

Instruction Scheduling

Adapted from Lectures by
Prof. Alex Aiken and George Necula (UCB)

Lecture Outline

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

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](#)

Instruction Dependencies

- Any two instructions cannot be reordered
- **Resource dependencies**
 - Two instructions that must use the same functional unit cannot execute at once
- **Data dependencies**: A must finish before B if
 - B reads a register written by A
 - Read-after-write dependency
 - B writes a register also written by A
 - Write-after-write dependency
 - B writes a register that A reads (write-after-read)

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

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)



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**

CS781(Prasad)

L29IS

7

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)

CS781(Prasad)

L29IS

8

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

CS781(Prasad)

L29IS

9

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

CS781(Prasad)

L29IS

10

Our Architecture

- Based on Motorola 88xxx but simplified

Operation	Description	Cycles
<code>i := j + k</code>	Integer add	1
<code>i := j + k</code>	Floating point add	2
<code>i := j * k</code>	Multiply (integer or float)	3
<code>i := j(k)</code>	Memory load from <code>j + k</code>	2
<code>if i = j goto L</code>	Branch	4

CS781(Prasad)

L29IS

11

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

CS781(Prasad)

L29IS

12



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

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 + 1
if i <= n goto top
```

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 + 1
if i <= n goto top
```

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

- Loop body (7 instr.) takes 15 cycles to execute
– less than 50% of the potential performance

The Example's Performance (Cont.)

- There is some parallelism between instructions
– E.g., `innerprod := t5 +F innerprod` and `i := i + 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

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

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

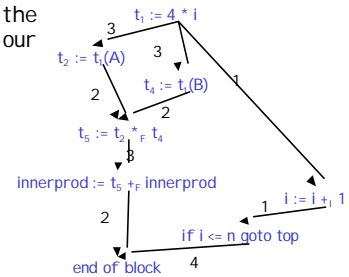
CS781(Prasad)

L29IS

19

The Dependence Graph. Example.

- The DG for the loop body in our example:



CS781(Prasad)

L29IS

20

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

CS781(Prasad)

L29IS

21

The Instruction Scheduling Algorithm (I)

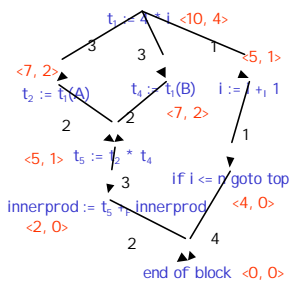
- Prioritize instructions according to how early in the computation they should be executed
- Assign to each instruction A a priority $\langle L, D \rangle$:
 - L is the weight of the longest path in the DG from A to the end of the block
 - D is the number of instructions depending on A

CS781(Prasad)

L29IS

22

Computing Priorities. Example.



CS781(Prasad)

L29IS

23

The Instruction Scheduling Algorithm (II)

- Build a schedule cycle by cycle
 - 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
 - Insert a `nop` if no eligible instr. in this cycle
 - Remove A from the DG and repeat from 1

CS781(Prasad)

L29IS

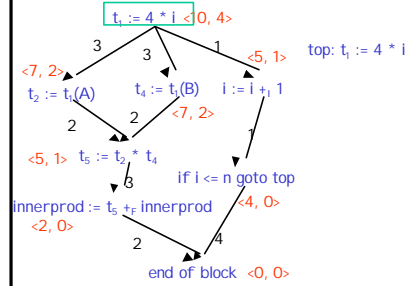
24



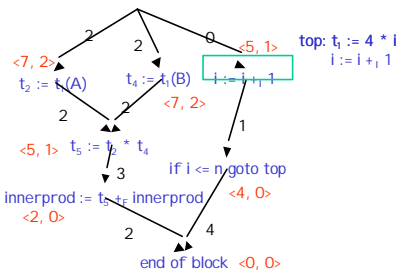
Intuition

- 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

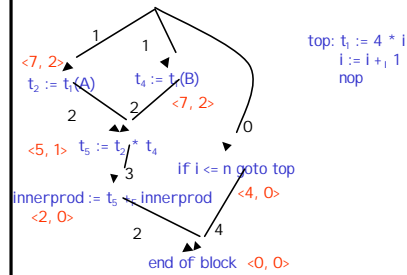
Constructing the Schedule. Example (1)



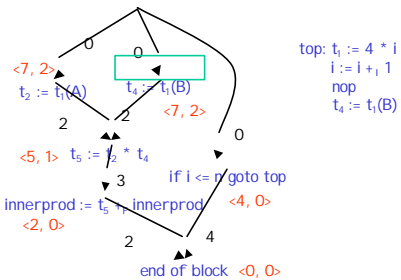
Constructing the Schedule. Example (2)



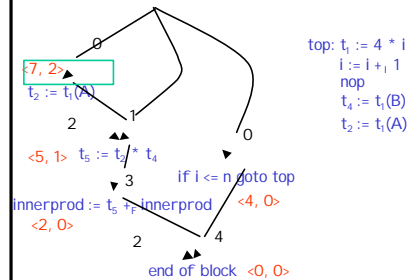
Constructing the Schedule. Example (3)



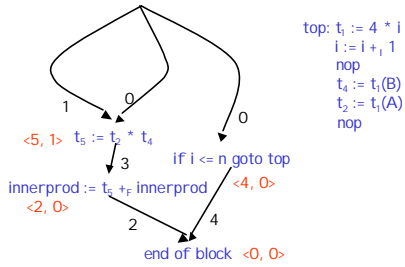
Constructing the Schedule. Example (4)



Constructing the Schedule. Example (5)



Constructing the Schedule. Example (6)



```

top: t1 := 4 * i
     i := i + 1
     nop
     t4 := t1(B)
     t2 := t1(A)
     nop
  
```

```

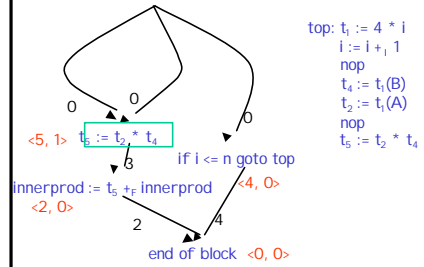
<5, 1> t5 := t2 * t4
innerprod := t5 +F innerprod <4, 0>
<2, 0>
  
```

CS781(Prasad)

L29IS

31

Constructing the Schedule. Example (7)



```

top: t1 := 4 * i
     i := i + 1
     nop
     t4 := t1(B)
     t2 := t1(A)
     nop
     t5 := t2 * t4
  
```

```

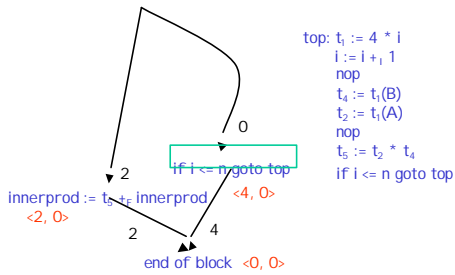
innerprod := t5 +F innerprod <4, 0>
<2, 0>
  
```

CS781(Prasad)

L29IS

32

Constructing the Schedule. Example (8)



```

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

```

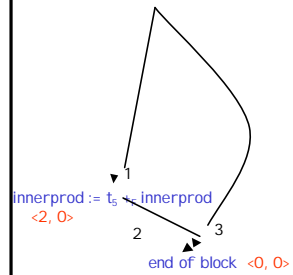
innerprod := t5 +F innerprod <4, 0>
<2, 0>
  
```

CS781(Prasad)

L29IS

33

Constructing the Schedule. Example (9)



```

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

```

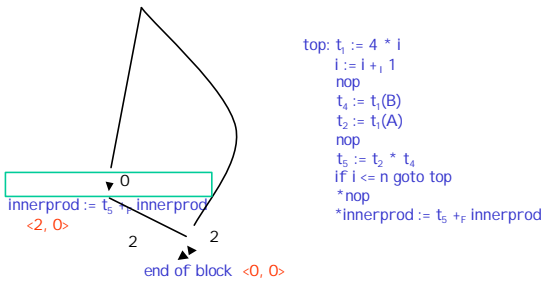
innerprod := t5 +F innerprod
<2, 0>
  
```

CS781(Prasad)

L29IS

34

Constructing the Schedule. Example (10)



```

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

```

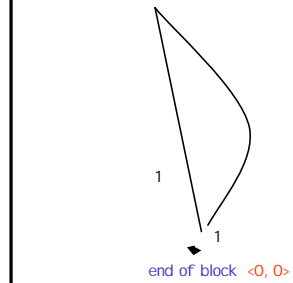
innerprod := t5 +F innerprod
<2, 0>
  
```

CS781(Prasad)

L29IS

35

Constructing the Schedule. Example (11)



```

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

```

innerprod := t5 +F innerprod
<1, 0>
  
```

CS781(Prasad)

L29IS

36



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 much better than that...

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

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;
}
```

Loop Unrolling (Cont.)

- In general, to unroll k times a loop with body E
 - Create k copies of E
 - In the j^{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

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

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
```

Conclusions

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