Lecture 9: Case Study— MIPS R4000 and Introduction to Advanced Pipelining

Professor Randy H. Katz

Computer Science 252

Spring 1996

Review: Evaluating Branch Alternatives

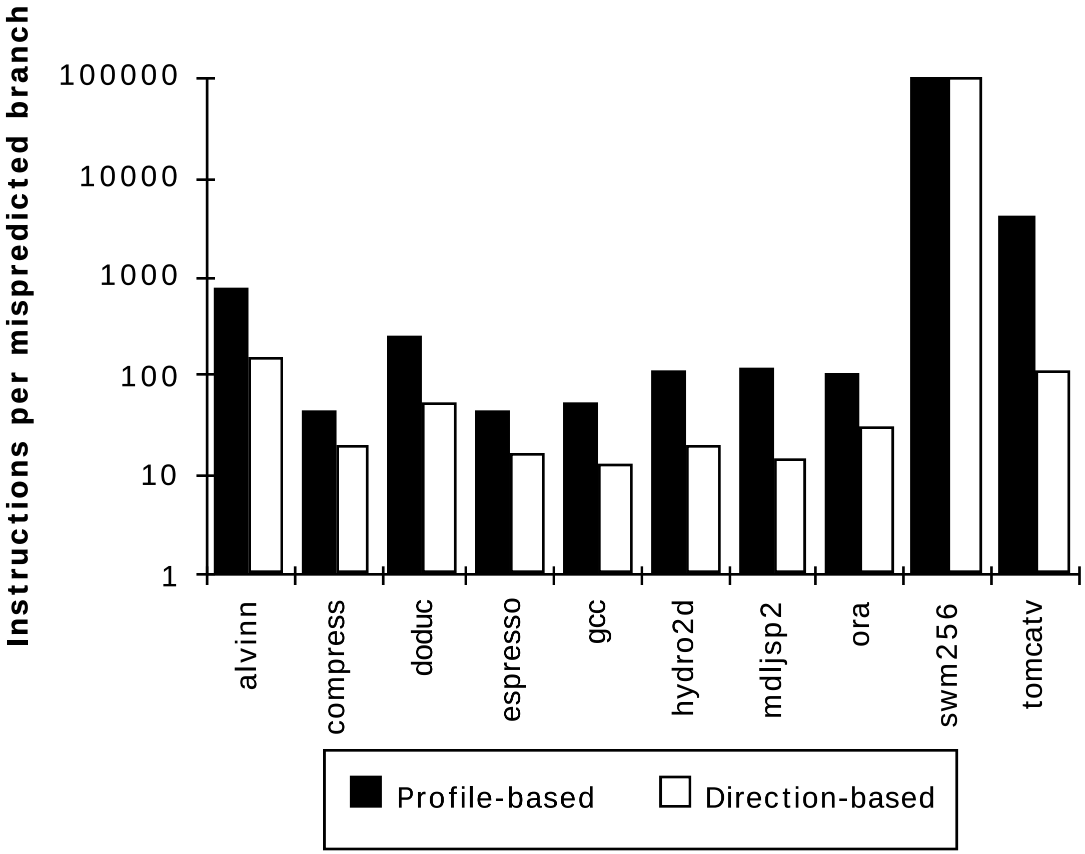
- **Two part solution:**
  - – Determine branch taken or not sooner, AND
  - – Compute taken branch address earlier

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	**3**	**1.42**	**3.5**	**1.0**
Predict taken	**1**	**1.14**	**4.4**	**1.26**
Predict not taken	**1**	**1.09**	**4.5**	**1.29**
Delayed branch	**0.5**	**1.07**	**4.6**	**1.31**

Review: Evaluating Branch Prediction Strategies

- **Two strategies**
  - **Backward branch predict taken, forward branch not taken**
  - **Profile-based prediction: record branch behavior, predict branch based on prior run**
- **"Instructions between mispredicted branches" a better metric than misprediction**



Review: Summary of Pipelining Basics

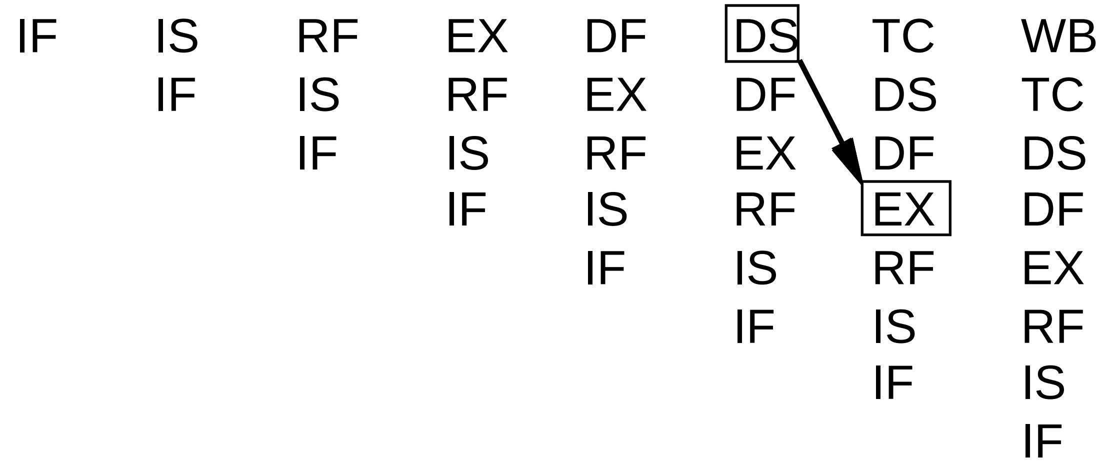
- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards; pipelining helps instruction bandwidth, not latency
- Interrupts, Instruction Set, FP makes pipelining harder
- Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction
- Today: Longer pipelines (R4000) => Better branch prediction, more instruction parallelism?

Case Study: MIPS R4000 (100 MHz to 200 MHz)

- 8 Stage Pipeline:
  - IF–first half of fetching of instruction; PC selection happens here as well as initiation of instruction cache access.
  - IS–second half of access to instruction cache.
  - RF–instruction decode and register fetch, hazard checking and also instruction cache hit detection.
  - EX–execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
  - DF–data fetch, first half of access to data cache.
  - DS–second half of access to data cache.
  - TC–tag check, determine whether the data cache access hit.
  - WB–write back for loads and register-register operations.
- 8 Stages: What is impact on Load delay? Branch delay? Why?

Case Study: MIPS R4000

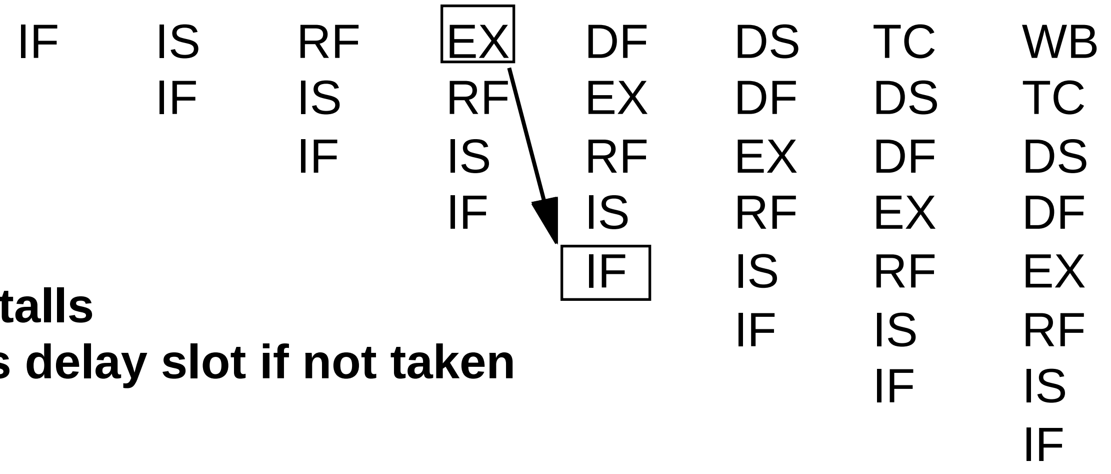
TWO Cycle
Load Latency



IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

THREE Cycle
Branch Latency

(conditions evaluated
during EX phase)



IF	IS	RF	EX	DF	DS	TC	WB
	IF	IS	RF	EX	DF	DS	TC
		IF	IS	RF	EX	DF	DS
			IF	IS	RF	EX	DF
				IF	IS	RF	EX
					IF	IS	RF
						IF	IS
							IF

Delay slot plus two stalls
Branch likely cancels delay slot if not taken

MIPS R4000 Floating Point

- FP Adder, FP Multiplier, FP Divider
- Last step of FP Multiplier/Divider uses FP Adder HW
- 8 kinds of stages in FP units:

Stage	*Functional unit*	*Description*
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

MIPS FP Pipe Stages

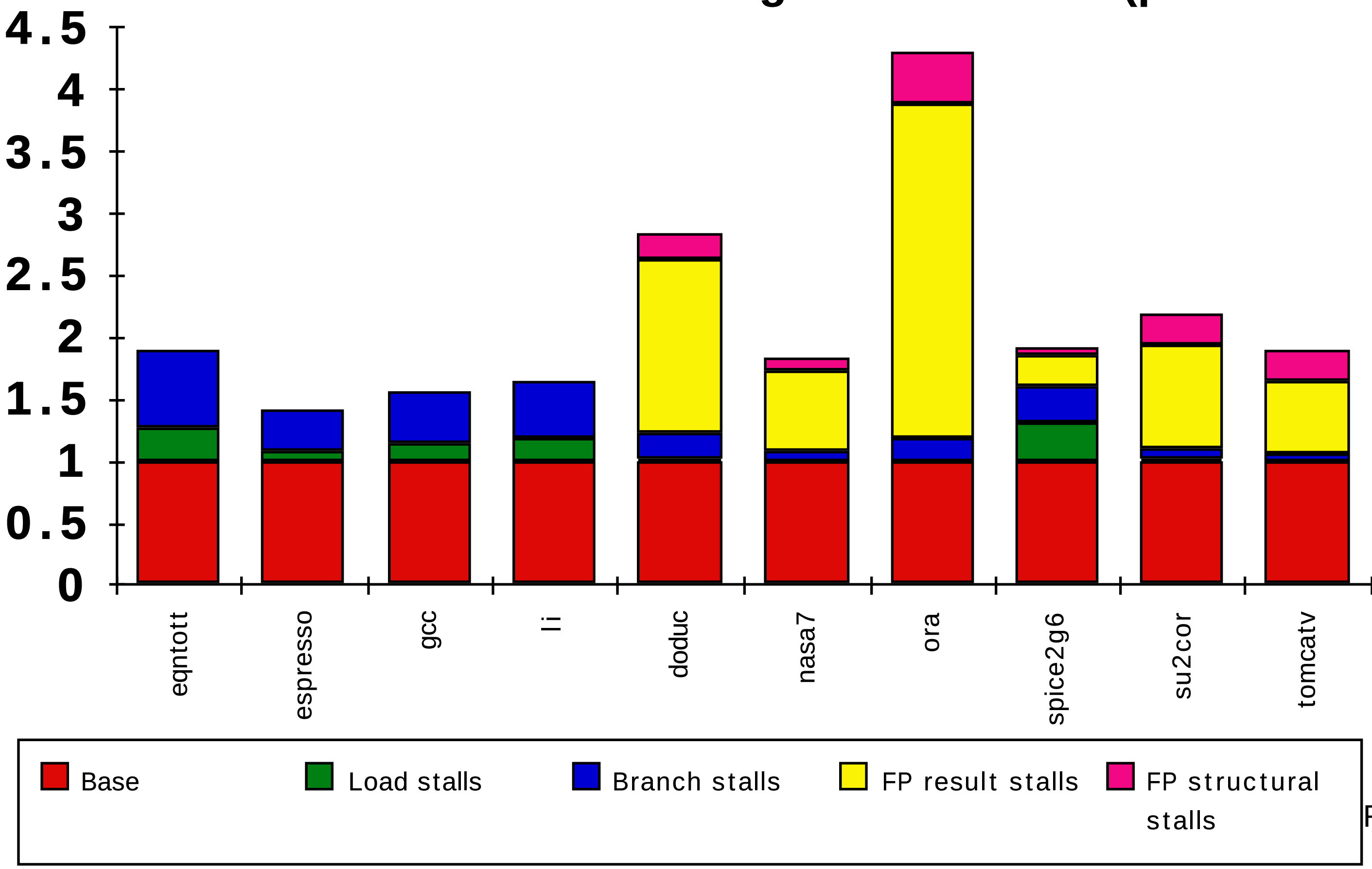
FP Instr	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*…*
Add, Subtract	**U**	**S+A**	**A+R**	**R+S**					
Multiply	**U**	**E+M**	**M**	**M**	**M**	**N**	**N+A**	**R**	
Divide	**U**	**A**	**R**	**D^{28}**	**…**	**D+A**	**D+R,**	**D+R,**	**D+A, D+R, A, R**
Square root	**U**	**E**	**(A+R)^{108}**		**…**	**A**	**R**		
Negate	**U**	**S**							
Absolute value	**U**	**S**							
FP compare	**U**	**A**	**R**						

Stages:

M	***First stage of multiplier***	***A***	***Mantissa ADD stage***
N	***Second stage of multiplier***	***D***	***Divide pipeline stage***
R	***Rounding stage***	***E***	***Exception test stage***
S	***Operand shift stage***		
U	***Unpack FP numbers***		

R4000 Performance

- Not ideal CPI of 1:
  - Load stalls (1 or 2 clock cycles)
  - Branch stalls (2 cycles + unfilled slots)
  - FP result stalls: RAW data hazard (latency)
  - FP structural stalls: Not enough FP hardware (parallelism)



Advanced Pipelining and Instruction Level Parallelism

- **gcc 17% control transfer**
  - **5 instructions + 1 branch**
  - **Beyond single block to get more instruction level parallelism**
- **Loop level parallelism one opportunity, SW and HW**
- **Do examples and then explain nomenclature**
- **DLX Floating Point as example**
  - **Measurements suggests R4000 performance FP execution has room for improvement**

FP Loop: Where are the Hazards?

```
Loop:  LD    F0,0(R1)   ;F0=vector element
       ADDD  F4,F0,F2   ;add scalar in F2
       SD    0(R1),F4   ;store result
       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
       BNEZ  R1,Loop    ;branch R1!=zero
       NOP              ;delayed branch slot
```

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
Integer op	Integer op	0

FP Loop Hazards

```
Loop:  LD    F0,0(R1)   ;F0=vector element
       ADDD  F4,F0,F2   ;add scalar in F2
       SD    0(R1),F4   ;store result
       SUBI  R1,R1,8    ;decrement pointer 8B (DW)
       BNEZ  R1,Loop    ;branch R1!=zero
       NOP              ;delayed branch slot
```

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
Integer op	Integer op	0

- **Where are the stalls?**

FP Loop Showing Stalls

```
1 Loop:  LD    F0,0(R1)   ;F0=vector element
2        stall
3        ADDD  F4,F0,F2   ;add scalar in F2
4        stall
5        stall
6        SD    0(R1),F4   ;store result
7        SUBI  R1,R1,8    ;decrement pointer 8B (DW)
8        BNEZ  R1,Loop    ;branch R1!=zero
9        stall            ;delayed branch slot
```

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

- **Rewrite code to minimize stalls?**

Revised FP Loop Minimizing Stalls

```
1 Loop:  LD     F0,0(R1)
2        stall
3        ADDD   F4,F0,F2
4        SUBI   R1,R1,8
5        BNEZ   R1,Loop    ;delayed branch
6        SD     8(R1),F4   ;altered when move past SUBI
```

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

Unroll loop 4 times code to make faster?

Unroll Loop Four Times

Rewrite loop to minimize stalls?

```
1  Loop: LD    F0,0(R1)
2        ADDD  F4,F0,F2
3        SD    0(R1),F4      ;drop SUBI & BNEZ
4        LD    F6,-8(R1)
5        ADDD  F8,F6,F2
6        SD    -8(R1),F8     ;drop SUBI & BNEZ
7        LD    F10,-16(R1)
8        ADDD  F12,F10,F2
9        SD    -16(R1),F12   ;drop SUBI & BNEZ
10       LD    F14,-24(R1)
11       ADDD  F16,F14,F2
12       SD    -24(R1),F16
13       SUBI  R1,R1,#32     ;alter to 4*8
14       BNEZ  R1,LOOP
15       NOP
```

15 + 4 x (1+2) = 27 clock cycles, or 6.8 per iteration

Assumes R1 is multiple of 4

Unrolled Loop That Minimizes Stalls

```
1 Loop: LD      F0,0(R1)
2       LD      F6,-8(R1)
3       LD      F10,-16(R1)
4       LD      F14,-24(R1)
5       ADDD    F4,F0,F2
6       ADDD    F8,F6,F2
7       ADDD    F12,F10,F2
8       ADDD    F16,F14,F2
9       SD      0(R1),F4
10      SD      -8(R1),F8
11      SD      -16(R1),F12
12      SUBI    R1,R1,#32
13      BNEZ    R1,LOOP
14      SD      8(R1),F16     ; 8-32 = -24
```

14 clock cycles, or 3.5 per iteration

- **What assumptions made when moved code?**
  - **OK to move store past SUBI even though changes register**
  - **OK to move loads before stores: get right data?**
  - **When is it safe for compiler to do such changes?**

Compiler Perspectives on Code Movement

- **Definitions: compiler concerned about dependencies in program, whether or not a HW hazard depends on a given pipeline**
- **(True) Data dependencies (RAW if a hazard for HW)**
  - **Instruction i produces a result used by instruction j, or**
  - **Instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.**
- **Easy to determine for registers (fixed names)**
- **Hard for memory:**
  - **Does 100(R4) = 20(R6)?**
  - **From different loop iterations, does 20(R6) = 20(R6)?**

Compiler Perspectives on Code Movement

- Another kind of dependence called name dependence: two instructions use same name but don't exchange data
- Antidependence (WAR if a hazard for HW)
  - Instruction j writes a register or memory location that instruction i reads from and instruction i is executed first
- Output dependence (WAW if a hazard for HW)
  - Instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

Compiler Perspectives on Code Movement

- **Again Hard for Memory Accesses**
  - Does 100(R4) = 20(R6)?
  - From different loop iterations, does 20(R6) = 20(R6)?
- **Our example required compiler to know that if R1 doesn't change then:**

  0(R1) ≠ -8(R1) ≠ -16(R1) ≠ -24(R1)

  There were no dependencies between some loads and stores so they could be moved by each other

Compiler Perspectives on Code Movement

- Final kind of dependence called control dependence
- Example

```
    if p1 {S1;};
    if p2 {S2;}
```

S1 is control dependent on p1 and S2 is control dependent on p2 but not on p1.

Compiler Perspectives on Code Movement

- Two (obvious) constraints on control dependences:
  - An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch.
  - An instruction that is not control dependent on a branch cannot be moved to after the branch so that its execution is controlled by the branch.
- Control dependencies relaxed to get parallelism; get same effect if preserve order of exceptions and data flow

When Safe to Unroll Loop?

- **Example: Where are data dependencies? (A,B,C distinct & nonoverlapping)**

```
for (i=1; i<=100; i=i+1) {
    A[i+1] = A[i] + C[i];    /* S1 */
    B[i+1] = B[i] + A[i+1];} /* S2 */
```

  1. S2 uses the value, A[i+1], computed by S1 in the same iteration.

  2. S1 uses a value computed by S1 in an earlier iteration, since iteration i computes A[i+1] which is read in iteration i+1. The same is true of S2 for B[i] and B[i+1].
  This is a "loop-carried dependence": between iterations

- **Implies that iterations are dependent, and can't be executed in parallel**
- **Not the case for our example; each iteration was distinct**

Summary

- Instruction Level Parallelism in SW or HW
- Loop level parallelism is easiest to see
- SW parallelism dependencies defined for program, hazards if HW cannot resolve
- SW dependencies/compiler sophistication determine if compiler can unroll loops