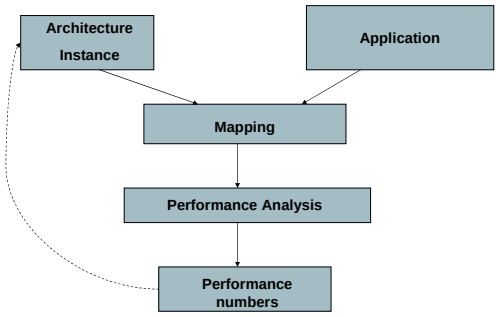
CS4272: HW SW Codesign
Software Timing Analysis

Dr. Abhik Roychoudhury
School of Computing
National University of Singapore

Reading

- Chapter 7 of Real-time Systems and Software by Alan Shaw
- Tulika Mitra and Abhik Roychoudhury, "Worst-case Execution Time and Energy Analysis", Chapter in Compiler Design Handbook (2nd Edition), CRC Press, To appear.
- Yau-Tsun Steven Li and Sharad Malik, *"Performance Analysis of Embedded Software Using Implicit Path Enumeration"*, in *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, June, 1995, pp. 456 - 461.
  - http://www.princeton.edu/~yauli/publication.html

The context



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 ?
- An Early solution -- Timing Schema.
- The two main steps.
  - Path Analysis.
  - Micro-architecture modeling.
- Modeling Program Flows.
  - 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.

Why need WCET ?

- 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 ?

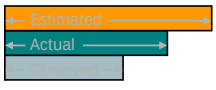
- To find WCET of a program, execute it for all possible inputs.
  - WCET by measurement.
  - Exponentially many possible inputs in terms of input size.
    - Insertion sort program
  - Similar problems will be encountered for WCET Analysis via platform simulation.
- Need access to platforms/simulators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

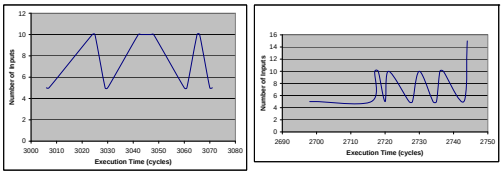
- **Analysis**
  - Employ static analysis to compute an upper bound on WCET (Estimated WCET)



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 ?

Restrictions

- Restrictions on control flow
  - 1. No unbounded loops
    - Common sense. Otherwise how to guarantee time?
  - 2. No unbounded recursion
    - Similar issue.
  - 3. No dynamic function calls
    - Need to statically know the functions called, and the possible call sites of these functions.

Organization

- What is Timing Analysis ?
- An Early solution -- Timing Schema.
- The two main steps.
  - Path Analysis.
  - Micro-architecture modeling.
- Modeling Program Flows.
  - 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(lhs := Exp) = T(addr_lhs) + T(:=) + T(Exp)
  - T(addr_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(Exp) is the time to evaluate the expression Exp.

Example

A := B + C

Load B, R1
Load C, R2
R1 := R1 + R2
Load [A], R2
Store R1, [R2]

T(addr_A) ≠ 0

Load B, R1
Load C, R2
R1 := R1 + R2
Store R1, A

T(addr_A) = 0

Schema: Procedure Calls

- T($p(e_1,...,e_n)$) =
  - T(call/ret) + n*T(par)+T(body_of_p) +
  - $T(e_1)$ + … + $T(e_n)$
- T(call/ret) is the time for call and return.
- T(par) is the time for parameter passing.
- $T(e_i)$ = 0 if expression e_i is a variable or constant.

If-then-else

- If B then S1 else S2
- T(if B then S1 else S2) = max(T1, T2)
  - T1 = T(B) + T(S1) + T(jump)
  - T2 = T(B) + T(S2) + T(jump)
  - Assembly code schematic:
    - if B=false then jump to L1
    - S1
    - jump to L2
    - L1: S2
    - L2:

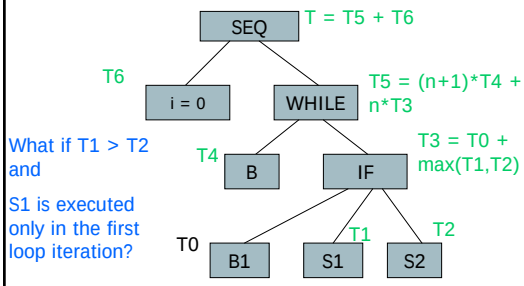
Loops

- While B do S
- Assembly code schematic:
  - Start: if B = false jump to end
  - S
  - jump to start
  - End:
- T(while B do S) =
  - (n+1)*T(B) + n*T(S) + (n+1)*T(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



Organization

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

Separated Approaches

```
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 <= x, y, z <= 100
x + y + z = 200
x + 2y <= 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 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 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 ?
- An Early solution -- Timing Schema.
- The two main steps.
  - Path Analysis.
  - Micro-architecture modeling.
- Modeling Program Flows.
  - 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

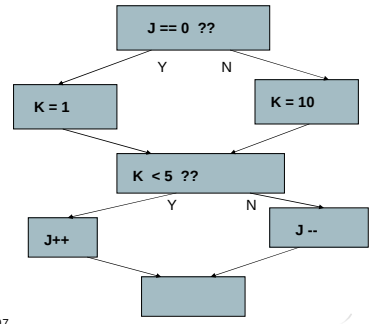
- Infeasible sequence of branches in general

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

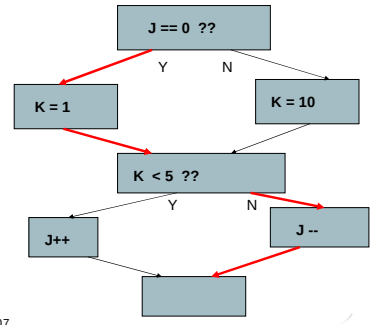
Cannot be executed together

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

```
Procedure Check_data()
{       int  i = 0, morecheck = 1, wrongone = -1, datasize = 10;
  L:    while (morecheck)
  LB:   {
              if  (data[i] < 0)
  A:              {  wrongone = i;   morecheck = 0; }
              else
  B:                  if  (++i >= datasize) morecheck = 0;
        }
        if (wrongone >= 0)
  C:            { handle_exception(wrongone); return 0; }
  C':   else  return 1;
}
```


Example

```
Procedure Check_data()
{       int  ....
  L:    while (...)
  LB:   {
              if  (...)
  A:              { .... }
              else
  B:                  if  (...)  ...;
        }
        if (...)
  C:            { ...}
  C':   else  ...
}
```

Alphabet=Control labels in code

Set of all paths

= **L. (LB.(A +B))* . (C+C')**

This set is obtained from the structure of the control flow graph.

Includes many infeasible paths.

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 ψ
- 2. Allow the user to input annotations in a "Description Language" : $I_1,...,I_n$
- 3. Convert $I_1,...,I_n$ to Regular Expressions $\varphi_1,..., \varphi_n$ *[Easy stuff : Not discussed here]*
- 4. Set of feasible paths then given by
  - $\psi \cap \varphi_1 \cap ... \cap \varphi_n$
  - How to use this for WCET Analysis ?

Overall approach

- 5. Eliminate intersections in $\psi \cap \varphi_1 \cap ... \cap \varphi_n$ to produce an equivalent disjunctive form
  - $X_1 \cup X_2 \cup ... \cup X_k$
  - Each X_i should be a regular expression
- 6. Compute $T_i = T(X_i)$ using timing schema approach.
  - WCET = max $\{T_1, T_2,..., T_k\}$
- Step 5 has high complexity.

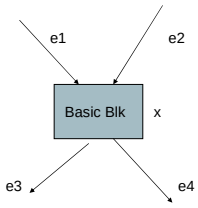
Complexity issues

- (a + b)^100 ∩ ((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 …

Complexity issues

- 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(ψ ∩ ((a+b)*a (a+b)*) ∩ ((a+b)* b (a+b)*)) :=
  - min(T(ψ∩((a+b)*a(a+b)*)), T(ψ∩((a+b)*b(a+b)*)))
- So, it is difficult to integrate infeasible path information into Timing Schema.

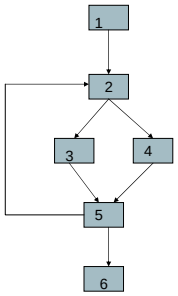
And now to ILP !



x = e1 + e2
= e3 + e4

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

ILP modeling of Control Flow



X1 = E(entry) = 1
X2 = E(loop) + E(entry) = X3 + X4
X5 = X3 + X4 = E(loop) + E(exit)
X6 = E(exit) = 1

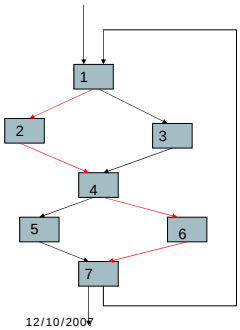
Need a loop bound
Say E(loop) ≤ 100
For a loop of the form

```
I = 0
while ( I < 100 && not flag) { ….
```


Timing Analysis via ILP

- 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 ?

Infeasible path info.



Assuming loop-bound =100
Add the constraint

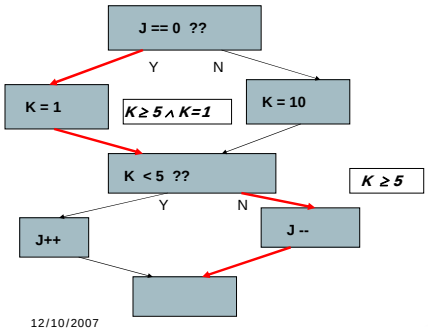
X2 + X6 ≤ 100

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



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 $\varphi(X_1,\ldots,X_k)$
- 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
      - $\psi_s(X, Y, Z, X', Y', Z') == (X'=X+1 \wedge Y' = Y \wedge Z' = Z)$
  - $WP(X_1,\ldots,X_k)$ =
  - $\forall X_1',\ldots,X_k' \;\; \psi_s(X_1,\ldots,X_k,X_1',\ldots,X_k') \Rightarrow \varphi(X_1',\ldots, X_k')$

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 path 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)**

```
for (I =1; I <= N ; I++){
   for (J=I; J <= N; J++){
```

$\sum_{1 \le I \le N} \sum_{I \le J \le N} 1$

$= \sum_{1 \le I \le N} (\sum_{1 \le J \le N} 1 - \sum_{1 \le J < I} 1)$

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 ?
- An Early solution -- Timing Schema.
- The two main steps.
  - Path Analysis.
  - Micro-architecture modeling.
- Modeling Program Flows.
  - 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 T1(hit) < T2(miss)
  - Find the maximum number of times I can miss #miss(I)
  - Then the contribution of I to WCET is
    - #miss(I)*T2 + (#I – #miss(I))*T1
  - Specialize an instruction based on hardware states rather than program paths
    - Need to develop bounds on #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.
- With 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)

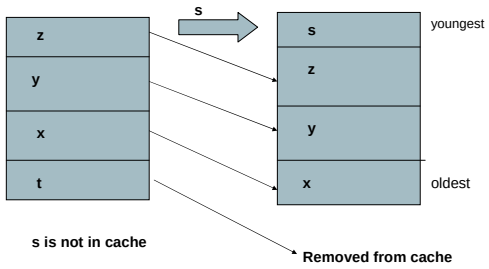
Cache-basics

- Redundant storage to reduce memory access time.
- Many memory blocks map to a single cache line
- F: Memory Block → 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)| is n, we have n-way set associative cache.
  - If F(m) = Set of all cache lines, then we have a fully associative cache (any memory block can map to any cache line).

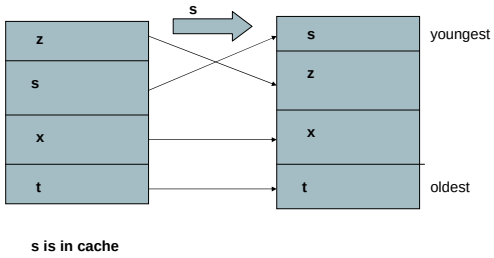
The cache

- Fully associative with LRU policy.
- Cache lines = L_1, L_2, …, 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_m
  - Any block S_i can map to any cache line L_j during program execution

Concrete cache update



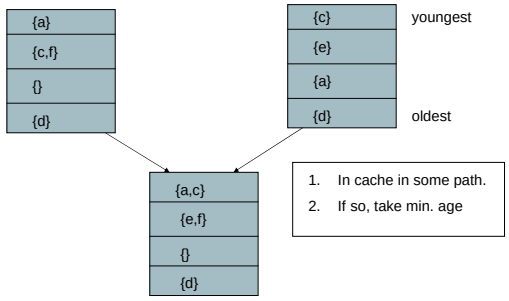
Concrete cache update



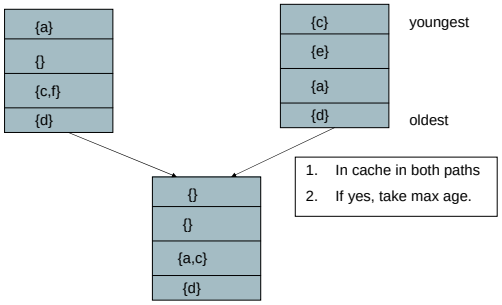
Abstract cache state

- In the concrete cache state c, if a block is in cache line x, its age is x
  - Cache line 1 is youngest.
- In the abstract cache state c', each line x contains a set of memory blocks
  - B ∈ c'(L_x) at a program point p means …
  - When control reaches p, B may (must) be in cache with min (max) age = x
  - Direction of approximation in abstraction.

May analysis



Must analysis



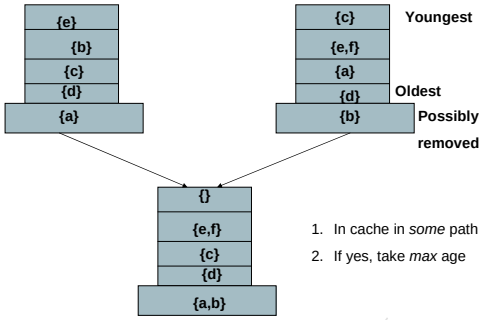
High-level view

- … of may/must analysis.
  - For each program point, initialize a default abstract cache state (empty cache)
  - Update abstract cache states at each program point by propagation based on control flow
    - Propagation at control flow merge points shown in past 2 slides.
    - Propagation done differently for may and must analysis.
  - Iterate the updates over and over, until the abstract cache states for all basic blocks become stable.
    - Why is termination guaranteed?

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 = $\sum_I$ #I * $wcet_I$
    - $wcet_I$ is a constant
    - #I is a 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)*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))*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

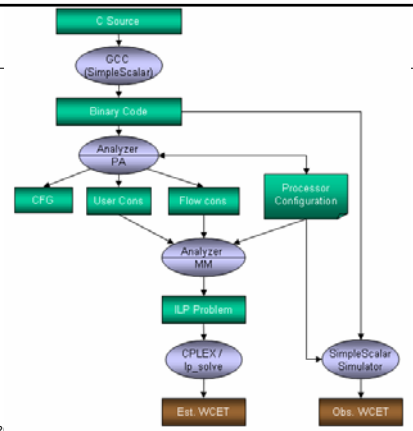
- **Modeling timing effects of I-cache**
  - Abstract Interpretation to categorize instr
  - ILP based modeling is more expensive.
- **I-cache does not have timing anomalies**
  - Can assume all accesses are misses.
  - Very pessimistic, but estimate still safe !
- For certain processors, even this is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

Organization

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

Chronos WCET estimation tool

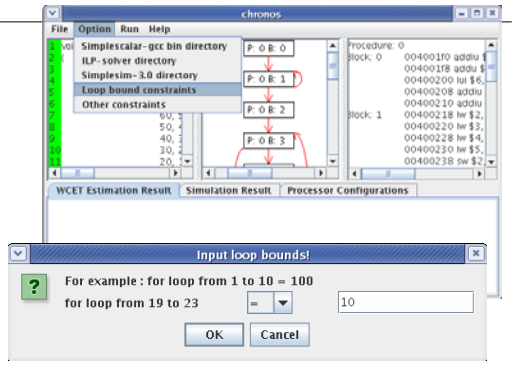
- Program path analysis
  - All paths in control flow graph are not feasible.
- Advanced Micro-architectural modeling
  - Dynamically variable instruction execution time
    - Cache, Branch Prediction
    - Out-of-order Pipelines
- http://www.comp.nus.edu.sg/~rpembed/chronos/

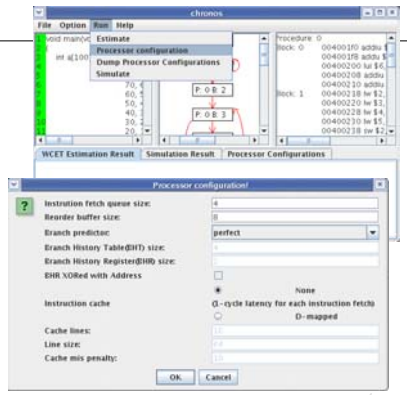


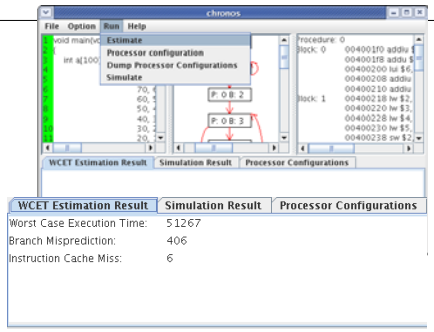
Pipeline + IC + BP

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

Program	Obs. WCET	Est. WCET	Ratio
matsum	101673	111779	1.10
fdct	10646	11958	1.12
fft	1098567	132997	1.12
whet	933647	106219	1.11
fir	46642	63679	1.37
ludcmp	12254	17414	1.42
minver	8514	13576	1.59







Summing up …

- Program flow modeling (typically by ILP)
  - Combine reasoning about timing of program fragments.
  - Exploiting Infeasible path information.
  - Difficult to use model checking for this purpose.
- Micro-architectural modeling (customized analysis)
  - Exec. Time of each instruction is not constant.
  - Worst-case not found by adding up worst-cases of code fragments – non compositional.
    - Efficient analysis tools (Chronos) to overcome this.

Processor Pipelines Additional Slides

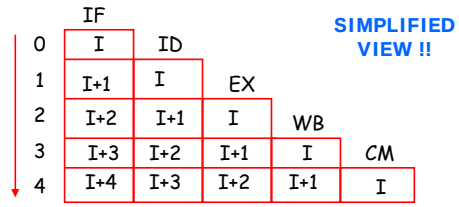
Abhik Roychoudhury

Pipelined exec.

Divide the execution of an instruction into stages

Instruction I+1 can proceed before I completes

Increased throughput, lower overall execution time



SIMPLIFIED VIEW !!

An O-o-O pipeline

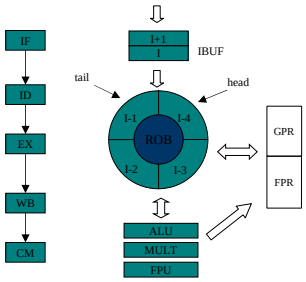
Mem => I-buffer (inorder)

IBUF => ROB (in-order)

ROB => FU (out-of-order), (Instr still in ROB)

FU => ROB (out-of-order) (forward data)

Update register file, free ROB entry (in-order)



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)

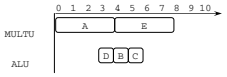
#	Ready Cycle	Instruction
A	0	mult r3 r1 r2
B	1	add r3 r3 8
C	2	and r3 r3 0xff
D	3	addu r5 r4 8
E	4	mult r5 r5 r6

Instruction sequence

MULTU	1 ~ 4 cycles
ALU	1 cycle

Latencies

Partial order of dependences



Instruction A executes 4 cycles

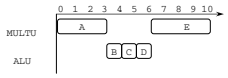
O-o-O execution (2)

#	Ready Cycle	Instruction
A	0	mult r3 r1 r2
B	1	add r3 r3 8
C	2	and r3 r3 0xff
D	3	addu r5 r4 8
E	4	mult r5 r5 r6

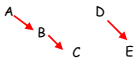
Instruction sequence

MULTU	1 ~ 4 cycles
ALU	1 cycle

Latencies



Instruction A executes 3 cycles



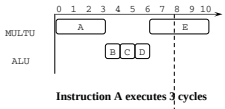
Difficulty in modeling

#	Ready Cycle	Instruction
A	0	mult r3 r1 r2
B	1	add r3 r3 8
C	2	and r3 r3 0xff
D	3	addu r5 r4 8
E	4	mult r5 r5 r6

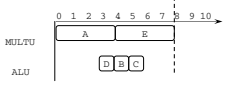
Instruction sequence

MULTU	1 ~ 4 cycles
ALU	1 cycle

Latencies



Instruction A executes 3 cycles



Instruction A executes 4 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 !