Recovery

- Completely remove wrong path instructions
  - Flush from IQ
  - Remove from ROB
  - Restore map table to before misprediction
  - Free destination registers

68

68

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add p7 ← p6, p4	[p4]
sub r3 ← r5, r2	sub p8 ← p5, p2	[p6]
addi r1 ← r3, 1	addi p9 ← p8, 1	[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

Free-list: p10

69

69

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add p7 ← p6, p4	[p4]
sub r3 ← r5, r2	sub p8 ← p5, p2	[p6]
addi r1 ← r3, 1	addi **p9** ← p8, 1	**[p1]**

r1	**p1**
r2	p2
r3	p8
r4	p7
r5	p5

Map table

Free-list: **p9**, p10

70

70

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add p7 ← p6, p4	[p4]
sub r3 ← r5, r2	sub **p8** ← p5, p2	**[p6]**

r1	p1
r2	p2
r3	**p6**
r4	p7
r5	p5

Map table

Free-list: **p8**, p9, p10

71

71

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor p6 ← p1, p2	[p3]
add r4 ← r3, r4	add **p7** ← p6, p4	**[p4]**

r1	p1
r2	p2
r3	p6
r4	**p4**
r5	p5

Map table

Free-list: **p7**, p8, p9, p10

72

72

Recovery example

Original insns	Renamed insns	Overwritten Reg
bne r1,r0 loop	bne p1,p0 loop	[ ]
xor r3 ← r1, r2	xor **p6** ← p1, p2	**[p3]**

r1	p1
r2	p2
r3	**p3**
r4	p4
r5	p5

Map table

Free-list: **p6**, p7, p8, p9, p10

73

73

Recovery example

Original insns	Renamed insns	Overwritten Reg
`bne r1,r0 loop`	`bne p1,p0 loop`	[ ]

r1	p1
r2	p2
r3	p3
r4	p4
r5	p5

Map table

p6
p7
p8
p9
p10

Free-list

74

74

What about stores

- Stores: Write D$, not registers
  - Can we rename memory?
  - Recover in the cache?

No (at least not easily)

- Cache writes unrecoverable
- Stores: only when certain
  - Commit

75

75

Commit

Original insns	Renamed insns	Overwritten Reg
`xor r3 ← r1, r2`	`xor p6 ← p1, p2`	[p3]
`add r4 ← r3, r4`	`add p7 ← p6, p4`	[p4]
`sub r3 ← r5, r2`	`sub p8 ← p5, p2`	[p6]
`addi r1 ← r3, 1`	`addi p9 ← p8, 1`	[p1]

- At commit: instruction becomes architected state
- In-order
- Only when instructions are finished
- Free overwritten register (why?)

76

76

Freeing over-written register

Original insns	Renamed insns	Overwritten Reg
`xor r3 ← r1, r2`	`xor p6 ← p1, p2`	**[p3]**
`add r4 ← r3, r4`	`add p7 ← p6, p4`	[p4]
`sub r3 ← r5, r2`	`sub p8 ← p5, p2`	[p6]
`addi r1 ← r3, 1`	`addi p9 ← p8, 1`	[p1]

- Before xor: **r3→ p3**
- After xor: **r3→ p6**
  - Insns older than xor reads p3
  - Insns younger than xor read p6 (until next r3-writing instruction)
- At commit of xor, no older instructions exist
  - No one else needs p3 → free it!

77

77

Commit Example

Original insns	Renamed insns	Overwritten Reg
`xor r3 ← r1, r2`	`xor p6 ← p1, p2`	[p3]
`add r4 ← r3, r4`	`add p7 ← p6, p4`	[p4]
`sub r3 ← r5, r2`	`sub p8 ← p5, p2`	[p6]
`addi r1 ← r3, 1`	`addi p9 ← p8, 1`	[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10

Free-list

78

78

Commit Example

Original insns	Renamed insns	Overwritten Reg
`xor r3 ← r1, r2`	`xor p6 ← p1, p2`	**[p3]**
`add r4 ← r3, r4`	`add p7 ← p6, p4`	[p4]
`sub r3 ← r5, r2`	`sub p8 ← p5, p2`	[p6]
`addi r1 ← r3, 1`	`addi p9 ← p8, 1`	[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3

Free-list

79

79

Commit Example

Original insns	Renamed insns	Overwritten Reg
add r4 ← r3, r4	add p7 ← p6, p4	[p4]
sub r3 ← r5, r2	sub p8 ← p5, p2	[p6]
addi r1 ← r3, 1	addi p9 ← p8, 1	[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3
p4

Free-list

80

80

Commit Example

Original insns	Renamed insns	Overwritten Reg
sub r3 ← r5, r2	sub p8 ← p5, p2	[p6]
addi r1 ← r3, 1	addi p9 ← p8, 1	[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3
p4
p6

Free-list

81

81

Commit Example

Original insns	Renamed insns	Overwritten Reg
addi r1 ← r3, 1	addi p9 ← p8, 1	[p1]

r1	p9
r2	p2
r3	p8
r4	p7
r5	p5

Map table

p10
p3
p4
p6
p1

Free-list

82

82

Out of order pipeline diagrams

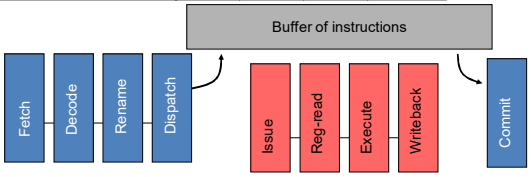
- Standard style: large and cumbersome
- Change layout slightly
  - Columns = stages (dispatch, issue, *etc.*)
  - Rows = instructions
  - Content of boxes = cycles
- For our purposes: issue/exec = 1 cycle
  - Ignore Preg read latency, *etc.*
  - Load-use, mul, div, and FP longer

83

83

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
lw p2 ← [p1]				
add p4 ← p2, p3				
xor p6 ← p4, p5				
lw p8 ← [p7]				



84

84

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
lw p2 ← [p1]				
add p4 ← p2, p3				
xor p6 ← p4, p5				
lw p8 ← [p7]				

2-wide
Infinite ROB, IQ, Pregs
Loads: 3 cycles

85

85

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`lw p2 ← [p1]`	1			
`add p4 ← p2, p3`	1			
`xor p6 ← p4, p5`				
`lw p8 ← [p7]`				

Cycle 1:
- Dispatch 1st lw and add

86

86

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`lw p2 ← [p1]`	1	2	5	
`add p4 ← p2, p3`	1			
`xor p6 ← p4, p5`	2			
`lw p8 ← [p7]`	2			

Cycle 2:
- Dispatch xor and 2nd lw
- 1st Lw issues -- also note WB cycle while you do this (Note: don't issue if WB ports full)

87

87

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`lw p2 ← [p1]`	1	2	5	
`add p4 ← p2, p3`	1			
`xor p6 ← p4, p5`	2			
`lw p8 ← [p7]`	2	3	6	

Cycle 3:
- add and xor are not ready
- 2nd load is → issue it

88

88

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`lw p2 ← [p1]`	1	2	5	
`add p4 ← p2, p3`	1	5	6	
`xor p6 ← p4, p5`	2			
`lw p8 ← [p7]`	2	3	6	

Cycle 4:
- nothing

Cycle 5:
- add can issue

89

89

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`lw p2 ← [p1]`	1	2	5	6
`add p4 ← p2, p3`	1	5	6	
`xor p6 ← p4, p5`	2	6	7	
`lw p8 ← [p7]`	2	3	6	

Cycle 6:
- 1st load can commit (oldest instruction & finished)
- xor can issue

90

90

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`lw p2 ← [p1]`	1	2	5	6
`add p4 ← p2, p3`	1	5	6	7
`xor p6 ← p4, p5`	2	6	7	
`lw p8 ← [p7]`	2	3	6	

Cycle 7:
- add can commit (oldest instruction & finished)

91

91

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`lw p2 ← [p1]`	1	2	5	6
`add p4 ← p2, p3`	1	5	6	7
`xor p6 ← p4, p5`	2	6	7	8
`lw p8 ← [p7]`	2	3	6	8

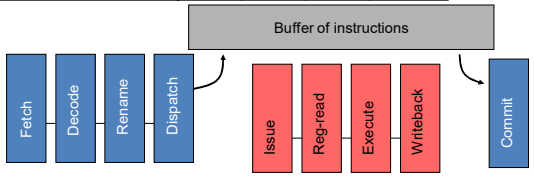
Cycle 8:
- xor and lw can commit (2-wide: can do both at once)

92

92

Out of order pipeline diagrams

Instruction	Disp	Issue	WB	Commit
`lw p2 ← [p1]`	1	2	5	6
`add p4 ← p2, p3`	1	5	6	7
`xor p6 ← p4, p5`	2	6	7	8
`lw p8 ← [p7]`	2	3	6	8



93

93

HANDLING MEMORY OPS

94

94

Dynamically Scheduling Memory Ops

- Compilers must schedule memory ops conservatively
- Options for hardware:
  - Hold loads until all prior stores execute (conservative)
  - Execute loads as soon as possible, detect violations (aggressive)
    - When a store executes, it checks if any later loads executed too early (to same address). If so, flush pipeline
  - Learn violations over time, selectively reorder (predictive)

Before

```
lw r2,4(sp)
lw r3,8(sp)
add r1←r3,r2 //stall
sw 0(sp),r1
lw r5,0(r8)
lw r6,4(r8)
sub r4←r5,r6 //stall
sw 8(r8),r4
```

Wrong(?)

```
lw r2,4(sp)
lw r3,8(sp)
lw r5,0(r8) //does r8==sp?
add r1←r3,r2
lw r6,4(r8) //does r8+4==sp?
sw 0(sp),r1
sub r4←r5,r6
sw 8(r8),r4
```

95

95

Loads and Stores

Instruction	Disp	Issue	WB	Commit
`fdiv p3 ← p1,p2`	1	2	25	
`st 0(p5) ← p4`	1	2	3	
`st 0(p6) ← p3`	2			
`ld p8 ← 0(p7)`	2			

Cycle 3:
- Can ld p8←[p7] execute? (why or why not?)

96

96

Loads and Stores

Instruction	Disp	Issue	WB	Commit
`fdiv p3 ← p1,p2`	1	2	25	
`st 0(p5) ← p4`	1	2	3	
`st 0(p6) ← p3`	2			
`ld p8 ← 0(p7)`	2			

Aliasing (again)
- p5 == p7 ?
- p6 == p7 ?

97

97

Loads and Stores

Instruction	Disp	Issue	WB	Commit
`fdiv p3 ← p1,p2`	1	2	25	
`st 0(p5) ← p4`	1	2	3	
`st 0(p6) ← p3`	2			
`ld p8 ← 0(p7)`	2			

Suppose p5 == p7 and p6 != p7
- Can ld p8←[p7] execute? (why or why not?)

98

98

Memory Forwarding

- Stores write cache at commit
  - Commit is in-order, delayed by all instructions
  - Allows stores to be "undone" on branch mis-predictions, etc.
- Loads read cache
  - Early execution of loads is critical
- Forwarding
  - Allow store → load communication before store commit
  - Conceptually like reg. bypassing, but different implementation
    - Why? Addresses unknown until execute

99

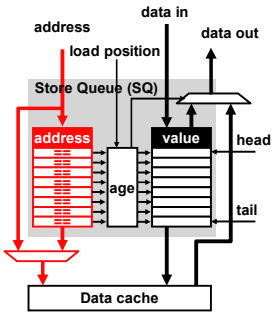
99

Forwarding: Store Queue

Store Queue
- Holds all in-flight stores
- CAM: searchable by address
- Age logic: determine youngest matching store older than load

Store execution
- Write Store Queue
  - Address + Data

Load execution
- Search SQ
  - Match? Forward
  - Else Read D$



100

100

Load scheduling

- Store→Load Forwarding:
  - Get value from executed (but not committed) store to load
- Load Scheduling:
  - Determine when load can execute with regard to older stores
- Conservative load scheduling:
  - All older stores have executed
  - Some architectures: split store address / store data
    - Only require known address
  - Advantage: always safe
  - Disadvantage: performance (limits out-of-orderness)

101

101

An example code

```
ld  r5 ← 0[r1]
ld  r6 ← 0[r2]
add r7 ← r5,r6
st  0[r3] ← r7
ld  r5 ← 4[r1]
ld  r6 ← 4[r2]
add r7 ← r5,r6
st  4[r3] ← r7
// loop control here
```

With conservative load scheduling, what can go out of order?

102

102

An example code

		Disp	Issue	WB	Commit
1	`ld p4 ← 0[p1]`	1			
2	`ld p5 ← 0[p2]`	1			
3	`add p6 ← p4,p5`				
4	`st 0[p3] ← p6`				
5	`ld p7 ← 4[p1]`				
6	`ld p8 ← 4[p2]`				
7	`add p9 ← p7,p8`				
8	`st 4[p3] ← p9`				

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 1:
Dispatch insns #1, #2

103

103

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	
2	ld p5 ← 0[p2]	1			
3	add p6 ← p4,p5	2			
4	st 0[p3] ← p6	2			
5	ld p7 ← 4[p1]				
6	ld p8 ← 4[p2]				
7	add p9 ← p7,p8				
8	st 4[p3] ← p9				

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 2:
Why don't we issue #2?

104

104

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	
2	ld p5 ← 0[p2]	1	3	6	
3	add p6 ← p4,p5	2			
4	st 0[p3] ← p6	2			
5	ld p7 ← 4[p1]	3			
6	ld p8 ← 4[p2]	3			
7	add p9 ← p7,p8				
8	st 4[p3] ← p9				

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 3:
Why don't we issue #3?
Why don't we issue #4?

105

105

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	
2	ld p5 ← 0[p2]	1	3	6	
3	add p6 ← p4,p5	2			
4	st 0[p3] ← p6	2			
5	ld p7 ← 4[p1]	3			
6	ld p8 ← 4[p2]	3			
7	add p9 ← p7,p8	4			
8	st 4[p3] ← p9	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 4:
Why don't we issue #5?

106

106

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	
3	add p6 ← p4,p5	2	6	7	
4	st 0[p3] ← p6	2			
5	ld p7 ← 4[p1]	3			
6	ld p8 ← 4[p2]	3			
7	add p9 ← p7,p8	4			
8	st 4[p3] ← p9	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 6:
Finally some action!

107

107

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	7
3	add p6 ← p4,p5	2	6	7	
4	st 0[p3] ← p6	2	7	8	
5	ld p7 ← 4[p1]	3			
6	ld p8 ← 4[p2]	3			
7	add p9 ← p7,p8	4			
8	st 4[p3] ← p9	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 7:
Getting somewhere….

108

108

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	7
3	add p6 ← p4,p5	2	6	7	8
4	st 0[p3] ← p6	2	7	8	
5	ld p7 ← 4[p1]	3	8	11	
6	ld p8 ← 4[p2]	3			
7	add p9 ← p7,p8	4			
8	st 4[p3] ← p9	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 8:
Etc…

109

109

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	7
3	add p6 ← p4,p5	2	6	7	8
4	st 0[p3] ← p6	2	7	8	9
5	ld p7 ← 4[p1]	3	8	11	
6	ld p8 ← 4[p2]	3	9	12	
7	add p9 ← p7,p8	4			
8	st 4[p3] ← p9	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 9:
Etc…

110

110

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	7
3	add p6 ← p4,p5	2	6	7	8
4	st 0[p3] ← p6	2	7	8	9
5	ld p7 ← 4[p1]	3	8	11	12
6	ld p8 ← 4[p2]	3	9	12	
7	add p9 ← p7,p8	4	12	13	
8	st 4[p3] ← p9	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 12:
Yawn…

111

111

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	7
3	add p6 ← p4,p5	2	6	7	8
4	st 0[p3] ← p6	2	7	8	9
5	ld p7 ← 4[p1]	3	8	11	12
6	ld p8 ← 4[p2]	3	9	12	13
7	add p9 ← p7,p8	4	12	13	
8	st 4[p3] ← p9	4	13	14	

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 13:
Stretch…

112

112

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	7
3	add p6 ← p4,p5	2	6	7	8
4	st 0[p3] ← p6	2	7	8	9
5	ld p7 ← 4[p1]	3	8	11	12
6	ld p8 ← 4[p2]	3	9	12	13
7	add p9 ← p7,p8	4	12	13	14
8	st 4[p3] ← p9	4	13	14	

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 14:
Zzzzzz…

113

113

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	7
3	add p6 ← p4,p5	2	6	7	8
4	st 0[p3] ← p6	2	7	8	9
5	ld p7 ← 4[p1]	3	8	11	12
6	ld p8 ← 4[p2]	3	9	12	13
7	add p9 ← p7,p8	4	12	13	14
8	st 4[p3] ← p9	4	13	14	15

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 15:
2-wide OoO = 1-wide inorder
I am very unhappy.

114

114

An example code

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	7
3	add p6 ← p4,p5	2	6	7	8
4	st 0[p3] ← p6	2	7	8	9
5	ld p7 ← 4[p1]	3	8	11	12
6	ld p8 ← 4[p2]	3	9	12	13
7	add p9 ← p7,p8	4	12	13	14
8	st 4[p3] ← p9	4	13	14	15

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

What was **#5** waiting for??

Can I speculate?

115

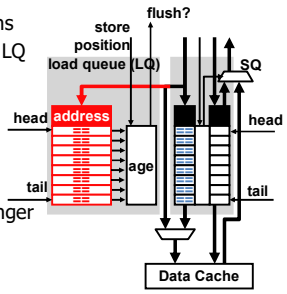
115

Load Speculation

- Speculation requires two things…..
  - Detection of mis-speculations
    - How can we do this?
  - Recovery from mis-speculations
    - Squash from offending load
    - Saw how to squash from branches: same method

Load Queue

- Detects ld ordering violations
- Execute load: write addr to LQ
  - Also note any store forwarded from
- Execute store: search LQ
  - Younger load with same addr?
  - Didn't forward from younger store?



Store Queue + Load Queue

- Store Queue: handles forwarding
  - Written by stores (@ execute)
  - Searched by loads (@ execute)
  - Read SQ when you write to the data cache (@ commit)
- Load Queue: detects ordering violations
  - Written by loads (@ execute)
  - Searched by stores (@ execute)
- Both together
  - Allows aggressive load scheduling
    - Stores don't constrain load execution

Our example from before

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	
2	ld p5 ← 0[p2]	1	3	6	
3	add p6 ← p4,p5	2			
4	st 0[p3] ← p6	2			
5	ld p7 ← 4[p1]	3	4	7	
6	ld p8 ← 4[p2]	3			
7	add p9 ← p7,p8	4			
8	st 4[p3] ← p9	4			

- 2 wide, **aggressive** scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 4:
Speculatively execute #5 before the store (#4).

Our example from before

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	
2	ld p5 ← 0[p2]	1	3	6	
3	add p6 ← p4,p5	2			
4	st 0[p3] ← p6	2			
5	ld p7 ← 4[p1]	3	4	7	
6	ld p8 ← 4[p2]	3	5	8	
7	add p9 ← p7,p8	4			
8	st 4[p3] ← p9	4			

- 2 wide, **aggressive** scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 5:
Speculatively execute #6 before the store (#4).

Our example from before

		Disp	Issue	WB	Commit
1	ld p4 ← 0[p1]	1	2	5	6
2	ld p5 ← 0[p2]	1	3	6	7
3	add p6 ← p4,p5	2	6	7	8
4	st 0[p3] ← p6	2	7	8	9
5	ld p7 ← 4[p1]	3	4	7	9
6	ld p8 ← 4[p2]	3	5	8	10
7	add p9 ← p7,p8	4	8	9	10
8	st 4[p3] ← p9	4	9	10	11

- 2 wide, **aggressive** scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Fast forward:
4 cycles faster
Actually OoO this time!

Aggressive Load Scheduling

- Allows loads to issue before older stores
  - Increases out-of-orderness
  - + When no conflict, increases performance
  - - Conflict → squash → worse performance than waiting
- Some loads might forward from stores
  - Always aggressive will squash a lot
- Can we have our cake AND eat it too?

Predictive Load Scheduling

- Predict which loads must wait for stores
- Fool me once, shame on you—fool me twice?
  - Loads default to aggressive
  - Keep table of load PCs that have been caused squashes
    - Schedule these conservatively
  - + Simple predictor
  - – Makes "bad" loads wait for *all* older stores: not great
- More complex predictors used in practice
  - Predict which stores loads should wait for

Out of Order: Window Size

- Scheduling scope = OoO window size
  - Larger = better
  - Constrained by physical registers (#Preg)
    - ROB roughly limited by #Preg = ROB size + #logical registers
    - Big register file = hard/slow
  - Constrained by issue queue
    - Limits number of un-executed instructions
    - CAM = can't make big (power + area)
  - Constrained by load + store queues
    - Limit number of loads/stores
    - CAMs
    - Active area of research: scaling window sizes
- Usefulness of large window: limited by branch prediction
  - 5% branch mis-prediction rate: 1 in 20 branches, 1 in 100 insns

Out of Order: Benefits

- Allows speculative re-ordering
  - Loads / stores
  - Branch prediction
- Schedule can change due to cache misses
  - Different schedule optimal from on cache hit
- Done by hardware
  - Compiler may want different schedule for different hw configs
  - Hardware has only its own configuration to deal with

Static vs. Dynamic Scheduling

- If we can do this in software…
- …why build complex (slow-clock, high-power) hardware?
  - + Performance portability
    - Don't want to recompile for new machines
  - + More information available
    - Memory addresses, branch directions, cache misses
  - + More registers available
    - Compiler may not have enough to schedule well
  - + Speculative memory operation re-ordering
    - Compiler must be conservative, hardware can speculate
  - – But compiler has a larger scope
    - Compiler does as much as it can (not much)
    - Hardware does the rest

Out of Order: Top 5 Things to Know

- Register renaming
  - How to perform it and how to recover it
- Commit
  - Precise state (ROB)
  - How/when registers are freed
- Issue/Select
  - Wakeup: CAM
  - Choose N oldest ready instructions
- Stores
  - Write at commit
  - Forward to loads via SQ
- Loads
  - Conservative/aggressive/predictive scheduling
  - Violation detection via LQ

Summary: Dynamic Scheduling

- Dynamic scheduling
  - Totally in the hardware
  - Also called "out-of-order execution" (OoO)
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline on branch misprediction
- Rename to avoid false dependencies
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies more tricky
- "Commit" instructions in order
  - Anything strange happens pre-commit, just flush the pipeline
- Current machines: 100+ instruction scheduling window

128