Cornell University

ECE4750/CS4420 Computer Architecture L1~~3~~4: Exploit ILP in Software

Edward Suh
Computer Systems Laboratory
suh@csl.cornell.edu

Overview

- Background
  - Very Long Instruction Word (VLIW) processors require compiler to handle hazards
  - Even superscalar processors can benefit from more parallelism
- How many parallel instructions do we need to keep a processor busy?
- Compiler techniques to extract parallelism
  - Loop unrolling
  - Software pipelining
  - Trace scheduling / predicates

Reading Material: Hennessy & Patterson Ch 2.2-2.7

Little's Law – How much parallelism do we need?

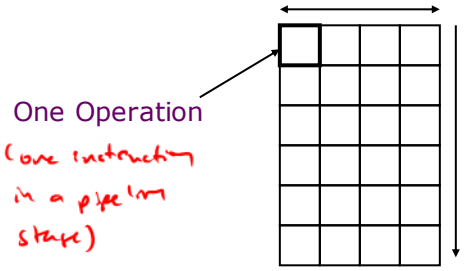
Parallelism = Throughput × Latency

$\left[\frac{inst}{cycle}\right] \times [cycles] = [inst]$

to fill pipeline, we need 24 instructions in this case

instructions issued per cycle

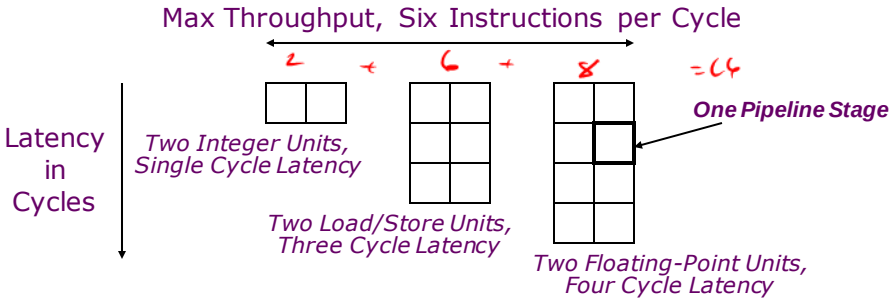
in this case, 4



cycles to finish an instruction

in this case, 6

Example (more realistic)



- How much instruction-level parallelism (ILP) required to keep machine pipelines busy? at least parallelism of 16, more if we have some kind of dependency

```
for (i=N-1; i>=0; i--) {
  a[i] = a[i]+c;
}
```

Basic Pipeline Scheduling

9 cycles/element

- Goal
  - find independent instructions that can be overlapped in the pipeline
  - separate dependent instructions by distance in clock cycles equal to the latency of the source instruction

plain

```
Loop: fld   f0,0($1)
      fadd  f4,f0,f2
      fst   f4,0($1)
      addi  $1,$1,-8
      bne   $1,$2,Loop
      nop
```

stall x1
stall x2

- Assumptions
  - basic five-stage pipeline
  - one-cycle delayed branches
  - fully pipelined/replicated Fus
    - LD/ST/Branch/Int ALU (1 cycle)
    - FPU (3 cycles)

```
fld   f0, 0($1)
addi  $1, $1, -8
fadd  f4, f0, f2
bne   $1, $2, Loop
fst   f4, 0($1)
```

still stall

need more independent instructions to get rid of stalls.

Loop Unrolling

```
for (i=N-1; i>=0; i-=2) {
  a[i] = a[i] + c
  a[i-1] = a[i-1] + c
}
```

- Replicate loop body several times
- Use different regs in each iteration
- Append loop termination code
  - if want to unroll k times
    - loop #1 (original) executes n%k times
    - loop #2 (unrolled) executes n/k times
- Advantages
  - more ILP
  - less overhead instructions
- Disadvantages
  - more complex compiler (miss
  - code size (↑ cache miss)
  - register pressure

```
loop: fld  f0, 0($1)
      fld  f6, -8($1)
      fadd f4, f0, f2
      fadd f8, f6, f2
      fst  f4, 0($1)
      fst  f8, -8($1)
      addi $1, $1, -16
      bne  0, $2, loop
```

Loop Unrolling - VLIW

fld - 2 cycles
fadd - 3 cycles

Unroll 4 times

```
loop:  fld f0, 0($1)
       fld f6, -8($1)
       fld f10, -16($1)
       fld f14, -24($1)
       fadd f4, f0, f2
       fadd f8, f0, f2
       fadd f12, f0, f2
       fadd f16, f0, f2
       fst f4, -0($1)
       fst f8, -8($1)
       fst f12, -16($1)
       fst f16, -24($1)
       addi $1, $1, -32
       bne $1, $2, loop
```

Schedule →

	Int1	Int 2	M1	M2	FP+	FPx
loop:			fld f0			
			fld f6			
			fld f10		fadd f4	
			fld f14		fadd f8	
		fld →			fadd f12	
		fld →	fst f4 →		fadd f16	
		fld →	fst f8 →		fadd	
	addi		fst f12 →		fadd	
	bne		fst f16 →		fadd	
				fst		
				fst		
				fst		

How many FLOPS/cycle?

floating point ops

↳ 4x fadd ↳ 9 cycles

4/9 = 2.25/element

what if we unroll 7 times?
assume infinite registers
and no cache miss

7 / 12
↑ flops

Software Pipelining

prologue

- Interleave operations from multiple iterations

iter 1 iter 2 iter 3 iter 4

fld
fadd fld } need to start w/ these

fst fadd fld
fst fadd fld
fst fadd
fst

finish w/ these → epilogue

fld f0, 0($1) } iter 0
fadd f4, f0, f2
fld f0, -8($1) } iter 1

loop
→ fst f4, 16($1) ← iter i-2
fadd f4, f0, f2 ← iter i-1
fld f0, 0($1) ← iter i
addi $1, $1, 8
bne $1, $2, loop

Software Pipelining + Loop Unrolling

Unroll 4 times

```
loop:  fld f0, 0($1)
       fld f6, -8($1)
       fld f10, -16($1)
       fld f14, -24($1)
       fadd f4, f0, f2
       fadd f8, f0, f2
       fadd f12, f0, f2
       fadd f16, f0, f2
       fst f4, -0($1)
       fst f8, -8($1)
       fst f12, -16($1)
       fst f16, -24($1)
       addi $1, $1, -32
       bne $1, $2, loop
```

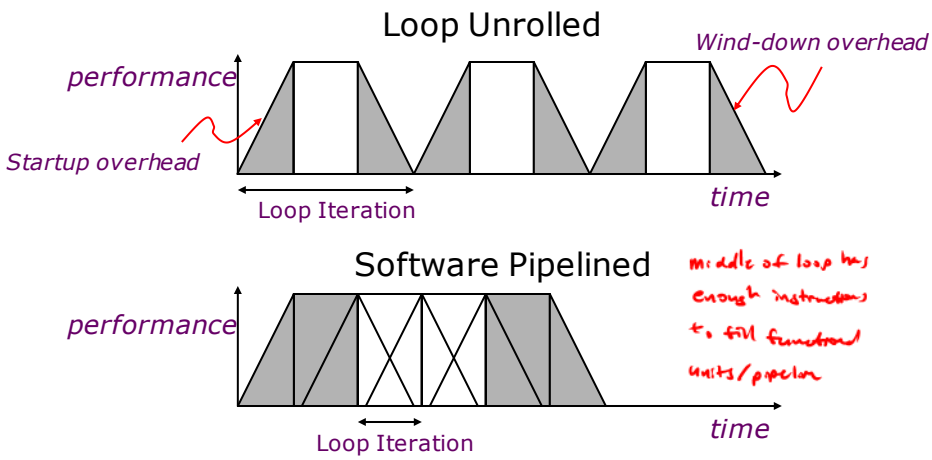
prologue ~~prolog~~

iterate

~~epilog~~ epilogue

	Int1	Int 2	M1	M2	FP+	FPx
			fld f0			
			fld f6			
			fld f10			
	add $1		fld f14			
			fld f0		fadd f4	
			fld f6		fadd f8	
			fld f10		fadd f12	
	add $1		fld f14		fadd f16	
loop:			fld f0	fst f4	fadd f4	
			fld f6	fst f8	fadd f8	
			fld f10	fst f12	fadd f12	
	add $1	bne	fld f14	fst f16	fadd f16	
				fst f4	fadd f4	
				fst f8	fadd f8	
				fst f12	fadd f12	
				fst f16	fadd f16	
				sd f4		

Software Pipelining vs. Loop Unrolling



middle of loop has enough instructions to fill functional units/pipelines

Dependence Analysis: Loop Example

```
for (j=0; j <=100; j++) {
     A[j] = B[j] * C[j];
     D[j] = A[j] * E[j];
}
```

no dependancy

A[j+1] = B[j+1] * C[j+1];
D[j+1] = A[j+1] * E[j+1];

Is the above loop parallel?

yes. No loop-carried dependancy

Another Loop Example

```
for (j=0; j <=100; j++) {
     A[j+1] = A[j] * C[j];
     B[j+1] = B[j] * A[j+1];
}
```

dependancy! :(

A[j+2] = A[j+1] * C[j+1]
B[j+2] = B[j+1] * A[j+2]

Uh-oh, loop-carried dependence! (where?)

Cannot run loop in parallel.

Yet Another Loop Example

```
for (j=0; j <=100; j++) {
      A[j] = A[j] + B[j];          /*S1*/
      B[j+1] = C[j] + D[j];        /*S2*/
}
```

A[0] = A[0] + B[0] j=0
B[1] = C[0] + D[0]
A[1] = A[1] + B[1] j=1
B[2] = C[1] + D[1]

What are the dependences between S1 and S2? Is the loop parallel? If so, how?

dependency is between S1 and S2

across iterations. we can parallelize?

A[0] = A[0] + B[0]
for(j=0; j<=99; j++){
  B[j+1] = C[j] + D[j]
  A[j+1] = A[j+1] + B[j+1]
}
B[101] = C[100] + D[100]

Recurrence

How about:

```
for (j=0; j <=100; j++) {
      A[j] = A[j] * A[j-5];        /* loop-carried? */
}
```

Dependence Analysis

In general how can a compiler determine if a loop is parallel?

- need to detect if there are any dependences

Nearly all algorithms assume array indices are *affine*

- array index is affine if it can be written in the form $a*i+b$
- a and b are constants and i is the loop index variable
- for multi-dimensional arrays, each index must be affine

Deciding whether there is a dependence between two accesses is same as deciding whether two affine functions can have the same value for different indices within the bounds of the loop

Dependence Analysis Algorithm

Assume we store an element with an index value $a*i + b$ and load an element with index $c*i + d$ where i runs from m to n

A dependence exists if both of the following hold

- there are 2 iterations j and k such that $m <= j,k <= n$
- there is a store to an element with index $a*j+b$ and a later load from the *same* array element $c*k+d$.
  - i.e. $a*j + b = c*k + d$

e.x. array[aj+b] = array[ck+d]+1

Unfortunately, in general cannot determine at compile time

- a,b,c,d may not be known (could be in other arrays!)
- might be known at compile time but expensive to compute
  - accesses depend on indices of multiple nested loops…

GCD Test

Luckily, many programs use indices known at compile time

a,b,c,d are often constants

One simple and efficient test is the GCD test

GCD is the *greatest common divisor* (remember 4th grade?)

Observation: if a loop-carried dependence exists then:

$aj + b = ck + d \Rightarrow aj - ck = d - b$

$g = GCD(a,c)$

$\frac{a}{g} j - \frac{c}{g} k = \frac{d-b}{g}$

if g doesn't divide $d-b$, we have no loop-carried dependency and thus can parallelize.

Example

Is the following loop parallel?

```
for (j=0; j <=100; j++) {
     A[2*j+3] = A[2*j] * 5.0;
}
```

a b c d = 0

First, are the indices affine?

what are values of *a,b,c,d*?

what is GCD(*a,c*)? (*d-b*)?

does GCD(*a,c*) divide *d-b*?

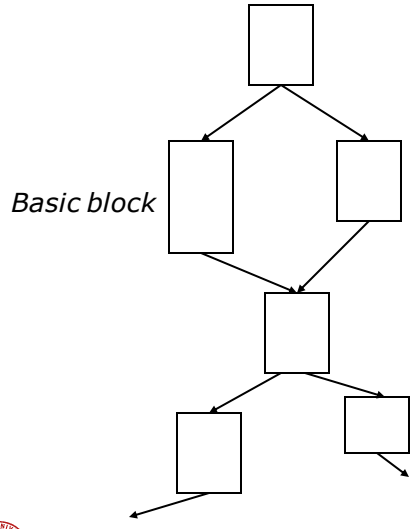
$g = GCD(2,2) = 2$

$(d-b) = -3$

$\frac{-3}{2} \notin \mathbb{Z}$

$\Rightarrow$ parallelizeable

What if there is no loop?



Static Branch Prediction

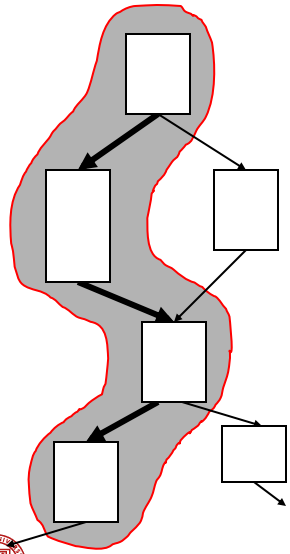
- Used when behavior highly predictable at compile time

assume no delay slot

```
        ld      $1,0($2)
        stall
        sub     $1,$1,$3
        beq     $1,$0,L
        or      $4,$5,$6
        add     $10,$11,$3
        ...
L:      add     $7,$8,$9
```

How
- forward: not taken
- backward: taken
- profiling

Trace Scheduling



pick a sequence of basic blocks, a trace, which is the most frequent path

Conditional (Predicated) Instructions

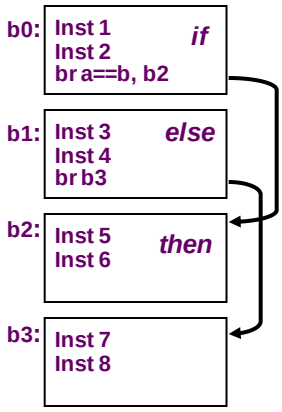
- Convert control dependence into data dependence

```
if (A == 0) { S = T; }
```

- Using a branch

- Using a conditional move

IA-64 Predicated Execution



Four basic blocks

Inst1
Inst2
(p1, p2) ← cmp(a==b)
(p1) inst3 || (p2) inst5
(p1) inst4 || (p2) inst6
inst7
inst8