CS4272: HW SW Codesign
Software Timing Analysis

Dr. Abhik Roychoudhury
School of Computing
National University of Singapore

Reading
- Chapter 7 of Real-time Systems and Software by Alan Shaw
  - http://www.princeton.edu/~yauli/publication.html

The context

<table>
<thead>
<tr>
<th>Architecture Instance</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping</td>
<td></td>
</tr>
<tr>
<td>Performance Analysis</td>
<td></td>
</tr>
<tr>
<td>Performance numbers</td>
<td></td>
</tr>
</tbody>
</table>

Performance Analysis
- Given a processor architecture A and a terminating program P
  - Provide the worst-case execution time estimate of P on A.
- Why do we care to do perf. analysis?
- May be we care, why worst-case?
  - Go for simulation?
- May be no simulation, how do we know the worst-case?
- Why is the architecture an issue at all?

Organization
- What is Timing Analysis?
  - The two main steps.
    - Path Analysis.
    - Micro-architecture modeling.
    - Primarily Control flow.
  - Modeling timing effects of Micro-architecture.
    - Cache, pipeline.
  - Chronos WCET Analysis tool for C programs

WCET
- Worst Case Execution Time of a program for a given hardware platform.
  - Sequential Terminating Programs.
  - Gets input, computes, produces output.
- Many inputs are possible.
  - Leads to different execution times.
- WCET : An upper bound on the execution time for all possible inputs.
Why need WCET?
- Performance estimation for Embedded system design.
  - Estimating uninterrupted software execution time on a given hardware (processor).
  - A building block for more complicated performance analysis.
  - Communicating multi-processor execution.
  - Helps estimate performance of a design point.
  - Serves as a sub-routine for Design Space Exploration.
- Schedulability analysis of Hard Real-time systems.
  - Such analysis assumes knowledge of WCET of each task being scheduled.
  - Rate Monotonic scheduling with tasks $T_1$, ..., $T_n$
    - Computation times $C_1$, ..., $C_n$
    - Period = deadline $D_1$, ..., $D_n$
    - Here $C_1$, ..., $C_n$ are the WCET (not average execution times of the programs)

Why need Analysis?
- To find WCET of a program, execute it for all possible inputs.
  - WCET by measurement.
  - Exponentially many possible inputs in terms of input size.
    - Insertion sort program
    - Similar problems will be encountered for WCET Analysis via platform simulation.
- Need access to platforms/simulators also!
  - Go for static analysis.

Measuring WCET
- What about single path programs such as matrix multiplication?
  - Execution path is independent of input data.
  - Still execution time can be variable.
    - Latency of floating point operation (e.g., multiplication) depends on the input data.
    - Not possible to try it on all possible platforms and then choose one.
    - Often trying to decide the platform as well.

WCET Analysis
- Analysis
  - Employ static analysis to compute an upper bound on WCET (Estimated WCET)
  - Observed
  - Actual

OK, analysis but ...
- ... why platform-aware analysis?
  - Exec. Time of an instr. can depend on
    - Operands
    - Context with which it is executed
      - Cache State
      - Pipeline State
    - ...
  - Exec Time distribution and WCET very diff. for diff. processors
Why platform-aware analysis

Distribution of execution times across inputs in a quicksort program on a simple and complex processor

But if I only analyze program...

I am still safe ---- No!

- Intra-task
  - Longest path in the program determined by time of instructions in the path!
- Inter-task
  - Additional context switch overhead due to sharing of HW data structures across tasks
  - Additional Cache Misses
  - What you deem as schedulable is not so!

WCET Analysis

- Program path analysis
  - All paths in control flow graph are not feasible.
- Micro-architectural modeling
  - Dynamically variable instruction execution time.
    - Cache, Pipeline, Branch Prediction
    - Out-of-order Pipelines

Restrictions

- Static analysis need not be on source program.
  - We can perform static analysis on assembly code of a given program.
  - The analysis is only for time taken, and not for the memory locations / values accessed.
  - No restriction on program data structures used for WCET analysis.
  - What about control flow?

Organization

- What is Timing Analysis?
- The two main steps.
  - Path Analysis.
  - Micro-architecture modeling.
  - Primarily Control flow.
- Modeling timing effects of Micro-architecture.
  - Cache, pipeline.
Timing Schema

- One of the first works on WCET analysis.
- Basically, perform control flow analysis to find the “longest” program path.
- The notion of “longest” is weighted
  - Take into account the cost of executing individual program elements.
  - Timing schema is a simple way of composing these costs.

Schema: Assignments

- Defined for elements in the source code, but considers a default assembly code.
  - \( T(\text{lhs} := \text{Exp}) = T(\text{addr}_\text{lhs}) + T(=) + T(\text{Exp}) \)
  - \( T(\text{addr}_\text{lhs}) \) is the time to calculate the address of v.
    - This is 0 if address is known at compile time.
  - \( T(=) \) is the time to do a store
  - \( T(\text{Exp}) \) is the time to evaluate the expression \( \text{Exp} \).

Example

\[
\text{A} := \text{B} + \text{C}
\]

\[
\begin{align*}
\text{Load B, R1} \\
\text{Load C, R2} \\
\text{R1 := R1 + R2} \\
\text{Load [A], R2} \\
\text{Store R1, [R2]} \\
\end{align*}
\]

\( T(\text{addr}_\text{A}) \neq 0 \)

\( T(\text{addr}_\text{A}) = 0 \)

Schema: Procedure Calls

- \( T(\text{p}(e_1,...,e_n)) = T(\text{call/ret}) + n \cdot T(\text{par}) + T(\text{body_of_p}) + T(e_1) + ... + T(e_n) \)
- \( T(\text{call/ret}) \) is the time for call and return.
- \( T(\text{par}) \) is the time for parameter passing.
- \( T(\text{e}_i) = 0 \) if expression \( \text{e}_i \) is a variable or constant.

If-then-else

- If \( B \) then \( S_1 \) else \( S_2 \)
- \( T(\text{if } B \text{ then } S_1 \text{ else } S_2) = \max(T_1, T_2) \)
- \( T_1 = T(B) + T(S_1) + T(\text{jump}) \)
- \( T_2 = T(B) + T(S_2) + T(\text{jump}) \)
- Assembly code schematic:
  - if \( B = \text{false} \) then jump to \( L_1 \)
  - \( S_1 \)
  - jump to \( L_2 \)
  - \( L_1: S_2 \)
  - \( L_2: \)

Loops

- While \( B \) do \( S \)
- Assembly code schematic:
  - Start: if \( B = \text{false} \) jump to \( \text{end} \)
  - \( S \)
  - jump to start
  - End:
  - \( T(\text{while B do S}) = (n+1)^*T(B) + n^*T(S) + (n+1)^*T(\text{jump}) \)
  - \( n = \) loop bound (which must be provided/computed)
Problems with timing schema

- **Language Level:**
  - Just a control flow analysis.
  - Insensitive to knowledge of infeasible paths.
- **Compiler level:**
  - How to integrate effect of compiler opt?
  - Easy to handle – schema on optimized code.
- **Architecture level:**
  - Instructions take constant time – Not true.
  - Cache hits, pipelining and other performance enhancing features.

Infeasible paths

```
SEQ
T = T5 + T6

WHILE
i = 0
T6

B
IF
T5 = (n+1)*T4 + n*T3

WHAT IF T1 > T2 and T3 = T0 + max(T1, T2)
S1 is executed only in the first loop iteration?
```

Organization

- What is Timing Analysis?
- The two main steps.
  - Path Analysis.
  - Micro-architecture modeling.
  - Primarily Control flow.
- Modeling timing effects of Micro-architecture.
- Chronos WCET Analysis tool for C programs

Two steps of ...

- WCET estimation
  - Weighted Longest path calculation
  - Detecting Infeasible paths.
  - Exploiting infeasible path information.
  - Micro-architectural modeling
    - Provides the "weights" for longest path calculation.
    - How to integrate the two steps?
      - Separated Approach --- more pragmatic
      - Integrated Approach (via ILP)

Program Flow Analysis

- Determine loop iterations, recursion depths
- Identify and exploit infeasible paths.
  ```
  if (i < 5) A;
  else B;
  if (i > 10) C; // A and C cannot
  else D; // execute together
  ```
- By manual annotations or automatically derived from data flow analysis.

Micro-architectural Modeling

- To determine the instruction timing
- Hardware affects program’s execution:
  - Clock cycles, ISAs, etc...
  - Performance speed-up features: cache, pipeline, branch prediction, etc...
- How significant?
  - Cache miss: 5 ~ 20+, ever increasing.
  - Branch misprediction: 3 ~ 19 clock cycles.
Separated Approaches

- A phase ordering problem:
  - Longest path is unknown without instr. timing.
  - Instr. timing cannot be determined without path info.
- Common practice in separated approaches:
  - Determine instr. timing first, then search longest path
  - Static Classification:
    - always hit,
    - always miss,
    - possible hit/miss,
  - Drawback: pessimism due to lack of path info.

```
for (i=0; i<100; i++) {
    if (...) A; // A maps to cache line X
    else B;
    C; // C maps to cache line X
}
```

This path to statement C always leads in a cache miss.
It might be the only path from start of program to statement C.

ILP – An Integrated Approach (1)

- ILP: Integer Linear Programming
  - Variables and linear constraints on them.
  - Cost function (linear) to optimize.

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

Optimal: \( f = 520; x = 40; y = 60; z = 100 \)
Non-Optimal: \( f = 480; x = 80; y = 30; z = 90 \)

ILP – An Integrated Approach (2)

- ILP framework: integrated \( \mu \)-arch modeling (instr. timing analysis) and longest path calc.
  - Constraints from Control Flow Graph (CFG).
  - Constraints from \( \mu \)-arch modeling.
  - Functional constraints (loop bounds, recursion depth, infeasible paths) by manual annotation or automatic data flow analysis.
  - Constraints together with the cost function are submitted to ILP solver.
  - In both approaches, program flow analysis via ILP.

Organization

- What is Timing Analysis?
  - The two main steps.
    - Path Analysis.
    - Micro-architecture modeling.
  - Primarily Control flow.
  - Modeling timing effects of Micro-architecture.
    - Cache, pipeline.
  - Chronos WCET Analysis tool for C programs

Infeasible paths

- \( J = 1; \)
- If \( J == 0 \) { \( K++; // this branch will never be taken \}
  } else { \( K--; \)
  }

Only possible to know via data flow analysis.
Infeasible paths

- Infeasible sequence of branches in general
  
  ```
  If (J== 0) {
    K = 1
  } else {
    K = 10
  }
  
  If (K < 5) {
    J++;
  } else {
    J--;
  }
  ```

- Cannot be executed together
- Such infeasible paths should not be a witness to our WCET estimate.

Control Flow Graph

An Infeasible path

Modeling Program Flows

- Path-based
  - Enumerate paths and find longest path
  - Expensive!
  - Need to remove longest path if it is infeasible.

- Tree-based
  - Bottom-up pass of Syntax Tree
  - Timing Schema
  - How to integrate infeasible path info?

Modeling Program Flows

- Integer Linear Programming
  - Modeling of control flow.
  - Can take into account certain infeasible path information if available.
  - Efficient solvers available e.g. CPLEX
  - Forms the back-end of most state-of-the-art timing analyzers.

Extending Timing Schema

- Timing schema is a Control Flow Analysis.
  - At each branch, it enumerates both choice to estimate the time of a code fragment.
  - These estimates are combined.
  - Effect of enumerating all possible program paths in the control flow graph and estimating their times.
  - But some of these paths are never taken due to data flow!
Path representations

- Terminating programs, Finite Paths.
- Paths for each control construct can be modeled via simple regular expressions.
- All feasible program paths can also be represented by regular expressions.
- How do we let the user input specific info. about infeasible paths?
  - We are not discussing the issue of infeasible path pattern detection (yet).

Example

```c
Procedure Check_data()
{
    int i = 0, morecheck = 1, wrongone = -1, datasize = 10;
    while (morecheck)
    {
        if (data[i] < 0)
        {  wrongone = i;   morecheck = 0; }
        else
            if  (++i >= datasize) morecheck = 0;
    }
    if (wrongone >= 0)
    { handle_exception(wrongone); return 0; }
    else  return 1;
}
```

User information

- loop L [1,10] times
  - Bound on loop iterations
- Samepath(A, C)
  - A and C are executed together
  - (not A) imply loop L 10 times
  - If A is not executed, L is iterated 10 times.
- Execute A [0,1] times inside L
  - A is executed at most once inside L

Overall Approach

- 1. Describe all paths in the CFG as a Regular Expression $\psi$
- 2. Allow the user to input annotations in a "Description Language": $I_1, \ldots, I_n$
- 3. Convert $I_1, \ldots, I_n$ to Regular Expressions $\psi_1, \ldots, \psi_n$ [Easy stuff: Not discussed here]
- 4. Set of feasible paths then given by $\psi \cap \psi_1 \cap \ldots \cap \psi_n$
- 5. Eliminate intersections in $\psi \cap \psi_1 \cap \ldots \cap \psi_n$ to produce an equivalent disjunctive form $X_1 \cup X_2 \cup \ldots \cup X_k$
- 6. Compute $T_i = T(X_i)$ using timing schema approach.

Overall approach

- 5. Eliminate intersections in $\psi \cap \psi_1 \cap \ldots \cap \psi_n$ to produce an equivalent disjunctive form $X_1 \cup X_2 \cup \ldots \cup X_k$
- 6. Compute $T_i = T(X_i)$ using timing schema approach.

WCET = max $\{T_1, T_2, \ldots, T_k\}$
- Step 5 has high complexity.
Complexity issues

- \((a + b)^{100} \cap ((a+b)^* a (a+b)^*)\)
- Models a loop with 100 iterations
- Captures every path via \(a\) in some loop iteration.
- Removal of intersection operator leads to enumeration of many cases.
- Essentially loop unrolling (undesirable!)
- No easy solutions to this problem
- But you can ...

A) Choose an user-annotation description language whose corresponding regular expressions can be intersected efficiently
- Loop path information is problematic
B) Delay the intersection removal until WCET analysis and perform approximations
- \(T(v \cap ((a+b)^*a(a+b)^*) \cap ((a+b)^*b(a+b)^*)) := \min(T(<(a+b)^*a(a+b)^*>), T(<(a+b)^*b(a+b)^*>))\)
- So, it is difficult to integrate infeasible path information into Timing Schema.

And now to ILP!

We are dealing with aggregated execution counts of nodes/edges of CFG.

ILP modeling of Control Flow

Subject to these constraints
- Maximize
  - \(c1*X1 + c2*X2 + c3*X3 + c4*X4 + c5*X5 + c6*X6\)
  - \(c1\) = Execution time of block 1 (constant).
  - \(X1\) = Execution count of block 1 (ILP variable).
- How to get \(c1,c2,c3,c4,c5,c6\) ?
  - Accurate estimates via micro-arch modeling
- How to integrate infeasible path info?

Timing Analysis via ILP

Need a loop bound
Say \(E\text{ (loop)} \leq 100\)
For a loop of the form
\(I = 0\)
while \((I < 100 \&\& \text{not flag})\) { ...}

Infeasible path info.

Assuming loop-bound = 100
Add the constraint
\(X2 = X6 \leq 100\)
Not an exact encoding of the infeasible path information, though
User information

- Many of the user information can be gleaned through (limited) dataflow analysis.
  - E.g. loop \([1,10]\) follows from value of datasize and loop termination condition.
- The discussion is not how to analyze infeas. paths
  - Less ambitious goal: if some info. resulting from data flow anal. is known, how to integrate it into WCET analysis.
- Infeasible path detection,
  - Many approaches exist, based on constraint propagation and solving.

Infeasible path detection - Example

```
J == 0 ??
K = 1 K = 10
K < 5 ??
J++
YN
YN
K ≥ 5
```

Constraint Propagation

- Over Control Flow Graph
  - Start from an outgoing edge of a branch
  - This gives an initial constraint.
  - Traverse the CFG backwards by transforming the constraint at each step.
    - How?
      - Stop when constraint store is unsatisfiable.
- Many issues -
  - Constraint solvers ?
  - Full-fledged loop unrolling ?
    - Heuristics to stop after few iterations
  - Limited detection – infeasible paths within a loop/loop-iteration.

Weakest pre-condition

- Constraint accumulated \(\phi(X_1, \ldots, X_k)\)
- One step weakest pre-condition computation w.r.t. statement \(s\)
  - Effect constraint of \(s\) is
    - \(\psi_s(X_1, \ldots, X_k, X_1', \ldots, X_k')\)
    - Effect constraint of \(X = X + 1\) over vars. \((X,Y,Z)\) is
      - \(\psi(X,Y,Z, X', Y', Z') = (X' = X + 1 \land Y' = Y \land Z' = Z)\)
    - \(WP(X_1, \ldots, X_k) = \forall X_1', \ldots, X_k' \psi(X_1, \ldots, X_k, X_1', \ldots, X_k') \Rightarrow \phi(X_1', \ldots, X_k')\)

Constraint Solvers

- Simplify Theorem Prover – Compaq SRC
  - Integrates automatic decision procedures.
    - Equality
    - Arithmetic
    - Arrays
  - Sound, incomplete
    - Unsatisfiable constraint may not be detected.
    - Incomplete detection of infeasible paths patterns – OK !

Loop Bound Detection

- Specific kind of infeasible path information
  - Develop offline customized analysis on source code instead of using generic constraint solvers.
  - Need to care for
    - Multiple exits of loop.
    - Dependence of loop counter on outer loop counters.
    - Full-fledged data-flow (never done, always overestimate)

\[
\text{for } (i = 1; i <= N; i++)\{ \\
\text{for } (j = 1; j <= N; j++) \\
\sum_{1 \leq i \leq N} \sum_{1 \leq j \leq N} 1 \\
\sum_{1 \leq i \leq N} (\sum_{1 \leq j \leq N} 1 - \\
\sum_{1 \leq i \leq N} 1) \\
\} \\
\]
***Summary so far***

- Program Flow analysis
  - Control flow modeled as ILP equations.
  - Limited data flow modeled as ILP inequalities.
    - Involves offline infeasible path detection.
    - Maximize objective function – Linear function of execution counts of basic blocks.
- Micro-architectural modeling
  - Constants denoting exec. time of basic blocks.
  - How to estimate these constants?
    - We will discuss this now

Why not use ILP alone?

- For many micro-architectural features.
  - Timing effects captured by ILP inequalities.
  - Plug these with the program flow modeling, and solve one huge ILP to get WCET estimate.
  - Not scalable in terms of solution time for modern processor features.
    - Big issue.
  - Problem size may explode --- varying of parameters of arch (cache size).
    - Smaller issue but the problem file itself may explode.

Organization

- What is Timing Analysis?
- The two main steps.
  - Path Analysis.
  - Micro-architecture modeling.
  - Primarily Control flow.
- Modeling timing effects of Micro-architecture.
  - Cache, pipeline.
- Chronos WCET Analysis tool for C programs

Micro-architectural modeling

- Cost of an instruction is not constant.
  - LD R2 [X] (I0)
  - R1 := R2 + R3 (I1)
  - R4 := R1 – R5 (I2)

Execution of each instruction may hit/miss in I-cache
Execution of I0 may hit/miss in D-cache
Pipeline stall may/may not occur at I2.

Basic ideas

- 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

Basic ideas

- Let the possible execution times of I under differing hardware states be T1< T2 < ...< Tn
  - Easy to enumerate this set for prediction based hardware data structures (cache, branch prediction)
  - Expensive for pipeline modeling, particularly consider pipelined execution of variable latency instructions ...
One possibility

- If we find that the possible execution times of I are $T_1(\text{hit}) < T_2(\text{miss})$
  - Find the maximum number of times I can miss $\#\text{miss}(I)$
  - Then the contribution of I to WCET is
    - $\#\text{miss}(I) * T_2 + (#I - \#\text{miss}(I)) * T_1$
  - Specialize an instruction based on hardware states rather than program paths
    - Need to develop bounds on $\#\text{miss}(I)$

Another possibility

- 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

- One concrete hardware data structure.
- Without no hardware modeling, all instructions should be taken as misses.
- Instead we can categorize some instructions as "always hit"
  - Coarse modeling.
  - For certain instructions, even the "worst case" may not be a miss!

Categorization ...

- ... of instructions
  - AH (always hit)
  - AM (always miss)
  - PS (Persistent: second and all further executions are guaranteed to produce a hit)
    - Effect of cold misses
  - NC (not AH, AM, PS)

Cache-basics

- Redundant storage to reduce memory access time.
- Many memory blocks map to a single cache line
- F: Memory Block $\rightarrow$ Cache lines
  - Given a memory block $m$, $F(m)$ returns the set of cache lines it can map to.
  - If $F(m)$ is always a singleton set, then we have a direct mapped cache.
  - If $|F(m)| = n$, we have $n$-way set associative cache.
  - If $F(m) = \text{Set of all cache lines}$, then we have a fully associative cache (any memory block can map to any cache line).

The cache

- 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
- Do not refer to physical cache lines
- Memory blocks $= S_1, S_2, \ldots, S_m$
  - Any block $S_i$ can map to any cache line $L_j$ during program execution
Concrete cache update

Concrete cache update

Abstract cache state

Abstract cache state

May analysis

May analysis

Must analysis

Must analysis

High-level view

High-level view
How to use such analysis?

- Let I be an instruction at control loc. CL.
- Let M be the memory block containing I.
  - Consider abstract cache state at CL obtained via “must analysis”.
  - If M is in some cache line within this abstract cache state, then I is Always Hit.
  - For cache state at L obtained via ‘may analysis’
    - If M is not in any cache line within this abstract cache state, 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 cache blocks from cache.

Persistence Analysis

Use of may-must analysis

- Separate micro-architectural modeling from program path analysis.
  - Use may-must analysis to find worst-case cache behavior of each instruction.
  - Sum up to get WCET with cache modeling.
  - Objective function = \( \Sigma I \cdot \#I \cdot wcetI \)
    - wcetI is a constant
    - \#I is an ILP variable as before, flow equations defined.
  - We can slightly better this formulation easily

Use of may-must analysis

- Let hit_time = t1, miss_time = t2
- Number of accesses of I == \#I (ILP variable)
  - I is AH
    - \#I * t1 = contribution of I to WCET
  - I is AM
    - \#I * t2 = contribution of I to WCET
  - I is PS
    - \((\#I - 1) \cdot t1 + t2 = contribution of I to WCET
  - Formulation is still linear, solve via ILP.

Can we improve precision?

- If we can bound the number of misses of instr. I (via constraints)
  - No need to reduce exec. Time of I to constant
  - Contribution of I to WCET
    - \#miss(I) * t2 + \((\#I - \#miss(I)) \cdot t1
  - Takes the idea of PS categorization one step further (distinguish between the different executions of I).
  - How to develop such constraints?
    - ILP, Expensive !!

Summary so far

- Modeling timing effects of I-cache
  - Abstract Interpretation to categorize instr
  - ILP based modeling is more expensive.
- I-cache does not have timing anomalies
  - Can assume all accesses are misses.
  - Very pessimistic, but estimate still safe!
  - For certain processors, even this is not true!
  - Adding worst-case of each instruction may produce an estimate lower than the global worst-case!
Organization

- What is Timing Analysis?
- The two main steps.
  - Path Analysis.
  - Micro-architecture modeling.
  - Primarily Control flow.
- Modeling timing effects of Micro-architecture.
  - Cache, pipeline.
- Chronos WCET Analysis tool for C programs

Chronos WCET estimation tool

- Program path analysis
  - All paths in control flow graph are not feasible.
- Advanced Micro-architectural modeling
  - Dynamically variable instruction execution time
    - Cache, Branch Prediction
- Out-of-order Pipelines
- http://www.comp.nus.edu.sg/~rprembed/chronos/

Pipeline + IC + BP

Parameters:
- Functional Units: ALU 1 cycle; MUL [1, 4]; FPU [1, 12]
- 4KB I-Cache: 4-way, 32 sets, 32 bytes/line, cache miss 10 cycles
- Gag dynamic branch predictor: 4-bit BHR, 16-entry BHT

<table>
<thead>
<tr>
<th>Program</th>
<th>Obs. WCET</th>
<th>Est. WCET</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix</td>
<td>124044</td>
<td>119546</td>
<td>1.05</td>
</tr>
<tr>
<td>fft</td>
<td>1109567</td>
<td>1122997</td>
<td>1.02</td>
</tr>
<tr>
<td>whet</td>
<td>933647</td>
<td>106219</td>
<td>1.11</td>
</tr>
<tr>
<td>fir</td>
<td>46462</td>
<td>63674</td>
<td>1.37</td>
</tr>
<tr>
<td>ludcmp</td>
<td>12254</td>
<td>17414</td>
<td>1.42</td>
</tr>
<tr>
<td>minver</td>
<td>8514</td>
<td>12576</td>
<td>1.49</td>
</tr>
</tbody>
</table>
**Processor Pipelines Additional Slides**
Abhik Roychoudhury

---

**O-o-O exec.**
Several instructions may reside in the same pipeline stage in the same clock cycle.
- An ADD instruction and MUL instruction in the EX stage since they use different func. units

**Pipeline stalls**
- Instruction I+1 may not proceed to EX since it depends on the result of instruction I

**Mask stall latency** by out-of-order exec
- If I+1 cannot proceed, let I+2 proceed if all its operands are available.
### O-o-O execution (1)

<table>
<thead>
<tr>
<th>Ready</th>
<th>Instruction</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>mult r3 r1 r2</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>add r3 r3 8</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>and r3 r3 0xff</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>addu r5 r4 8</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>mult r5 r5 r6</td>
<td>4</td>
</tr>
</tbody>
</table>

**Latencies**
- **MULTU**: 1 ~ 4 cycles
- **ALU**: 1 cycle

**Instruction sequence**
- MULTU: 1 - 4 cycles
- ALU: 1 cycle

**Partial order of dependences**

```
A -> B -> C
D -> E
```

Instruction A executes 4 cycles

### O-o-O execution (2)

<table>
<thead>
<tr>
<th>Ready</th>
<th>Instruction</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>mult r3 r1 r2</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>add r3 r3 8</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>and r3 r3 0xff</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>addu r5 r4 8</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>mult r5 r5 r6</td>
<td>4</td>
</tr>
</tbody>
</table>

**Latencies**
- **MULTU**: 1 ~ 4 cycles
- **ALU**: 1 cycle

**Instruction sequence**
- MULTU: 1 - 4 cycles
- ALU: 1 cycle

Instruction A executes 3 cycles

### Difficulty in modeling

<table>
<thead>
<tr>
<th>Ready</th>
<th>Instruction</th>
<th>Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>mult r3 r1 r2</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>add r3 r3 8</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>and r3 r3 0xff</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>addu r5 r4 8</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>mult r5 r5 r6</td>
<td>4</td>
</tr>
</tbody>
</table>

**Latencies**
- **MULTU**: 1 ~ 4 cycles
- **ALU**: 1 cycle

**Instruction sequence**
- MULTU: 1 - 4 cycles
- ALU: 1 cycle

Instruction A executes 3 cycles

### Timing Anomaly

- Overall WCET of an instruction sequence cannot be obtained from WCET of each instruction.
- Need to consider all possible execution times of each instruction to safely estimate WCET!
  - Expensive enumeration
- Very different from cache modeling.
  - Worst-case cache behavior of an instruction sequence can be safely estimated by considering all cache accesses as misses.
- Modeling of out-of-order pipeline behavior is extremely complex, and not discussed here!