COSC4201
Instruction Level Parallelism

Prof. Mokhtar Aboelaze

Parts of these slides are taken from Notes by
Prof. David Patterson (UCB)

Outline

- Data dependence and hazards
- Exposing parallelism (loop unrolling and scheduling)
- Reducing branch costs (prediction)
- Dynamic scheduling
- Speculation
- Multiple issue and static scheduling
- Advanced techniques
- Example

ILP

- What is Instruction Level Parallelism?
- For average MIPS progem, the average branch frequency 15%-25%
- That means average size of Basic Block is 5-7 instructions.
- Instructions in a basic block is likely to depend on one another, not much room for ILP
- Solution: Exploit ILP across multiple basic blocks.

ILP Concept

- Data and control hazards put a limit on the ability of the processor to exploit parallelism.
- There are two approaches to exploit ILP, software and hardware based.
  - Hardware: Depends on the hardware to locate and exploit parallelism between the instructions, scoreboard and Tomasulo's algorithm(dynamic)
  - Software: Depends on the compiler to exploit ILP (static)

Vector Processing

- Consider the following loop

```
for(i=1; i<=1000;i++)
    x[i]=x[i]+y[i]
```

- No dependence between iterations.
- All iterations can be done in parallel (if no structural hazards).
- Vector processing is ideal for such a case.
- Not widely used in general purpose applications.

Data Dependences and Hazards

- Objective is to determine *parallel instructions*, those instructions that can run in parallel.
- If they can run in parallel, then they can overlap in a pipeline.
- There are three different type of dependence, data dependence, name dependence, and control dependence.

Data Dependence

- An instruction *j* is data dependent on instruction *i, if either*
  - instruction *i produces a data that is used by j,*

```
i: add r1,r2,r3
J: sub r4,r1,r3
```

  - Instruction *j* is data dependent on *k*, and *k* is data dependent on *l*
- True dependence, we cannot ignore or avoid, data must be produced before it is consumed.

Data Dependence

- The arrows show the data dependence.
- Either the compiler must not schedule the instruction this way, or the processor interlock detects it and stall.
- Dependence is a property of the program, whether it results in a hazard or not, that depends on the pipeline (hardware)
- Much more difficult is data passed through memory rather than registers

```
Loop:LOAD  F0,0(R1)
     ADD   F4,F0,F2
     SD    F4,0(R1)
     ADD   R1,R1,#-8
     BNE   R1,R2,Loop
```

Name Dependence

° There are two types of name dependence, *antidependence*, and *name dependence*.

- Anitdependence between instructions *i*, and *j* when *j* writes a register or memory location that *i* reads.

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```


Name Dependence

- An output dependence is when two instructions write to the same register or memory location.

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

° There is no value being transmitted between instructions, could be solved by *register renaming*

Control Dependence

```
If p1 {
     S1;
};
```

- S1 is control dependent on p1
- Can not move S1 before if, and can not move another instruction in the loop.
- We can violate control dependence and execute instruction that is not suppose to, if that will maintain the *program correctness*.
- We mostly care about *exceptions behavior* and *data flow*

Control Dependence

```
        ADD    R2,R3,R4
        BEQZ   R2,L1
        LW     R1,0(R2)
L1:     ….
```

No data dependence,
can we move LW
before BEQZ?

Control Dependence

```
      ADD   R1,R2,R3
      BEQZ  R4,L1
      SUB   R1,R5,R6
      ...
L1:   OR    R7,R1,R8
```

```
      ADD   R1,R2,R3
      BEQZ  R4,L1
      SUB   R4,R5,R6
      ADD   R5,R4,R9
L1:   OR    R7,R1,R8
```


Basic Techniques for Exposing ILP

- Consider a MIPS like pipeline with the following specs

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**

Basic Techniques for Exposing ILP

° Consider the following code

```
For (i=1000; i>0; i--)
        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    F4, 0(R1); store result
      DADDUI R1,R1,-8;  decrement pointer 8B (DW)
      BNE    R1,R2Loop; branch R1!=zero
```


Basic Techniques for Exposing ILP

1 Loop:	L.D	F0,0(R1)
2	stall	
3	ADD.D	F4,F0,F2
4	stall	
5	stall	
6	S.D	0(R1),F4
7	DADDUI	R1,R1,-8
8	stall	
9	BNE	R1,R2,Loop

After rescheduling

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	BNE	R1,R2,Loop	

Loop Unrolling

```
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
26      BNEZ   R1,LOOP
```

27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4), what if it is not?

Loop Unrolling

```
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

Loop Unrolling

- Loop unrolling decreases the impact of branching on performance by amortizing the cost over many iterations.
- Increases the size of the code
- Requires the use of more registers