Review: Evaluating Branch Alternatives

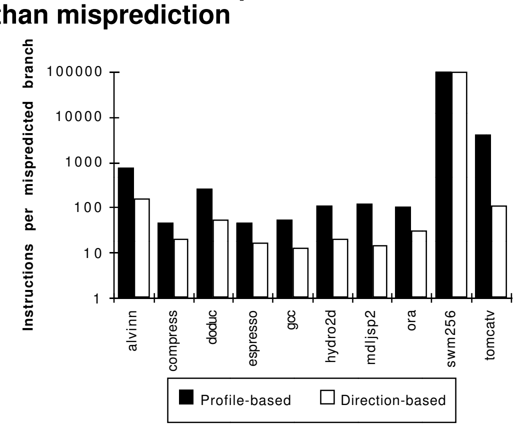
- 2 part solution:
  - –Determine branch taken or not sooner, AND
  - –Compute taken branch address earlier

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

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 untaken
  - – Profile-based prediction: record branch behavior, predict branch based on prior run
- "Instructions between mispredicted branches" a better metric than misprediction

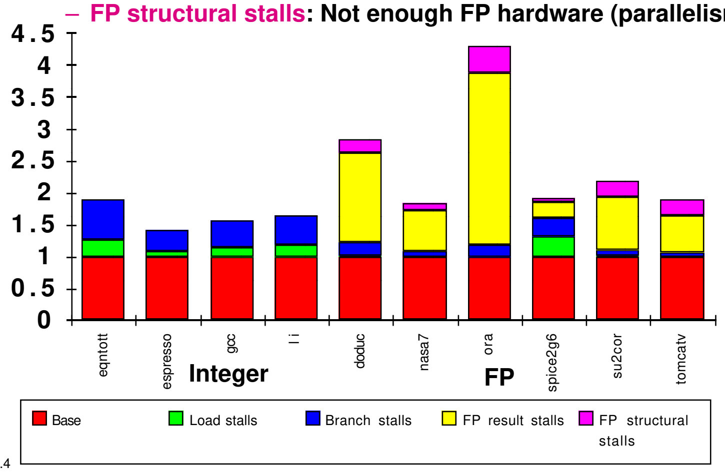


Review: MIPS R4000 (100 MHz to 200 MHz)

- 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 => 2 cycle Load delay & 3 cycle Branch delay

Review: Performance of R4000 8 stage pipeline

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



Review: Summary of Pipelining Basics

- Hazards limit performance on computers:
  - 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 since pipelining helps instruction bandwidth, not latency
- Interrupts, Instruction Set, FP makes pipelining harder
- Compilers key to reducing cost of data and control hazards
  - load delay slots
  - branch delay slots
  - branch prediction
- Longer pipelines (R4000) => Better branch prediction, more instruction 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 from last time suggests R4000 performance suggests 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 4 times

```
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

Rewrite loop to minimize stalls?

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:
2 instructions use same name but don t exchange data

Antidependance (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 dependance (WAW if a hazard for HW)

- ≅ instruction i and instruction j write the same register or memory location; ordering between instructions must be preserved.

Again Hard for memory:
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.

2 (obvious) constraints on control dependences:

1. 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.
2. An instruction that is not control dependent on a branch cannot be moved to after the branch so that it s 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)

```
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]. (Called a ɘloop-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

Hardware Schemes for Instruction Parallelism

Why in HW at run time?
- ≅ Works when can t know dependence at run time
- ≅ compiler simpler
- ≅ code for one machine runs well on another

Key idea: Allow instructions behind stall to proceed

```
DIVD  F0,F2,F4
ADDD  F10,F0,F8
SUBD  F8,F8,F14
```

enables out-of-order execution => out-of-order completion

ID stage checked both for structural & data dependencies

Out-of-order execution divides ID stage:

1. Issue decode instructions, check for structural hazards
2. Read operands wait until no data hazards, then read operands

Scoreboards allow instruction to execute whenever 1 & 2 hold, not waiting for prior instructions

Scoreboard implications

Out-of-order completion => WAR, WAW hazards?

Solutions for WAR
- ≅ queue both the operation and copies of its operands
- ≅ read registers only during Read Operands stage

For WAW, must detect hazard: stall until other completes

Need to have multiple instructions in execution phase => multiple execution units or pipelined execution units

Scoreboard keeps track of dependencies, state or operations

Scoreboard replaces ID, EX, WB with 4 stages

4 Stages of Scoreboard Control

1. **Issue** decode instructions & check for structural hazards (ID1)

    If a functional unit for the instruction is free and no other active instruction has the same destination register (WAW), the scoreboard issues the instruction to the functional unit and updates its internal data structure. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared.

2. **Read operands** wait until no data hazards, then read operands (ID2)

    A source operand is available if no earlier issued active instruction is going to write it, or if the register containing the operand is being written by a currently active functional unit. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step, and instructions may be sent into execution out of order.

4 Stages of Scoreboard Control (cont d)

3. **Execution** operate on operands (EX)

    The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution.

4. **Write result** finish execution (WB)

    Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards. If none, it writes results. If WAR, then it stalls the instruction.

    ≅ Example

    ```
    DIVD   F0,F2,F4
    ADDD   F10,F0,F8
    SUBD   F8,F8,F14
    ```

    CDC 6600 scoreboard would stall SUBD until ADDD reads operands

3 Parts of the Scoreboard

1. **Instruction status** which of 4 steps the instruction is in.
2. **Functional unit status** Indicates the state of the functional unit (FU). 9 fields for each functional unit:
   - **Busy** Indicates whether the unit is busy or not
   - **Op** Operation to perform in the unit (e.g., + or ≅)
   - **Fi** Destination register
   - **Fj, Fk** Source-register numbers
   - **Qj, Qk** Functional units producing source registers Fj, Fk
   - **Rj, Rk** Flags indicating when Fj, Fk are ready
3. **Register result status** Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.

Detailed Scoreboard Pipeline Control

Instruction status	Wait until	Bookkeeping
Issue	Not busy (FU) and not result(D)	Busy(FU)← yes; Op(FU)← op; Fi(FU)← `D ; Fj(FU)←`S1 ; Fk(FU)← `S2 ; Qj← Result(≅S1 ); Qk← Result(`S2); Rj← not Qj; Rk← not Qk; Result(≅D)← FU;
Read operands	Rj and Rk	Rj← No; Rk← No
Execution complete	Functional unit done	
Write result	∀f((Fj(f) Fi(FU) or Rj(f)=No) & (Fk(f) Fi(FU) or Rk(f)=No))	∀f(if Qj(f)=FU then Rj(f)← Yes); ∀f(if Qk(f)=FU then Rj(f)← Yes); Result(Fi(FU))← 0; Busy(FU)← No

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
- Scoreboard is one dynamic HW scheme; Drawbacks of 6600 scoreboard
  - ≅ Speedup 1.7 from compiler; 2.5 by hand BUT slow memory (no cache)
  - ≅ No forwarding
  - ≅ Limited to instructions in basic block (small *window*)
  - ≅ Number of functional units
  - ≅ Presence of WAR, WAW hazards