Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to maximize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

Chapter 3

Instruction-Level Parallelism and Its Exploitation

Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)
- Challenges:
  - Data dependency
    - Instruction *j* is data dependent on instruction *i* if
      - Instruction *i* produces a result that may be used by instruction *j*
      - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously

Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”
- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

Other Factors

- Data Hazards
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)
- Control Dependence
  - Ordering of instruction i with respect to a branch instruction
    - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
    - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall
- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect

Examples

- Example 1:

```
    add x1,x2,x3
    beq x4,x0,L
    sub x1,x1,x6
L:  …
    or x7,x1,x8
```

- or instruction dependent on add and sub

- Example 2:

```
    add x1,x2,x3
    beq x12,x0,skip
    sub x4,x5,x6
    add x5,x4,x9
skip:
    or x7,x8,x9
```

- Assume x4 isn't used after skip
  - Possible to move sub before the branch

Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, but is a problem when reordering instructions
  - Antidependence: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - Output dependence: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use register renaming techniques

Loop Unrolling

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   fld f0,0(x1)
        fadd.d f4,f0,f2
        fsd f4,0(x1) //drop addi & bne
        fld f6,-8(x1)
        fadd.d f8,f6,f2
        fsd f8,-8(x1) //drop addi & bne
        fld f0,-16(x1)
        fadd.d f12,f0,f2
        fsd f12,-16(x1) //drop addi & bne
        fld f14,-24(x1)
        fadd.d f16,f14,f2
        fsd f16,-24(x1)
        addi x1,x1,-32
        bne x1,x2,Loop
```

- note: number of live registers vs. original loop

Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

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

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```
Loop:   fld f0,0(x1)
        fld f6,-8(x1)
        fld f8,-16(x1)
        fld f14,-24(x1)
        fadd.d f4,f0,f2
        fadd.d f8,f6,f2
        fadd.d f12,f0,f2
        fadd.d f16,f14,f2
        fsd f4,0(x1)
        fsd f8,-8(x1)
        fsd f12,-16(x1)
        fsd f16,-24(x1)
        addi x1,x1,-32
        bne x1,x2,Loop
```

- 14 cycles
- 3.5 cycles per element

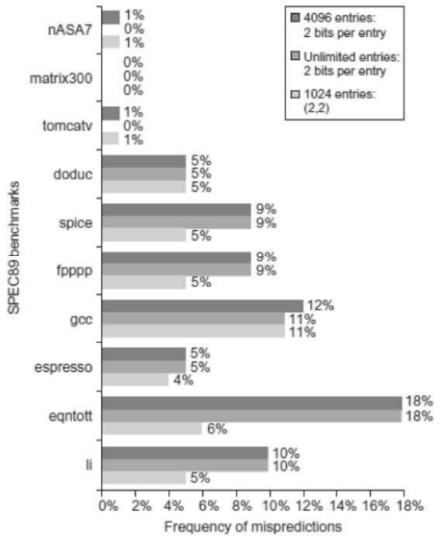
Pipeline Stalls


```
Loop:   fld      f0,0(x1)
        stall
        fadd.d f4,f0,f2
        stall
        stall
        fsd f4,0(x1)
        addi x1,x1,-8
        bne x1,x2,Loop
```

```
Loop:   fld      f0,0(x1)
        addi x1,x1,-8
        fadd.d f4,f0,f2
        stall
        stall
        fsd f4,0(x1)
        bne x1,x2,Loop
```

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

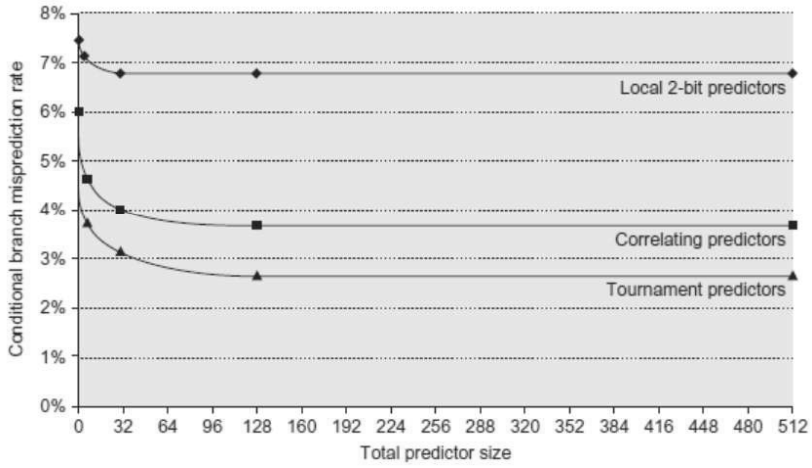
Branch Prediction Performance



Strip Mining

- Unknown number of loop iterations?
  - Number of iterations = *n*
  - Goal: make *k* copies of the loop body
  - Generate pair of loops:
    - First executes *n* mod *k* times
    - Second executes *n* / *k* times
    - "Strip mining"

Branch Prediction Performance



Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches
    - (*m*,*n*) predictor: behavior from last *m* branches to choose from 2^m *n*-bit predictors
- Tournament predictor:
  - Combine correlating predictor with local predictor

Dynamic Scheduling

- Example 2:

  fdiv.d f0,f2,f4

  fmul.d f6,f0,f8

  fadd.d f0,f10,f14

  - fadd.d is not dependent, but the antidependence makes it impossible to issue earlier without register renaming

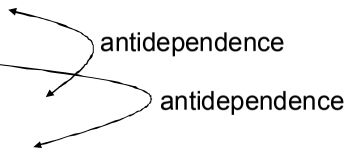
Dynamic Scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow
- Advantages:
  - Compiler doesn't need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time
- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions

Register Renaming

- Example 3:

fdiv.d f0,f2,f4
fadd.d f6,f0,f8
fsd f6,0(x1)
fsub.d f8,f10,f14
fmul.d f6,f10,f8



  - name dependence with f6

Dynamic Scheduling

- Dynamic scheduling implies:
  - Out-of-order execution
  - Out-of-order completion
- Example 1:

  fdiv.d f0,f2,f4

  fadd.d f10,f0,f8

  fsub.d f12,f8,f14

  - fsub.d is not dependent, issue before fadd.d

Register Renaming

- RS fetches and buffers an operand as soon as it becomes available (not necessarily involving register file)
- Pending instructions designate the RS to which they will send their output
  - Result values broadcast on a result bus, called the common data bus (CDB)
- Only the last output updates the register file
- As instructions are issued, the register specifiers are renamed with the reservation station
- May be more reservation stations than registers
- Load and store buffers
  - Contain data and addresses, act like reservation stations

Register Renaming

- Example 3:

```
fdiv.d f0,f2,f4
fadd.d S,f0,f8
fsd S,0(x1)
fsub.d T,f10,f14
fmul.d f6,f10,T
```

- Now only RAW hazards remain, which can be strictly ordered

Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results if prediction was correct
- Instruction commit: allowing an instruction to update the register file when instruction is no longer speculative
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - I.e. updating state or taking an execution

Register Renaming

- Tomasulo's Approach
  - Tracks when operands are available
  - Introduces register renaming in hardware
    - Minimizes WAW and WAR hazards
- Register renaming is provided by reservation stations (RS)
  - Contains:
    - The instruction
    - Buffered operand values (when available)
    - Reservation station number of instruction providing the operand values

Fallacies and Pitfalls

- Processors with lower CPIs / faster clock rates will also be faster

Processor	Implementation technology	Clock rate	Power	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	90 nm	3.8 GHz	115 W	11.5	12.2
Intel Itanium 2	90 nm	1.66 GHz	104 W approx. 70 W one core	14.5	17.3
Intel i7 920	45 nm	3.3 GHz	130 W total approx. 80 W one core	35.5	38.4

  - Pentium 4 had higher clock, lower CPI
  - Itanium had same CPI, lower clock

Reorder Buffer

- Reorder buffer – holds the result of instruction between completion and commit
- Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number
  - Value field: output value
  - Ready field: completed execution?
- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit

问题？

Fallacies and Pitfalls

- It is easy to predict the performance/energy efficiency of two different versions of the same ISA if we hold the technology constant