

Instruction Scheduling

Last time

- Instruction scheduling using list scheduling

Today

- Improvements on list scheduling
 - Register renaming
 - Unrolling
- Software pipelining

Improving Instruction Scheduling

Techniques

- Register renaming
 - Scheduling loads
 - Loop unrolling
 - Software pipelining
 - Predication and speculation
- } Deal with data hazards
- } Deal with control hazards

Register Renaming

Idea

- Reduce false data dependencies by reducing register reuse
- Give the instruction scheduler greater freedom

Example

```
add $r1, $r2, 1      add $r1, $r2, 1
st  $r1, [$fp+52]    st  $r1, [$fp+52]
mul $r1, $r3, 2      mul $r11, $r3, 2
st  $r1, [$fp+40]    st  $r11, [$fp+40]
```

```
add $r1, $r2, 1
mul $r11, $r3, 2
st  $r1, [$fp+52]
st  $r11, [$fp+40]
```

Scheduling Loads

Reality

- Loads can take many cycles (slow caches, cache misses)
- Many cycles may be wasted

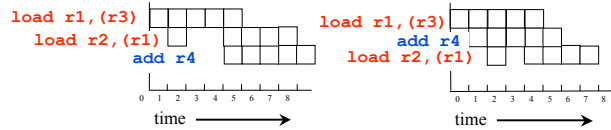
Most modern architectures provide non-blocking (delayed) loads

- Loads never stall
- Instead, the use of a register stalls if the value is not yet available
- Scheduler should try to place loads well before the use of target register

Scheduling Loads (cont)

Hiding latency

- Place independent instructions behind loads



- How many instructions should we insert?
 - Depends on latency
 - Difference between cache miss and cache hits are growing
 - If we underestimate latency: Stall waiting for the load
 - If we overestimate latency: Hold register longer than necessary
- Wasted parallelism

Loop Unrolling

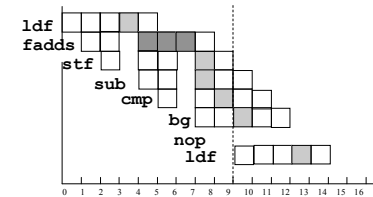
Idea

- Replicate body of loop and iterate fewer times
- Reduces loop overhead (test and branch)
- Creates larger loop body \Rightarrow more scheduling freedom

Example

```
L: ldf f0, [r1]
   fadds f2, f0, f1
   stf [r1], f2
   sub r1, 4, r1
   cmp r1, 0
   bg L
   nop
```

Loop overhead

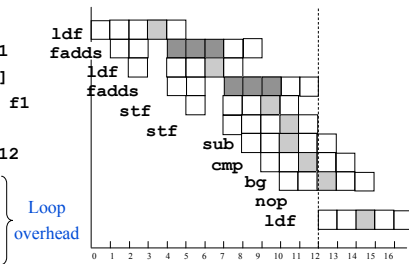


Cycles per iteration: 9

Loop Unrolling Example

Sample loop

```
L: ldf f0, [r1]
   fadds f2, f0, f1
   ldf f10, [r1-4]
   fadds f12, f10, f1
   stf [r1], f2
   stf [r1-4], f12
   sub r1, 8, r1
   cmp r1, 0
   bg L
   nop
```



Cycles per iteration: $12/2 = 6$
(1.5 speedup!)

The larger window lets us hide some of the latency of the `fadds` instruction

Loop Unrolling Summary

Benefit

- Loop unrolling allows us to schedule code across iteration boundaries, providing more scheduling freedom

Issues

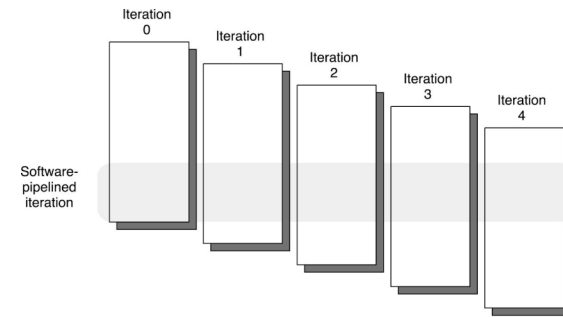
- How much unrolling should we do?
 - Try various unrolling factors and see which provides the best schedule?
 - Unroll as much as possible within a code expansion budget?
- An alternative: **Software pipelining**

Software Pipelining

Basic idea

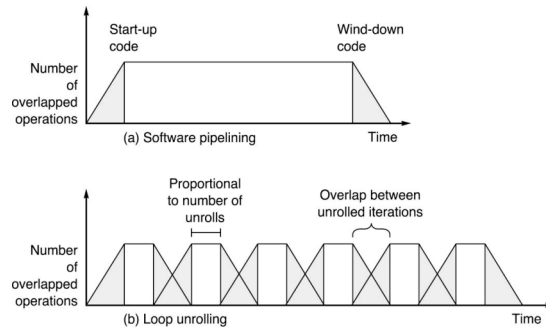
- **Software pipelining** is a systematic approach to scheduling across iteration boundaries without doing loop unrolling
- Try to move the long latency instructions to **previous** iterations of the loop
- Use independent instructions to hide their latency
- Three parts of a software pipeline
 - **Kernel:** Steady state execution of the pipeline
 - **Prologue:** Code to fill the pipeline
 - **Epilogue:** Code to empty the pipeline

Visualizing Software Pipelining



© 2003 Elsevier Science (USA). All rights reserved.

Software Pipelining versus Loop Unrolling

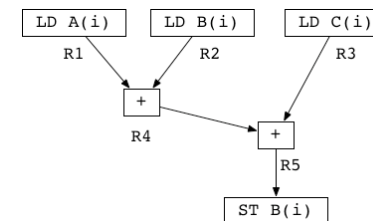


© 2003 Elsevier Science (USA). All rights reserved.

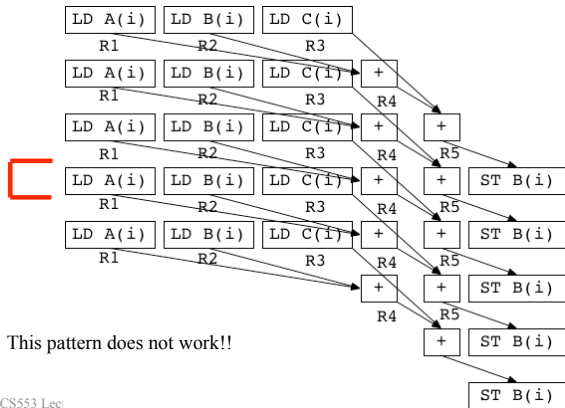
SW Pipelining (Step 1: Construct DAG and Assign Registers)

```
int A[100], B[100], C[100];
for (i=0; i<100; i++) {
    B[i] = A[i] + B[i] + C[i];
}
```

Example assumes infinite functional units and single-cycle latency.



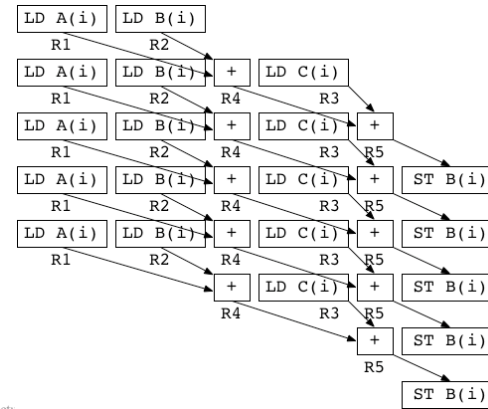
SW Pipelining (Step 2: "Unroll", Satisfy Latencies, Find Pattern)



CS553 Lec

14

SW Pipelining (Step 3: Satisfy register constraints)



CS553 Lecture

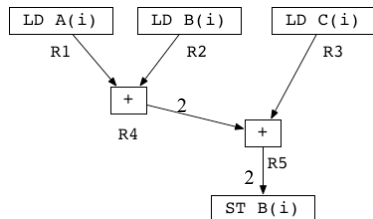
INSTRUCTION SCHEDULING II

15

SW Pipelining (Step 1: Construct DAG and Assign Registers)

```
int A[100], B[100], C[100];
for (i=0; i<100; i++) {
    B[i] = A[i] + B[i] + C[i];
}
```

Example assumes pg. 739 machine.

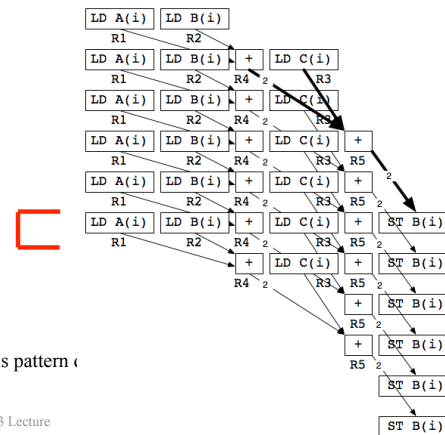


CS553 Lecture

Instruction Scheduling II

16

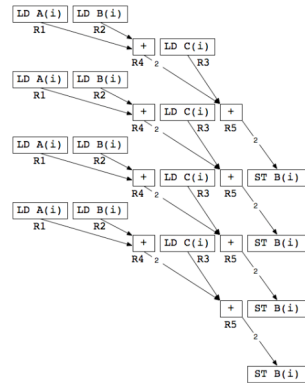
SW Pipelining (Step 2: "Unroll", Satisfy Latencies, Find Pattern)



CS553 Lecture

17

SW Pipelining (Step 3: Satisfy register constraints)



SW Pipelining and Loop Unrolling Summary

Unrolling removes branching overhead and helps tolerate data dependence latency

SW pipelining maintains max parallelism in steady state through continuous tolerance of data dependence latency

Both work best with loops that are parallel, getting ILP by taking instructions from different iterations

Software Pipelining

Complications

- What if there is control flow within the loop?
 - Use control-flow profiles to identify most frequent path through the loop
 - Optimize for the most frequent path
- How do we identify the most frequent path?
 - Profiling

Concepts

Improving instruction scheduling

- Register renaming
- Scheduling loads
- Loop unrolling

Instruction scheduling across basic blocks

- Software pipelining

Next Time

Lecture

- Data-flow analysis