB ~ MB
    - 1 ~ 10 cycles
- **Main Memory**
  - GB
    - 100 ~ 200 cycles
Cache Analysis

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

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Result</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Analysis</td>
<td>Assuming all accesses are cache miss for safety</td>
<td>10,000</td>
</tr>
<tr>
<td>With Analysis</td>
<td>90% of the cache hits are successfully identified</td>
<td>5,590</td>
</tr>
</tbody>
</table>

44.1% reduction in the estimated WCET!
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

<table>
<thead>
<tr>
<th></th>
<th>way-1</th>
<th>way-2</th>
<th>way-3</th>
<th>way-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>set-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>set-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cache in a Nutshell

- Set-associative caches

<table>
<thead>
<tr>
<th></th>
<th>way-1</th>
<th>way-2</th>
<th>way-3</th>
<th>way-4</th>
<th>Replacement policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>set-1</td>
<td>adr1</td>
<td>adr5</td>
<td>adr9</td>
<td>adr13</td>
<td>adr17  adr21</td>
</tr>
<tr>
<td>set-2</td>
<td>adr2</td>
<td>adr6</td>
<td>adr10</td>
<td>adr14</td>
<td>adr18  ...</td>
</tr>
<tr>
<td>set-3</td>
<td>adr3</td>
<td>adr7</td>
<td>adr11</td>
<td>adr15</td>
<td>adr19  ...</td>
</tr>
<tr>
<td>set-4</td>
<td>adr4</td>
<td>adr8</td>
<td>adr12</td>
<td>adr16</td>
<td>adr20  ...</td>
</tr>
</tbody>
</table>

(set nr = adr mod #set)
Cache in a Nutshell

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

---

**Access “x”**

The **MISS** case

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>x</td>
<td>a</td>
<td>b</td>
<td>c</td>
</tr>
</tbody>
</table>
```

---

**Access “x”**

The **HIT** case

```
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>x</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>a</td>
<td>b</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>
```
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)
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?
Example cache states

Always Hit

Always Miss

First Miss

(cache size = 2)
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
WCET Calculation

- Integration cache analysis results into IPET

// hit latency = 2; miss latency = 10

Maximize

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

// cache constraints

\[X0 = x_{0h} + x_{0m}\]
\[0 \leq x_{0m} \leq X0\]
\[x_{0h} = 0\]

\[X1 = x_{1h} + x_{1m}\]
\[0 \leq x_{1h} \leq X1\]
\[x_{1m} \leq 1\]

\[X2 = x_{2h} + x_{2m}\]
\[0 \leq x_{2h} \leq X2\]
\[x_{2m} = 0\]

......