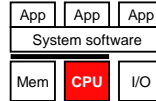


# CSE 560 Computer Systems Architecture

## Static Scheduling

1

## This Unit: Static Scheduling



- Code scheduling to
  - Reduce pipeline stalls
  - Increase ILP
- Two approaches to scheduling
  - **This Unit:**
    - **Static scheduling by the compiler**
  - Coming Soon:
    - Dynamic scheduling by the hardware

4

4

## Code Scheduling

- Scheduling: act of finding independent instructions
  - **Static:** at compile time by the compiler (software)
  - **Dynamic:** at runtime by the processor (hardware)
- Why schedule code?
  - **Scalar pipelines:** fill load-to-use delays to improve CPI
  - **Superscalar:** place independent instructions together
    - As above, load-to-use delay slots
    - Allow multiple-issue decode logic to let them execute at the same time

5

5

## Scheduling Requirements

- **Independent insns**
  - no ILP → game over
- **Large Scheduling Scope**
  - Scope = code region we are scheduling
  - The bigger the better (more independent insns to play with)
  - Once scope is defined, schedule is pretty obvious
  - Trick is creating a large scope (schedule across branches?)
- **Enough registers**
  - To hold additional "live" values
- **Alias analysis**
  - Whether load/store reference same memory locations
    - Can they be legally rearranged?

6

6

## Scheduling Techniques

- **Stall Removal**
  - Separate load-use pairs
- **Scope enlarging**
  - For Loops: **loop unrolling**
  - For Non-loops:
    - **Superblocks**
    - **Predication**
- **Exploit Data-Level Parallelism**
  - Vectors

7

7

## New Metric: Utilization

- Utilization:** actual performance / peak performance
- Important metric for performance/cost
  - Why pay for hardware you rarely use?
  - Adding hardware usually ↑ performance, ↓ utilization
    - New hardware cannot always be exploited
    - Diminishing marginal returns
  - Compiler can help make better use of existing hardware
    - Important for superscalar

8

8

## Running Code Example: SAXPY

- SAXPY** (Single-precision A X Plus Y)
  - Linear algebra routine (for solving systems of equations)
  - Part of early **Livermore Loops** benchmark suite
  - floating point uses "F" registers and "F" instructions

```
for (i=0;i<N;i++)
    Z[i]=(A*X[i])+Y[i];
```

```
0: ldf X(r1)→f1          // loop           LOAD1
1: mulf f0,f1→f2        // A in f0       USE1
2: ldf Y(r1)→f3         // X,Y,Z constants LOAD2
3: addf f2,f3→f4        //              USE2
4: stf f4→Z(r1)
5: addi r1,4→r1         // i in r1
6: blt r1,r2,0          // N*4 in r2
```

9

## SAXPY Performance and Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20			
ldf X(r1)→f1	F	D	X	M	W																		
mulf f0,f1→f2		F	D	d*	E*	E*	E*	E*	E*	W													
ldf Y(r1)→f3			F	D	X	M	W																
addf f2,f3→f4			F	p*	d*	d*	d*	d*	D	E+	E+	W											
stf f4→Z(r1)				F	p*	p*	p*	p*	D	X	M	W											
addi r1,4→r1					F	p*	p*	p*	p*	D	X	M	W										
blt r1,r2,0										F	D	X	M	W									
ldf X(r1)→f1											F	D	X	M	W								

Scalar pipeline

- Full bypassing, 5-cycle E\*, 2-cycle E+, predict branches taken
- Single iteration (7 insns) latency:  $16-5 = 11$  cycles
- Performance:** 7 insns / 11 cycles = 0.64 IPC
- Utilization:** actual/peak IPC =  $0.64 / 1 = 64\%$

A word about stalls

- mulf stalls due to a **data dependence** on ldf x
- ldf Y stalls due to a **pipeline hazard** because mulf is occupying D

10

## SAXPY Performance and Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20			
ldf X(r1)→f1	F	D	X	M	W																		
mulf f0,f1→f2	F	D	d*	d*	E*	E*	E*	E*	E*	W													
ldf Y(r1)→f3		F	D	X	M	W																	
addf f2,f3→f4		F	p*	p*	d*	d*	d*	d*	D	E+	E+	W											
stf f4→Z(r1)			F	p*	p*	p*	p*	p*	D	X	M	W											
addi r1,4→r1				F	p*	p*	p*	p*	p*	D	X	M	W										
blt r1,r2,0					F	p*	p*	p*	p*	D	X	M	W										
ldf X(r1)→f1											F	D	X	M	W								

2-way superscalar pipeline

- Any two insns per cycle + split integer and FP pipelines
- Performance:** 7 insns / 10 cycles = 0.70 IPC
- Utilization:** actual/peak IPC =  $0.70 / 2 = 35\%$
- More hazards → more stalls
- Each stall is more expensive

11

## Eliminate Load-Use Pairs?

```
for (i=0;i<N;i++)
    Z[i]=(A*X[i])+Y[i];
```

```
0: ldf X(r1)→f1          LOAD1
1: mulf f0,f1→f2        USE1
2: ldf Y(r1)→f3         LOAD2
3: addf f2,f3→f4        USE2
4: stf f4→Z(r1)
5: addi r1,4→r1
6: blt r1,r2,0

0: ldf X(r1)→f1          LOAD1
1: mulf f0,f1→f2        USE1
2: ldf Y(r1)→f3         LOAD2
3: addf f2,f3→f4        USE2
4: stf f4→Z(r1)
5: addi r1,4→r1
6: blt r1,r2,0
```

Problem solved?

12

## Loop Unrolling SAXPY

- Goal:** separate dependent insns from one another
- SAXPY problem: not enough flexibility within one iteration
  - Longest chain of insns is 9 cycles
    - Load (1)
    - Forward to multiply (5)
    - Forward to add (2)
    - Forward to store (1)
  - Can't hide a 9-cycle chain using only 7 insns
  - But how about two 9-cycle chains using 14 insns?
- Loop unrolling:** schedule 2+ iterations together
  - Fuse iterations
  - Schedule to reduce stalls
  - Schedule introduces ordering problems → rename registers

13

## Unrolling SAXPY I: Fuse Iterations

- Combine two (in general K) iterations of loop
  - Fuse loop control: induction variable ( $i=r1$ ) increment + branch
  - Adjust (implicit) induction uses: constants → constants + 4

```
ldf X(r1),f1
mulf f0,f1,f2
ldf Y(r1),f3
addf f2,f3,f4
stf f4,Z(r1)
addi r1,4,r1
blt r1,r2,0
-- increment i
-- jump back

ldf X(r1),f1
mulf f0,f1,f2
ldf Y(r1),f3
addf f2,f3,f4
stf f4,Z(r1)

ldf X+4(r1),f1
mulf f0,f1,f2
ldf Y+4(r1),f3
addf f2,f3,f4
stf f4,Z+4(r1)
addi r1,8,r1
blt r1,r2,0
-- increment i
-- jump back
```

14

## Unrolling SAXPY II: Pipeline Schedule

- Pipeline schedule to reduce stalls
  - Have already seen this: pipeline scheduling

```

ldf X(r1), f1
mulf f0, f1, f2
ldf Y(r1), f3
addf f2, f3, f4
stf f4, Z(r1)
ldf X+4(r1), f1
mulf f0, f1, f2
ldf Y+4(r1), f3
addf f2, f3, f4
stf f4, Z+4(r1)
addi r1, 8, r1
blt r1, r2, 0
    
```



```

ldf X(r1), f1
ldf X+4(r1), f1
mulf f0, f1, f2
mulf f0, f1, f2
ldf Y(r1), f3
ldf Y(r1), f3
ldf Y+4(r1), f3
ldf Y+4(r1), f3
addf f2, f3, f4
addf f2, f3, f4
addf f2, f3, f4
addf f2, f3, f4
stf f4, Z(r1)
stf f4, Z+4(r1)
stf f4, Z+4(r1)
stf f4, Z+4(r1)
addi r1, 8, r1
addi r1, 8, r1
blt r1, r2, 0
    
```

15

15

## Unrolling SAXPY III: "Rename" Registers

- Pipeline scheduling causes reordering violations
  - Rename registers to correct

```

ldf X(r1), f1
ldf X+4(r1), f1
mulf f0, f1, f2
mulf f0, f1, f2
ldf Y(r1), f3
ldf Y+4(r1), f3
addf f2, f3, f4
addf f2, f3, f4
stf f4, Z(r1)
stf f4, Z+4(r1)
addi r1, 8, r1
blt r1, r2, 0
    
```



```

ldf X(r1), f1
ldf X+4(r1), f5
mulf f0, f1, f2
mulf f0, f1, f6
ldf Y(r1), f3
ldf Y+4(r1), f7
addf f2, f3, f4
addf f2, f3, f4
stf f4, Z(r1)
stf f4, Z+4(r1)
addi r1, 8, r1
blt r1, r2, 0
    
```

Do we have enough registers to do this?

Are we sure we can move these loads above these stores?

Alias analysis must be conservative.

16

16

## Unrolled SAXPY Performance/Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
ldf X(r1) → f1	F	D	X	M	W															
ldf X+4(r1) → f5		F	D	X	M	W														
mulf f0, f1 → f2			F	D	E*	E*	E*	E*	E*	W										
mulf f0, f5 → f6				F	D	E*	E*	E*	E*	E*	W									
ldf Y(r1) → f3					F	D	X	M	W											
ldf Y+4(r1) → f7						F	D	X	M	W	s*	s*	W							
addf f2, f3 → f4							F	D	d*	E+	E+	s*	W							
addf f6, f7 → f8								F	p*	D	E+	p*	E+	W						
stf f4 → Z(r1)										F	D	X	M	W						
stf f8 → Z+4(r1)											F	D	X	M	W					
addi r1 → 8, r1												F	D	X	M	W				
blt r1, r2, 0													F	D	X	M	W			
ldf X(r1) → f1														F	D	X	M	W		

- + Performance: 12 insns / 13 cycles = 0.92 IPC
- + Utilization: actual/peak IPC = 0.92 / 1 = 92%
- + **Speedup**: (2 \* 11 cycles) / 13 cycles = 1.69
- ? But improvement in IPC is only 0.92/0.64 = 1.43, what gives?

17

17

## Loop Unrolling Shortcomings

- Static code growth → more I\$ misses (limits unrolling)
- Needs more registers to hold values (ISA limits this)
- Doesn't handle: non-loops, inter-iteration dependences

```

for (i=0; i<N; i++)
    X[i]=A*X[i-1];
    
```

```

ldf X-4(r1), f1
mulf f0, f1, f2
stf f2, X(r1)
addi r1, 4, r1
blt r1, r2, 0
ldf X-4(r1), f1
mulf f0, f1, f2
stf f2, X(r1)
addi r1, 4, r1
blt r1, r2, 0
    
```



```

ldf X-4(r1), f1
mulf f0, f1, f2
stf f2, X(r1)
mulf f0, f2, f3
stf f3, X+4(r1)
addi r1, 4, r1
blt r1, r2, 0
    
```

- Two mulf's are not parallel
- Other (more advanced) techniques help

18

18

## Summary: Static Scheduling Limitations

- Limited number of registers (set by ISA)
- Scheduling scope
  - Example: hard to move memory insns past branches
- Inexact memory aliasing information
  - Often prevents reordering of loads above stores
- Caches misses (or any runtime event) confound scheduling
  - How can the compiler know which loads will miss/hit?
  - Can impact the compiler's scheduling decisions

19

19

## Scheduling Techniques

- **Stall Removal**
  - Separate load-use pairs
- **Scope enlarging**
  - For Loops: **loop unrolling**
  - **For Non-loops:**
    - **Superblocks** (biased branches)
    - **Predication** (non-biased branches)
- **Exploit Data-Level Parallelism**
  - Vectors

20

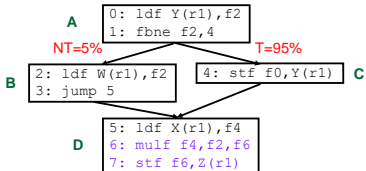
20

## Superblocks

### Source code

```
A = Y[i];
if (A == 0)
  A = W[i];
else
  Y[i] = 0;
Z[i] = A*X[i];
```

### 4 basic blocks: A,B,C,D



### Machine code

```
0: ldf Y(r1),F2
1: fbne F2,4
2: ldf W(r1),F2
3: jump 5
4: stf f0,Y(r1)
5: ldf X(r1),F4
6: mulf F4,F2,F6
7: stf F6,Z(r1)
```

- Use when branch is highly biased
- Fuse blocks of most frequent path: ACD
- Schedule
- Create **repair code** in case path = ABD

21

21

## Superblock and Repair Code

### Superblock

```
0: ldf Y(r1),F2
1: fbeq F2,2
4: stf f0,Y(r1)
5: ldf X(r1),F4
6: mulf F4,F2,F6
7: stf F6,Z(r1)
```

### Repair code

```
2: ldf W(r1),F2
5': ldf X(r1),F4
6': mulf F4,F2,F6
7': stf F6,Z(r1)
```

- What did we do?
  - Change sense (test) of branch 1
    - Original taken target now fall-thru
  - Created repair block
    - May need to duplicate some code (here basic-block D)
  - Haven't actually scheduled superblock yet

22

22

## Superblocks Scheduling I

### Superblock

```
0: ldf Y(r1),F2
1: fbeq F2,2
5: ldf X(r1),F4
6: mulf F4,F2,F6
4: stf f0,Y(r1)
7: stf F6,Z(r1)
```

### Repair code

```
2: ldf W(r1),F2
5': ldf X(r1),F4
6': mulf F4,F2,F6
7': stf F6,Z(r1)
```

- First scheduling move: move insns 5 and 6 above insn 4
  - Hmmm: moved load (5) above store (4)
  - We can tell this is OK, but can the compiler
    - If yes, fine
    - Otherwise, need to do something

23

23

## Predication

- Conventional control
  - Conditionally executed insns also conditionally fetched
- Predication**
  - Conditionally executed insns unconditionally fetched
  - Full predication** (ARM, IA-64)
    - Tag every insn with predicate, costs extra bits
  - Conditional moves** (Alpha, IA-32)
    - Construct appearance of full predication from one primitive
 

```
cmovreq r1,r2,r3 // if (r1==0) r3←r2;
```

      - May require some code duplication to achieve desired effect
      - Only good way of adding predication to an existing ISA
  - If-conversion**: replacing control with predication
    - Good if branch is unpredictable (save mis-prediction)
    - But more instructions fetched and "executed"

24

24

## Avoiding Branches via ISA: Predication

- Conventional control
  - Conditionally executed insns also conditionally fetched

	1	2	3	4	5	6	7	8	9
beq r3,targ	F	D	X	M	W				
sub r6,1,r5		F	D	--	--	--	flushed: wrong path		
targ:add r4,r5,r4			F	--	--	--	flushed: why?		
targ:add r4,r5,r4				F	D	X	M	W	

- If **beq** mis-predicts, both **sub** and **add** must be flushed
  - Waste: **add** is independent of mis-prediction
- Predication**: not prediction, predication
  - ISA support for conditionally-executed unconditionally-fetched insns
  - If **beq** mis-predicts, annul **sub** in place, preserve **add**
    - Example is if-then, but if-then-else can be predicated too
  - How is this done? How does **add** get correct value for **r5**

25

25

## Full Predication

### Full predication

- Every insn can be annulled, annulment controlled by...
- Predicate registers: additional register in each insn (e.g., IA64)

	1	2	3	4	5	6	7	8	9
setp.eq r3,p3	F	D	X	M	W				
sub.p r6,1,r5,p3		F	D	X	--	--	annulled		
targ:add r4,r5,r4			F	D	X	M	W		

- Predicate codes: condition bits in each insn (e.g., ARM)

	1	2	3	4	5	6	7	8	9
setcc r3	F	D	X	M	W				
sub.nz r6,1,r5		F	D	X	--	--	annulled		
targ:add r4,r5,r4			F	D	X	M	W		

- Only ALU insn shown (**sub**), but this applies to all insns, even stores
- Branches replaced with "set-predicate" insns

26

26

## Conditional Register Moves (CMOVs)

### Conditional (register) moves

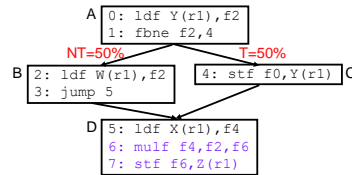
- Construct appearance of full predication from one primitive  
`cmovneq r1, r2, r3 // if (r1==0) r3 ← r2;`
- May require some code duplication to achieve desired effect
- Painful, potentially impossible for some insn sequences
- Requires more registers
- Only good way of retro-fitting predication onto ISA (e.g., IA32, Alpha)

	1	2	3	4	5	6	7	8	9
<code>sub r6,1,r9</code>	F	D	X	M	W				
<code>cmovne r3,r9,r5</code>		F	D	X	M	W			
<code>targ:add r4,r5,r4</code>			F	D	X	M	W		

27

27

## Non-Biased Branches: Use Predication



Using Predication

```

0: ldf Y(r1), f2
1: fbpne f2, p1
2: ldf.p p1, W(r1), f2
4: stf.np p1, f0, Y(r1)
5: ldf X(r1), f4
6: mulf f4, f2, f6
7: stf f6, Z(r1)
  
```

28

28

## ISA Support for Predication

```

0: ldf Y(r1), f2
1: fbpne f2, p1
2: ldf.p p1, W(r1), f2
4: stf.np p1, f0, Y(r1)
5: ldf X(r1), f4
6: mulf f4, f2, f6
7: stf f6, Z(r1)
  
```

- IA-64: change branch 1 to **set-predicate insn fbpne**
- Change insns 2 and 4 to **predicated insns**
  - `ldf.p` performs `ldf` if predicate `p1` is true
  - `stf.np` performs `stf` if predicate `p1` is false

29

29

## Predication Performance

- Cost/benefit analysis
  - Benefit:** predication avoids branches
    - Thus avoiding mis-predictions
    - Also reduces pressure on predictor table (few branches to track)
  - Cost:** extra (annulled) instructions
- Since branch predictors are highly accurate...
  - Might not help:
    - 5-stage pipeline, two instruction on each path of if-then-else
    - No performance gain, likely slower if branch predictable
  - Or even hurt!
  - But can help:
    - Deeper pipelines, hard-to-predict branches, and few added insns
- Predication is useful, but not a panacea

30

30

## Aside: Profiling

### How do we know whether a branch is biased or not?

- Profile:** statistical information about program tendencies
- Collect from previous program runs (different inputs)
- ± Works OK depending on information
  - Memory latencies (cache misses)
    - + Which loads miss frequently independent of inputs?
    - Depends on cache configuration
  - Memory dependences
    - Which loads & stores communicate with each other?
    - + Stable across inputs
  - Branch outcomes
    - Which branches are usually taken/not-taken?
    - Not so stable across inputs
- Popular research topic

31

31

## Scheduling Techniques

- Stall Removal**
  - Separate load-use pairs
- Scope enlarging**
  - For Loops: **loop unrolling**
  - For Non-loops:**
    - Superblocks** (biased branches)
    - Predication** (non-biased branches)
- Exploit Data-Level Parallelism**
  - Vectors

32

32

## Data-Level Parallelism

### Data-level parallelism (DLP)

- Single operation repeated on multiple data elements
  - SIMD (Single-Instruction, Multiple-Data)
- Less general than ILP: parallel insns are all same operation
- Exploit with **vectors**

Old idea: Cray-1 supercomputer from late 1970s

- Eight 64-entry x 64-bit floating point "Vector registers"
  - 4096 bits (0.5KB) in each register! 4KB vector register file
- Special vector instructions to perform vector operations
  - Load vector, store vector (wide memory operation)
  - Vector+Vector addition, subtraction, multiply, etc.
  - Vector+Constant addition, subtraction, multiply, etc.
  - In Cray-1, each instruction specifies 64 operations!

33

33

## Example Vector ISA Extensions

Extend ISA with floating point (FP) vector storage ...

- **Vector register:** fixed-size array of 32- or 64- bit FP elements
- **Vector length:** For example: 4, 8, 16, 64, ...

• ... and example operations for vector length of 4, 8-bit elements

- Load vector: `ldf.v X(r1), v1 =`

```
ldf X+0(r1), v1[0]
ldf X+1(r1), v1[1]
ldf X+2(r1), v1[2]
ldf X+3(r1), v1[3]
```
- Add two vectors: `addf.vv v1, v2, v3`

```
addf v1[i], v2[i], v3[i] (where i is 0,1,2,3)
```
- Add vector to scalar: `addf.vs v1, f2, v3`

```
addf v1[i], f2, v3[i] (where i is 0,1,2,3)
```

34

34

## Example Use of Vectors – 4-wide

<p>7x1024 insns</p> <pre>ldf X(r1), f1 mulf f0, f1, f2 ldf Y(r1), f3 addf f2, f3, f4 stf f4, Z(r1) addi r1, 4, r1 blti r1, 4096, 0</pre>		<pre>ldf.v X(r1), v1 mulf.vs v1, f0, v2 ldf.v Y(r1), v3 addf.vv v2, v3, v4 stf.v v4, Z(r1) addi r1, 16, r1 blti r1, 4096, 0</pre> <p>7x256 insns (4x fewer insns)</p>
----------------------------------------------------------------------------------------------------------------------------------------------	--	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Operations

- Load vector: `ldf.v X(r1), v1`
- Multiply vector to scalar: `mulf.vs v1, f2, v3`
- Add two vectors: `addf.vv v1, v2, v3`
- Store vector: `stf.v v1, X(r1)`

### Performance?

- If CPI = 1, 4x speedup
- CPI not always 1
  - Execution width (implementation) ≠ vector width (ISA)

35

35

## Why Vectorization is Awesome

Have your cake and eat it, too

All the benefits of a wider machine, without superscalar costs

- Single instruction fetch
- Wide reads & writes (without multiple \$ or regfile ports)
- Wider data to bypass ≠ N<sup>2</sup> bypass

Execution width (implementation) vs vector width (ISA)

- Example: Pentium 4 and Core 1 execute vector ops at half width
- Core 2 executes them at full width
- Intel's Sandy Bridge brings 256-bit vectors to x86
- Intel's Larrabee graphics chip brings 512-bit vectors to x86

Vector + superscalar? Sure!

- Multiple n-wide vector instructions per cycle

36

36

## Scheduling: Compiler or Hardware

### Compiler

- + Large scheduling scope (full program)
- + Simple hardware → fast clock, short pipeline, and low power
- Low branch prediction accuracy (profiling?)
- Little information on memory dependences (profiling?)
- Can't dynamically respond to cache misses (or anything really)
- Hard to speculate, recover from mis-speculation (h/w support?)

### Hardware

- Finite buffering resources fundamentally limit scheduling scope
- Scheduling machinery adds pipeline stages and consumes power
- + High branch prediction accuracy
- + Dynamic information about memory dependences
- + Can respond to cache misses
- + Easy to speculate and recover from mis-speculation

37

37