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

} Deal with data hazards

- Loop unrolling
- Software pipelining
- Predication and speculation (next week)

} Deal with control hazards

Register Renaming

Idea

- Reduce false data dependences 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

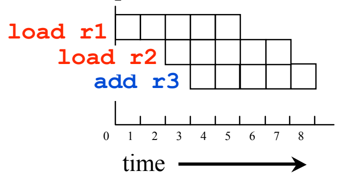
Most modern architectures provide non-blocking (delayed) loads

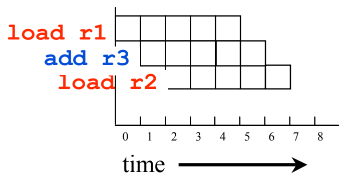
- Loads never stall
- Instead, the use of a register stalls if the value is not yet available
- Scheduler should try to place loads well before the use of target register

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

Balanced Scheduling [Kerns and Eggers'92]

Idea

- Impossible to know the latencies statically
- Instead of estimating latency, balance the ILP (instruction-level parallelism) across all loads
- Schedule for characteristics of the code instead of for characteristics of the machine

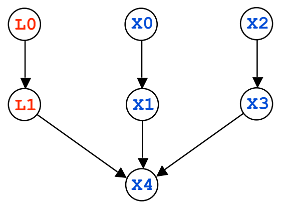
Balancing load

- Compute **load level parallelism**

$$\text{LLP} = 1 + \frac{\text{\# independent instructions}}{\text{\# of loads that can use this parallelism}}$$

Balanced Scheduling Example

Example



LLP for L0 = 1+4/2 = 3

LLP for L1 = 1+4/2 = 3

list scheduling		balanced scheduling
w=5	w=1	
L0	L0	L0
X0	L1	X0
X1	X0	X1
X2	X1	L1
X3	X2	X2
L1	X3	X3
X4	X4	X4

Pessimistic   Optimistic

Loop Unrolling

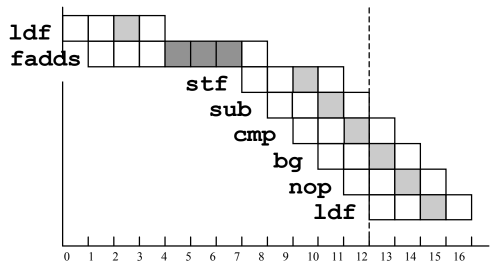
Idea

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

Example

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

Loop overhead



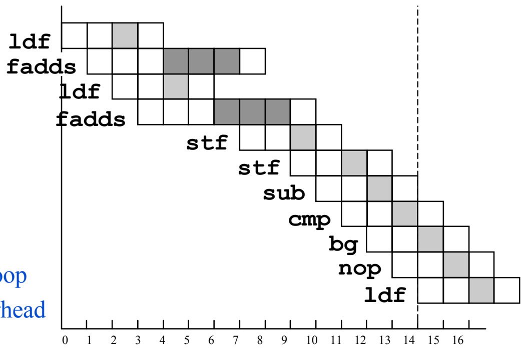
Cycles per iteration: 12

Loop Unrolling Example

Sample loop

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

Loop overhead



Cycles per iteration: 14/2 = 7
(71% 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**

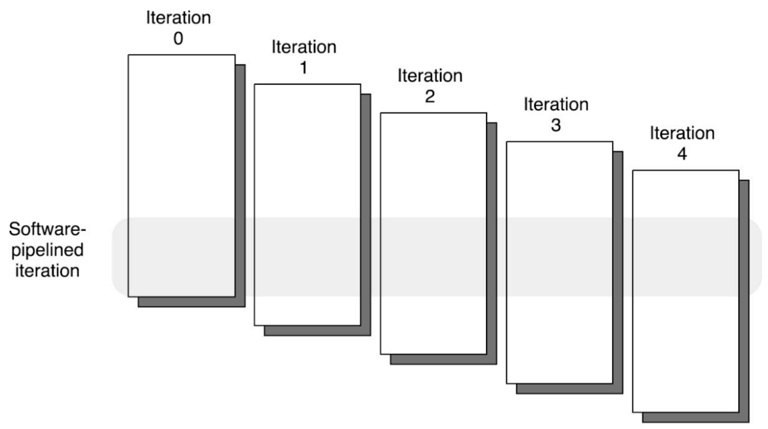
Software Pipelining

Basic idea

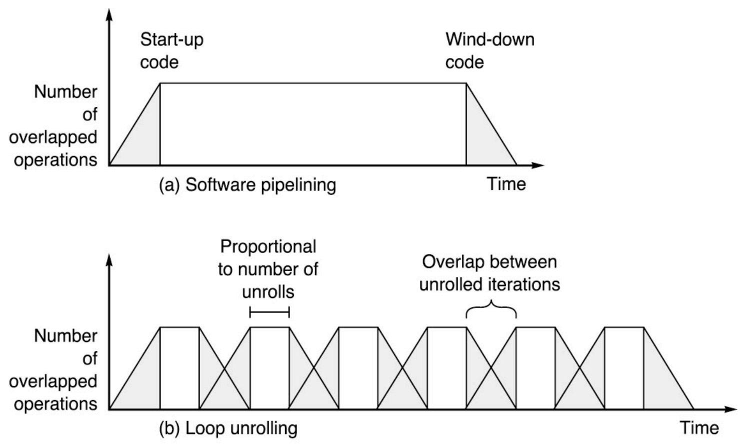
- **Software pipelining** is a systematic approach to scheduling across iteration boundaries without doing loop unrolling
- Try to move the long latency instructions to **previous** iterations of the loop
- Use independent instructions to hide their latency

- 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



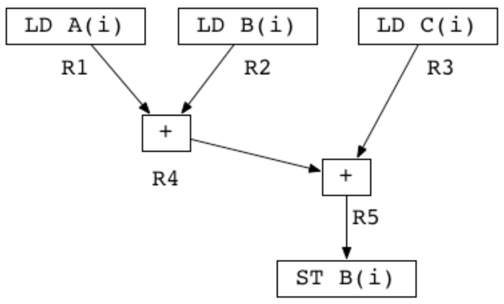
Software Pipelining versus Loop Unrolling



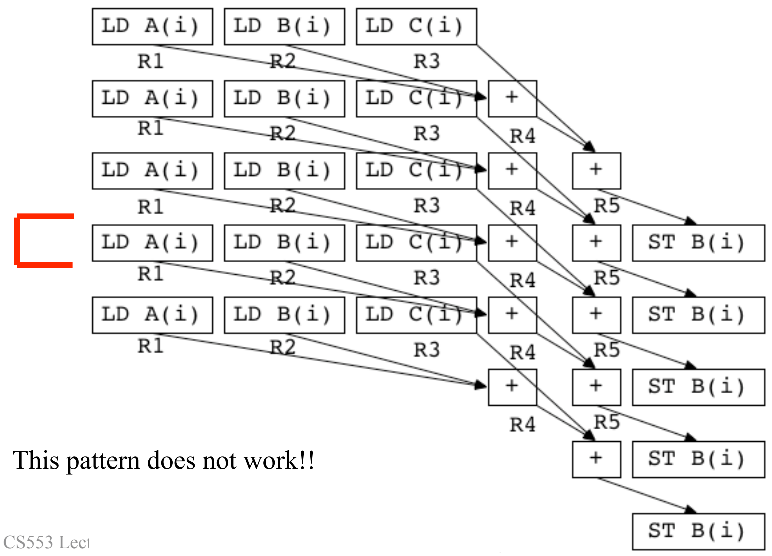
Start-up code
Wind-down code
Number of overlapped operations
(a) Software pipelining
Time
Proportional to number of unrolls
Overlap between unrolled iterations
Number of overlapped operations
(b) Loop unrolling
Time

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];
}
```

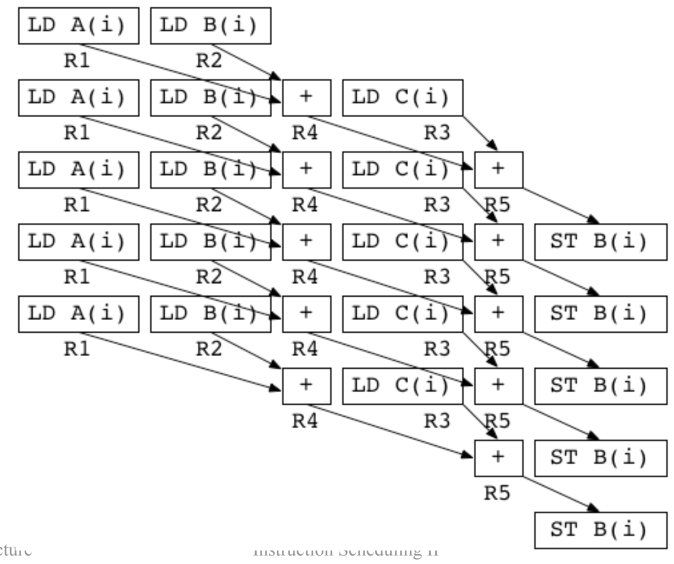


SW Pipelining (Step 2: "Unroll", Schedule, Find Pattern)



This pattern does not work!!

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
- Balanced load scheduling
- Loop unrolling

Instruction scheduling across basic blocks

- Software pipelining

Next Time

Lecture

- More instruction scheduling
  - profiling
  - trace scheduling