Programming for Performance
Single-Thread Performance: Compiler Scheduling for Pipelines

Adopted from

Siddhartha Chatterjee

Spring 2009

Review of Pipelining

- Pipelining improves throughput of an instruction sequence but not the latency of an individual instruction
- Speedup due to pipelining limited by hazards
  - Structural hazards lead to contention for limited resources
  - Data hazards require stalling or forwarding to maintain sequential semantics
  - Control hazards require cancellation of instructions (to maintain sequential branch semantics) or delayed branches (to define a new branch semantics)
- Hazard
  - Detection: interlocks in hardware
  - Elimination: renaming, branch elimination
  - Resolution: stalling, forwarding, scheduling

CPI of a Pipelined Machine

Issuing multiple instructions per cycle
Compiler dependence analysis
Software pipelining
Trace scheduling

Pipeline CPI = Ideal pipeline CPI
+ Structural stalls
+ RAW stalls
+ WAR stalls
+ WAW stalls
+ Control stalls

Dynamic scheduling with register renaming
Compiler dependence analysis
Software pipelining
Trace scheduling
Speculation

Basic pipeline scheduling
Dynamic scheduling with scoreboarding
Dynamic memory disambiguation (for stalls involving memory)
Compiler dependence analysis
Software pipeline
Trace scheduling
Speculation

Loop unrolling
Dynamic branch prediction
Speculation
Predication

Instruction-Level Parallelism (ILP)

- Pipelining is most effective when we have parallelism among instructions
- Parallelism within a basic block is limited
  - Branch frequency of 15% implies about six instructions in basic block
  - These instructions are likely to depend on each other
  - Need to look beyond basic blocks
- Loop-level parallelism
  - Parallelism among iterations of a loop
  - To convert loop-level parallelism into ILP, we need to “unroll” the loop
    - Statically, by the compiler
    - Dynamically, by the hardware
    - Using vector instructions

Motivating Example for Loop Unrolling

```
for (u = 0; u < 1000; u++)
    x[u] = x[u] + s;
```

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

```
LOOP:  LD      F0, 0(R1)
       ADDD    F4, F0, F2
       SD      0(R1), F4
       SUBI    R1, R1, 8
       BNEZ    R1, LOOP
       NOP
```

Assumptions

- Loop is being run backwards
- Scalar s is in register pair F2:F3
- Array x starts at memory address 0
- 1-cycle branch delay
- No structural hazards

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LD	F	D	X	M	W										
ADDD		-	F	D	E	E	E	E	M	W					
SD				-	-	F	D	X	M	W					
SUBI							F	D	X	M	W				
BNEZ								-	F	D	X	M	W		
NOP										-					
LD											F	D	X	M	W

10 cycles per iteration

How Far Can We Get With Scheduling?

```
LOOP:  LD      F0, 0(R1)
       ADDD    F4, F0, F2
       SD      0(R1), F4
       SUBI    R1, R1, 8
       BNEZ    R1, LOOP
       NOP
```

```
LOOP:  LD      F0, 0(R1)
       SUBI    R1, R1, 8
       ADDD    F4, F0, F2
       BNEZ    R1, LOOP
       SD      8(R1), F4
```

```
for (u = 999; u >= 0; ){
     register double d = x[u];
     u--; d += s; x[u+1] = d;
}
```

	1	2	3	4	5	6	7	8	9	10	11
LD	F	D	X	M	W						
SUBI		F	D	X	M	W					
ADDD			F	D	E	E	E	E	M	W	
BNEZ				F	D	X	M	W			
SD					-	F	D	X	M	W	
LD							F	D	X	M	W

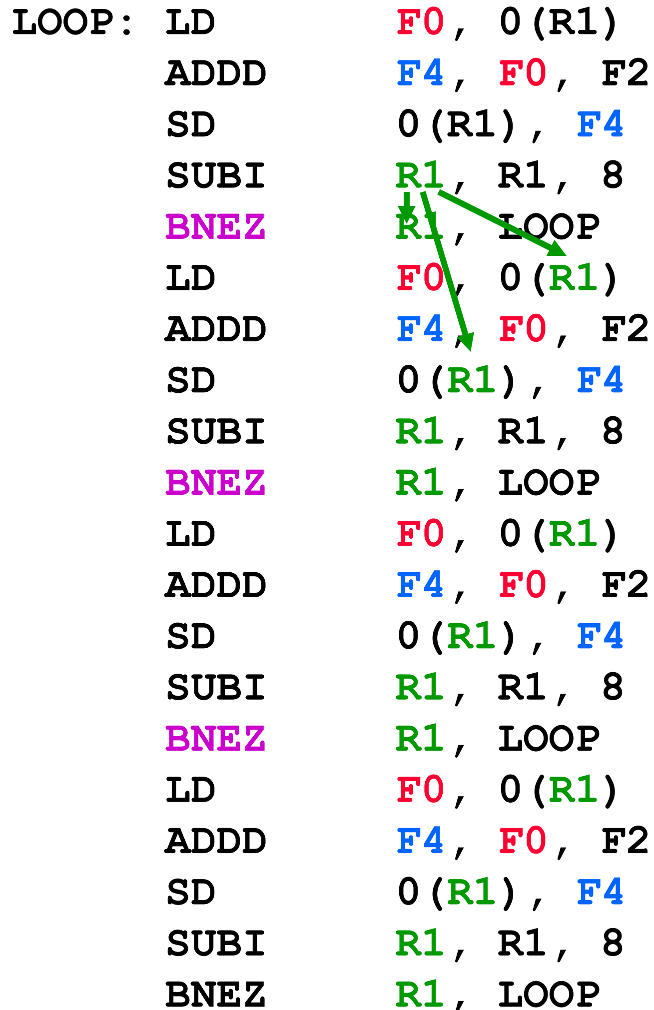
6 cycles per iteration

Note change in `SD` instruction, from `0(R1)` to `8(R1)`; this is a non-trivial change.

Observations on Scheduled Code

- 3 out of 5 instructions involve FP work
- The other two constitute loop overhead
- Could we improve loop performance by unrolling the loop?
- Assume number of loop iterations is a multiple of 4, and unroll loop body four times
  - In real life, would need to handle the fact that loop trip count may not be a multiple of 4

Unrolling: Take 1



- This is not any different from situation before unrolling
- Branches induce control dependence
  - Can't move instructions much during scheduling
- However, the whole point of unrolling was to guarantee that the three internal branches will fall through
- So, maybe we can delete the intermediate branches
- There is an implicit NOP after the final branch

Unrolling: Take 2

```
LOOP:  LD      F0, 0(R1)
       ADDD    F4, F0, F2
       SD      0(R1), F4
       SUBI    R1, R1, 8
       LD      F0, 0(R1)
       ADDD    F4, F0, F2
       SD      0(R1), F4
       SUBI    R1, R1, 8
       LD      F0, 0(R1)
       ADDD    F4, F0, F2
       SD      0(R1), F4
       SUBI    R1, R1, 8
       LD      F0, 0(R1)
       ADDD    F4, F0, F2
       SD      0(R1), F4
       SUBI    R1, R1, 8
       BNEZ    R1, LOOP
```

- Even though we got rid of the control dependences, we have flow dependences through R1
- We could remove flow dependences by observing that R1 is decremented by 8 each time
  - Adjust the address specifiers
  - Delete the first three SUBIs
  - Change the constant in the fourth SUBI to 32
- These are non-trivial inferences for a compiler to make

```
for (u = 999; u >= 0; ){
    register double d;
    d = x[u]; d += s; x[u] = d; u--;
    d = x[u]; d += s; x[u] = d; u--;
    d = x[u]; d += s; x[u] = d; u--;
    d = x[u]; d += s; x[u] = d; u--;
}
```

Unrolling: Take 3

```
LOOP:  LD      F0, 0(R1)
       ADDD    F4, F0, F2
       SD      0(R1), F4
       LD      F0, -8(R1)
       ADDD    F4, F0, F2
       SD      -8(R1), F4
       LD      F0, -16(R1)
       ADDD    F4, F0, F2
       SD      -16(R1), F4
       LD      F0, -24(R1)
       ADDD    F4, F0, F2
       SD      -24(R1), F4
       SUBI    R1, R1, 32
       BNEZ    R1, LOOP
```

- Performance is now limited by the anti-dependences and output dependences on F0 and F4
- These are name dependences
  - The instructions are not in a producer-consumer relation
  - They are simply using the same registers, but they don't have to
  - We can use different registers in different loop iterations, subject to availability
- Let's rename registers

```
 for (u = 999; u >= 0; u -= 4){
     register double d;
     d = x[u]; d += s; x[u] = d;
     d = x[u-1]; d += s; x[u-1] = d;
     d = x[u-2]; d += s; x[u-2] = d;
     d = x[u-3]; d += s; x[u-3] = d;
}
```

Unrolling: Take 4

```
LOOP:  LD      F0,  0(R1)
       ADDD    F4,  F0,  F2
       SD      0(R1),  F4
       LD      F6,  -8(R1)
       ADDD    F8,  F6,  F2
       SD      -8(R1),  F8
       LD      F10,  -16(R1)
       ADDD    F12,  F10,  F2
       SD      -16(R1),  F12
       LD      F14,  -24(R1)
       ADDD    F16,  F14,  F2
       SD      -24(R1),  F16
       SUBI    R1,  R1,  32
       BNEZ    R1,  LOOP
```

- Time for execution of 4 iterations
  - 14 instruction cycles
  - 4 LD→ADDD stalls
  - 4 ADDD→SD stalls (2 cycles each)
  - 1 SUBI→BNEZ stall
  - 1 branch delay stall
- 36 cycles for 4 iterations, or 9 cycles per iteration
- Slower than scheduled version of original loop
- Let's schedule the unrolled loop

```
for (u = 999; u >= 0; u -= 4){
    register double d0, d1, d2, d3;
    d0 = x[u]; d0 += s; x[u] = d0;
    d1 = x[u-1]; d1 += s; x[u-1] = d1;
    d2 = x[u-2]; d2 += s; x[u-2] = d2;
    d3 = x[u-3]; d3 += s; x[u-3] = d3;
}
```

Unrolling: Take 5

```
LOOP:  LD      F0, 0(R1)
       LD      F6, -8(R1)
       LD      F10, -16(R1)
       LD      F14, -24(R1)
       ADDD    F4, F0, F2
       ADDD    F8, F6, F2
       ADDD    F12, F10, F2
       ADDD    F16, F14, F2
       SD      0(R1), F4
       SD      -8(R1), F8
       SUBI    R1, R1, 32
       SD      16(R1), F12
       BNEZ    R1, LOOP
       SD      8(R1), F16
```

```
 for (u = 999; u >= 0; ){
     register double d0, d1, d2, d3;
     d0 = x[u]; d1 = x[u-1]; d2 = x[u-2]; d3 = x[u-3];
     d0 += s; d1 += s; d2 += s; d3 += s;
     x[u] = d0; x[u-1] = d1; u -= 4;
     x[u+2] = d2; x[u+1] = d3;
}
```

- This code runs without stalls
  - 14 cycles for 4 iterations
  - 3.5 cycles per iteration
  - Performance is limited by loop control overhead once every four iterations
- Note that original loop had three FP instructions that were not independent
- Loop unrolling exposed independent instructions from multiple loop iterations
- By unrolling further, can approach asymptotic rate of 3 cycles per iteration
  - Subject to availability of registers

What Did The Compiler Have To Do?

- Determine that it was legal to move the SD after the SUBI and BNEZ, and find the amount to adjust the SD offset
- Determine that loop unrolling would be useful by discovering independence of loop iterations
- Rename registers to avoid name dependences
- Eliminate extra tests and branches and adjust loop control
- Determine that LDs and SDs can be interchanged by determining that (since R1 is not being updated) the address specifiers 0(R1), -8(R1), -16(R1), -24(R1) all refer to different memory locations
- Schedule the code, preserving dependences
- Resources consumed: Code space, architectural registers

Dependences

- Three kinds of dependences
  - Data dependence
  - Name dependence
  - Control dependence
- In the context of loop-level parallelism, data dependence can be
  - Loop-independent
  - Loop-carried
- Data dependences act as a limit of how much ILP can be exploited in a compiled program
- Compiler tries to identify and eliminate dependences
- Hardware tries to prevent dependences from becoming stalls

Data and Name Dependences

- Instruction *v* is data-dependent on instruction *u* if
  - *u* produces a result that *v* consumes
- Instruction *v* is anti-dependent on instruction *u* if
  - *u* precedes *v*
  - *v* writes a register or memory location that *u* reads
- Instruction *v* is output-dependent on instruction *u* if
  - *u* precedes *v*
  - *v* writes a register or memory location that *u* writes
- A data dependence that cannot be removed by renaming corresponds to a RAW hazard
- Anti-dependence corresponds to a WAR hazard
- Output dependence corresponds to a WAW hazard

Control Dependences

- A control dependence determines the ordering of an instruction with respect to a branch instruction so that the non-branch instruction is executed only when it should be

```
if (p1) {s1;}
if (p2) {s2;}
```

- Control dependence constrains code motion
  - An instruction that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
  - An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Data Dependence in Loop Iterations

```
A[u+1] = A[u]+C[u];
B[u+1] = B[u]+A[u+1];
```

```
A[u+1] = A[u]+C[u];
B[u+1] = B[u]+A[u+1];
A[u+2] = A[u+1]+C[u+1];
B[u+2] = B[u+1]+A[u+2];
```

```
A[u] = A[u]+B[u];
B[u+1] = C[u]+D[u];
```

```
A[u] = A[u]+B[u];
B[u+1] = C[u]+D[u];
A[u+1] = A[u+1]+B[u+1];
B[u+2] = C[u+1]+D[u+1];
```

```
B[u+1] = C[u]+D[u];
A[u+1] = A[u+1]+B[u+1];
```

```
B[u+1] = C[u]+D[u];
A[u+1] = A[u+1]+B[u+1];
B[u+2] = C[u+1]+D[u+1];
A[u+2] = A[u+2]+B[u+2];
```

Loop Transformation

- Sometimes loop-carried dependence does not prevent loop parallelization
- Example: Second loop of previous slide
- In other cases, loop-carried dependence prohibits loop parallelization
- Example: First loop of previous slide

```
A[u] = A[u]+B[u];
B[u+1] = C[u]+D[u];
```

```
A[u] = A[u]+B[u];
B[u+1] = C[u]+D[u];
A[u+1] = A[u+1]+B[u+1];
B[u+2] = C[u+1]+D[u+1];
A[u+2] = A[u+2]+B[u+2];
B[u+3] = C[u+2]+D[u+2];
```

```
A[u] = A[u]+B[u];
B[u+1] = C[u]+D[u];
A[u+1] = A[u+1]+B[u+1];
B[u+2] = C[u+1]+D[u+1];
A[u+2] = A[u+2]+B[u+2];
B[u+3] = C[u+2]+D[u+2];
```