Instruction Level Parallelism

Instruction level parallelism (ILP) = fine-grained parallelism

- from instruction overlap
- from issuing instructions in parallel (later)

Want to increase it

Contrast to:

- **loop-level** parallelism (medium-grained)
- **process-level** parallelism (coarse-grained)

Convert higher-level parallelism into ILP

- from loop level: loop unrolling & code scheduling
- from thread-level: simultaneous multithreading

Instruction Level Parallelism

Dependences vs. hazards

- dependences are a property of the code
  - dependence: flow, antidependence, output
  - instructions with dependences can’t be reordered
  - instructions with dependences can’t completely overlap
  - determine lower limit on ILP
- hazards stem from pipeline structure
  - RAW, WAR, WAW
  - pipeline implementation can eliminate hazards
- HW & SW techniques to increase ILP
  - eliminate dependences
  - reduce stalls

Dependences

data dependence: flow or true

name dependence: antidependence, output

control dependence

Must preserve:

- **data flow** (not merely dependence)

```
         add r1, r2, r3
         beqz r4, target
         sub r1, r5, r6
target:  or r7, r1, r8
```

- **exception behavior**: code scheduling cannot introduce new exceptions:

```
         beqz r2, target
         lw r1, 0(r2)
target:  add r1, ...
```

Dependences inhibit ILP

- how can we eliminate the dependence?

Register Renaming

Eliminates name dependences (**anti**, output)

- static register renaming

original code	after renaming
`add r1, r8, r3`	`add r7, r8, r3`
`sub r2, r3, r4`	`sub r6, r3, r4`
`and r5, r1, r2`	`and r5, r7, r6`
`ld r8, 0(r4)`	`ld r9, 0(r4)`

\+ increases ILP by eliminating name dependences

\- increases register pressure

- hardware register renaming *(later)*

Dependences

Loop level parallelism: parallelism across iterations of a loop

- analysis of loop-level parallelism determines dependences across loop iterations
- **loop-independent dependences**
  - occur within an iteration
  - can parallelize the loop
- **loop-carried dependences**
  - exist between different iterations of a loop
  - cyclical - cannot parallelize

```
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
     }
```

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


Dependences

- **loop-carried dependences**
  - no cycle - can parallelize

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

```
     A[1] = A[1] + B[1]
for (i=1; i<=99; i=i+1) {
     B[i+1] = C[i] + D[i];         S2
     A[i+1] = A[i+1] + B[i+1];  S1
     }
     B[101] = C[100] + D[100];
```

Loop Unrolling

```
loop:  ld f0, 0(r1)
       fadd f4, f0, f2
       sd 0(r1), f4
       subi r1, r1, #8
       bnez r1, loop
```

```
loop:  ld f0, 0(r1)
       fadd f4, f0, f2
       sd 0(r1), f4
       ld f0, -8(r1)
       fadd f4, f0, f2
       sd -8(r1), f4
       ld f0, -16(r1)
       fadd f4, f0, f2
       sd -16(r1), f4
       subi r1, r1, #24
       bnez r1, loop
```

Unrolling factor: number of times a loop is unrolled

Adv: execute fewer instructions

eliminate some data/control dependences

Disadv: increase code side, bigger cache footprint

Loop Unrolling

Can not eliminate all data hazards

loop: ld f0, 0(r1)	delayed load
stall	
fadd f4, f0, f2	3 cycle latency
stall	
stall	
sd 0(r1), f4	
ld f0, -8(r1)	delayed load
stall	
fadd f4, f0, f2	3 cycle latency
stall	
stall	
sd -8(r1), f4	
subi r1, r1, #16	
bnez r1, loop	delayed branch
stall	

Loop Unrolling

```
loop: ld f0, 0(r1)          load delay slot filled
      ld f6, -8(r1)         load delay slot filled
      fadd f4, f0, f2
      fadd f8, f6, f2
      stall                 1 cycle not hidden
      sd 0(r1), f4
      subi r1, r1, #16
      bnez r1, loop
      sd 8(r1), f8          branch delay slot filled
```

Combine with code scheduling

- adv: fewer stalls because loop unrolling exposes more instructions to use in hiding delays
- disadv: increase in register pressure