

6.035

Fall 2006

Loop Optimizations

Instruction Scheduling

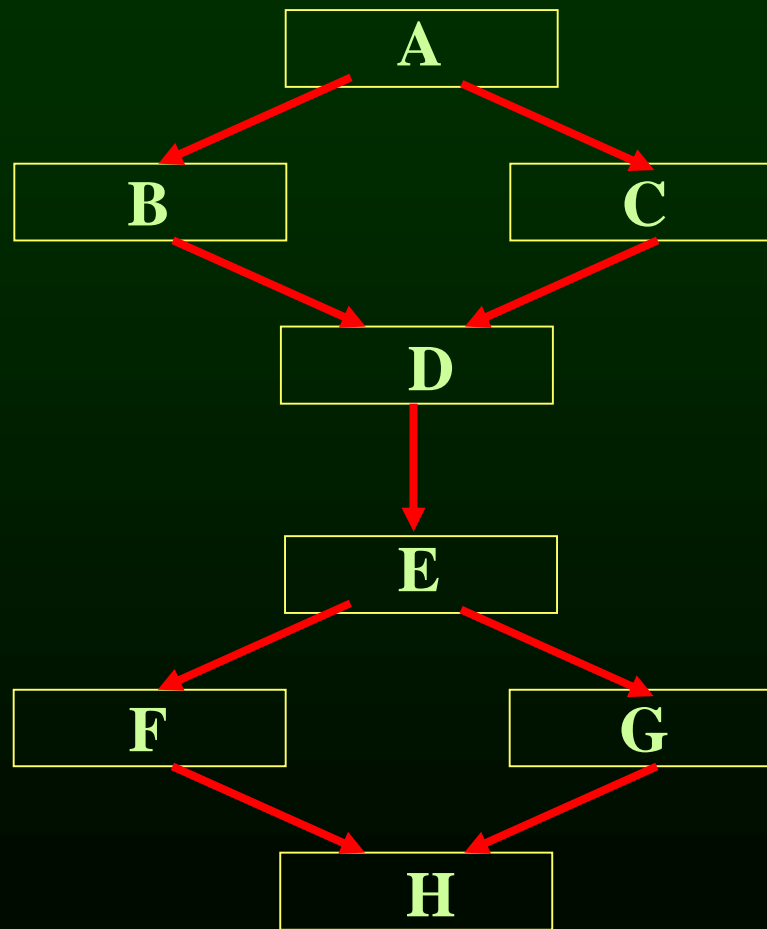
Control Dependencies: Scheduling across basic blocks

- Number of instructions in a basic block is small
 - Cannot keep a multiple units with long pipelines busy by just scheduling within a basic block
- Need to handle control dependence
 - Scheduling constraints across basic blocks
 - Scheduling policy

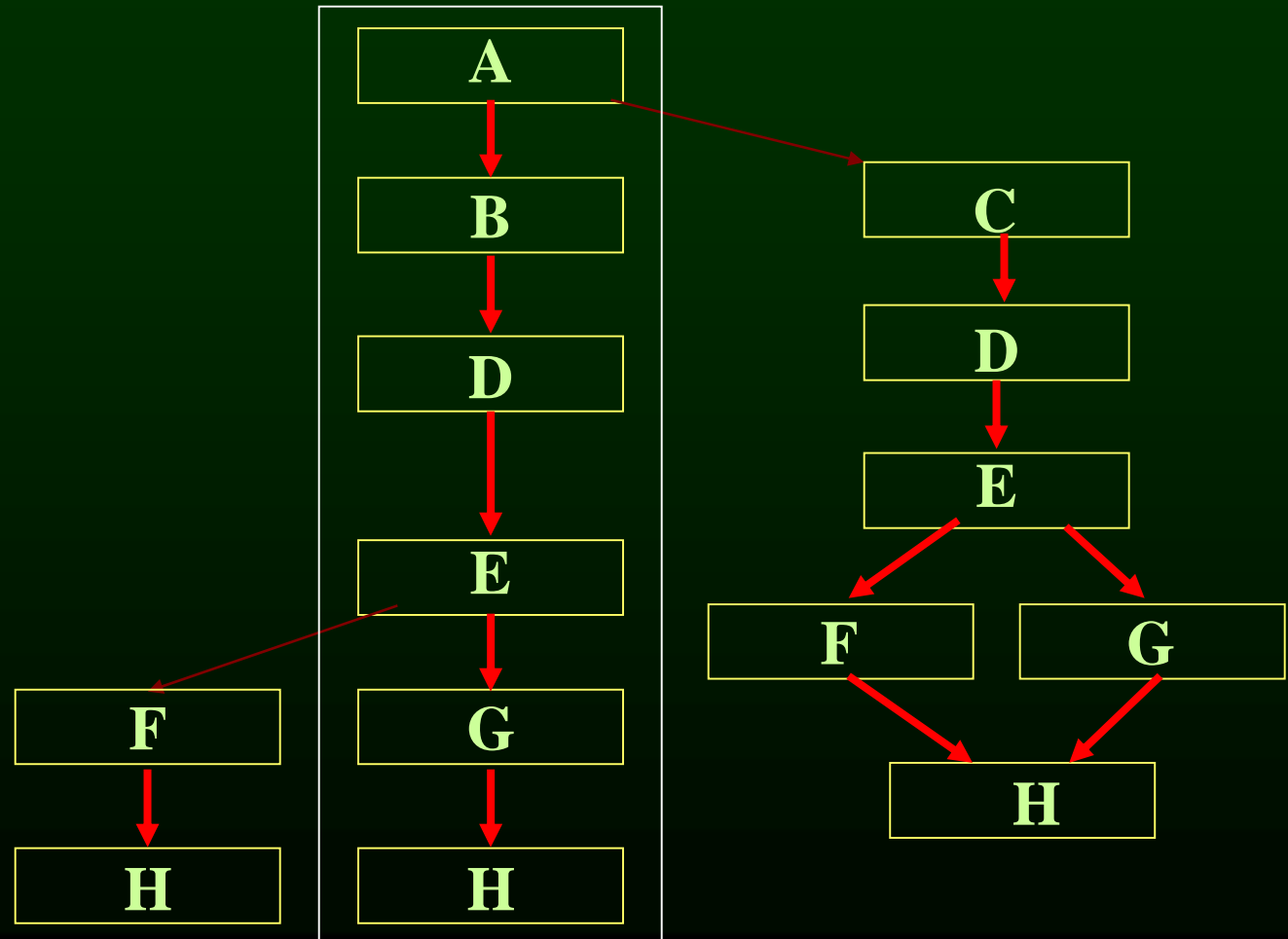
Trace Scheduling

- Find the most common trace of basic blocks
 - Use profile information
- Combine the basic blocks in the trace and schedule them as one block
- Create clean-up code if the execution goes off-trace

Trace Scheduling



Trace Scheduling



Outline

- Trace scheduling
- **Scheduling for loops**
- Loop unrolling
- Software pipelining
- Interaction with register allocation
- Hardware vs. Compiler
- loop invariant code motion
- Induction Variable Recognition
- Strength Reduction
- Loop Test Replacement

Scheduling Loops

- Loop bodies are small
- But, lot of time is spend in loops due to large number of iterations
- Need better ways to schedule loops

Loop Example

- Machine
 - One load/store unit
 - load 2 cycles
 - store 2 cycles
 - Two arithmetic units
 - add 2 cycles
 - branch 2 cycles
 - multiply 3 cycles
 - Both units are pipelined (initiate one op each cycle)

- Source Code

```
for i = 1 to N
    A[i] = A[i] * b
```

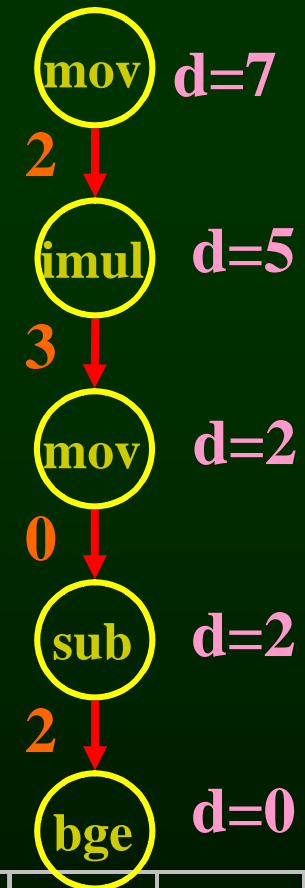
Loop Example

- Assembly Code

```

loop:
    mov    (%rdi,%rax), %r10
    imul  %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub   $4, %rax
    bge   loop
    
```

- Schedule (9 cycles per iteration)



mov					mov							
	mov					mov						
		imul					bge					
			imul					bge				
				imul								
					sub							
						sub						

Loop Unrolling

- Unroll the loop body few times
- Pros:
 - Create a much larger basic block for the body
 - Eliminate few loop bounds checks
- Cons:
 - Much larger program
 - Setup code (# of iterations < unroll factor)
 - beginning and end of the schedule can still have unused slots

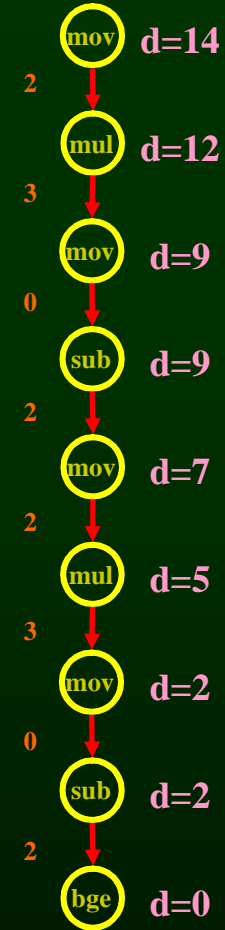
Loop Example

```
loop:  
  mov    (%rdi,%rax), %r10  
  imul  %r11, %r10  
  mov    %r10, (%rdi,%rax)  
  sub   $4, %rax  
  bge   loop
```

Loop Example

```

loop:
    mov    (%rdi,%rax), %r10
    imul  %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub   $4, %rax
    mov    (%rdi,%rax), %r10
    imul  %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub   $4, %rax
    bge   loop
    
```



- Schedule (8 cycles per iteration)

mov					mov		mov					mov			
	mov					mov		mov					mov		
		imul							imul					bge	
			imul							imul					bge
				imul							imul				
					sub							sub			
						sub							sub		

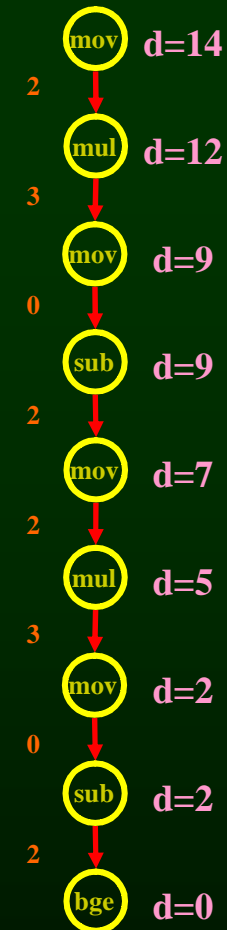
Loop Unrolling

- Rename registers
 - Use different registers in different iterations
- Eliminate unnecessary dependencies
 - again, use more registers to eliminate true, anti and output dependencies
 - eliminate dependent-chains of calculations when possible

Loop Example

loop:

```
mov    (%rdi,%rax), %r10
imul   %r11, %r10
mov    %r10, (%rdi,%rax)
sub    $4, %rax
mov    (%rdi,%rax), %rcx
imul   %r11, %rcx
mov    %rcx, (%rdi,%rax)
sub    $4, %rax
bge    loop
```



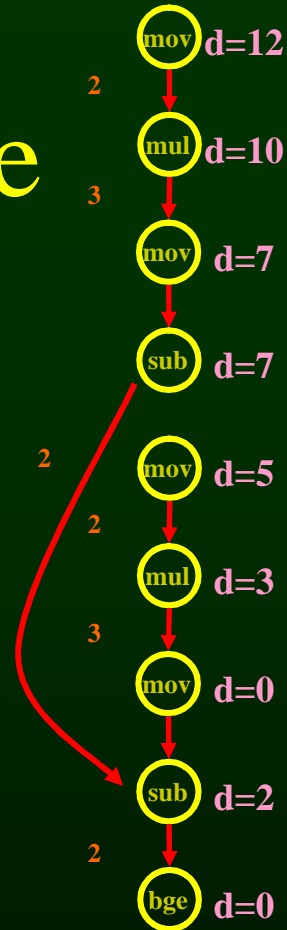
Loop Example

loop:

```

mov    (%rdi,%rax), %r10
imul  %r11, %r10
mov    %r10, (%rdi,%rax)
sub    $8, %rax
mov    (%rdi,%rbx), %rcx
imul  %r11, %rcx
mov    %rcx, (%rdi,%rbx)
sub    $8, %rbx
bge   loop
    
```

- Schedule (4.5 cycles per iteration)



mov		mov			mov		mov			
	mov		mov			mov		mov		
		imul		imul			bge			
			imul		imul			bge		
				imul		imul				
					sub		sub			
						sub		sub		

Software Pipelining

- Try to overlap multiple iterations so that the slots will be filled
- Find the steady-state window so that:
 - all the instructions of the loop body is executed
 - but from different iterations

Loop Example

- Assembly Code

```

loop:
    mov    (%rdi,%rax), %r10
    imul  %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub   $4, %rax
    bge   loop
    
```

- Schedule

mov		mov1		mov2	st	mov3	st1	mov4	st2	mov5	st3	mov6
	mov		mov1		mov2	mov	mov3	mov1	mov4	mov2	ld5	mov3
		mul		mul1		mul2	bge	mul3	bge1	mul4	bge2	mul5
			mul		mul1		mul2	bge	mul3	bge1	mul4	bge2
				mul		mul1		mul2		mul3		mul4
					sub		sub1		sub2		sub3	
						sub		sub1		sub2		sub3

Loop Example

- 4 iterations are overlapped
 - value of `%r11` don't change
 - 4 regs for `(%rdi,%rax)`
 - each addr. incremented by $4*4$
 - 4 regs to keep value `%r10`
 - Same registers can be reused after 4 of these blocks
generate code for 4 blocks,
otherwise need to move

mov4	st2
mov1	mov4
mul3	bge1
bge	mul3
mul2	
	sub2
sub1	

```
loop:
    mov    (%rdi,%rax), %r10
    imul  %r11, %r10
    mov    %r10, (%rdi,%rax)
    sub   $4, %rax
    bge   loop
```

Software Pipelining

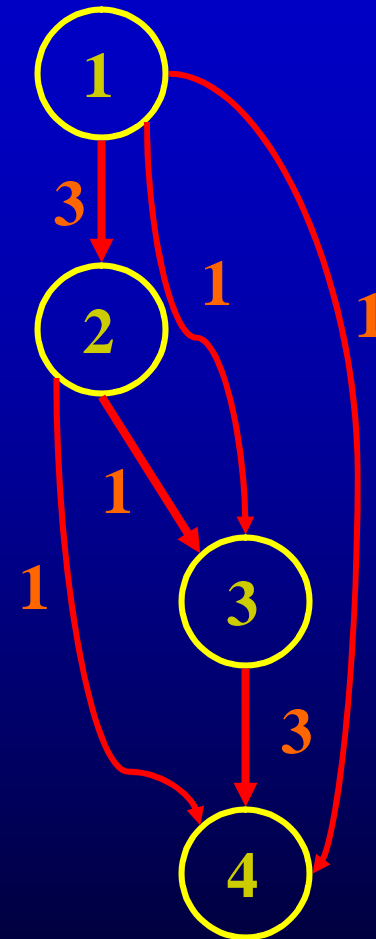
- Optimal use of resources
- Need a lot of registers
 - Values in multiple iterations need to be kept
- Issues in dependencies
 - Executing a store instruction in an iteration before branch instruction is executed for a previous iteration (writing when it should not have)
 - Loads and stores are issued out-of-order (need to figure-out dependencies before doing this)
- Code generation issues
 - Generate pre-amble and post-amble code
 - Multiple blocks so no register copy is needed

Register Allocation and Instruction Scheduling

- If register allocation is before instruction scheduling
 - restricts the choices for scheduling

Example

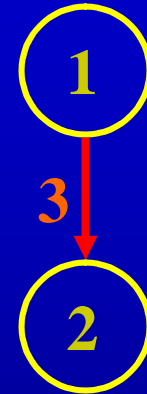
```
1: mov    4(%rbp), %rax
2: add    %rax, %rbx
3: mov    8(%rbp), %rax
4: add    %rax, %rcx
```



ALUop			2		4
MEM 1	1			3	
MEM 2		1			3

Example

```
1: mov    4(%rbp), %rax
2: add    %rax, %rbx
3: mov    8(%rbp), %r10
4: add    %r10, %rcx
```



ALUop			2	4
MEM 1	1	3		
MEM 2		1	3	

Register Allocation and Instruction Scheduling

- If register allocation is before instruction scheduling
 - restricts the choices for scheduling

Register Allocation and Instruction Scheduling

- If register allocation is before instruction scheduling
 - restricts the choices for scheduling
- If instruction scheduling before register allocation
 - Register allocation may spill registers
 - Will change the carefully done schedule!!!

Superscalar: Where have all the transistors gone?

- Out of order execution
 - If an instruction stalls, go beyond that and start executing non-dependent instructions
 - Pros:
 - Hardware scheduling
 - Tolerates unpredictable latencies
 - Cons:
 - Instruction window is small

Superscalar: Where have all the transistors gone?

- Register renaming
 - If there is an anti or output dependency of a register that stalls the pipeline, use a different hardware register
 - Pros:
 - Avoids anti and output dependencies
 - Cons:
 - Cannot do more complex transformations to eliminate dependencies

Hardware vs. Compiler

- In a superscalar, hardware and compiler scheduling can work hand-in-hand
- Hardware can reduce the burden when not predictable by the compiler
- Compiler can still greatly enhance the performance
 - Large instruction window for scheduling
 - Many program transformations that increase parallelism
- Compiler is even more critical when no hardware support
 - VLIW machines (Itanium, DSPs)

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
```

```
for i = 1 to N
```

```
  x = x + 1
```

```
  for j = 1 to N
```

```
    a(i,j) = t1 + 10*i + j + x
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
```

```
for i = 1 to N
```

```
  x = x + 1
```

```
  for j = 1 to N
```

```
    a(i,j) = t1 + 10*i + j + x
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N
```

```
for i = 1 to N
```

```
  x = x + 1
```

```
  t2 = t1 + 10*i + x
```

```
  for j = 1 to N
```

```
    a(i,j) = t2 + j
```

Induction Variables

- Example

```
i = 200
```

```
for j = 1 to 100
```

```
  a(i) = 0
```

```
  i = i - 1
```

Induction Variables

- Example

```
for j = 1 to 100  
  a(201 - j) = 0
```

Basic Induction variable:

J = 1, 2, 3, 4,

Index Variable i in a(i):

I = 200, 199, 198, 197.... = 201 - J

What are induction variables?

- x is an induction variable of a loop L if
 - variable changes its value every iteration of the loop
 - the value is a function of number of iterations of the loop
- In compilers this function is normally a linear function
 - Example: for loop index variable j , function $c*j + d$

What can we do with induction variables?

- Use them to perform strength reduction
- Get rid of them

Classification of induction variables

- Basic induction variables
 - Explicitly modified by the same constant amount once during each iteration of the loop
 - Example: loop index variable
- Dependent induction variables
 - Can be expressed in the form: $a*x + b$ where a and b are loop invariant and x is an induction variable
 - Example: $202 - 2*j$

Classification of induction variables

- Class of induction variables: All induction variables with same basic variable in their linear equations
- Basis of a class: the basic variable that determines that class

Finding Basic Induction Variables

- Look inside loop nodes
- Find variables whose only modification is of the form $j = j + d$ where d is a loop constant

Finding Dependent Induction Variables

- Find all the basic induction variables
- Search variable k with a single assignment in the loop
- Variable assignments of the form $k = e \text{ op } j$ or $k = -j$ where j is an induction variable and e is loop invariant

Finding Dependent Induction Variables

- Example

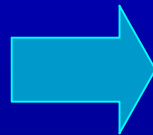
```
for i = 1 to 100
```

```
    j = i*c
```

```
    k = j+1
```

A special case

```
t = 202
for j = 1 to 100
  t = t - 2
  a(j) = t
  t = t - 2
  b(j) = t
```



```
u1 = 200
u2 = 202
for j = 1 to 100
  u1 = u1 - 4
  a(j) = u1
  u2 = u2 - 4
  b(j) = u2
```

Strength Reduction

- Replace expensive operations in an expression using cheaper ones
 - Not a data-flow problem
 - Algebraic simplification
 - Example: $a*4 \Rightarrow a \ll 2$

Strength Reduction

- In loops reduce expensive operations in expressions in to cheaper ones by using the previously calculated value

Strength Reduction

`t = 202`

`for j = 1 to 100`

`t = t - 2`

`*(abase + 4*j) = t`




Basic Induction variable:

J = 1,  2,  3,  4,

Induction variable $200 - 2*j$

t = 202,  200,  198,  196,

Induction variable $abase+4*j$:

$abase+4*j = abase+4, abase+8, abase+12, abase+14, \dots$
  

Strength Reduction Algorithm

- For a dependent induction variable $k = a*j + b$
- Add a pre-header $k' = a*j_{init} + b$
- Next to $j = j + c$ add $k' = k' + a*c$
- Use k' instead of k

```
t = abase + 4*1
for j = 1 to 100
  *(t) = j
  t = t + 4
```

Example

```
double A[256], B[256][256]
```

```
j = 1
```

```
while(j < 100)
```

```
    A[j] = B[j][j]
```

```
    j = j + 2
```

Example

```
double A[256], B[256][256]
j = 1
a = &A + 4

while(j>100)
    *a = *(&B + 4*(256*j + j))
    j = j + 2
    a = a + 8
```

Base Induction Variable: j

Dependent Induction Variable: $a = \&A + 4*j$

Example

```
double A[256], B[256][256]
j = 1
a = &A + 4
b = &B + 1028
while(j > 100)
    *a = *b
    j = j + 2
    a = a + 8
    b = b + 2056
```

Base Induction Variable: j

Dependent Induction Variable: $b = \&B + 4 * 257 * j$

Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```
double A[256], B[256][256]
j = 1
while(j>100)
    A[j] = B[j][j]
```

Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```
double A[256], B[256][256]
j = 1
a = &A + 4
b = &B + 1028
while(j>100)
    *a = *b
    j = j + 2
    a = a + 8
    b = b + 2056
```

Loop Test Replacement

- Eliminate basic induction variable used only for calculating other induction variables

```
double A[256], B[256][256]
```

```
a = &A + 4
```

```
b = &B + 1028
```

```
while(a > &A + 800)
```

```
    *a = *b
```

```
    a = a + 8
```

```
    b = b + 2056
```

- J is only used for the loop bound

- Use a dependent IV (a or b)

- Lets choose a

$$j > 100 \Rightarrow a > \&A + 800$$

- Replace the loop condition

- Get rid of j

Loop Test Replacement Algorithm

- If basic induction variable J is only used for calculating other induction variables
- Select an induction variable k in the family of J
($K = a*J + b$)

- Replace a comparison such as

```
if (J > X) goto L1
```

by

```
if(K' > a*X + b) goto L1 if a is positive
```

```
if(K' < a*X + b) goto L1 if a is negative
```

- If J is live at any exit from loop, recompute

$$J = (K' - b) / a$$