CMSC 411
Computer Systems Architecture
Lecture 8
Instruction Level Parallelism

Alan Sussman
als@cs.umd.edu

Administrivia

- **HW #2, on pipelining, due Thursday**
- **Continue reading Chapter 2 of H&P**
- **First exam scheduled for Thursday, March 5**
  - **on Units 1-3**

Outline

- **ILP**
- **Compiler techniques to increase ILP**
- **Loop Unrolling**
- **Static Branch Prediction**
- **Dynamic Branch Prediction**
- **Overcoming Data Hazards with Dynamic Scheduling**
- **(Start) Tomasulo Algorithm**
- **Conclusion**

Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

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

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

Control Dependence Ignored

- Control dependence need not be preserved
  - willing to execute instructions that should not have been executed, thereby violating the control dependences, *if* can do so without affecting correctness of the program
- Instead, 2 properties critical to program correctness are

  1) exception behavior and

  2) data flow

Exception Behavior

- Preserving exception behavior
  - ➢ any changes in instruction execution order must not change how exceptions are raised in program (no new exceptions)
- Example:

```
        DADDU    R2,R3,R4
        BEQZ     R2,L1
        LW       R1,0(R2)
L1:
```

  - (Assume branches not delayed)
- Problem with moving LW before BEQZ?

Data Flow

- Data flow: actual flow of data values among instructions that produce results and those that consume them
  - branches make flow dynamic, determine which instruction is supplier of data
- Example:

```
        DADDU   R1,R2,R3
        BEQZ    R4,L
        DSUBU   R1,R5,R6
L:      …
        OR      R7,R1,R8
```

- OR depends on DADDU or DSUBU?
  Must preserve data flow on execution

Outline

- ILP
- Compiler techniques to increase ILP
- **Loop Unrolling**
- **Static Branch Prediction**
- **Dynamic Branch Prediction**
- **Overcoming Data Hazards with Dynamic Scheduling**
- **(Start) Tomasulo Algorithm**
- **Conclusion**

Software Techniques - Example

- This code, add a scalar to a vector:

```
for (i=1000; i>0; i=i–1)
    x[i] = x[i] + s;
```

- Assume following latencies for all examples
  - Ignore delayed branch in these examples

Instruction producing result	*Instruction using result*	*Latency in cycles*	*stalls between in cycles*
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0

FP Loop: Where are the Hazards?

- First translate into MIPS code:
  - To simplify, assume 8 is lowest address

```
for (i=1000; i>0; i=i–1)
    x[i] = x[i] + s;
```

```
Loop:  L.D    F0,0(R1) ;F0=vector element
       ADD.D  F4,F0,F2 ;add scalar from F2
       S.D    0(R1),F4 ;store result
       DADDUI R1,R1,-8 ;decrement pointer 8B (DW)
       BNEZ   R1,Loop  ;branch R1!=zero
```


FP Loop Showing Stalls

```
for (i=1000; i>0; i=i–1)
    x[i] = x[i] + s;
```

```
1 Loop:  L.D    F0,0(R1)   ;F0=vector element
2        stall
3        ADD.D  F4,F0,F2   ;add scalar in F2     plus branch delay!
4        stall
5        stall
6        S.D    0(R1),F4   ;store result
7        DADDUI R1,R1,-8   ;decrement pointer 8B (DW)
8        stall             ;assumes can't forward to branch
9        BNEZ   R1,Loop    ;branch R1!=zero
```

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

- 9 clock cycles: Rewrite code to minimize stalls?

Revised FP Loop Minimizing Stalls

```
1 Loop:  L.D    F0,0(R1)
2        DADDUI R1,R1,-8
3        ADD.D  F4,F0,F2
4        stall
5        stall
6        S.D    8(R1),F4;altered offset when move DSUBUI
7        BNEZ   R1,Loop
```

Swap DADDUI and S.D by changing address of S.D

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**

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How make faster?

Unroll Loop Four Times (straightforward way)

```
1 Loop:L.D    F0,0(R1)
3      ADD.D  F4,F0,F2
6      S.D    0(R1),F4     ;drop DSUBUI & BNEZ
7      L.D    F6,-8(R1)
9      ADD.D  F8,F6,F2
12     S.D    -8(R1),F8    ;drop DSUBUI & BNEZ
13     L.D    F10,-16(R1)
15     ADD.D  F12,F10,F2
18     S.D    -16(R1),F12  ;drop DSUBUI & BNEZ
19     L.D    F14,-24(R1)
21     ADD.D  F16,F14,F2
24     S.D    -24(R1),F16
25     DADDUI R1,R1,#-32   ;alter to 4*8
27     BNEZ   R1,LOOP
```

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4)

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n, and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
  - 1st executes (n mod k) times and has a body that is the original loop
  - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n, most of the execution time will be spent in the unrolled loop

Unrolled Loop That Minimizes Stalls

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

14 clock cycles, or 3.5 per iteration

5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:

1. Determine loop unrolling useful by finding that loop iterations were *independent* (except for maintenance code)
2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
   - Transformation requires analyzing memory addresses and finding that they do not refer to the same address
5. Schedule the code, preserving any dependences needed to yield the same result as the original code

3 Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling
   - Amdahl's Law
2. Growth in code size
   - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not be possible to allocate all live values to registers, may lose some or all of its advantage

- Loop unrolling reduces impact of branches on pipeline; another way is *branch prediction*

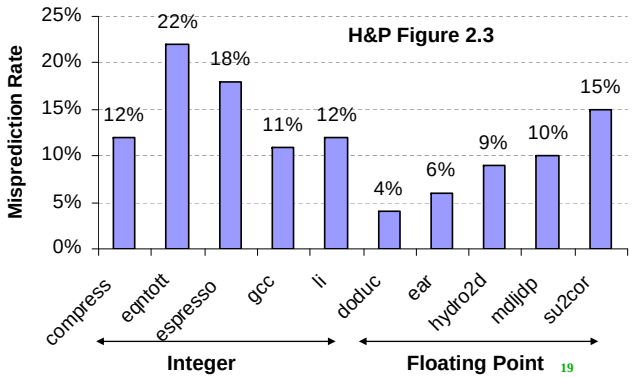
Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- **Static Branch Prediction**
- **Dynamic Branch Prediction**
- **Overcoming Data Hazards with Dynamic Scheduling**
- **(Start) Tomasulo Algorithm**
- **Conclusion**

Static Branch Prediction

- Prior lecture showed scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
  - Average misprediction = untaken branch frequency = 34% SPEC92

More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run:



Dynamic Branch Prediction

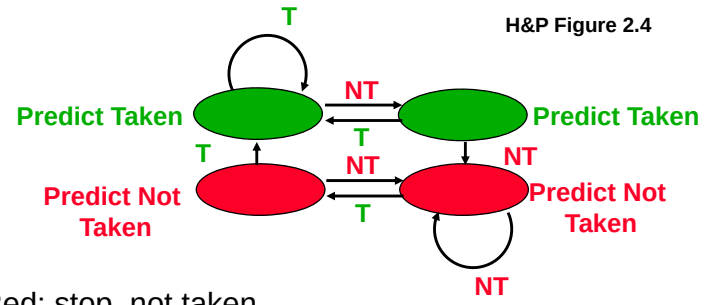
- Why does prediction work?
  - Underlying algorithm has regularities
  - Data that is being operated on has regularities
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems
- Is dynamic branch prediction better than static branch prediction?
  - Seems to be
  - There are a small number of important branches in programs that have dynamic behavior

Dynamic Branch Prediction

- Performance = *ƒ*(accuracy, cost of misprediction)
- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 loop iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on *next* time through code, when it predicts exit instead of looping

Dynamic Branch Prediction

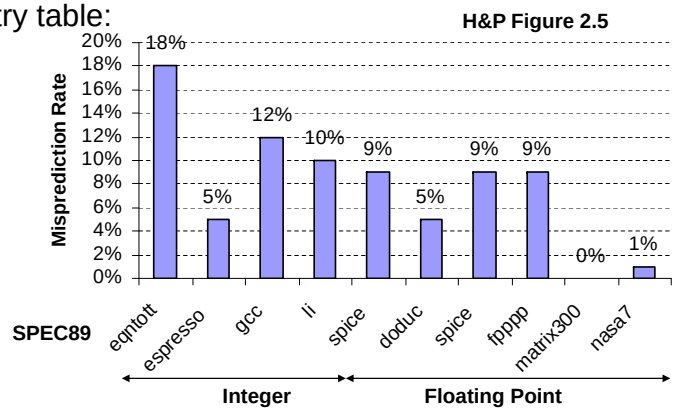
- Solution: 2-bit scheme where change prediction only if get misprediction *twice*



- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

BHT Accuracy

- Mispredict because either:
  - Wrong guess for that branch
  - Got branch history of wrong branch when index the table
- 4096 entry table:



Accuracy of Different Schemes