



Computer Architecture
A Quantitative Approach, Sixth Edition

Chapter 3

Instruction-Level Parallelism and Its Exploitation

Copyright © 2019, Elsevier Inc. All rights Reserved

1

Introduction

Introduction

- Pipelining became universal technique in 1985
 - Overlaps execution of instructions
 - Exploits “Instruction Level Parallelism”
- Beyond this, there are two main approaches:
 - Hardware-based dynamic approaches
 - Used in server and desktop processors
 - Not used as extensively in PMP processors
 - Compiler-based static approaches
 - Not as successful outside of scientific applications

Copyright © 2019, Elsevier Inc. All rights Reserved

2

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI
 - Pipeline CPI =
 - Ideal pipeline CPI +
 - Structural stalls +
 - Data hazard stalls +
 - Control stalls
 - Parallelism with basic block is limited
 - Typical size of basic block = 3-6 instructions
 - Must optimize across branches



Data Dependence

- Loop-Level Parallelism
 - Unroll loop statically or dynamically
 - Use SIMD (vector processors and GPUs)
- Challenges:
 - Data dependency
 - Instruction j is data dependent on instruction i if
 - Instruction i produces a result that may be used by instruction j
 - Instruction j is data dependent on instruction k and instruction k is data dependent on instruction i
 - Dependent instructions cannot be executed simultaneously



Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall
- Data dependence conveys:
 - Possibility of a hazard
 - Order in which results must be calculated
 - Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect



Copyright © 2019, Elsevier Inc. All rights Reserved

5

Name Dependence

- Two instructions use the same name but no flow of information
 - Not a true data dependence, but is a problem when reordering instructions
 - Antidependence: instruction j writes a register or memory location that instruction i reads
 - Initial ordering (i before j) must be preserved
 - Output dependence: instruction i and instruction j write the same register or memory location
 - Ordering must be preserved
- To resolve, use register renaming techniques



Copyright © 2019, Elsevier Inc. All rights Reserved

6

Other Factors

- Data Hazards
 - Read after write (RAW)
 - Write after write (WAW)
 - Write after read (WAR)
- Control Dependence
 - Ordering of instruction i with respect to a branch instruction
 - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
 - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch



Examples

- Example 1:
 $\text{add } x1, x2, x3$
 $\text{beq } x4, x0, L$
 $\text{sub } x1, x1, x6$
 $L: \dots$
 $\text{or } x7, x1, x8$
 - or instruction dependent on add and sub
- Example 2:
 $\text{add } x1, x2, x3$
 $\text{beq } x12, x0, \text{skip}$
 $\text{sub } x4, x5, x6$
 $\text{add } x5, x4, x9$
 skip:
 $\text{or } x7, x8, x9$
 - Assume $x4$ isn't used after skip
 - Possible to move sub before the branch



Compiler Techniques for Exposing ILP

- Pipeline scheduling
 - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
 - Example:


```
for (i=999; i>=0; i=i-1)
x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0



Copyright © 2019, Elsevier Inc. All rights Reserved

9

Pipeline Stalls

```
Loop:    fld      f0,0(x1)
         stall
         fadd.d f4,f0,f2
         stall
         stall
         fsd f4,0(x1)
         addi x1,x1,-8
         bne x1,x2,Loop
```

```
Loop:    fld      f0,0(x1)
         addi x1,x1,-8
         fadd.d f4,f0,f2
         stall
         stall
         fsd f4,0(x1)
         bne x1,x2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0



Copyright © 2019, Elsevier Inc. All rights Reserved

10

Loop Unrolling

- Loop unrolling
 - Unroll by a factor of 4 (assume # elements is divisible by 4)
 - Eliminate unnecessary instructions
- ```

Loop: fld f0,0(x1)
 fadd.d f4,f0,f2
 fsd f4,0(x1) //drop addi & bne
 fld f6,-8(x1)
 fadd.d f8,f6,f2
 fsd f8,-8(x1) //drop addi & bne
 fld f0,-16(x1)
 fadd.d f12,f0,f2
 fsd f12,-16(x1) //drop addi & bne
 fld f14,-24(x1)
 fadd.d f16,f14,f2
 fsd f16,-24(x1)
 addi x1,x1,-32
 bne x1,x2,Loop

```
- note: number of live registers vs. original loop



Copyright © 2019, Elsevier Inc. All rights Reserved

11

## Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```

Loop: fld f0,0(x1)
 fld f6,-8(x1)
 fld f8,-16(x1)
 fld f14,-24(x1)
 fadd.d f4,f0,f2
 fadd.d f8,f6,f2
 fadd.d f12,f0,f2
 fadd.d f16,f14,f2
 fsd f4,0(x1)
 fsd f8,-8(x1)
 fsd f12,-16(x1)
 fsd f16,-24(x1)
 addi x1,x1,-32
 bne x1,x2,Loop

```

- 14 cycles
- 3.5 cycles per element



Copyright © 2019, Elsevier Inc. All rights Reserved

12

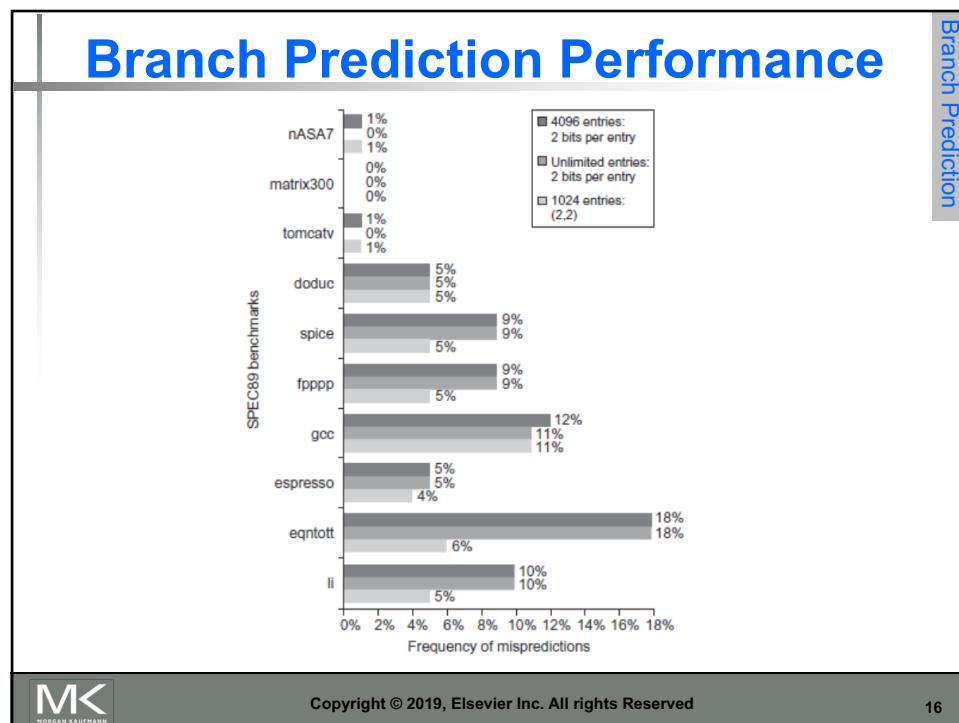
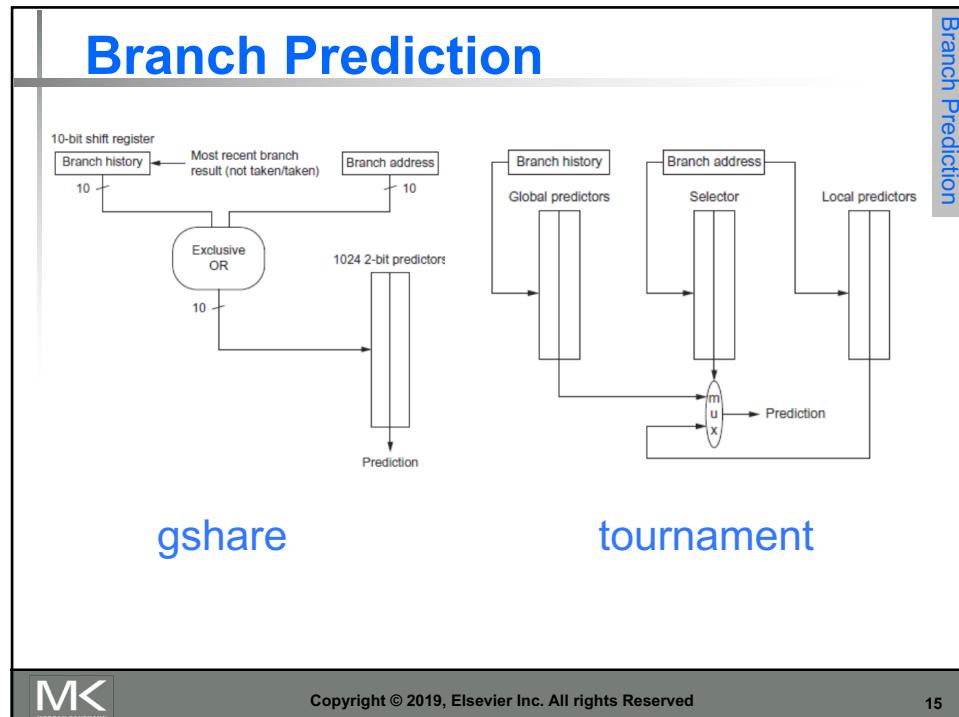
## Strip Mining

- Unknown number of loop iterations?
  - Number of iterations =  $n$
  - Goal: make  $k$  copies of the loop body
  - Generate pair of loops:
    - First executes  $n \bmod k$  times
    - Second executes  $n / k$  times
    - “Strip mining”

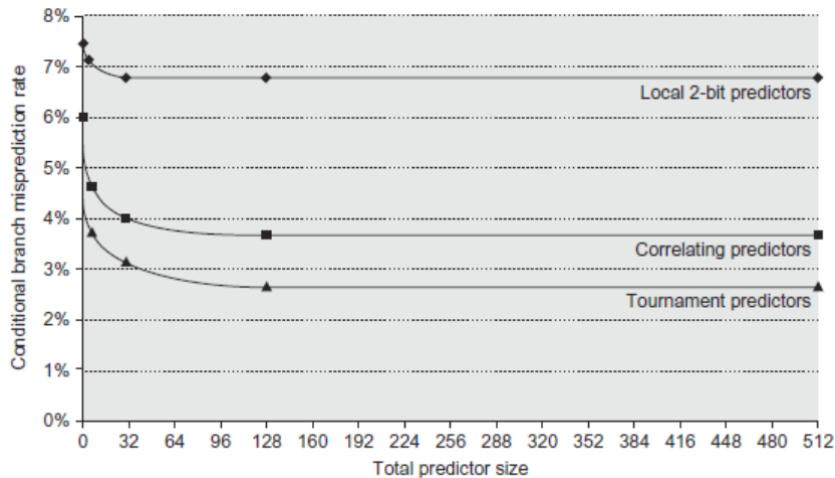


- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding  $n$  branches
    - $(m,n)$  predictor: behavior from last  $m$  branches to choose from  $2^m$   $n$ -bit predictors
- Tournament predictor:
  - Combine correlating predictor with local predictor





## Branch Prediction Performance



Copyright © 2019, Elsevier Inc. All rights Reserved

17

Branch Prediction

## Tagged Hybrid Predictors

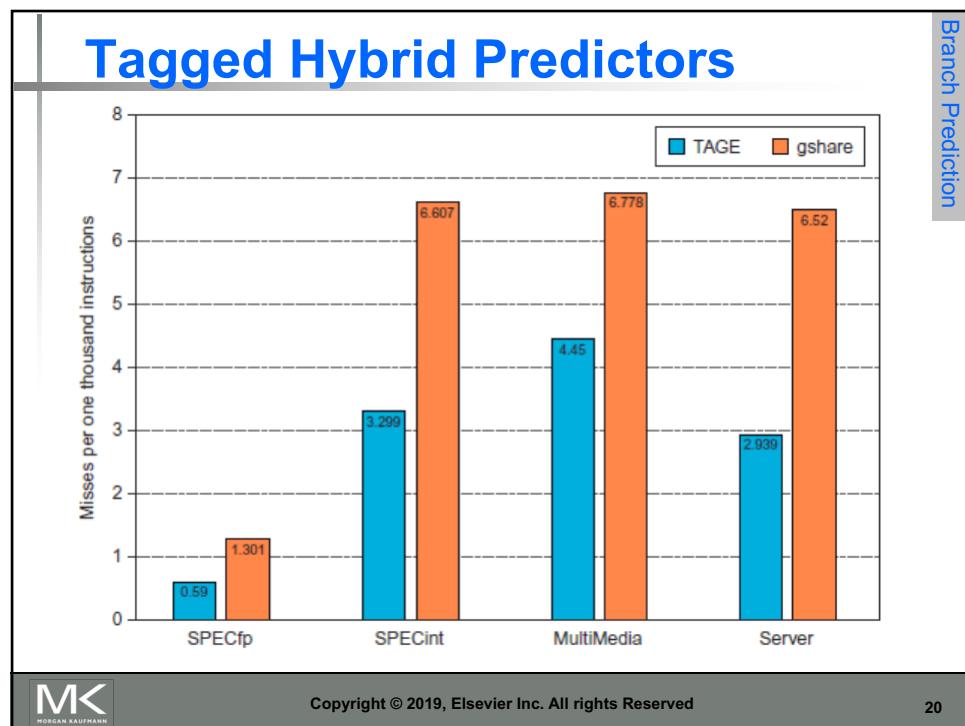
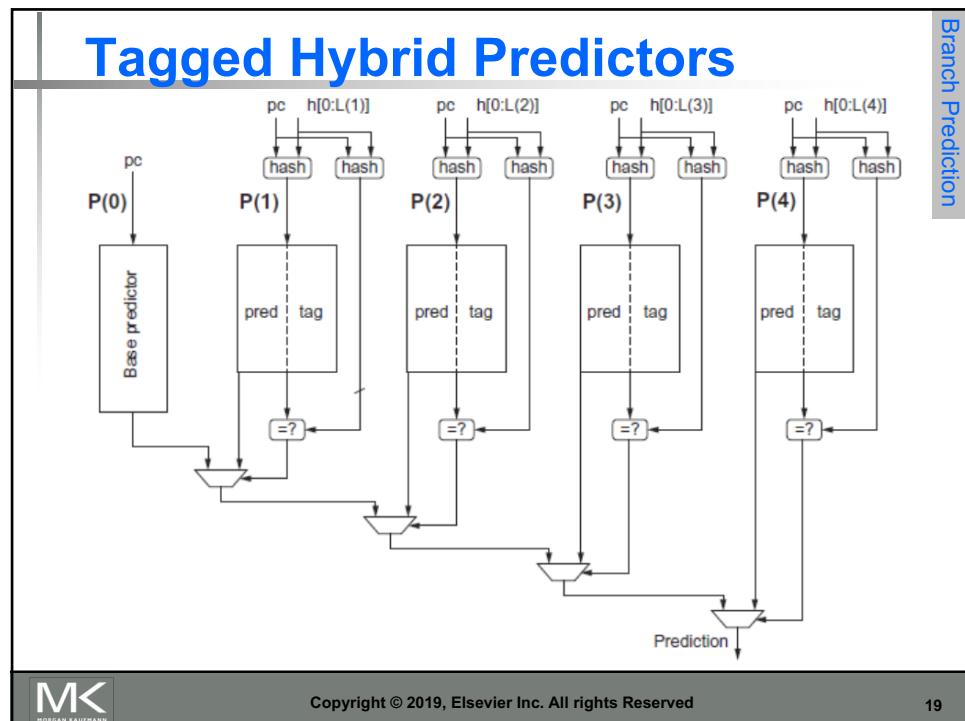
- Need to have predictor for each branch and history
  - Problem: this implies huge tables
  - Solution:
    - Use hash tables, whose hash value is based on branch address and branch history
    - Longer histories may lead to increased chance of hash collision, so use multiple tables with increasingly shorter histories



Copyright © 2019, Elsevier Inc. All rights Reserved

18

Branch Prediction



## Dynamic Scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow
- Advantages:
  - Compiler doesn't need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time
- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions



Copyright © 2019, Elsevier Inc. All rights Reserved

21

## Dynamic Scheduling

- Dynamic scheduling implies:
  - Out-of-order execution
  - Out-of-order completion
- Example 1:  
`fdiv.d f0,f2,f4  
fadd.d f10,f0,f8  
fsub.d f12,f8,f14`
  - fsub.d is not dependent, issue before fadd.d



Copyright © 2019, Elsevier Inc. All rights Reserved

22

## Dynamic Scheduling

- Example 2:

    fdiv.d f0,f2,f4  
    fmul.d f6,f0,f8  
    fadd.d f0,f10,f14

- fadd.d is not dependent, but the antidependence makes it impossible to issue earlier without register renaming



Copyright © 2019, Elsevier Inc. All rights Reserved

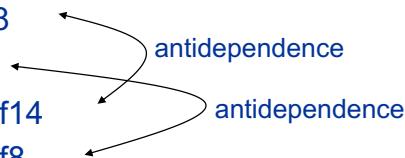
23

Dynamic Scheduling

## Register Renaming

- Example 3:

    fdiv.d f0,f2,f4  
    fadd.d f6,f0,f8  
    fsd f6,0(x1)  
    fsub.d f8,f10,f14  
    fmul.d f6,f10,f8



- name dependence with f6



Copyright © 2019, Elsevier Inc. All rights Reserved

24

Dynamic Scheduling

## Register Renaming

- Example 3:

```
fdiv.d f0,f2,f4
fadd.d S,f0,f8
fsd S,0(x1)
fsub.d T,f10,f14
fmul.d f6,f10,T
```

- Now only RAW hazards remain, which can be strictly ordered



## Register Renaming

- Tomasulo's Approach
  - Tracks when operands are available
  - Introduces register renaming in hardware
    - Minimizes WAW and WAR hazards
- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Reservation station number of instruction providing the operand values

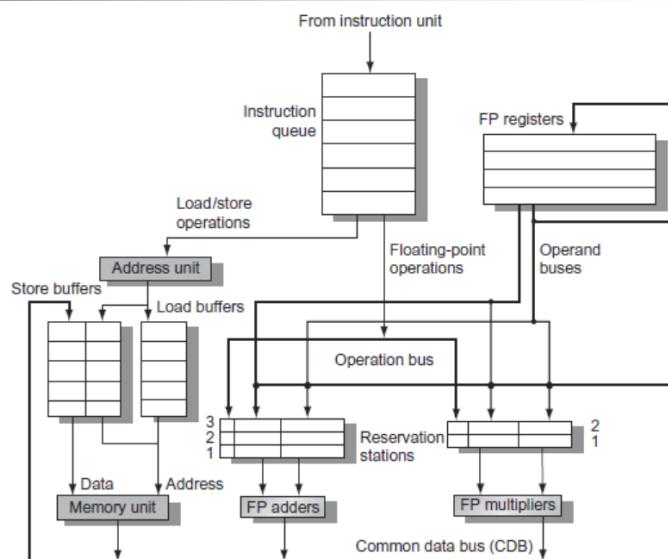


## Register Renaming

- RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
- Pending instructions designate the RS to which they will send their output
  - Result values broadcast on a result bus, called the common data bus (CDB)
- Only the last output updates the register file
- As instructions are issued, the register specifiers are renamed with the reservation station
- May be more reservation stations than registers
- Load and store buffers
  - Contain data and addresses, act like reservation stations



## Tomasulo's Algorithm



## Tomasulo's Algorithm

- Three Steps:
  - Issue
    - Get next instruction from FIFO queue
    - If available RS, issue the instruction to the RS with operand values if available
    - If operand values not available, stall the instruction
  - Execute
    - When operand becomes available, store it in any reservation stations waiting for it
    - When all operands are ready, issue the instruction
    - Loads and store maintained in program order through effective address
    - No instruction allowed to initiate execution until all branches that proceed it in program order have completed
  - Write result
    - Write result on CDB into reservation stations and store buffers
      - (Stores must wait until address and value are received)



Copyright © 2019, Elsevier Inc. All rights Reserved

29

## Example

| Instruction status |       |         |              |  |  |  |  |
|--------------------|-------|---------|--------------|--|--|--|--|
| Instruction        | Issue | Execute | Write result |  |  |  |  |
| f1d f6,32(x2)      | ✓     | ✓       | ✓            |  |  |  |  |
| f1d f2,44(x3)      | ✓     | ✓       |              |  |  |  |  |
| fmul.d f0,f2,f4    | ✓     |         |              |  |  |  |  |
| fsub.d f8,f2,f6    | ✓     |         |              |  |  |  |  |
| fdiv.d f0,f0,f6    | ✓     |         |              |  |  |  |  |
| fadd.d f6,f8,f2    | ✓     |         |              |  |  |  |  |

| Reservation stations |      |      |                    |    |       |       |               |
|----------------------|------|------|--------------------|----|-------|-------|---------------|
| Name                 | Busy | Op   | Vj                 | Vk | Qj    | Qk    | A             |
| Load1                | No   |      |                    |    |       |       |               |
| Load2                | Yes  | Load |                    |    |       |       | 44 + Regs[x3] |
| Add1                 | Yes  | SUB  | Mem[32 + Regs[x2]] |    | Load2 |       |               |
| Add2                 | Yes  | ADD  |                    |    | Add1  | Load2 |               |
| Add3                 | No   |      |                    |    |       |       |               |
| Mult1                | Yes  | MUL  | Regs[f4]           |    | Load2 |       |               |
| Mult2                | Yes  | DIV  | Mem[32 + Regs[x2]] |    | Mult1 |       |               |

| Register status |       |       |    |      |      |       |     |     |     |
|-----------------|-------|-------|----|------|------|-------|-----|-----|-----|
| Field           | f0    | f2    | f4 | f6   | f8   | f10   | f12 | ... | f30 |
| Qi              | Mult1 | Load2 |    | Add2 | Add1 | Mult2 |     |     |     |



Copyright © 2019, Elsevier Inc. All rights Reserved

30

## Tomasulo's Algorithm

- Example loop:

```
Loop: fld f0,0(x1)
 fmul.d f4,f0,f2
 fsd f4,0(x1)
 addi x1,x1,8
 bne x1,x2,Loop // branches if x16 != x2
```

Dynamic Scheduling



Copyright © 2019, Elsevier Inc. All rights Reserved

31

## Tomasulo's Algorithm

| Instruction status |                |       |         |              |  |
|--------------------|----------------|-------|---------|--------------|--|
| Instruction        | From iteration | Issue | Execute | Write result |  |
| fld f0,0(x1)       | 1              | ✓     | ✓       |              |  |
| fmul.d f4,f0,f2    | 1              | ✓     |         |              |  |
| fsd f4,0(x1)       | 1              | ✓     |         |              |  |
| fld f0,0(x1)       | 2              | ✓     | ✓       |              |  |
| fmul.d f4,f0,f2    | 2              | ✓     |         |              |  |
| fsd f4,0(x1)       | 2              | ✓     |         |              |  |

| Reservation stations |      |       |              |          |       |       |              |
|----------------------|------|-------|--------------|----------|-------|-------|--------------|
| Name                 | Busy | Op    | Vj           | Vk       | Qj    | Qk    | A            |
| Load1                | Yes  | Load  |              |          |       |       | Regs[x1] + 0 |
| Load2                | Yes  | Load  |              |          |       |       | Regs[x1] - 8 |
| Add1                 | No   |       |              |          |       |       |              |
| Add2                 | No   |       |              |          |       |       |              |
| Add3                 | No   |       |              |          |       |       |              |
| Mult1                | Yes  | MUL   |              | Regs[f2] | Load1 |       |              |
| Mult2                | Yes  | MUL   |              | Regs[f2] | Load2 |       |              |
| Store1               | Yes  | Store | Regs[x1]     |          |       | Mult1 |              |
| Store2               | Yes  | Store | Regs[x1] - 8 |          |       | Mult2 |              |

| Register status |       |    |    |       |    |     |     |     |     |
|-----------------|-------|----|----|-------|----|-----|-----|-----|-----|
| Field           | f0    | f2 | f4 | f6    | f8 | f10 | f12 | ... | f30 |
| Qj              | Load2 |    |    | Mult2 |    |     |     |     |     |

Dynamic Scheduling



Copyright © 2019, Elsevier Inc. All rights Reserved

32

## Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results if prediction was correct
- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - I.e. updating state or taking an execution

Hardware-Based Speculation



Copyright © 2019, Elsevier Inc. All rights Reserved

33

## Reorder Buffer

- Reorder buffer – holds the result of instruction between completion and commit
- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?
- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit

Hardware-Based Speculation



Copyright © 2019, Elsevier Inc. All rights Reserved

34

## Reorder Buffer

- Issue:
  - Allocate RS and ROB, read available operands
- Execute:
  - Begin execution when operand values are available
- Write result:
  - Write result and ROB tag on CDB
- Commit:
  - When ROB reaches head of ROB, update register
  - When a mispredicted branch reaches head of ROB, discard all entries

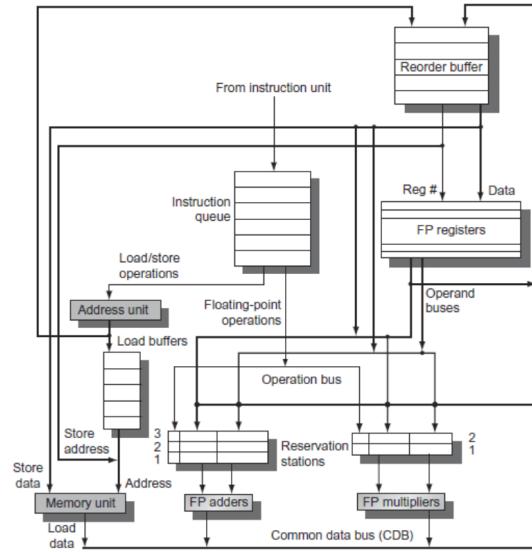


## Reorder Buffer

- Register values and memory values are not written until an instruction commits
- On misprediction:
  - Speculated entries in ROB are cleared
- Exceptions:
  - Not recognized until it is ready to commit



## Reorder Buffer



Copyright © 2019, Elsevier Inc. All rights Reserved

37

## Reorder Buffer

| Reorder buffer |      |             |           |              |       |
|----------------|------|-------------|-----------|--------------|-------|
| Entry          | Busy | Instruction | State     | Destination  | Value |
| 1              | No   | f1d         | f6,32(x2) | Commit       | f6    |
| 2              | No   | f1d         | f2,44(x3) | Commit       | f2    |
| 3              | Yes  | fmul.d      | f0,f2,f4  | Write result | f0    |
| 4              | Yes  | fsub.d      | f8,f2,f6  | Write result | f8    |
| 5              | Yes  | fdiv.d      | f0,f0,f6  | Execute      | f0    |
| 6              | Yes  | fadd.d      | f6,f8,f2  | Write result | f6    |

| Reservation stations |      |        |                    |                    |    |    |      |   |
|----------------------|------|--------|--------------------|--------------------|----|----|------|---|
| Name                 | Busy | Op     | Vj                 | Vk                 | Qj | Qk | Dest | A |
| Load1                | No   |        |                    |                    |    |    |      |   |
| Load2                | No   |        |                    |                    |    |    |      |   |
| Add1                 | No   |        |                    |                    |    |    |      |   |
| Add2                 | No   |        |                    |                    |    |    |      |   |
| Add3                 | No   |        |                    |                    |    |    |      |   |
| Mult1                | No   | fmul.d | Mem[44 + Regs[x3]] | Regs[f4]           |    |    | #3   |   |
| Mult2                | Yes  | fdiv.d |                    | Mem[32 + Regs[x2]] | #3 |    | #5   |   |

| FP register status |     |    |    |    |    |    |     |     |     |     |
|--------------------|-----|----|----|----|----|----|-----|-----|-----|-----|
| Field              | f0  | f1 | f2 | f3 | f4 | f5 | f6  | f7  | f8  | f10 |
| Reorder #          | 3   |    |    |    |    |    | 6   |     | 4   | 5   |
| Busy               | Yes | No | No | No | No | No | Yes | ... | Yes | Yes |



Copyright © 2019, Elsevier Inc. All rights Reserved

38

## Multiple Issue and Static Scheduling

- To achieve CPI < 1, need to complete multiple instructions per clock
- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors

Multiple Issue and Static Scheduling



Copyright © 2019, Elsevier Inc. All rights Reserved

39

## Multiple Issue

| Common name               | Issue structure  | Hazard detection   | Scheduling               | Distinguishing characteristic                                       | Examples                                                             |
|---------------------------|------------------|--------------------|--------------------------|---------------------------------------------------------------------|----------------------------------------------------------------------|
| Superscalar (static)      | Dynamic          | Hardware           | Static                   | In-order execution                                                  | Mostly in the embedded space: MIPS and ARM, including the Cortex-A53 |
| Superscalar (dynamic)     | Dynamic          | Hardware           | Dynamic                  | Some out-of-order execution, but no speculation                     | None at the present                                                  |
| Superscalar (speculative) | Dynamic          | Hardware           | Dynamic with speculation | Out-of-order execution with speculation                             | Intel Core i3, i5, i7; AMD Phenom; IBM Power 7                       |
| VLIW/LIW                  | Static           | Primarily software | Static                   | All hazards determined and indicated by compiler (often implicitly) | Most examples are in signal processing, such as the TI C6x           |
| EPIC                      | Primarily static | Primarily software | Mostly static            | All hazards determined and indicated explicitly by the compiler     | Itanium                                                              |

Multiple Issue and Static Scheduling



Copyright © 2019, Elsevier Inc. All rights Reserved

40

## VLIW Processors

- Package multiple operations into one instruction
- Example VLIW processor:
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
- Must be enough parallelism in code to fill the available slots

Multiple Issue and Static Scheduling



Copyright © 2019, Elsevier Inc. All rights Reserved

41

## VLIW Processors

| Memory reference 1 | Memory reference 2 | FP operation 1     | FP operation 2    | Integer operation/branch |
|--------------------|--------------------|--------------------|-------------------|--------------------------|
| fld f0,0(x1)       | fld f6,-8(x1)      |                    |                   |                          |
| fld f10,-16(x1)    | fld f14,-24(x1)    |                    |                   |                          |
| fld f18,-32(x1)    | fld f22,-40(x1)    | fadd.d f4,f0,f2    | fadd.d f8,f6,f2   |                          |
| fld f26,-48(x1)    |                    | fadd.d f12,f0,f2   | fadd.d f16,f14,f2 |                          |
|                    |                    | fadd.d f20,f18,f2  | fadd.d f24,f22,f2 |                          |
| fsd f4,0(x1)       | fsd f8,-8(x1)      | fadd.d f28,f26,f24 |                   |                          |
| fsd f12,-16(x1)    | fsd f16,-24(x1)    |                    |                   | addi x1,x1,-56           |
| fsd f20,24(x1)     | fsd f24,16(x1)     |                    |                   |                          |
| fsd f28,8(x1)      |                    |                    |                   | bne x1,x2,Loop           |

Multiple Issue and Static Scheduling

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
  - Binary code compatibility



Copyright © 2019, Elsevier Inc. All rights Reserved

42

## Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches:
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
- Issue logic is the bottleneck in dynamically scheduled superscalars

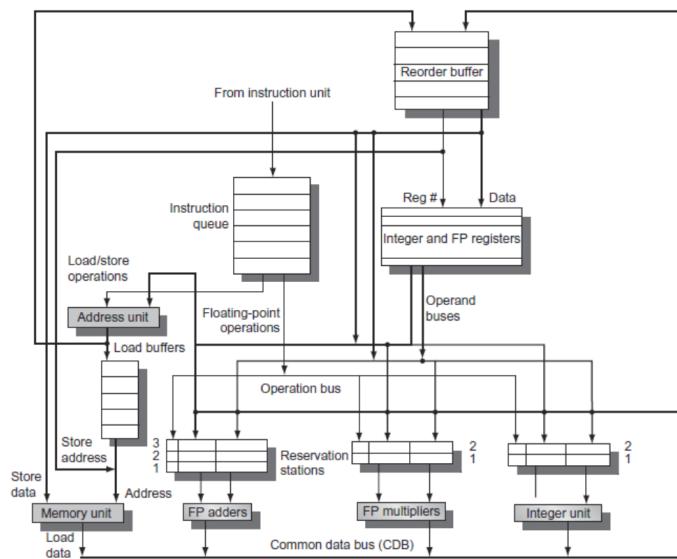


Copyright © 2019, Elsevier Inc. All rights Reserved

43

Dynamic Scheduling, Multiple Issue, and Speculation

## Overview of Design



Copyright © 2019, Elsevier Inc. All rights Reserved

44

Dynamic Scheduling, Multiple Issue, and Speculation

## Multiple Issue

- Examine all the dependencies among the instructions in the bundle
- If dependencies exist in bundle, encode them in reservation stations
- Also need multiple completion/commit
- To simplify RS allocation:
  - Limit the number of instructions of a given class that can be issued in a “bundle”, i.e. on FP, one integer, one load, one store



## Example

```
Loop: ld x2,0(x1) //x2=array element
 addi x2,x2,1 //increment x2
 sd x2,0(x1) //store result
 addi x1,x1,8 //increment pointer
 bne x2,x3,Loop //branch if not last
```



## Example (No Speculation)

| Iteration number | Instructions   | Issues at clock cycle number | Executes at clock cycle number | Memory access at clock cycle number | Write CDB at clock cycle number | Comment          |
|------------------|----------------|------------------------------|--------------------------------|-------------------------------------|---------------------------------|------------------|
| 1                | ld x2,0(x1)    | 1                            | 2                              | 3                                   | 4                               | First issue      |
| 1                | addi x2,x2,1   | 1                            | 5                              |                                     | 6                               | Wait for ld      |
| 1                | sd x2,0(x1)    | 2                            | 3                              | 7                                   |                                 | Wait for addi    |
| 1                | addi x1,x1,8   | 2                            | 3                              |                                     | 4                               | Execute directly |
| 1                | bne x2,x3,Loop | 3                            | 7                              |                                     |                                 | Wait for addi    |
| 2                | ld x2,0(x1)    | 4                            | 8                              | 9                                   | 10                              | Wait for bne     |
| 2                | addi x2,x2,1   | 4                            | 11                             |                                     | 12                              | Wait for ld      |
| 2                | sd x2,0(x1)    | 5                            | 9                              | 13                                  |                                 | Wait for addi    |
| 2                | addi x1,x1,8   | 5                            | 8                              |                                     | 9                               | Wait for bne     |
| 2                | bne x2,x3,Loop | 6                            | 13                             |                                     |                                 | Wait for addi    |
| 3                | ld x2,0(x1)    | 7                            | 14                             | 15                                  | 16                              | Wait for bne     |
| 3                | addi x2,x2,1   | 7                            | 17                             |                                     | 18                              | Wait for ld      |
| 3                | sd x2,0(x1)    | 8                            | 15                             | 19                                  |                                 | Wait for addi    |
| 3                | addi x1,x1,8   | 8                            | 14                             |                                     | 15                              | Wait for bne     |
| 3                | bne x2,x3,Loop | 9                            | 19                             |                                     |                                 | Wait for addi    |

Dynamic Scheduling, Multiple Issue, and Speculation



Copyright © 2019, Elsevier Inc. All rights Reserved

47

## Example (Multiple Issue with Speculation)

| Iteration number | Instructions   | Issues at clock number | Executes at clock number | Read access at clock number | Write CDB at clock number | Commits at clock number | Comment           |
|------------------|----------------|------------------------|--------------------------|-----------------------------|---------------------------|-------------------------|-------------------|
| 1                | ld x2,0(x1)    | 1                      | 2                        | 3                           | 4                         | 5                       | First issue       |
| 1                | addi x2,x2,1   | 1                      | 5                        |                             | 6                         | 7                       | Wait for ld       |
| 1                | sd x2,0(x1)    | 2                      | 3                        |                             |                           | 7                       | Wait for addi     |
| 1                | addi x1,x1,8   | 2                      | 3                        |                             | 4                         | 8                       | Commit in order   |
| 1                | bne x2,x3,Loop | 3                      | 7                        |                             |                           | 8                       | Wait for addi     |
| 2                | ld x2,0(x1)    | 4                      | 5                        | 6                           | 7                         | 9                       | No execute delay  |
| 2                | addi x2,x2,1   | 4                      | 8                        |                             | 9                         | 10                      | Wait for ld       |
| 2                | sd x2,0(x1)    | 5                      | 6                        |                             |                           | 10                      | Wait for addi     |
| 2                | addi x1,x1,8   | 5                      | 6                        |                             | 7                         | 11                      | Commit in order   |
| 2                | bne x2,x3,Loop | 6                      | 10                       |                             |                           | 11                      | Wait for addi     |
| 3                | ld x2,0(x1)    | 7                      | 8                        | 9                           | 10                        | 12                      | Earliest possible |
| 3                | addi x2,x2,1   | 7                      | 11                       |                             | 12                        | 13                      | Wait for ld       |
| 3                | sd x2,0(x1)    | 8                      | 9                        |                             |                           | 13                      | Wait for addi     |
| 3                | addi x1,x1,8   | 8                      | 9                        |                             | 10                        | 14                      | Executes earlier  |
| 3                | bne x2,x3,Loop | 9                      | 13                       |                             |                           | 14                      | Wait for addi     |

Dynamic Scheduling, Multiple Issue, and Speculation

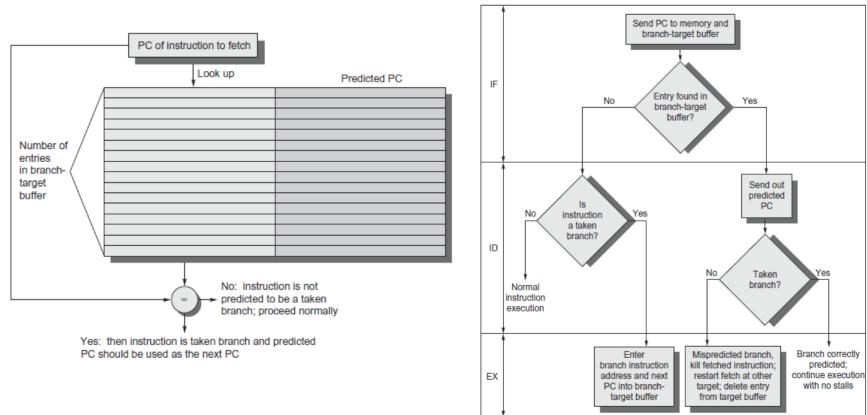


Copyright © 2019, Elsevier Inc. All rights Reserved

48

## Branch-Target Buffer

- Need high instruction bandwidth
  - Branch-Target buffers
    - Next PC prediction buffer, indexed by current PC



Copyright © 2019, Elsevier Inc. All rights Reserved

49

## Branch Folding

- Optimization:
  - Larger branch-target buffer
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer
  - “Branch folding”



Copyright © 2019, Elsevier Inc. All rights Reserved

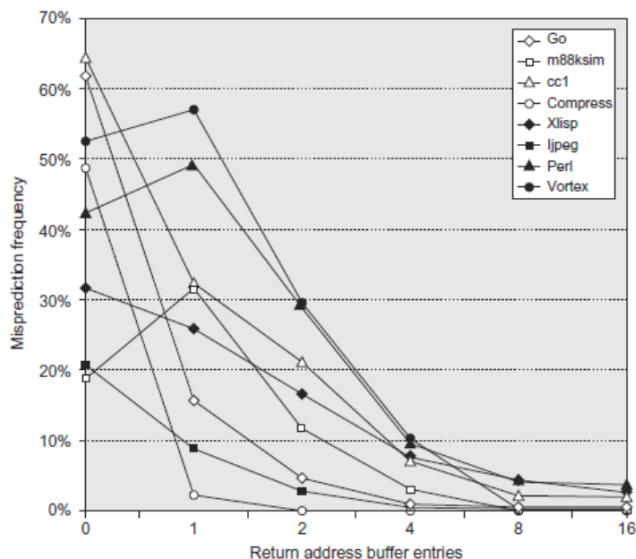
50

## Return Address Predictor

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack



## Return Address Predictor



## Integrated Instruction Fetch Unit

- Design monolithic unit that performs:
  - Branch prediction
  - Instruction prefetch
    - Fetch ahead
  - Instruction memory access and buffering
    - Deal with crossing cache lines

Adv. Techniques for Instruction Delivery and Speculation



Copyright © 2019, Elsevier Inc. All rights Reserved

53

## Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue
  - WAW and WAR hazards are avoided
  - Speculation recovery occurs by copying during commit
  - Still need a ROB-like queue to update table in order
  - Simplifies commit:
    - Record that mapping between architectural register and physical register is no longer speculative
    - Free up physical register used to hold older value
    - In other words: SWAP physical registers on commit
  - Physical register de-allocation is more difficult
    - Simple approach: deallocate virtual register when next instruction writes to its mapped architecturally-visible register

Adv. Techniques for Instruction Delivery and Speculation



Copyright © 2019, Elsevier Inc. All rights Reserved

54

## Integrated Issue and Renaming

- Combining instruction issue with register renaming:
  - Issue logic pre-reserves enough physical registers for the bundle
  - Issue logic finds dependencies within bundle, maps registers as necessary
  - Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary

| Instr. # | Instruction  | Physical register assigned or destination | Instruction with physical register numbers | Rename map changes |
|----------|--------------|-------------------------------------------|--------------------------------------------|--------------------|
| 1        | add x1,x2,x3 | p32                                       | add p32,p2,p3                              | x1->p32            |
| 2        | sub x1,x1,x2 | p33                                       | sub p33,p32,p2                             | x1->p33            |
| 3        | add x2,x1,x2 | p34                                       | add p34,p33,x2                             | x2->p34            |
| 4        | sub x1,x3,x2 | p35                                       | sub p35,p3,p34                             | x1->p35            |
| 5        | add x1,x1,x2 | p36                                       | add p36,p35,p34                            | x1->p36            |
| 6        | sub x1,x3,x1 | p37                                       | sub p37,p3,p36                             | x1->p37            |



Copyright © 2019, Elsevier Inc. All rights Reserved

55

## How Much?

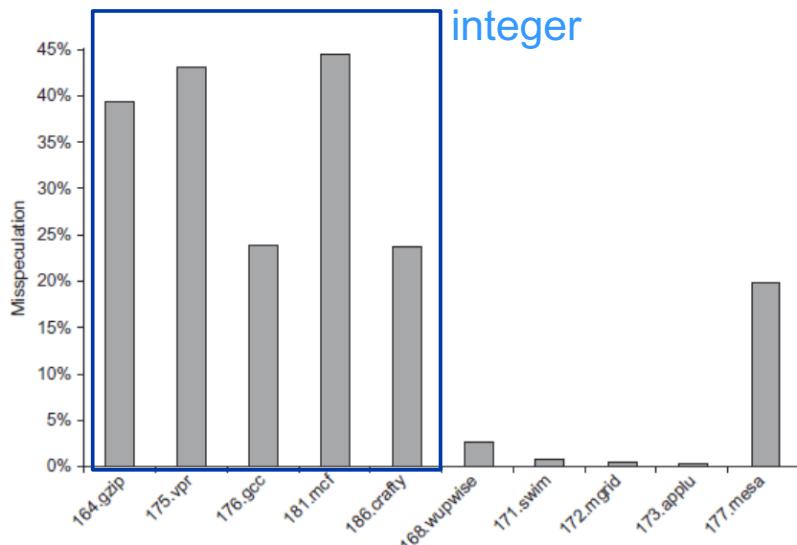
- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)
- Speculating through multiple branches
  - Complicates speculation recovery
- Speculation and energy efficiency
  - Note: speculation is only energy efficient when it significantly improves performance



Copyright © 2019, Elsevier Inc. All rights Reserved

56

## How Much?



Copyright © 2019, Elsevier Inc. All rights Reserved

57

## Energy Efficiency

- Value prediction
  - Uses:
    - Loads that load from a constant pool
    - Instruction that produces a value from a small set of values
  - Not incorporated into modern processors
  - Similar idea--*address aliasing prediction*--is used on some processors to determine if two stores or a load and a store reference the same address to allow for reordering

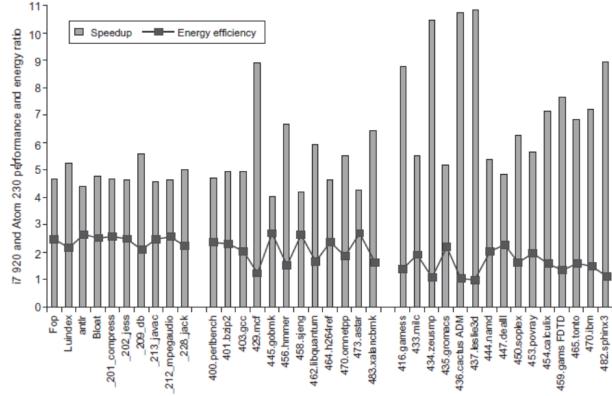


Copyright © 2019, Elsevier Inc. All rights Reserved

58

## Fallacies and Pitfalls

- It is easy to predict the performance/energy efficiency of two different versions of the same ISA if we hold the technology constant



Copyright © 2019, Elsevier Inc. All rights Reserved

59

Fallacies and Pitfalls

## Fallacies and Pitfalls

- Processors with lower CPIs / faster clock rates will also be faster

| Processor           | Implementation technology | Clock rate | Power                             | SPECInt2006 base | SPECFFP2006 baseline |
|---------------------|---------------------------|------------|-----------------------------------|------------------|----------------------|
| Intel Pentium 4 670 | 90 nm                     | 3.8 GHz    | 115 W                             | 11.5             | 12.2                 |
| Intel Itanium 2     | 90 nm                     | 1.66 GHz   | 104 W approx. 70 W one core       | 14.5             | 17.3                 |
| Intel i7 920        | 45 nm                     | 3.3 GHz    | 130 W total approx. 80 W one core | 35.5             | 38.4                 |

- Pentium 4 had higher clock, lower CPI
- Itanium had same CPI, lower clock



Copyright © 2019, Elsevier Inc. All rights Reserved

60

Fallacies and Pitfalls

## Fallacies and Pitfalls

- Sometimes bigger and dumber is better
  - Pentium 4 and Itanium were advanced designs, but could not achieve their peak instruction throughput because of relatively small caches as compared to i7
  
- And sometimes smarter is better than bigger and dumber
  - TAGE branch predictor outperforms gshare with less stored predictions



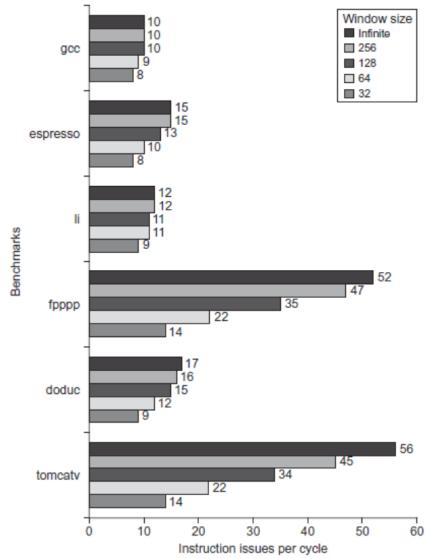
Copyright © 2019, Elsevier Inc. All rights Reserved

61

Fallacies and Pitfalls

## Fallacies and Pitfalls

- Believing that there are large amounts of ILP available, if only we had the right techniques



Copyright © 2019, Elsevier Inc. All rights Reserved

62

Fallacies and Pitfalls