IN4343 Real-Time Systems

Static Timing Analysis
Recall

Estimating $C_i$ is not easy
Predictability vs. Efficiency

Estimating $C_i$ is not easy

- **hard** RTS based on Worst Case Exec. Time
- **soft** RTS based on Average Case Exec. Time
Execution Time Analysis

Timing analysis must be carried out for
- Program code + libraries
- VM code
- RTOS code
- Hardware
  - processor, busses, caches, memories, pipelines
  - peripherals, external sensors & actuators

Can’t be ignored
Execution Time Analysis

Two main approaches

- **Static** timing analysis
  - Look at the code and try to figure out how long it will take under certain circumstances

- **Dynamic** timing analysis
  - Execute the code and measure the time it takes

Determining which is the best/worst scenario is a hard (search) problem
Static vs. Dynamic Timing Analysis

- **Static** timing analysis is always “safe”
  - If it is performed correctly
  - The prediction is always worse than the real worst case

- **Dynamic** timing analysis is always “unsafe”
  - The prediction is always lower than the real worst case

In practice, usually a combination of both, static and dynamic timing analysis, is applied.
Static Timing Analysis

Definition

• Full analysis of a program to determine its worst/best case execution time
  ➢ it is only based on documentation, without execution of the program
  ➢ **but** it tries to include various execution conditions
Static Timing Analysis

Example

Step 1
• determine best/worst case execution path

```c
void median(const char* image, char* result, int N, int M) {
    // Move window through all elements of the image
    for (int m = 1; m < M - 1; m++)
        for (int n = 1; n < N - 1; n++) {
            // Pick up window elements
            int k = 0;
            char window[9];
            for (int j = m - 1; j < m + 2; j++)
                for (int i = n - 1; i < n + 2; i++)
                    window[k++] = image[j * N + i];
            // Order elements (only half of them)
            for (int j = 0; j < 5; j++) {
                // Find position of minimum element
                int min = j;
                for (int l = j + 1; l < 9; l++)
                    if (window[l] < window[min])
                        min = l;
                // Put found minimum element in its place
                const char temp = window[j];
                window[j] = window[min];
                window[min] = temp;
            }
            // Get result - the middle element
            result[(m - 1) * (N - 2) + n - 1] = window[4];
        }
}
```
Static Timing Analysis

Example

Step 1
- determine best/worst case execution path

Step 2
- map to assembly and annotate how often each line is executed

median:
.LFB0:

leal -1(%rcx), %eax
cmp $1, %eax
jle .L22
pushq %r15
pushq %r14
pushq %r13
pushq %r12
pushq %rbp
pushq %rbx
movq %rsi, %r12
leal -2(%rdx), %eax
movl %eax, -28(%rsp)
movl %edx, %ebp
leal -2(%rcx), %eax
movl %eax, -24(%rsp)
movl $0, %r14d
movl $0, -32(%rsp)
movl $0, -36(%rsp)
leal -1(%rdx), %eax
movl %eax, -20(%rsp)
leal -2(%rdx), %eax
movl %eax, -40(%rsp)
jmp .L3

.L14:
Static Timing Analysis

Example

Step 3
- consult processor manual to determine \#cycles/instruction and \#cycles/mem.acc.

Step 4
- multiply with line counts and sum

Complication:
modern processors have
- execution pipelines
- branch prediction
- cache memories
Static Timing Analysis

Characteristics

- Assumes that the time for all parts of a program can be calculated
  - deterministic hardware
  - deterministic system calls
  - deterministic programming constructs

- Does not incorporate the application context of a program, any input data, or state information

Cannot be done for arbitrary programs and platforms
Deterministic programs

Loops

It must be guaranteed that every loop terminates

- predetermined number of iterations (bounded loops)
- predetermined amount of time

Difficult to extract from the loop condition only

Additional information by programmer

- pragmas
- trigger exceptions on overruns

```c
for (expr1; expr2; expr3)
MAX_COUNT (const_expr) | MAX_TIME (const_expr)
loop_body;
[ON_OVERRUN|ON_TIMEOUT exception];
```
Deterministic programs

Recursion

It must be guaranteed that the recursion stops
- predetermined call depth
- implies bounded stack space

Difficult to extract from code
- pragmas
- forbid recursion
Deterministic programs

Additional restrictions

- **NO** function pointers
  - callee is unknown
  - calls might be recursive
  - no OO / dynamic typing / virtual method tables

- Parameters, user input, file I/O, ...
  - fix values (pragmas)
  - forbid
WCET Calculation

Case analysis

• Simple language construct (e.g., expr, assign)
  ➢ time of the corresponding machine instructions
  ➢ either the binary/assembly code is analyzed, or the concretization of the compiler must be known

• Sequence of simple constructs (;)
  ➢ sum of the estimations for the simple constructs in the sequence
WCET Calculation

Case analysis

- Alternative construct (if-then-else)
  - time for the evaluation of the condition
  - plus the maximum of the two alternatives

- Time-bounded loop
  - Maximum time specified for the loop

- Iteration-bounded loop
  - Time for condition evaluation, multiplied by the maximal number of iterations
  - Loop body multiplied by the max number of iterations
## WCET Calculation

### Timing schema

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive</td>
<td>WCET(primitive) = T(primitive)</td>
</tr>
<tr>
<td>Sequence</td>
<td>WCET(sequence) = ( \sum_{i} WCET(construct_i) )</td>
</tr>
<tr>
<td>Alternative</td>
<td>WCET(alternative) = WCET(condition) + ( \max( WCET(construct_1), WCET(construct_2)) )</td>
</tr>
<tr>
<td>LoopNumber</td>
<td>WCET(loopTail) = count * (WCET(body) + WCET(condition)) + WCET(overrun_statement)</td>
</tr>
<tr>
<td></td>
<td>WCET(loopHead) = WCET(init) + WCET(condition) + WCET(loopTail)</td>
</tr>
<tr>
<td>LoopTime</td>
<td>WCET(loopTime) = \min( Time-bound, WCET(loopTail) )</td>
</tr>
<tr>
<td>Subroutine</td>
<td>WCET(subroutine) = T(organization) + WCET(body)</td>
</tr>
</tbody>
</table>
Improvement

Can we derive additional program information in order to improve the outcome?

Standard Static Timing Analysis Produces very Conservative Estimates
based on slides by Zonghua Gu (顾宗华)

WCET ANALYSIS AS ILP FORMULATION
Problems with Timing Schemas

• High-level:
  ✓ worst-case assumptions everywhere

• Low-level:
  ✓ Assumes each instruction takes constant time

Advanced approach: WCET as part of compiler
  ✓ integrated approach possible
  ✓ ILP formulation(s)
Infeasible Paths

Motivating example

- Highly pessimistic estimation!
Program Flow Analysis

• Detect and exploit infeasible path information to improve WCET accuracy
• A and C cannot both execute in the code below

```c
if (i < 5) A;
else B;
if (i > 10) C;
else D;
```

• Either manual annotations or automatically-derived
Micro-architectural modeling

- To determine instruction timing

- Processor features affect program’s execution:
  - Clock cycles, ISAs, etc ...
  - Performance speed-up features: cache, pipeline, branch prediction, etc ...

- How significant?
  - Cache miss: 5 ~ 20 clock cycles, and increasing.
  - Branch misprediction: 3 ~ 19 clock cycles.
Separated approach

• Chicken-and-egg problem:
  ➢ Longest path is unknown without instr. timing
  ➢ Instr. timing can’t be determined without path info

• Common practice:
  ➢ Determine instr. timing first, then search longest path
  ➢ Static classification of cache state:
    • *always hit*,
    • *always miss*,
    • *possible hit/miss*.

(very) pessimistic timings
ILP: Integer Linear Programming

Crash course

• Linear constraints and cost function
  - maximize $c^T x$, subject to $Ax \leq b$
  - Efficient solvers available (CPLEX)

• Example: maximize $f$:

  $f = 3x + 5y + z$
  $0 \leq x, y, z \leq 100$
  $x + y + z = 200$
  $x + 2y \leq 160$

  - Optimal solution: $f = 520; x = 40; y = 60; z = 100$
  - Sub-optimal solution: $f = 480; x = 80; y = 30; z = 90$
Integrated Approach using ILP

- ILP framework: integrated μ-arch modeling (instr. Timing analysis) and longest path calc.
  - Constraints from Control Flow Graph (CFG)
  - Constraints from μ-arch modeling
  - Functional constraints (loop bounds, recursion depth, infeasible paths) by manual annotation or automatic derivation

- Constraints together with the cost function are submitted to ILP solver
Infeasible Paths

• The branch (K++) will never be taken

```java
J = 1;
If (J == 0)
    K++;  // symbolic execution
else
    K--;  // abstract interpretation
```

• Dataflow analysis can determine this
• Use it to improve WCET analysis accuracy
Infeasible Paths

- K=1 and J-- cannot execute both
  - symbolic execution along red path: K=1 ∧ K≥5

C code

```c
If (J== 0)
    K = 1;
else
    K = 10;
If (K < 5)
    J++;    
else
    J--; 
```

Control Flow Graph (CFG)
ILP for Modeling Program Flow

Basic constraints

- Sum of incoming exec counts equals Sum of outgoing exec counts

- $x = e_1 + e_2 = e_3 + e_4$
Program Flow Constraints

- $X_1 = E(\text{entry}) = 1$
- $X_2 = E(\text{loop}) + E(\text{entry}) = X_3 + X_4$
- $X_5 = X_3 + X_4 = E(\text{loop}) + E(\text{exit})$
- $X_6 = E(\text{exit}) = 1$
- Need a loop bound
  - Say $E(\text{loop}) \leq 100$
WCET Analysis via ILP

• Maximize WCET
  \[ WCET = c_1 X_1 + c_2 X_2 + c_3 X_3 + c_4 X_4 + c_5 X_5 + c_6 X_6 \]
  - \( c_i \) = Execution time of block \( i \) (constant)
  - \( X_i \) = Execution count of block \( i \) (ILP variable)

• Subject to flow constraints

• How to obtain \( c_1, c_2, c_3, c_4, c_5, c_6 \) ?
  - Estimates from micro-arch modeling
  - model execution times as ILP constraints
Integrating Infeasible Path Info

- When BBs 2 and 6 cannot execute together
  - due to red path being infeasible

- Add constraint on loop-bound
  - $X_2 + X_6 \leq X_1 = \text{loop bound} + 1$

- Not an exact encoding of infeasible path information
Micro-Architectural Modeling

- Exec time of an instruction is not constant

```plaintext
LD R2 [X]
R1 := R2 + R3
R4 := R1 − R5
```

- Execution of each instruction may hit/miss in I-cache
- Execution of LD may hit/miss in D-cache
- Pipeline stall may/may not occur at lines 2 and 3
Basic Idea

• For each instruction find out the maximum possible time I can take in any execution
  ➢ Exec. Time of I estimated to a constant
  ➢ specialize I w.r.t. diff. contexts (approximation of paths leading to I)
• For each exec of I with context c, find the maximum exec. time
  ➢ Need to find out # of times I is exec. with c
• Let the possible execution times of I under different hardware states be T1 < T2 < ... < Tn
Approach

• Statically analyze program flows to verify whether
  ➢ Instruction I will always hit
  ➢ Instruction I will always miss
  ➢ ...
• Reduce Execution time of I to constant
  ➢ More approximate, but more scalable
  ➢ Abstract Interpretation based approach
Instruction Cache

- With no hardware modeling, all instructions should be taken as misses

- Instead, categorize instructions as
  - AH (always hit)
  - AM (always miss)
  - PS (Persistent: second and all further executions are guaranteed to produce a hit)
    - Effect of cold misses – loops
  - NC (others)
Simulate cache contents

**Example: Fully associative with LRU policy**

- **Cache lines** = \( L_1, L_2, \ldots, L_n \)
  - \( L_1 \) is the youngest line
  - \( L_n \) is the oldest line
- **Memory blocks** = \( M_1, \ldots, M_m \)
  - Any block \( M_i \) can map to any cache line \( L_j \) during program execution

- Track set of memory blocks per cache line
  - must (conservative: at most 1 block)
  - may (optimistic: multiple blocks)
- Iterate over CFG until convergence
Cache Update; instruction fetch

Fully associative with LRU policy

- Cache hit
- Cache miss
Cache update; join operation

**Must Analysis**

1. In cache in both paths
2. If yes, take max age.
Cache update; join operation

May Analysis

1. In cache in some path.
2. If so, take min. age
Use of May/Must Analysis

Instruction classification

- Abstract cache per instruction I
  - Let M be the memory block containing I
  - If M is in some cache line within **must** cache, then I is **Always Hit**
  - If M is **not** in any cache line within **may** cache, then I is **Always Miss**

- How to categorize an instruction as “persistent”?
  - Misses the first time, but hits subsequently
  - Need to conservatively model removal of memory blocks from cache
Use of May/Must Analysis

**ILP formulation**

- Sum up to get WCET with cache modeling

- **Objective function** = $\sum_{I} #I \times wcet_{I}$
  - $#I$ is a ILP variable as before (flow equations defined)
  - $wcet_{I}$ is cost of instruction fetch

- $wcet_{I}$ depends on instr. classification
  - Let $hit\text{\_time} = t_{hit}$, $miss\text{\_time} = t_{miss}$
  - I is AH: $wcet_{I} = t_{hit}$
  - I is AM: $wcet_{I} = t_{miss}$
  - I is PS: $wcet_{I} = (t_{miss} + (#I -1)\times t_{hit}) / #I$
Static Timing Analysis

Recent advances

• Reconstruction of a binary file
  ➢ Generation of a flow graph out of the binary code

• Value-domain analysis
  ➢ Determines the memory addresses used and approximates the time for these accesses

• Cache & pipeline analysis
  ➢ Requires a good model of the hardware

• Execution path analysis
  ➢ User input on loop iterations, recursion depth, unreachable paths, etc.
Static Timing Analysis

aiT – WCET tool
Static Timing Analysis

Evaluation

- Complicated models for software behavior
  - Lack of real execution information
- Complicated models for hardware behavior
- Provision of additional information through system designers
  - Problem: Correctness & Completeness of this information
- Lack of professional tools
  - SoTA: research tools with limited applicability