

## Instruction Scheduling

### Lecture 22

Prof. Necula CS 164

1

## Lecture Outline

- Instruction-Level Parallelism
- Instruction Scheduling
- List Scheduling
  - A simple and effective heuristic for instruction scheduling
  - An extended example

Prof. Necula CS 164

2

## Instruction-Level Parallelism (ILP)

- Modern CPUs can execute multiple instructions concurrently
- Two sources of parallelism are exploited
  - Some machines issue multiple instructions in one cycle  $\Rightarrow$  superscalar machine
  - Some machines overlap various execution phases of different instructions  $\Rightarrow$  pipelining
  - Most modern machines do both
- ILP can be improved by reordering instructions  $\Rightarrow$  instruction scheduling

Prof. Necula CS 164

3

## Instruction Dependencies

- Not any two instructions can be reordered
- Resource dependencies
  - Two instructions that must use the same functional unit cannot execute at once
- Data dependencies
  - Two instructions use the same registers for operands or results

Prof. Necula CS 164

4

## Data Dependencies

- A must finish before B if
  - B reads a register written by A  
 $a \leftarrow b + 1; b \leftarrow a * 2$ 
    - Read-after-write dependency
  - B writes a register also written by A  
 $a \leftarrow b + 1; \dots; a \leftarrow c * 2$ 
    - Write-after-write dependency
  - B writes a register that A reads  
 $a \leftarrow b + 1; b \leftarrow c * 2$ 
    - Write-after-read dependency

Prof. Necula CS 164

5

## Two Kinds of Scheduling Techniques

- Dynamic-scheduling
  - The processor decides the order at run-time
  - Also called out-of-order execution
- Static-scheduling
  - The compiler decides the order at compiler time
  - We will look at this technique

Prof. Necula CS 164

6

## The MIPS

- On the MIPS most operations execute in 1 cycle
- Conditional branches require 2 cycles to complete

```
addiu $t1 $t1 1
addiu $t2 $t2 1
beq $t2 $t3 label
```
- This code requires 4 cycles to complete (1 for each add and 2 for the branch)

Prof. Necula CS 164

7

## MIPS Branch Delay Slots

- We can insert a `nop` to make the second branch cycle explicit

```
addiu $t1 $t1 1
addiu $t2 $t2 1
beq $t2 $t3 label
*nop
```
- The \* means that the `nop` executes in the branch's second cycle
- This cycle is called branch delay slot

Prof. Necula CS 164

8

## MIPS Branch Delay Slots (Cont.)

- The code can be improved by scheduling something useful in the delay slot:

```
addiu $t2 $t2 1
beq $t2 $t3 label
* addiu $t1 $t1 1
```
- This code is equivalent to the original
  - The final state of the machine is the same
- But executes 25% faster (3 cycles)

Prof. Necula CS 164

9

## MIPS Branch Delay Slots (Cont.)

- Note that not any instruction can go in the delay slot.
- If we try:

```
addiu $t1 $t1 1
beq $t2 $t3 label
* addiu $t2 $t2 1
```
- This code is no longer correct
- It uses the wrong value of `$t2` in the comparison

Prof. Necula CS 164

10

## Beyond MIPS

- On the MIPS the only scheduling problem is filling the delay slot
- On newer architectures the scheduling problem is both
  - more complex, and
  - more critical to good performance
- We will show this on a "made-up" architecture

Prof. Necula CS 164

11

## Our Architecture

- Based on Motorola 68xxx but simplified

Operation	Description	Cycles
$i := j +_I k$	Integer add	1
$i := j +_F k$	Floating-point add	2
$i := j * k$	Multiply (integer or float)	3
$i := j(k)$	Memory load from $j + k$	2
$\text{if } i = j \text{ goto } L$	Branch	4

Prof. Necula CS 164

12

## Our Architecture (Cont.)

- Instructions issue in the program order
- All instructions are fully pipelined
  - The machine can issue one instruction per cycle
  - Any instruction can issue in any cycle
- The results of an instruction are unavailable until the instruction completes
- An instruction is delayed if it depends on results not yet available

Prof. Necula CS 164

13

## An Example

- We'll use the following loop as a running example:
 

```
innerprod := 0;
for(i = 1; i <= n; i++) {
    innerprod := A[i] * B[i] + innerprod;
}
```
- Where *A* and *B* are two floating-point arrays

Prof. Necula CS 164

14

## The Code for the Example

- The generated code might be like this:

```
innerprod := 0
i := 1
top: t1 := 4 * i
    t2 := t1(A)
    t3 := 4 * i
    t4 := t3(B)
    t5 := t2 * t4
    innerprod := t5 +F innerprod
    i := i +T 1
    if i <= n goto top
```

Prof. Necula CS 164

15

## The Optimized Code for the Example

- Note that  $t_1$  and  $t_3$  are common subexpr.
- After local optimization:

```
innerprod := 0
i := 1
top: t1 := 4 * i
    t2 := t1(A)
    t4 := t1(B)
    t5 := t2 * t4
    innerprod := t5 +F innerprod
    i := i +T 1
    if i <= n goto top
```

Prof. Necula CS 164

16

## The Example's Performance

- We insert `nop` where the processor stalls to wait for a value
 

```
innerprod := 0          nop
i := 1                  nop
top: t1 := 4 * i         innerprod := t5 +F innerprod
nop                     i := i +T 1
nop                     if i <= n goto top
t2 := t1(A)             *nop
t4 := t1(B)             *nop
nop                     *nop
t5 := t2 * t4
```
- Loop body (7 instr.) takes 15 cycles to execute
  - less than 50% of the potential performance

Prof. Necula CS 164

17

## The Example's Performance (Cont.)

- There is already some parallelism between instructions
  - E.g., `innerprod := t5 +F innerprod` and `i := i +T 1`
- But there are many "bubbles" in the pipeline where the processor is stalled waiting for a previous instruction to complete
- We try to reorder instructions to reduce the number of stalls

Prof. Necula CS 164

18

## An Instruction Scheduling Method

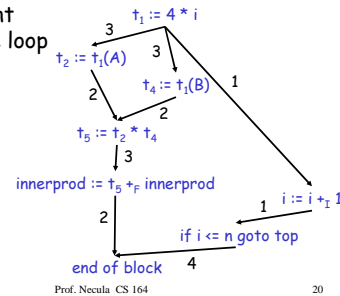
- We will look only at basic-block scheduling
  - An optimal schedule is too expensive to compute
    - We will use heuristics
- Step 1:
- Calculate dependencies between instructions
- Step 2:
- Pick instructions whose dependencies are satisfied

Prof. Necula CS 164

19

## The Dependence Graph. Example.

- The Dependent Graph for the loop body in our example:



Prof. Necula CS 164

20

## The Dependence Graph

- Dependencies between instructions can be shown using a directed graph
  - Each instruction is a node
  - If B reads the output of A and A completes in k cycles
    - draw an edge from A to B with weight k
  - If B writes the input of A
    - draw an edge from A to B with weight 1
  - If B writes the output of A
    - draw an edge from A to B with weight 1

Prof. Necula CS 164

21

## Dependence Graphs Describe Legal Orderings

- If  $A \rightarrow^k B$  in the DG then
  - A must appear before B, and
  - At least k-1 instructions must separate A and B
- Thus,  $t_1 := 4 * i$  must be the first instruction
- Followed by either
  - $t_2 := t_1(A)$  after 2 instructions
  - $t_4 := t_1(B)$  after 2 instructions
  - $i := i + 1$  after 0 instructions
- We'll pick  $i := i + 1$  next

Prof. Necula CS 164

22

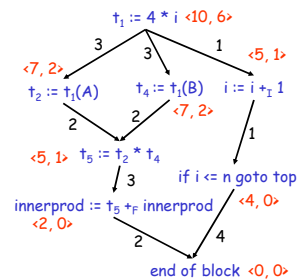
## The Instruction Scheduling Algorithm (I)

- Prioritize instructions according to how early in the computation they should be executed
- Assign to each instruction A a priority  $\langle L, D \rangle$ :
  - L is the weight of the longest path in the DG from A to the end of the block
  - D is the number of instructions depending on A

Prof. Necula CS 164

23

## Computing Priorities. Example.



Prof. Necula CS 164

24

## The Instruction Scheduling Algorithm (II)

- Build a schedule cycle by cycle
- 1. Pick an eligible instruction A such that:
  - It is a root in the current DG, and
  - Its inputs are available in this cycle, and
  - If A is a branch then all unscheduled instructions can complete in the delay slots
  - Among eligible instructions pick that with largest L
  - Break ties in favor of instructions with larger D
- 2. Insert a `nop` if no eligible instr. in this cycle
- 3. Remove A from the DG and repeat from 1

Prof. Necula CS 164

25

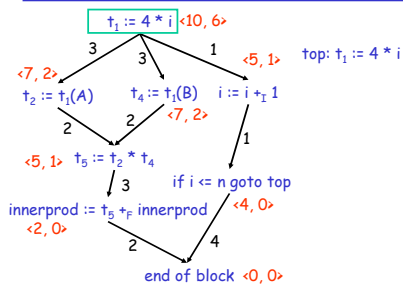
## Intuition

- The most important instructions are those on the critical path (the longest chain of dependencies)
- Delaying instructions on the critical path is likely to result in a longer schedule
- Picking instructions with more dependents will make more instructions eligible later

Prof. Necula CS 164

26

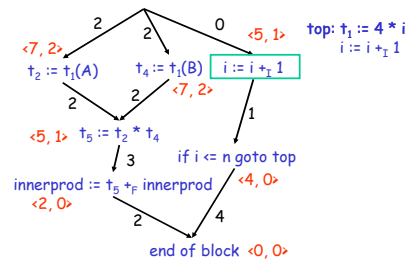
### Constructing the Schedule. Example (1)



Prof. Necula CS 164

27

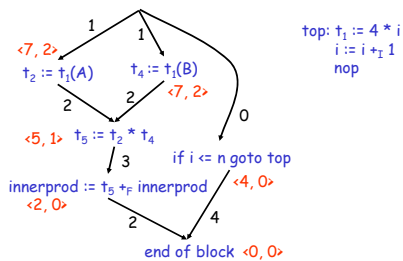
### Constructing the Schedule. Example (2)



Prof. Necula CS 164

28

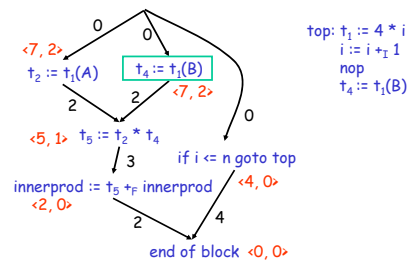
### Constructing the Schedule. Example (3)



Prof. Necula CS 164

29

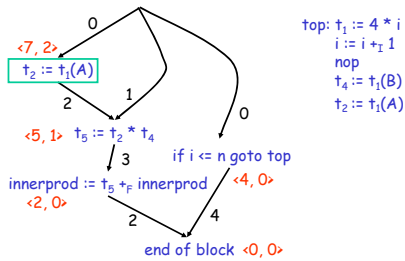
### Constructing the Schedule. Example (4)



Prof. Necula CS 164

30

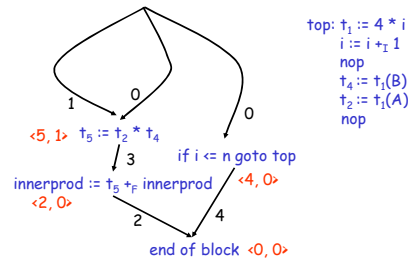
### Constructing the Schedule. Example (5)



Prof. Necula CS 164

31

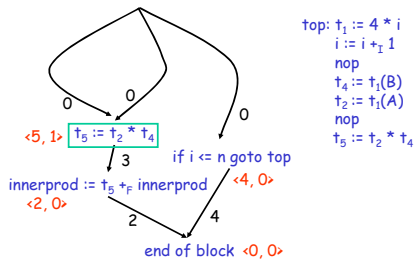
### Constructing the Schedule. Example (6)



Prof. Necula CS 164

32

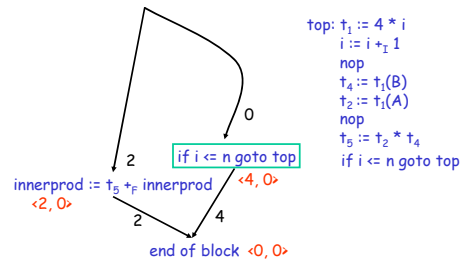
### Constructing the Schedule. Example (7)



Prof. Necula CS 164

33

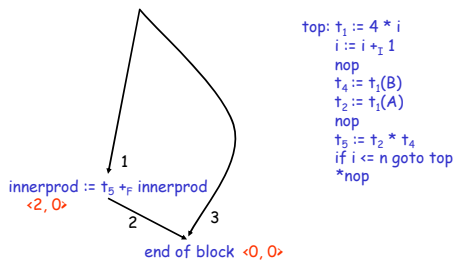
### Constructing the Schedule. Example (8)



Prof. Necula CS 164

34

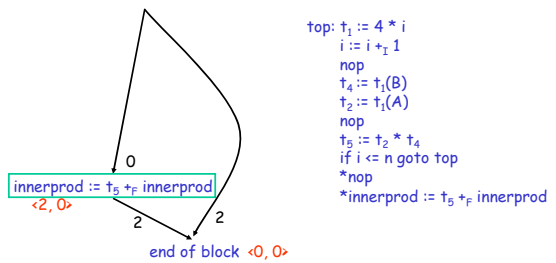
### Constructing the Schedule. Example (9)



Prof. Necula CS 164

35

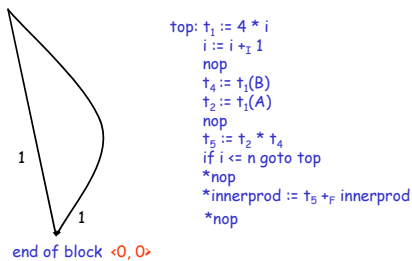
### Constructing the Schedule. Example (10)



Prof. Necula CS 164

36

## Constructing the Schedule. Example (11)



Prof. Necula CS 164

37

## Notes

- Now one iteration takes 11 cycles
  - 26% faster than before
  - 64% utilization of the machine (7 busy cycles)
  - It seems that this is almost the best we can do since  $t_1 := 4 * i$  has a critical path of length 10
- However we can do much better than that...

Prof. Necula CS 164

38

## Loop Unrolling

- The problem with the example is that the basic block is too small
- The scheduler does not have enough instructions to fill the bubbles
- We get a bigger block by unrolling the loop
- Loop unrolling duplicates the loop body and combines two iterations in one

Prof. Necula CS 164

39

## Loop Unrolling. Example.

- The unrolled loop for computing the inner product (assuming  $n$  is even):
 

```

innerprod := 0;
for(i:=1; i<=n; i += 2) {
  innerprod := A[i] * B[i] + innerprod;
  innerprod := A[i+1] * B[i+1] + innerprod;
}
      
```

Prof. Necula CS 164

40

## Loop Unrolling (Cont.)

- In general, to unroll  $k$  times a loop with body  $E$ 
  - Create  $k$  copies of  $E$
  - In the  $j^{\text{th}}$  copy replace iteration variable  $i$  by  $i+j-1$
  - Make the iteration variable step by  $k$
  - Adjust loop termination tests
- Two advantages:
  - It gives the scheduler more instructions
  - Eliminates conditional tests between iterations
- Disadvantage: code size growth

Prof. Necula CS 164

41

## The Example Unrolled

```

innerprod := 0
i := 1
top: t1 := 4 * i
    t2 := t1(A)
    t4 := t1(B)
    t5 := t2 *F t4
    innerprod := t5 +F innerprod
    j := i + 1
    s1 := 4 * j
    s2 := s1(A)
    s4 := s1(B)
    s5 := s2 *F s4
    innerprod := s5 +F innerprod
    i := i + 2
    if i <= n goto top
  
```

- The subexpression  $4 * (i+1)$  is stored in  $s_1$
- In this form one iteration takes  $2 * 15 - 4$  cycles
  - as before, less a branch

Prof. Necula CS 164

42

## A New Schedule

- This loop takes 15 cycles
- Twice as fast as before
- 50% faster than the single iteration schedule
- 86% machine utilization

```
innerprod := 0
i := 1
top: t1 := 4 * i
j := i + 1
s1 := 4 * j
t2 := t1(A)
t4 := t1(B)
s2 := s1(A)
s4 := s1(B)
t5 := t2 *F t4
s5 := s2 *F s4
i := i + 2
innerprod := t5 +F innerprod
if i <= n goto top
*innerprod := s5 +F innerprod
*nop
*nop
```

Prof. Necula CS 164

43

## Conclusions

- Instruction scheduling is useful for modern pipelined and superscalar architectures
  - Otherwise, you waste processor speed
- Typical programs have small basic blocks
  - Limits the effectiveness of instruction scheduler
  - There are global scheduling algorithms, whose cost and complexity are high
- Loop unrolling exposes more opportunities for instruction scheduling

Prof. Necula CS 164

44