Advanced Computer Architecture

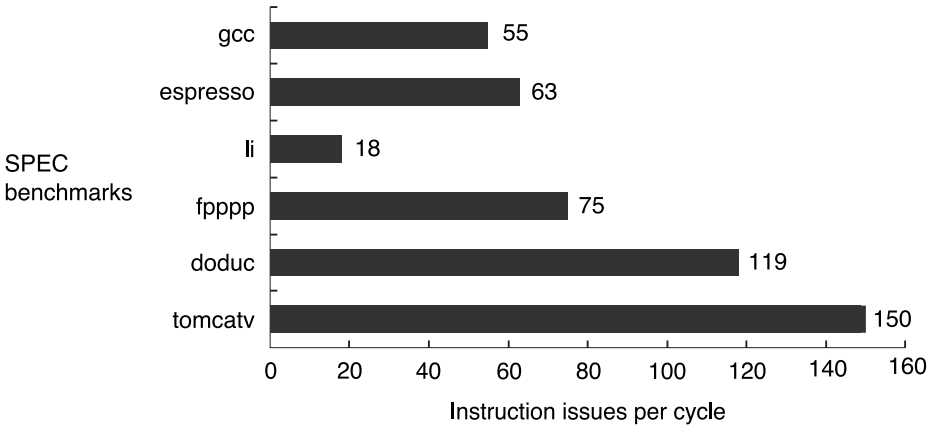
Limits to ILP

Lecture 3

Factors limiting ILP

- Rename registers
  - Reservation stations, reorder buffer, rename file
- Branch prediction
- Jump prediction
  - Harder than branches
- Memory aliasing
- Window size
- Issue width

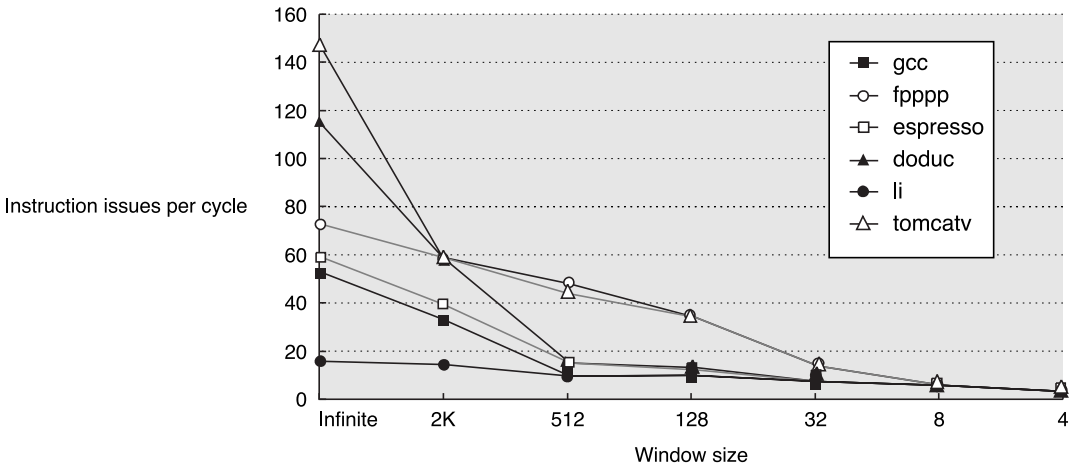
ILP in a “perfect” processor



Window size limits

- How far ahead can you look?
- 50 instruction window
  - 2,450 register comparisons looking for RAW, WAR, WAW
  - Assuming only register-register operations
- 2,000 instruction window
  - ~4M comparisons
- Commercial machines: window size up to 128

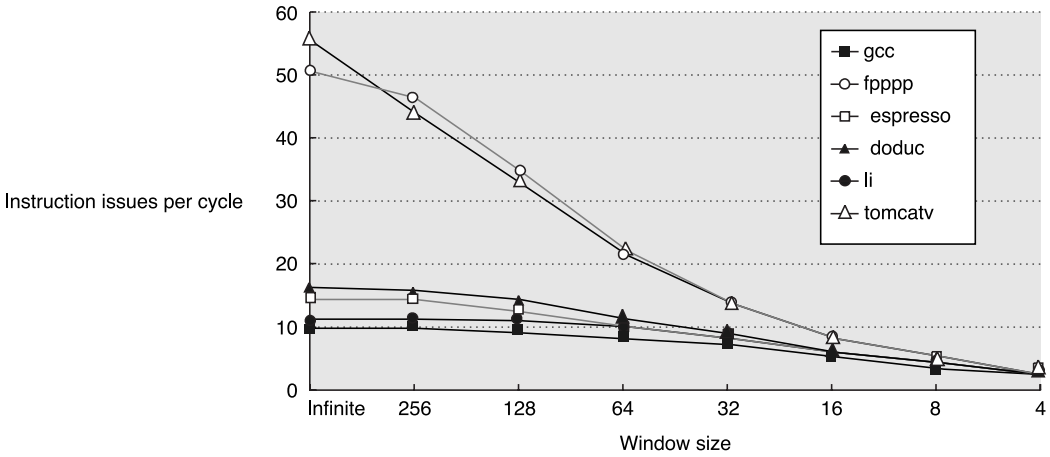
Window size effects



Assuming all instructions take 1 cycle,

Assuming perfect branch prediction

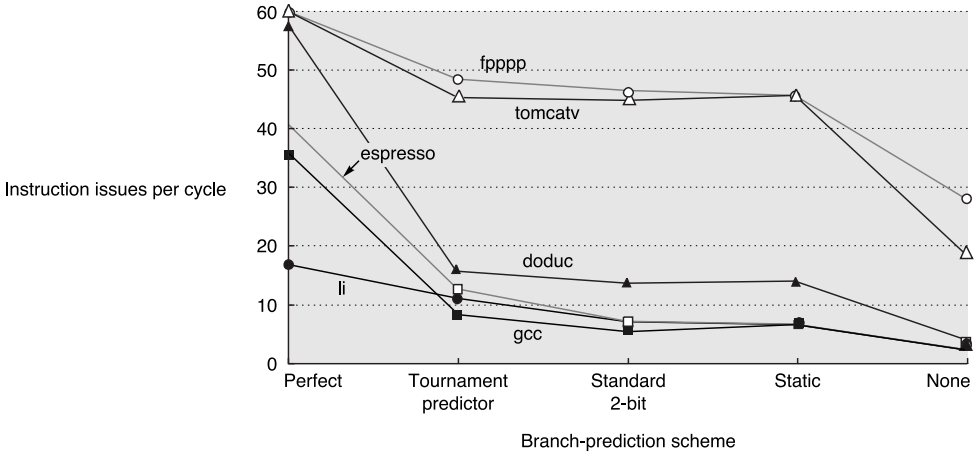
Window size effects, limited issue



Limited to 64 issues per clock

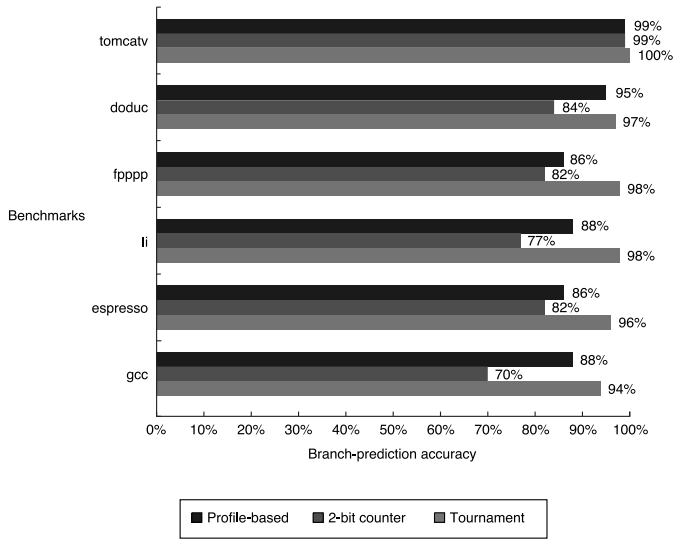
(note: current limits closer to 6

Branch prediction effects



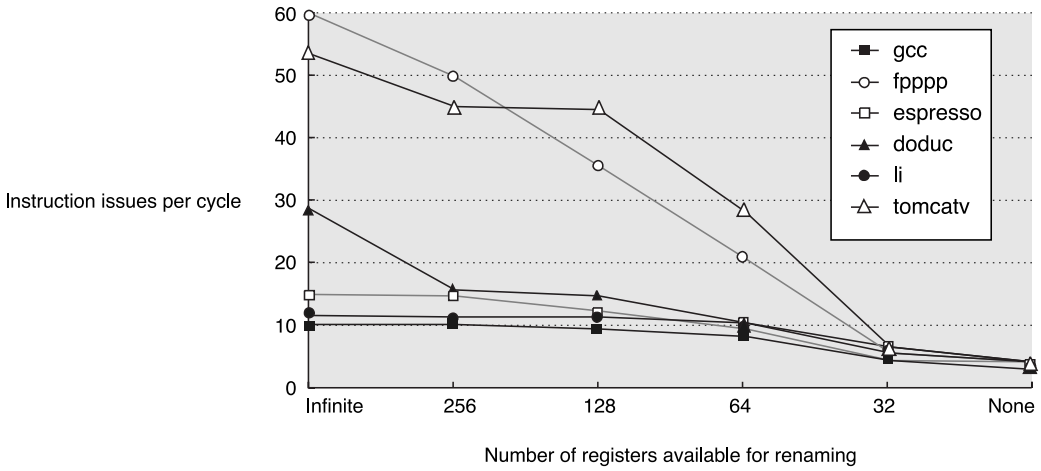
Assuming 2K instruction window

How much misprediction affects ILP?



Not much

Register limitations

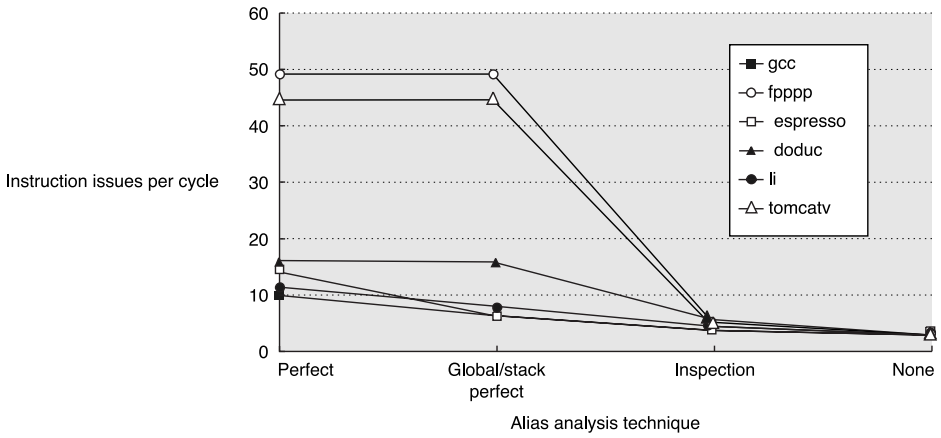


Current processors near 256 – not much left!

Memory disambiguation

- Memory aliasing
  - Like RAW, WAW, WAR, but for loads and stores
  - Even compiler can't always know
    - Indirection, base address changes, pointers
- How to avoid?
  - Don't: do everything in order
  - Speculate: fix up later
  - Value prediction

Alias analysis



Current compilers somewhere between Inspection and Global/stack perfect

Instruction issue complexity

- "Classic" RISC instructions
  - Only 32-bit instructions
- Current RISC
  - Most have 16-bit as well as 32-bit forms
- x86
  - 1 to 17 bytes per instruction

Tradeoff

- More instruction issue
  - More gates in decode – lower clock rate
  - More pipe stages in decode – more branch penalty
- More instruction execution
  - More register ports – lower clock rate
  - More comparisons – lower clock rate or more pipe stages
  - Chip area grows faster than performance?

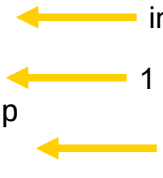
Help from the compiler

- Standard FP loop:

```
Loop:  L.D    F0,0(R1)      <-- 1 cycle load stall
       ADD.D  F4,F0,F2      <-- 2 cycle execute stall
       S.D    F4,0(R1)
       DADDUI R1,R1,#-8
       BNE    R1,R2,Loop    <-- 1 cycle branch stall
```

Same code, reordered

```
Loop:  L.D    F0,0(R1)
       DADDUI R1,R1,#-8
       ADD.D  F4,F0,F2
       BNE    R1,R2,Loop
       S.D    F4,0(R1)
```



Saves 3 cycles per loop

Loop Unrolling

- In previous loop
  - 3 instructions do “work”
  - 2 instructions “overhead” (loop control)
- Can we minimize overhead?
  - Do more “work” per loop
  - Repeat loop body multiple times for each counter/branch operation

Unrolled Loop

```
Loop:  L.D    F0,0(R1)
       ADD.D  F4,F0,F2
       S.D    F4,0(R1)
       L.D    F0,-8(R1)
       ADD.D  F4,F0,F2
       S.D    F4,-8(R1)
       L.D    F0,-16(R1)
       ADD.D  F4,F0,F2
       S.D    F4,-16(R1)
       L.D    F0,-24(R1)
       ADD.D  F4,F0,F2
       S.D    F4,-24(R1)
       DADDUI R1,R1,#-32
       BNE    R1,R2,Loop
```


Tradeoffs

- Fewer ADD/BR instructions
  - 1/n for n unrolls
- Code expansion
  - Nearly n times for n unrolls
- Prologue
  - Most loops aren't 0 mod n iterations
  - Need a loop start (or end) of single iterations from 0 to n-1

Unrolled loop, dependencies minimized

```
Loop:  L.D    F0,0(R1)
       ADD.D  F4,F0,F2
       S.D    F4,0(R1)
       L.D    F6,-8(R1)
       ADD.D  F8,F6,F2
       S.D    F8,-8(R1)
       L.D    F10,-16(R1)
       ADD.D  F12,F10,F2
       S.D    F12,-16(R1)
       L.D    F14,-24(R1)
       ADD.D  F16,F14,F2
       S.D    F16,-24(R1)
       DADDUI R1,R1,#-32
       BNE    R1,R2,Loop
```

This is what register renaming does in hardware

Unrolled loop, reordered

```
Loop:  L.D    F0,0(R1)
       L.D    F6,-8(R1)
       L.D    F10,-16(R1)
       L.D    F14,-24(R1)
       ADD.D  F4,F0,F2
       ADD.D  F8,F6,F2
       ADD.D  F12,F10,F2
       ADD.D  F16,F14,F2
       S.D    F4,0(R1)
       S.D    F8,-8(R1)
       DADDUI R1,R1,#-32
       S.D    F12,16(R1)
       BNE    R1,R2,Loop
       S.D    F16,8(R1)
```

This is what dynamic scheduling does in hardware

Limitations on software rescheduling

- Register pressure
  - Run out of architectural registers
  - Why Itanium has 128 registers
- Data path pressure
  - How many parallel loads?
- Interaction with issue hardware
  - Can hardware see far enough ahead to notice?
  - One reason there are different compilers for each processor implementation
- Compiler sees dependencies
  - Hardware sees hazards

Superscalar Issue

	Integer pipeline		FP pipeline	
Loop:	L.D	F0,0(R1)		
	L.D	F6,-8(R1)		
	L.D	F10,-16(R1)	ADD.D	F4,F0,F2
	L.D	F14,-24(R1)	ADD.D	F8,F6,F2
	L.D	F18,-32(R1)	ADD.D	F12,F10,F2
	S.D	F4,0(R1)	ADD.D	F16,F14,F2
	S.D	F8,-8(R1)	ADD.D	F20,F18,F2
	S.D	F12,-16(R1)		
	DADDUI	R1,R1,# -40		
	S.D	F16,16(R1)		
	BNE	R1,R2,Loop		
	S.D	F20,8(R1)		

Can the compiler tell the hardware to issue in parallel?

Limit on loop unrolling

- Keeping the pipeline filled
  - Each iteration is load-execute-store
  - Pipeline has to wait – at least at start of loop
- Load delays
- Dependencies
- Code explosion
  - Cache effects

Loop dependencies

```
for(i=0;i<1000;i++) {
      A[i] = B[i];
}
```

← Embarrassingly parallel

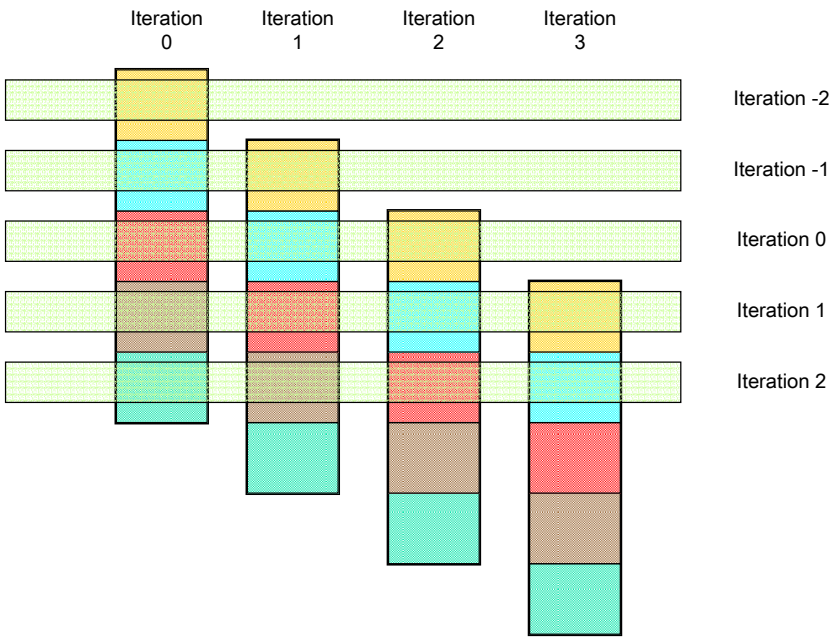
```
for(i=0;i<1000;i++) {
      A[i] = B[i] + x;
      B[i+1] = A[i] + B[i];
}
```

← Iteration i depends on i-1

Software pipelining

- Separate program loop from data loop
  - Each iteration has different phases of execution
  - Each iteration of original loop occurs in 2 or more iterations of pipelined loop
- Analogous to Tomasulo
  - Reservation stations hold different iterations of loop

Software pipelining



Compiling for Software Pipeline

Original

```
for(i=0; i<1000; i++ ) {
        x = load(a[i]);
        y = calculate(x);
        b[i] = y;
}
```

Pipelined

```
x = load(a[0]);
y = calculate(x);
x = load(a[0]);
for(i=1; i<1000; i++ ) {
        b[i-1] = y;
        y = calculate(x);
        x = load(a[i+1]);
}
```


Unrolling vs. Software pipelining

- Loop unrolling minimizes branch overhead
- Software pipelining minimizes dependencies
  - Especially useful for long latency loads/stores
  - No idle cycles within iterations
- Can combine unrolling with software pipelining
- Both techniques are standard practice in high performance compilers

Static Branch Prediction

- Some of the benefits of dynamic techniques
- Delayed branch
- Compiler guesses
  - `for(i=0;i<10000;i++)` is pretty easy
- Compiler hints
  - Some instruction sets include hints
  - Especially useful for trace-based optimization

More branch troubles

- Even unrolled, loops still have some branches
  - Taken branches can break pipeline – nonsequential fetch
  - Unpredicted branches can break pipeline
- Especially annoying for short branches

```
a = f();
b = g();
if( a > 0 )
        x = a;
else
        x = b;
```

Predication

- Add a predicate to an instruction
  - Instruction only executes if predicate is true
  - Several approaches
    - flags, registers
    - On every instruction, or just a few (e.g., load predicated)
  - Used in several architectures
    - HP Precision, ARM, IA-64

```
LD      R2,b[]
LD      R1,a[]
ifGT    MOV     R4,R1
ifLE    MOV     R4,R2
…
```


Cost/Benefit of Predication

- Enables longer basic blocks
  - Much easier for compilers to find parallelism
- Can eliminate instructions
  - Branch folded into instruction
- Instruction space pressure
  - How to indicate predication?
- Register pressure
  - May need extra registers for predicates
- Stalls can still occur
  - Still have to evaluate predicate and forward to predicated instruction