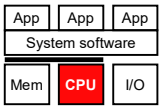
CSE 5600
Computer Systems Architecture

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

2

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

3

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?

4

Scheduling Techniques

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

5

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

6

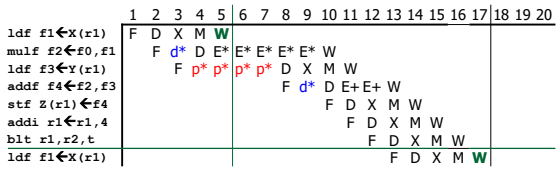
Running Code Example: SAXPY

- **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];

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

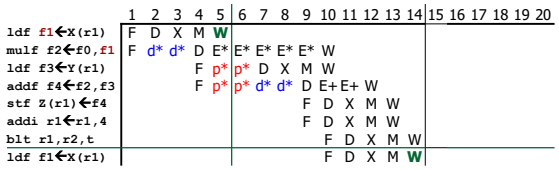

SAXPY Performance and Utilization



Scalar pipeline

- Full bypassing, 5-cycle E*, 2-cycle E+, predict branches taken
- Single iteration (7 insns) latency: **17–5 = 12 cycles**
- **Performance**: 7 insns / 12 cycles = 0.58 IPC
- **Utilization**: actual/peak IPC = 0.58 / 1 = 58%

SAXPY Performance and Utilization



2-way superscalar pipeline

- Any two insns per cycle + split integer and FP pipelines
- \+ **Performance**: 7 insns / 9 cycles = 0.78 IPC
- – **Utilization**: actual/peak IPC = 0.78 / 2 = 39%
- – More hazards → more stalls
- – Each stall is more expensive

Eliminate Load-Use Pairs?

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

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

Problem solved?

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

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

Unrolling SAXPY II: Pipeline Schedule

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

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

→

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


13

Unrolling SAXPY III: "Rename" Registers

- Pipeline scheduling causes reordering violations
  - Rename registers to correct

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

→

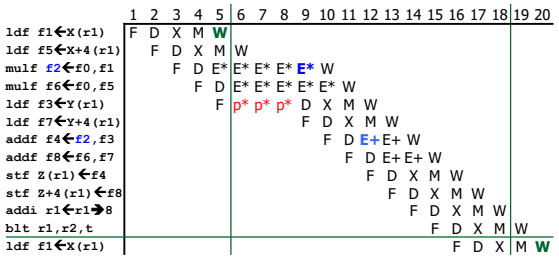
```
ldf f1←X(r1)
ldf f5←X+4(r1)
mulf f2←f0,f1
mulf f6←f0,f5
ldf f3←Y(r1)
ldf f7←Y+4(r1)
addf f4←f2,f3
addf f8←f6,f7
stf Z(r1)←f4
stf Z+4(r1)←f8
addi r1←r1,8
blt r1,r2,t
```

Do we have enough registers to do this?

Are we sure we can move these loads above these stores?
Alias analysis must be conservative.

14

Unrolled SAXPY Performance/Utilization



+ Performance: 12 insn / 15 cycles = 0.8 IPC
+ Utilization: actual/peak IPC = 0.8/1 = 80%
+ **Speedup**: (2 * 12 cycles) / 15 cycles = 1.6
? But improvement in IPC is only 0.8/0.58 = 1.38, what gives?

15

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

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

→

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

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

16

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

17

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

18

Superblocks

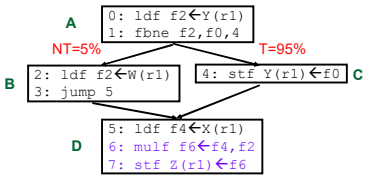
Source code

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

Machine code

```
0: ldf f2←Y(r1)
1: fbne f2,f0,4
2: ldf f2←W(r1)
3: jump 5
4: stf Y(r1)←f0
5: ldf f4←X(r1)
6: mulf f6←f4,f2
7: stf Z(r1)←f6
```

4 basic blocks: A,B,C,D



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

19

19

Superblock and Repair Code

Superblock

```
0: ldf f2←Y(r1)
1: fbeq f2,f0,2
4: stf Y(r1)←f0
5: ldf f4←X(r1)
6: mulf f6←f4,f2
7: stf Z(r1)←f6
```

Repair code

```
2: ldf f2←W(r1)
5': ldf f4←X(r1)
6': mulf f6←f4,f2
7': stf Z(r1)←f6
```

- 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

20

20

Superblocks Scheduling I

Superblock

```
0: ldf f2←Y(r1)
1: fbeq f2,f0,2
5: ldf f4←X(r1)
6: mulf f6←f4,f2
4: stf Y(r1)←f0
7: stf Z(r1)←f6
```

Repair code

```
2: ldf f2←W(r1)
5': ldf f4←X(r1)
6': mulf f6←f4,f2
7': stf Z(r1)←f6
```

- 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

21

21

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

```
cmoveq 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"

22

22

Avoiding Branches via ISA: Predication

- Conventional control
  - Conditionally executed insns also conditionally fetched

	1	2	3	4	5	6	7	8	9	
`beq r3,r7,targ`	F	D	X	M	W					
`sub r5←r6,1`		F	D	--	--	--				flushed: wrong path
`targ:add r4←r4,r5`			F	--	--	--	--			flushed: why?
`targ:add r4←r4,r5`				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**

23

23

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

24

24

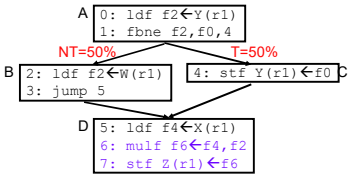
Conditional Register Moves (CMOVs)

Conditional (register) moves

- Construct appearance of full predication from one primitive
  `cmoveq 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
sub r6,1,r9        D  X  M  W
cmovne r3,r9,r5    F  D  X  M  W
targ:add r4,r5,r4     F  D  X  M  W
```


Non-Biased Branches: Use Predication



Using Predication

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


ISA Support for Predication

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

- IA-64: change branch 1 to **set-predicate insn** `fspne`
- 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

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

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

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

Data-Level Parallelism

Data-level parallelism (DLP)

- Single operation repeated on multiple data elements
  - SIMD (**S**ingle-**I**nstruction, **M**ultiple-**D**ata)
- 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!

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 v1←X(r1) =`

```
ldf v1[0]←X+0(r1)
ldf v1[1]←X+1(r1)
ldf v1[2]←X+2(r1)
ldf v1[3]←X+3(r1)
```

  - 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←v2,f3`

```
addf v1[i]←v2[i],f3  (where i is 0,1,2,3)
```


Example Use of Vectors – 4-wide

7x1024 insns

```
ldf f1←X(r1)
mulf f2←f0,f1
ldf f3←Y(r1)
addf f4←f2,f3
stf Z(r1)←f4
addi r1←r1,4
blti r1,4096,t
```

7x256 insns (4x fewer insns)

```
ldf.v v1←X(r1)
mulf.vs v2←v1,f0
ldf.v v3←Y(r1)
addf.vv v4←v2,v3
stf.v Z(r1)←v4
addi r1←r1,16
blti r1,4096,0
```

Operations

- Load vector: `ldf.v v1←X(r1)`
- Multiply vector to scalar: `mulf.vs v2←v1,f0`
- Add two vectors: `addf.vv v4←v2,v3`
- Store vector: `stf.v X(r1)←v4`

Performance?

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

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^2 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

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