Computer Systems Architecture CMSC 411 Unit 4 – Instruction-level parallelism

Alan Sussman
March 28, 2006

What we already know about pipelining

- We need to avoid structural hazards, data hazards and control hazards in order to get optimal performance from the pipeline
  - Pipeline CPI = Ideal pipeline CPI + Structural stalls + Data hazard stalls + Control stalls
- Accomplish this by techniques such as
  - instruction reordering
  - hardware modifications to detect branches earlier
  - compiler approaches to reduce branch delays
- Each technique reduces one or more of the stall components of the Pipeline CPI

What's new in Units 4 & 4b

- Some instructions can be executed independently of others. In fact, they could be executed **in parallel** if there was the hardware to do it
- Idea is to take advantage of this **instruction level parallelism** to do major rearrangements to how compilers generate code (Unit 4b) and how the MIPS (and other modern processors) pipeline executes code (Unit 4)

New techniques

- Hardware
  - dynamic pipeline scheduling
    - with scoreboarding
    - with register renaming (Tomasulo)
  - dynamic branch prediction
  - issuing multiple instructions per cycle
  - speculation
  - dynamic memory disambiguation
- Software (compiler)
  - loop unrolling
  - compiler dependence analysis
  - software pipelining and trace scheduling
  - compiler speculation

Data dependences (background)

- If 2 instructions are *parallel*, can execute simultaneously in pipeline without stalls
  - assuming no structural hazards
- If 2 instructions are *dependent*, must be executed in order, but may sometimes be partially overlapped

Data dependences (cont.)

- Instruction *j* is **data dependent** on instruction *i* if either
  - instruction *i* produces a result that may be used by instruction *j*, or
  - instruction *j* is data dependent on instruction *k*, and instruction *k* is data dependent on instruction *I* (transitivity)
- Dependence implies a chain of one or more data hazards between the 2 instructions
  - potentially causing a pipelined processor to stall
- Dependences are properties of *programs*
  - and whether one causes a stall depends on the properties of the *pipeline organization*

Computer Systems Architecture CMSC 411 Unit 4 – Instruction-level parallelism

Alan Sussman
April 6, 2006

Administrivia

- Midterm returned Tuesday
- Next project posted by tomorrow
  - talk about it on Tuesday
- Read A.8 and 3.1-3.7, 3.10

Last time

- Virtual memory protection
  - main memory shared by multiple *processes*
  - each process has its own virtual address space that no other process can touch or see
  - enforced by base/bound registers and OS/kernel that manages page tables
- Hardware/software instruction-level parallelism
  - data dependences – prevent instructions from executing in parallel
    - direct vs. indirect dependences
    - property of programs, and *may* cause a pipeline stall

Data dependences (cont.)

- A dependence
  - indicates the possibility of a hazard
  - determines the order results must be produced
  - sets an upper bound on available parallelism
- Ways to overcome dependences
  - maintain the dependence, but avoid the hazard (schedule the code – hardware or software)
  - eliminate the dependence – by transforming the code (software)

Name dependences

- When 2 instructions use the same register or memory location (harder to detect), called a *name*, but no flow of data between them
- 2 types
  - *antidependence* between instruction *i* and instruction *j* when *j* writes a register or memory location that *i* reads
  - *output dependence* occurs when *i* and *j* write the same register or memory location
- In both cases, the instructions can execute at the same time, or be reordered, if the *name* is changed so the instructions don't conflict
  - statically by compiler or dynamically by hardware (usually only for registers – why?)

Hazards

- True data dependences correspond to RAW data hazards
- Output dependences correspond to WAW hazards
- Antidependences correspond to WAR hazards

Control dependences

- To determine the ordering of an instruction, *j*, with respect to a branch instruction so that it is executed in correct program order, and only when it should be
  - e.g., the statements in *then* part of an *if* statement are control dependent on the branch

Control dependences (cont.)

- 2 constraints from control dependences
  - an 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
- But, can violate control dependences if don't affect the correctness of the program
  - by preserving *exception behavior* and *data flow*
  - implemented by control hazard detection that causes control stalls, which can be reduced by various techniques (e.g., delayed branch)

Dynamic pipeline scheduling

- A hardware technique that can do a good job of scheduling, since it has perfect information about hazards
- Basic idea is to start each instruction as early as possible
- Means have to deal with instructions that complete out-of-order
- There is a lot to keep track of, and two main schemes to do the bookkeeping
  - scoreboard method
  - Tomasulo's method

Scoreboard method

- Origins: CDC 6600, late 1960s
- Scoreboard controls the progress of each instruction to ensure that hazards such as RAW, etc. are avoided
- ID basic pipeline stage split into 2 stages
  - Issue – decode instructions, check for structural hazards
  - Read operands – wait until no data hazards, then read operands
- Distinguish when instruction *begins execution* and when *completes* execution – in between it is *in execution*
- All instructions pass through issue stage in order, but can stall or bypass each other in second stage
  - so can get WAR hazards when instructions execute out of order
  - to have multiple instructions in EX stage simultaneously, use **multiple functional units**, or pipelined units, or both – equivalent for purposes of pipeline control

Functional units

- Processor has various **functional units**, perhaps
  - 1 ALU,
  - 2 multiply units,
  - 1 floating point adder,
  - and 1 divide unit
- Idea is to try to keep all of them busy

Scoreboard (cont.)

- One segment of the scoreboard keeps track of which pipe stage each instruction is in – Fig. A.52

Instruction	Issue	Read operands	Execution complete	Write result
L.D F6,34(R2)	x	x	x	x
L.D F2, 45(R3)	x	x	x	
MUL.D F0,F2,F4	x			
SUB.D F8,F6,F2	x			
DIV.D F10,F0,F6	x			
ADD.D F6,F8,F2				

Functional units

- One segment of the scoreboard keeps track of, for each functional unit:
  - Busy (yes or no)
  - Op: operation to perform
  - Fi: name of destination register
  - Fj, Fk: names of source registers
  - Qj, Qk: functional units producing the sources
  - Rj, Rk: ready flags for Fj and Fk:
    - "yes" if the source is ready and still need it
    - "no" if the source is not ready, or if no longer need it

Scoreboard for functional units

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register status

- One segment of the scoreboard keeps track of which registers are expecting outputs from which functional units

	F0	F2	F4	F6	F8	F10	F12	…	F30
FU	Mult1	Integer			Add	Divide			

- Next, an example of the scoreboard method

The bookkeeping

Each functional unit keeps track of:

Unit	Opn	Fj	Fk	Qj	Qk

For our convenience, we'll also indicate:

Inst	Stage

And we'll assume these initial register contents and flags:

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
0	0	0	0	0	0	0	0

Scoreboard - bookkeeping

Mult1	x	x	x	x	x
x	x				

Mult2	x	x	x	x	x
x	x				

Add1	x	x	x	x	x
x	x				

Add2	x	x	x	x	x
x	x				

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
0	0	0	0	0	0	0	0

Computer Systems Architecture CMSC 411 Unit 4 – Instruction-level parallelism

Alan Sussman

April 11, 2006

Administrivia

- Midterm returned Thursday
  - answers posted
- Project posted
  - due April 27

Last time

- Data dependences
  - true, or flow, dependences – RAW hazards
  - name dependences
    - antidependences – WAR hazards
    - output dependences – WAW hazards
- Control dependences
  - statement orderings with respect to branches
  - must preserve data flow and exception behavior
- Dynamic pipeline scheduling
  - start each instruction as early as possible – once operands are available
  - instructions can complete out-of-order
  - scoreboard and Tomasulo scheduling
- Scoreboard
  - split ID stage into issue and read operands stages
  - instructions issue in order, can pass each other in read operands stage, enter functional units when ready
  - keeps track of which pipe stage each inst. is in, state of each functional unit, state of registers (which functional unit will supply data)

Time 0

S1: ADD.D F0, F2, F4

Assume 2 cycle add

Mult1	x	x	x	x	x
x	x				
Mult2	x	x	x	x	x
x	x				
Add1	Add.d	3	4	0	0
S1	Issue				
Add2	x	x	x	x	x
x	x				

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
a1	0	0	0	0	0	0	0

Time 1

S2: MULT.D F2, F6, F8

Assume 4 cycle mult

Mult1	Mult.d	6	8	0	0
S2	Issue				
Mult2	x	x	x	x	x
x	x				
Add1	Add.d	3	4	0	0
S1	Exec1				
Add2	x	x	x	x	x
x	x				

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
a1	m1	0	0	0	0	0	0

Time 2

S3: MULT.D F10, F0, F2

Mult1	Mult.d	6	8	0	0
S2	Exec1				
Mult2	Mult.d	x	x	a1	m1
S3	Issue				
Add1	Add.d	3	4	0	0
S1	Exec2				
Add2	x	x	x	x	x
x	x				

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
a1	m1	0	0	0	m2	0	0

Time 3

S4: ADD.D F0, F8, F6

Stalled - WAW hazard on F0

Mult1	Mult.d	6	8	0	0
S2	Exec2				
Mult2	Mult.d	7	x	0	m1
S3	Issue				
Add1	Add.d	3	4	0	0
S1	Write				
Add2	x	x	x	x	x
x	x				

F0	F2	F4	F6	F8	F10	F12	F14
7	3	4	6	8	10	12	14
0	m1	0	0	0	m2	0	0

Time 4

S4: ADD.D F0, F8, F6

Mult1	Mult.d	6	8	0	0
S2	Exec3				

Mult2	Mult.d	7	x	0	m1
S3	Issue				

Add1	x	x	x	x	x
x	x				

Add2	Add.d	8	6	x	x
S4	Issue				

F0	F2	F4	F6	F8	F10	F12	F14
7	3	4	6	8	10	12	14
a2	m1	0	0	0	m2	0	0

CMSC 411 - Alan Sussman 31

Time 5

S5: ADD.D F4, F6, F6

Mult1	Mult.d	6	8	0	0
S2	Exec4				

Mult2	Mult.d	7	x	0	m1
S3	Issue				

Add1	Add.d	6	6	0	0
S5	Issue				

Add2	Add.d	8	6	0	0
S4	Exec1				

F0	F2	F4	F6	F8	F10	F12	F14
7	3	4	6	8	10	12	14
a2	m1	a1	0	0	m2	0	0

CMSC 411 - Alan Sussman 32

Time 6

S6: ADD.D F2, F2, F6 stalled

Mult1	Mult.d	6	8	0	0
S2	Write				

Mult2	Mult.d	7	48	0	0
S3	Issue				

Add1	Add.d	6	6	0	0
S5	Exec1				

Add2	Add.d	8	6	0	0
S4	Exec2				

F0	F2	F4	F6	F8	F10	F12	F14
7	48	4	6	8	10	12	14
a2	0	a1	0	0	m2	0	0

CMSC 411 - Alan Sussman 33

Time 7

S6: ADD.D F2, F2, F6 stalled

Mult1	x	x	x	x	x
x	x				

Mult2	Mult.d	7	48	0	0
S3	Exec1				

Add1	Add.d	6	6	0	0
S5	Exec2				

Add2	Add.d	8	6	0	0
S4	Write				

F0	F2	F4	F6	F8	F10	F12	F14
14	48	4	6	8	10	12	14
0	0	a1	0	0	m2	0	0

CMSC 411 - Alan Sussman 34

Time 8

S6: ADD.D F2, F2, F6

Mult1	x	x	x	x	x
x	x				

Mult2	Mult.d	7	48	0	0
S3	Exec2				

Add1	Add.d	6	6	0	0
S5	Write				

Add2	Add.d	48	6	0	0
S6	Issue				

F0	F2	F4	F6	F8	F10	F12	F14
14	48	12	6	8	10	12	14
0	a2	0	0	0	m2	0	0

CMSC 411 - Alan Sussman 35

Limits to the Scoreboard method

- Number and types of functional units
  - for structural hazards
- Number of entries in the scoreboard table (how many instructions can be issued simultaneously)
  - determines how far ahead pipeline can look for independent instructions (the *window*)
  - in our discussions, never goes beyond a branch
- Output dependences and antidependences
  - that lead to WAW and WAR stalls
- Parallelism available in the program
  - to find independent instructions to execute

CMSC 411 - Alan Sussman 36

Tomasulo’s dynamic scheduling method

- Origins: IBM 360/91, 3 years after CDC 6600
- Scoreboard controls the progress of each instruction and makes sure that hazards such as RAW, etc. are avoided
- In Tomasulo's method, these functions are decentralized in **reservation stations**

Tomasulo’s method

- Main differences with scoreboard method:
  - decentralized execution control
  - decentralized hazard detection
  - **register renaming** by having operands passed to reservation stations
    - eliminates WAW and WAR hazards

Instruction status

- Only 3 kinds of pipeline stages, not 4:
  - **Issue**: send instruction to an empty reservation station, in FIFO order, along with any register contents it needs
    - also renames registers, removing WAR and WAW hazards
  - **Execute**: wait for all operands, check for RAW hazards, then execute the instruction
    - if multiple instructions ready in same cycle for same functional unit, for f.p. units choose one arbitrarily
    - for loads/stores, compute effective address first, execute loads from load buffer as soon as memory unit available, for stores wait for value to be stored before send to memory unit
  - **Write result**: put result on the **common data bus** (CDB) to get it back to a register and to any other reservation stations that need it
    - stores write to memory here too

Instruction status (cont.)

- Note differences from scoreboard:
  - no check for WAW and WAR hazards, since renaming prevents them
  - CDB broadcasts results, so don't have to wait for registers to be filled
  - Load and Store are also treated as functional units

Advantages over scoreboard

- Distribution of hazard detection logic
  - from distributed reservation stations and CDB
  - can release multiple instructions at once from single result (if other operands available)
- Eliminates stalls for WAW and WAR hazards
  - from renaming registers using reservation stations
  - and from storing operands into reservation station as soon as they are available

Bookkeeping for Tomasulo method

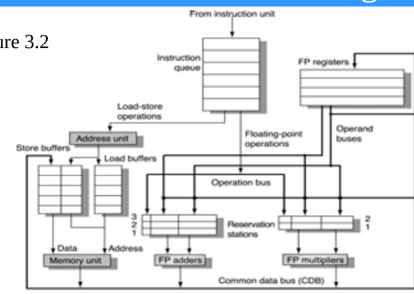
- Each reservation station keeps track of
  - Busy flag
  - Op: the operation to be performed
  - Qj, Qk: sources of operands
    - already present
    - or to come from another reservation station (functional unit)
  - Vj, Vk: values of the operands
  - A: the memory address info for a load or store (eventually the effective address)

Bookkeeping (cont.)

- Register hardware keeps track of which reservation station will fill each register
  - Qi – number of reservation station containing the operation whose result will be stored into the register
- The load and store buffers have
  - busy flag
  - value to be stored
  - A: effective address

MIPS FP with Tomasulo's algorithm

Figure 3.2



Instructions (Fig. 3.3)

Instruction	Issue	Execute	Write Result
L.D F6,34(R2)	x	x	x
L.D F2, 45(R3)	x	x	
MUL.D F0,F2,F4	x		
SUB.D F8,F2,F6	x		
DIV.D F10,F0,F6	x		
ADD.D F6,F8,F2	x		

Reservation Stations (Fig. 3.3)

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					45+R[R3]
Add1	yes	SUB		Mem[34+R[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		R[F4]	Load2		
Mult2	yes	DIV		Mem[34+R[R2]]	Mult1		

Register Status (Fig. 3.3)

Field	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Computer Systems Architecture CMSC 411 Unit 4 – Instruction-level parallelism

Alan Sussman
April 13, 2006

Administrivia

- Midterm returned today
  - Average: 71 Median: 68 25%: 62 75%: 79
  - questions?
- Project 1 and homework 4 graded by tomorrow
  - project grades by email
- Project 2 questions?
- anyone looking for a summer job?

CMSC 411 - Alan Sussman 49

Last time

- Scoreboard limitations
  - structural hazards from number and type of functional units
  - number entries in scoreboard table
  - output and anti-dependences
  - parallelism available in program
- Tomasulo dynamic scheduling
  - decentralize execution control to *reservation stations*
  - decentralize hazard detection
  - register renaming through putting operands directly in reservation stations – no WAW or WAR hazards
  - 3 stage pipeline
    - issue - send to reservation station
    - execute - in reservation station, once operands available
    - write result to CDB

CMSC 411 - Alan Sussman 50

An example of the Tomasulo method

The bookkeeping

Each functional unit keeps track of:

Unit	Opn	Vj	Vk	Qj	Qk

For our convenience, we'll also indicate:

Inst	Stage

And we'll assume these initial register contents and flags:

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
0	0	0	0	0	0	0	0

CMSC 411 - Alan Sussman 52

The bookkeeping

Mult1	x	x	x	x	x
x	x				

Mult2	x	x	x	x	x
x	x				

Add1	x	x	x	x	x
x	x				

Add2	x	x	x	x	x
x	x				

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
0	0	0	0	0	0	0	0

CMSC 411 - Alan Sussman 53

Time 0

S1: ADD.D F0, F2, F4

Assume 2 cycle add

Mult1	x	x	x	x	x
x	x				

Mult2	x	x	x	x	x
x	x				

Add1	Add.d	3	4	0	0
S1	Issue				

Add2	x	x	x	x	x
x	x				

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
a1	0	0	0	0	0	0	0

CMSC 411 - Alan Sussman 54

Time 1

S2: MULT.D F2, F6, F8

Assume 4 cycle mult.

Mult1	Mult.d	6	8	0	0
S2	Issue				
Mult2	x	x	x	x	x
x	x				
Add1	Add.d	3	4	0	0
S1	Exec1				
Add2	x	x	x	x	x
x	x				

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
a1	m1	0	0	0	0	0	0

Time 2

S3: MULT.D F10, F0, F2

Mult1	Mult.d	6	8	0	0
S2	Exec1				
Mult2	Mult.d	x	x	a1	m1
S3	Issue				
Add1	Add.d	3	4	0	0
S1	Exec2				
Add2	x	x	x	x	x
x	x				

F0	F2	F4	F6	F8	F10	F12	F14
0	3	4	6	8	10	12	14
a1	m1	0	0	0	m2	0	0

Time 3

S4: ADD.D F0, F8, F6

NOT STALLED - F0's contents already passed to all units that need it!

Mult1	Mult.d	6	8	0	0
S2	Exec2				
Mult2	Mult.d	7	x	0	m1
S3	Issue				
Add1	Add.d	3	4	0	0
S1	Write				
Add2	Add.d	8	6	0	0
S4	Issue				

To m2, but F0 not really necessary

F0	F2	F4	F6	F8	F10	F12	F14
7	3	4	6	8	10	12	14
a2	m1	0	0	0	m2	0	0

Time 4

S5: ADD.D F4, F6, F6

Mult1	Mult.d	6	8	0	0
S2	Exec3				
Mult2	Mult.d	7	x	0	m1
S3	Issue				
Add1	Add.d	6	6	0	0
S5	Issue				
Add2	Add.d	8	6	0	0
S4	Exec1				

F0	F2	F4	F6	F8	F10	F12	F14
7	3	4	6	8	10	12	14
a2	m1	a1	0	0	m2	0	0

Time 5

S6: ADD.D F2, F2, F6

stalled

Mult1	Multd	6	8	0	0
S2	Exec4				
Mult2	Mult.d	7	x	0	m1
S3	Issue				
Add1	Add.d	6	6	0	0
S5	Exec1				
Add2	Add.d	8	6	0	0
S4	Exec2				

F0	F2	F4	F6	F8	F10	F12	F14
7	3	4	6	8	10	12	14
a2	m1	a1	0	0	m2	0	0

Time 6

S6: ADD.D F2, F2, F6

stalled

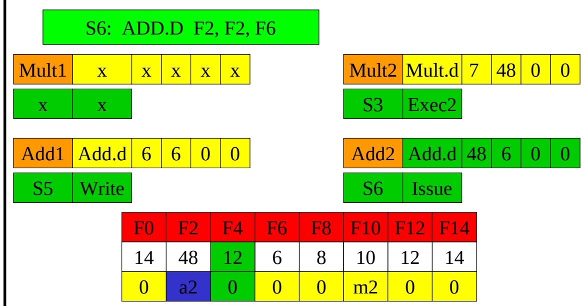
Mult1	Mult.d	6	8	0	0
S2	Write				
Mult2	Mult.d	6	48	0	0
S3	Issue				
Add1	Add.d	6	6	0	0
S5	Exec2				
Add2	Add.d	8	6	0	0
S4	Write				

stalled

F0	F2	F4	F6	F8	F10	F12	F14
7	48	4	6	8	10	12	14
a2	0	a1	0	0	m2	0	0

Time 7

Time 8



Loop-based example

- To show power of eliminating WAR and WAW hazards from dynamic register renaming

```
Loop: L.D      F0, 0(R1)
      MUL.D    F4,F0,F2
      S.D      F4,0(R1)
      DADDUI   R1,R1,-8
      BNE      R1,R2,Loop
```

- If predict branches are taken, multiple loop iterations can execute at same time – dynamic loop unrolling

Example (Fig. 3.6)

- Assume all instructions issued in two successive iterations of loop, but none of f.p. loads/stores or operations has completed
  - don't show integer ALU op, assume branch predicted as taken
- Once system reaches this state, 2 copies of loop could be sustained, with a CPI close to 1.0
  - if multiplies take less than 4 cycles to complete

Instruction status (Fig. 3.6)

Instruction	From iteration	Issue	Execute	Write Result
L.D F0,0(R1)	1	x	x	
MUL.D F4,F0,F2	1	x		
S.D F4,0(R1)	1	x		
L.D F0,0(R1)	2	x	x	
MUL.D F4,F0,F2	2	x		
S.D F4,0(R1)	2	x		

Reservation stations (Fig. 3.6)

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	yes	Load					R[R1]+0
Load2	yes	Load					R[R1]-8
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL		R[F2]	Load1		
Mult2	yes	MUL		R[F2]	Load2		
Store1	yes	Store	R[R1]			Mult1	
Store2	yes	Store	R[R1]-8			Mult2	

Register status (Fig. 3.6)

Field	F0	F2	F4	F6	F8	F10	F12	…	F30
Qi	Load2		Mult2						

More on Loads/Stores

- Can be done in different order, if access different addresses
- If load and store access same address, either
  - load is before store in program order, so after interchange get WAR hazard
  - store is before load in program order, so after interchange get RAW hazard
- Interchanging 2 stores to same address results in WAW hazard

Loads / Stores (cont.)

- To decide whether a load can be executed at a given time, check if any uncompleted store (that precedes the load in program order) shares same memory address
  - similarly, for stores, wait until no earlier *loads or stores* share same address
- To find the address conflicts
  - for a load, after effective address computed, check against **A** field of all active store buffers
    - if a match, send to load buffer after conflicting store completes
  - for a store, check for conflicts in both load and store buffers

Branch prediction

Dynamic branch prediction

- Dynamic scheduling techniques (scoreboard and Tomasulo algorithm) reduce **data hazards**
- Now: ways to reduce branch costs from **control hazards**

Branch-prediction buffers

- Simplest dynamic branch-prediction scheme
  - also called *branch history table*
- Records how frequently a branch has been taken in the past
- What is the buffer?
  - a small memory/cache area
- If there are more branches than locations in the buffer:
  - then some locations are used for more than one branch instruction, which can reduce the accuracy of predictions
- How to find the right place in the buffer:
  - from the low order bits of the address of the branch instruction

Branch-prediction buffers (cont.)

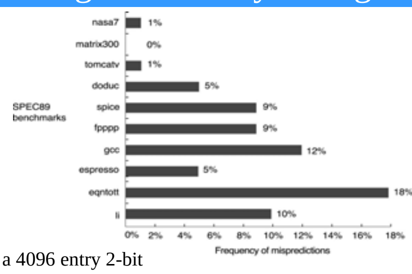
- When to access the buffer:
  - fetch the contents during the ID pipeline cycle
  - update the contents after the branch is resolved
- What is stored in the buffer:
  - perhaps one bit per branch, indicating whether the branch was last taken or not
    - predict taken if it was taken last time.
    - otherwise predict not taken

    Example: If the branch is for an inner loop, the prediction will be wrong on the 1st and last iterations, each time the loop is executed

Branch-prediction buffers (cont.)

- What is stored in the buffer (cont.):
  - perhaps a counter for each branch. If have 3 bits, for example, start the count at 4
    - add one every time the branch is taken
    - subtract one every time the branch is not taken
    - predict taken if the count is 4 or more
    - predict not taken if the count is 3 or less.
  - perhaps a **correlating predictor** that counts for this branch and for 1 or more earlier ones

How good are they? – Fig. 3.8



For a 4096 entry 2-bit prediction buffer

Computer Systems Architecture CMSC 411 Unit 4 – Instruction-level parallelism

Alan Sussman

April 18, 2006

Administrivia

- Midterm questions?
- Project 1 grades emailed (with minor error)
- Project 2 questions?
- anyone looking for a summer job?

Last time

- Tomasulo dynamic scheduling
  - detailed example showed benefit of forwarding results directly to reservation stations, no WAR/WAW hazards
  - still get structural hazards (not enough functional units, CDB)
  - loop example showed instructions from multiple loop iterations in execution at same time, to get CPI close to 1.0
  - need care with loads/stores to same address to avoid WAR, WAW hazards
    - check contents of load/store buffers (res. stations)
- Branch prediction
  - to reduce control hazards
  - branch prediction buffer – keep track of recent branch history, to predict which way it will go in future
    - fetch during ID stage, to get next instruction
    - update after branch resolved

Correlating branch predictors

- To improve prediction accuracy, look at behavior of recent *other* branches than the one trying to predict
- Take advantage of *correlation* between behavior of different branches
  - also called *two-level* predictors

Simple example:

```
if (d == 0)
   d = 1;
if (d == 1)
   …
```

MIPS code, with d in R1:

```
    BNEZ  R1,L1   ; branch b1
    DADDIU R1,R0,#1
L1: DADDIU R3,R1,#-1
    BNEZ R3,L2   ; branch b2
    …
L2:
```


Example (cont.) – Fig. 3.10

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	yes	not taken	1	yes	not taken
1	no	taken	1	yes	not taken
2	no	taken	2	no	taken

b1 not taken → b2 not taken

Example – 1 bit predictor

Predictor initialized to not taken – Fig. 3.11

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

All branches mispredicted!

1-bit prediction with 1 bit correlation

Initialized to not taken/not taken – Fig. 3.13

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	**NT**/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

Only misprediction is on first iteration/row

Correlating predictors (cont.)

- Predictor on last slide is a (1,1) predictor
  - uses behavior of last branch to choose from among a pair of 1-bit branch predictors
- In general, an (*m*,*n*) predictor uses behavior of last *m* branches to choose from 2^m branch predictors
  - each is an *n*-bit predictor for a single branch
  - simple hardware to do this – can keep global history of most recent *m* branches in an *m*-bit shift register, and each bit records whether branch taken/not taken
    - and index branch prediction buffer using low-order bits of branch address with *m*-bit global history

Comparing predictors

- Correlating vs. standard 2-bit scheme
  - using same number of state bits
- Number of state bits for (*m*,*n*) predictor is:
  - $2^m \times n \times$ # prediction entries to select from with the branch address
- 2-bit predictor with no global history is a (0,2) predictor
  - (0,2) predictor from Fig. 3.8 had 4K entries selected by branch address
    - total number of bits is $2^0 \times 2 \times 4K = 8K$ bits
  - (2,2) predictor from Fig. 3.14 has 64 entries, with 4 entries per branch address
    - total number of bits is $2^2 \times 2 \times 16 = 128$ bits

Comparing 2-bit predictors

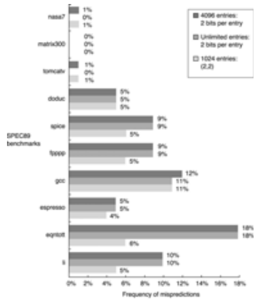
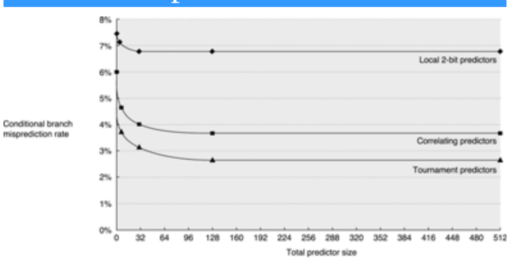


Fig. 3.15

Combining local and global predictors

- *Tournament* predictors use multiple predictors, usually one global and one local, combined with a selector
  - a form of *multilevel* branch prediction
  - selector is often just a 2-bit saturating counter per branch to choose from 2 predictors that is used with a finite state machine (see Fig. 3.16)
    - increment counter when “predicted” predictor is correct and other predictor is incorrect, decrement in opposite situation

Misprediction rates



on SPEC89 benchmarks

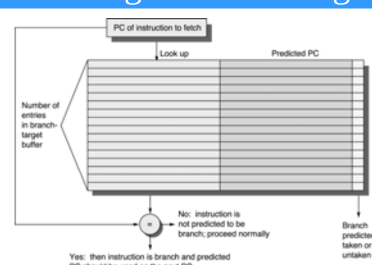
Branch target buffers

- Another branch penalty reduction method
- BTB stores:
  - branch prediction information
  - predicted location of the instruction following the branch
- What is the buffer:
  - a small memory area, accessed with the address of the PC of the instruction fetched
- If there are more branches than locations in the buffer:
  - then some locations are used for more than one branch instruction, limiting the amount of information stored

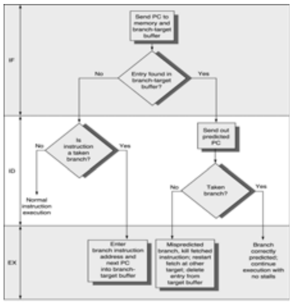
Branch target buffers (cont.)

- When to access the buffer:
  - fetch the contents during the IF pipeline cycle
  - update the contents after the branch is resolved
- How to find the right entry in the buffer:
  - look for an entry that matches the PC
- What is stored in the buffer:
  - addresses of branch instructions
  - predicted target of each branch
  - branch prediction information
- Allows *branch folding* for unconditional branches (jumps)
  - zero cycle instruction!
  - by having BTB replace jump instruction in pipeline with branch target instruction returned from BTB

Branch target buffer – Fig. 3.19



BTB algorithm – Fig. 3.20



BTB penalties – Fig. 3.21

Instruction in buffer	Prediction	Actual branch	Penalty cycles
yes	taken	taken	0
yes	taken	not taken	2
no		taken	2
no		not taken	0

Multiple Issue

Issuing multiple instructions at one time

- Scoreboard and Tomasulo’s method reduce stalls from **data hazards**
- Branch buffers reduce stalls from **control hazards**
- Now talk about getting more parallelism by issuing several instructions at once
- Two currently used methods:
  - superscalar processors
  - VLIW (very long instruction word) processors

5 approaches – Fig. 3.23

Common name	Issue structure	Hazard detection	Scheduling	Interesting feature	Examples
Superscalar (static)	dynamic	hardware	static	in-order execution	Sun UltraSparc II/III
Superscalar (dynamic)	dynamic	hardware	dynamic	some o-o-o execution	IBM Power2
Superscalar (speculative)	dynamic	hardware	dynamic, with spec.	o-o-o, with speculation	PIII/4, R10K, Alpha 21264
VLIW/LIW	static	software	static	no hazards bet. packets	Trimedia, Intel i860
EPIC	static	mostly software	mostly static	explicit dep. by compiler	Itanium

Superscalar MIPS

- Idea: issue several non-interfering instructions at each clock cycle
- The *hardware* has the burden of detecting interfering instructions
- Can be *statically* scheduled (using compiler techniques), or *dynamically* scheduled (using scoreboard or Tomasulo’s algorithm)
- For example, look at a superscalar MIPS processor that can issue two instructions at once:
  - one floating point instruction
  - one non-floating point (integer or branch) instruction

Superscalar MIPS (cont.)

- 3 steps in instruction fetch and issue:
  - fetch 2 instructions from cache
  - determine whether 0, 1, or 2 can issue
    - look at hazards from earlier instructions issued, and from within the 2 fetched instructions
    - since 1 integer and 1 FP instruction can issue, mostly just look at opcodes for hazards within fetched pair, unless integer instruction is FP load/store/move – possible RAW hazard
    - also need another read/write port on FP register file, to allow loads/stores to issue with FP operations, and (lots) more bypass paths
  - issue the instructions to the correct functional unit

Computer Systems Architecture CMSC 411 Unit 4 – Instruction-level parallelism

Alan Sussman
April 20, 2006

Administrivia

- Midterm questions?
- Project 2 questions?
  - due next Thursday, April 27
- Guest lecturer Tuesday, (short) class next Thursday to finish up Unit 4b

Last time

- Branch prediction
  - for a given branch instruction, local predictor uses history of that branch inst.
  - correlating predictor uses history of *all* recently executed branches
  - two-level predictor combines both – also called a tournament predictor
  - branch target buffer also stores and uses address of next predicted instruction, to reduce branch penalty further
- Multiple instruction issue
  - to issue and complete more than one instruction per cycle
  - superscalar and VLIW are dynamic and static techniques, respectively
  - superscalar MIPS from example can issue 0, 1, or 2 instructions per cycle, depending on structural and data hazards

Example

x is an array of 1000 double precision numbers, indexed from 1 to 1000

for i=1000, 999, ..., 1
  x[i] = x[i] + s;

Unroll the loop – execute 4 iterations before branch (for now don't worry about loop test)

```
Loop:
  L.D    F0, 0(R1)     x[i] = x[i] + s
  ADD.D  F4, F0,F2     uses F0 and F4
  S.D    F4,0(R1)
  L.D    F6,-8(R1)     x[i-1] = x[i-1] + s
  ADD.D  F8,F6,F2      uses F6 and F8
  S.D    F8,-8(R1)
  L.D    F10,-16(R1)   x[i-2] = x[i-2] + s
  ADD.D  F12,F10,F2    uses F10 and F12
  S.D    F12,-16(R1)
  L.D    F14,-24(R1)   x[i-3] = x[i-3] + s
  ADD.D  F16, F14,F2   uses F10 and F12
  S.D    F16, -24(R1)
  DSUBI  R1,R1,#32     point to next element
  BNEZ   R1, Loop
```


Example (cont.)

On single issue MIPS pipeline, takes 14 cycles

Cycle	FP instruction	non-FP instruction
1		L.D F0, 0(R1)
2		L.D F6, -8(R1)
3	ADD.D F4, F0, F2	L.D F10, -16(R1)
4	ADD.D F8, F6, F2	L.D F14, -24(R1)
5	ADD.D F12, F10, F2	DSUBI R1, R1, #32 (changes later offsets)
6	ADD.D F16, F14,F2	S.D F4, 32(R1)
7		S.D F8, 24(R1)
8		S.D F12, 16(R1)
9		BNEZ R1, Loop (delayed branch)
10		S.D F16, 8(R1)

Example (cont.)

- Need to use care not to exceed register traffic bounds
- It may be impossible to load an F register at the same time that the floating point unit is accessing/reading another F register
  - a structural hazard

CMSC 411 - Alan Sussman 103

VLIW MIPS

- Idea: have several non-interfering instructions stored in each instruction fetched
- Then the software (the compiler) has the burden of detecting interfering instructions
- Must be *statically* scheduled by compiler

CMSC 411 - Alan Sussman 104

Example

- Assume a VLIW instruction can contain 2 memory references, 2 floating point operations, and 1 other operation each clock cycle
- Then the example loop takes 6 cycles, because latencies still limit performance

CMSC 411 - Alan Sussman 105

Example (cont.)

Cycle	Memory	Memory	FP	FP	Other
1	L.D F0	L.D F6			
2	L.D F10	L.D F14			
3			ADD.D F4	ADD.D F8	
4			ADD.D F12	ADD.D F16	DSUBI
6	S.D F4	S.D F8			BNEZ
6	S.D F12	S.D F16			

CMSC 411 - Alan Sussman 106

Limitations on multiple issue

- Programs have limited parallelism and cannot keep all functional units busy – still have control hazards
- CPU hardware gets very complicated for issuing and executing
- Memory bandwidth requirements increase
- Superscalar:
  - many decisions to be made very fast
- VLIW:
  - code length increases, because of loop unrolling and many NOP instructions/operations
- Vector processors (Appendix G online) generally cheaper and faster.
- ... but recently, VLIW becoming more popular again (Intel/HP EPIC)

CMSC 411 - Alan Sussman 107

Hardware speculation

- A method for overcoming control dependences/hazards
- Branch prediction reduces stalls, but still want to execute instructions past a branch to get more instruction level parallelism, esp. with multiple issue
- *Speculate* on the outcome of branches, and *execute* the program as if the guesses are correct
  - more than just dynamic scheduling plus branch prediction
  - instructions are fetched, issued, and *executed*
  - then fix things up if the speculation was incorrect

CMSC 411 - Alan Sussman 108

Hardware speculation (cont.)

- 3 key ideas
  - *dynamic branch prediction*, to choose which instructions to execute
  - *speculation*, to allow executing instructions before branches resolved (and effects undone if speculation was wrong)
  - *dynamic scheduling*, to deal with scheduling combinations of basic blocks
- Essentially a *data flow execution*
  - operations execute as soon as operands available
- We'll use Tomasulo's algorithm for dynamic scheduling, and just for FP unit

Extensions to Tomasulo's algorithm

- Separate bypassing of results between instructions from actual completion of an instruction
  - so that an instruction can execute and produce results for other instructions, without performing any updates that can't be undone – until know that the instruction should have been executed (control dependences resolved)
  - when instruction is no longer speculative:
    - update the register file, or memory
    - call this *instruction commit*

Implementing speculation

- Allow instructions to execute out-of-order, but make them commit *in order*
  - and prevent any action that can't be undone (update state, take an exception, etc.) until instruction commits
- Requires some changes to standard pipeline sequence, and hardware buffers to hold results of instructions that have completed execution, but have not committed yet
  - a **reorder buffer**
  - also used to pass results between speculated instructions

Reorder buffer (ROB)

- Provides registers, like reservation stations
- Holds instruction result from time operation completes until instruction commits
- But no writeback to register file until instruction commits
  - so ROB supplies operands in this time interval
- Similar to store buffer in Tomasulo's algorithm
  - in design shown, store buffer integrated into ROB

Reorder buffer (cont.)

- 4 fields in an ROB entry
  - **instruction type** – branch, store, register op (ALU or load)
  - **destination** – register number or memory address
  - **value** – result of instruction
  - **ready** – set when instruction completes execution, and value is ready

MIPS with Tomasulo + speculation

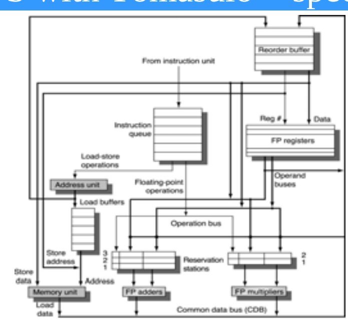


Fig. 3.29

Instruction execution – 4 steps

- **Issue**
  - get instruction from instruction queue
  - issue if both reservation station and empty ROB slot available
  - send operands to reservation station if available in either registers or ROB
  - send ROB entry number to reservation station for writing result
  - if reservations stations or ROB full, instruction stalls

Instruction execution (cont.)

- **Execute**
  - If an operand is not yet available, monitor CDB until it appears
  - When all operands available at reservation station, execute the operation
  - Instructions can take multiple cycles in this stage, loads require 2 steps (compute effective address and memory access), stores only need to compute effective address

Instruction execution (cont.)

- **Write result**
  - When result available, write on CDB (with ROB tag set in **issue** step), and from CDB to ROB, plus any reservation stations waiting
  - Mark reservation station available
  - For stores, if value to be stored is available, write into Value field of ROB entry for store
    - otherwise, wait until it appears on CDB, then update Value field of ROB entry

Instruction execution (cont.)

- **Commit**
  - For "normal" case (instruction reaches head of ROB and its result is in the buffer), update the register with the result and remove instruction from ROB
  - For a store, update memory instead of result register
  - For an incorrectly predicted branch, the speculation was wrong – flush the ROB and reservation stations, and restart execution at correct branch target
    - correctly predicted branch is just finished

Example – Fig. 3.30

```
L.D     F6,34(R2)
L.D     F2,45(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F6,F2
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

- Assume 2 cycle add, 10 cycle multiply, 40 cycle divide
- Status tables will show when MUL.D is ready to commit

Example (cont.)

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	no							
Load2	no							
Add1	no							
Add2	no							
Add3	no							
Mult1	no	MUL.D	Mem[45+R[R3]]	R[F4]			#3	
Mult2	yes	DIV.D		Mem[34+R[R2]]	#3		#5	

Example (cont.)

Reorder buffer

Entry	Busy	Instruction	State	Dest.	Value
1	no	L.D	Commit	F6	Mem[34+R[R2]]
2	no	L.D	Commit	F2	Mem[45+R[R3]]
3	yes	MUL.D	Write result	F0	#2 × R[F4]
4	yes	SUB.D	Write result	F8	#1 - #2
5	yes	DIV.D	Execute	F10	
6	yes	ADD.D	Write result	F6	#4 + #2

Example (cont.)

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
ROB #	3						6		4	5
Busy	yes	no	no	no	no	no	yes	no	yes	yes

Speculation - Exceptions

- Processor with ROB can provide precise exceptions/interrupts
  - don't take exception caused by program until commit
  - at that point, flush any pending instructions
  - works because commit happens in program order
- Much harder to do precise exceptions with basic Tomasulo algorithm
  - instructions can complete before earlier issued instructions, with register or memory writes that can't be undone – *imprecise* exceptions
  - makes some kinds of exceptions (e.g., page faults) hard to handle

Loop example – Fig. 3.31

```
Loop:
   L.D F0,0(R1)
   MUL.D F4,F0,F2
   S.D  F4,0(R1)
   DADDIU R1,R1,#-8
   BNE  R1,R2,Loop
```

- Assume all instructions in loop issued twice
- L.D and MUL.D from iteration 1 committed, all other instructions finished executing
- Effective address for stores already computed, and have left ROB

Loop example (cont.)

Reorder buffer

Entry	Busy	Inst.	State	Dest.	Value
1	no	L.D	Commit	F0	Mem[0+R[R1]]
2	no	MUL.D	Commit	F4	#1 × R[F2]
3	yes	S.D	Write result	0+R[R1]	#2
4	yes	DADDUI	Write result	R1	R[R1] – 8
5	yes	BNE	Write result		
6	yes	L.D	Write result	F0	Mem[#4]
7	yes	MUL.D	Write result	F4	#6 × R[F2]
8	yes	S.D	Write result	0+#4	#7
9	yes	DADDUI	Write result	R1	#4 – 8
10	yes	BNE	Write result		

Loop example (cont.)

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
ROB #	6				7				
Busy	yes	no	no	no	yes	no	no	no	no

- If branch mispredicted, all instructions prior to branch commit, and when branch commits, ROB cleared and processor starts fetching from other path
  - this is expensive, but shouldn't happen very often

Speculation with multiple issue

- Similar to extending Tomasulo algorithm for multiple issue
  - process multiple instructions per clock, assign reservation stations and reorder buffer entries
- Major challenges are instruction issue and monitoring CDBs for instruction completion
  - and must be able to handle multiple instruction commits per cycle
- As example in book shows (Figs. 3.33 and 3.34), speculation helps when there are data dependent branches that limit performance
  - requires good branch prediction
  - incorrect speculation could hurt performance

Design considerations

- Register *renaming*
  - an alternative to ROB
  - employ a larger set of physical registers than the ones visible in the architecture (R and F)
  - replace the ones in the reservations stations and ROB
  - renaming happens at instruction issue
  - renaming dest register removes WAW and WAR hazards
  - speculation recovery easy since physical register holding instruction dest doesn't become architectural register until commit
  - hard part is knowing when deallocation is OK
    - when no longer corresponds to a physical register, and nothing will use it

Design considerations (cont.)

- How much to speculate
  - cost comes from incorrect speculation of an expensive event (e.g., a 2nd level cache miss)
  - solution is to only allow low-cost exceptional events in speculative mode (e.g., 1st level cache miss)
  - processor waits until instruction is not speculative before handling expensive exceptional event

Design considerations (cont.)

- Speculating through multiple branches
  - when is this beneficial?
    - very high branch frequencies
    - clustering of branches
    - long delays in functional units
  - really just makes recovery more complicated
  - IBM Power2 can even predict and speculate on more than 1 branch per cycle!

Example: P6 Microarchitecture

P6 microarchitecture

- Dynamically scheduled
- Translates each IA-32 instruction into series of micro-operations (uops) to be executed in pipeline
  - each uop is like a MIPS instruction
- Up to 3 IA-32 instructions fetched, decoded, and translated per cycle, max of 6 uops generated per cycle
- Pipeline is out-of-order and speculative, with register renaming and a reorder buffer (ROB)

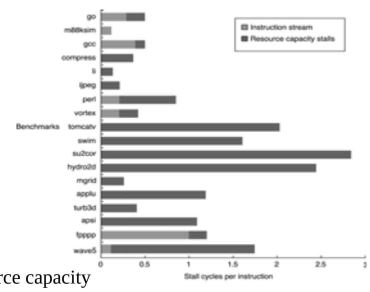
P6 uops – Fig. 3.48

Instruction	Pipeline stages	Repeat rate
Integer ALU	1	1
Integer load	3	1
Integer multiply	4	1
FP add	3	1
FP multiply	5	2
FP divide (64-bit)	32	32

Pipeline stalls

- Lead to processor not committing as many instructions as possible in a cycle
  - fewer than 3 insts fetched – inst cache miss
  - fewer than 3 insts issued – too many uops
  - not all uops generated can issue – lack of reservation stations or ROB entries
  - data dependences everywhere – reservation stations and ROBs
  - data cache miss – all res. stations and ROBs waiting
  - branch mispredicts – pipeline flushes and refills

Decode stalls – Fig. 3.51



resource capacity is the big problem

Computer Systems Architecture CMSC 411 Unit 4 – Instruction-level parallelism

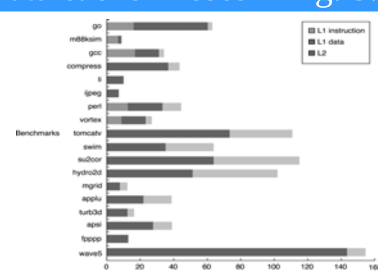
Alan Sussman

April 25, 2006

Last time

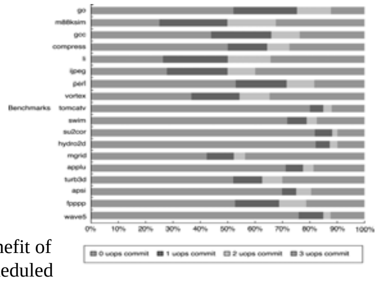
- Multiple issue
  - with superscalar, loop unrolling example showed can execute loop body in fewer cycles, on average
  - VLIW – compiler packs operations for multiple functional units into one instruction – static scheduling
  - limits include parallelism in program, complexity of CPU hardware, memory bandwidth requirements
- Hardware speculation
  - fetch, issue, execute instructions through branches
  - ROB helps fix things up if branch prediction and speculation was wrong – don't *commit* those instructions – no register or memory updates until commit
  - and instructions commit in order, allowing precise exceptions
  - don't allow expensive events to speculative – 2nd level cache miss
- P6 microarchitecture
  - IA-32 instructions translated to uops that are executed in out-of-order, speculative pipeline

Data cache misses – Fig. 3.53



hard to read, but L2 misses are the problem

uop completions per cycle
Fig. 3.56



shows benefit of a dyn. scheduled pipeline

Fallacies

- Processors with lower CPIs will always be faster
- Processors with faster clock rates will always be faster
- Sophisticated pipelines with slower clocks sometimes win from using more ILP

Pitfalls

- Emphasizing improved CPI by increasing issue rate, but sacrificing clock rate, can lead to lower performance
  - complexity introduced to improve CPI can slow clock rate
- Improving only one part of a multiple-issue processor may not improve overall performance
  - need to find the bottleneck (i.e. Amdahl’s law)
- Sometimes bigger and dumber is better
  - different applications can produce different behaviors