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 (next week)

Deal with data hazards
Deal with control hazards
Register Renaming

Idea

– Reduce false data dependences by reducing register reuse
– Give the instruction scheduler greater freedom

Example

\[
\begin{align*}
\text{add} & \quad $r1, $r2, 1 & \quad \text{add} & \quad $r1, $r2, 1 \\
\text{st} & \quad $r1, \left[$fp+52\right] & \quad \text{st} & \quad $r1, \left[$fp+52\right] \\
\text{mul} & \quad $r1, $r3, 2 & \quad \text{mul} & \quad $r11, $r3, 2 \\
\text{st} & \quad $r1, \left[$fp+40\right] & \quad \text{st} & \quad $r11, \left[$fp+40\right]
\end{align*}
\]

\[
\begin{align*}
\text{add} & \quad $r1, $r2, 1 \\
\text{mul} & \quad $r11, $r3, 2 \\
\text{st} & \quad $r1, \left[$fp+52\right] \\
\text{st} & \quad $r11, \left[$fp+40\right]
\end{align*}
\]

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
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

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

Balanced Scheduling (cont)
Balanced Scheduling Example

Example

<table>
<thead>
<tr>
<th>list scheduling</th>
<th>balanced scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>w=5</td>
<td>w=1</td>
</tr>
<tr>
<td>L0</td>
<td>L0</td>
</tr>
<tr>
<td>X0</td>
<td>L0</td>
</tr>
<tr>
<td>X1</td>
<td>X0</td>
</tr>
<tr>
<td>X2</td>
<td>X1</td>
</tr>
<tr>
<td>X3</td>
<td>X2</td>
</tr>
<tr>
<td>X4</td>
<td>X3</td>
</tr>
</tbody>
</table>

Pessimistic

Optimistic

LLP for L0 = 1+4/2 = 3
LLP for L1 = 1+4/2 = 3

Loop Unrolling

Idea

\- Replicate body of loop and iterate fewer times
\- Reduces loop overhead (test and branch)
\- Creates larger loop body \iff 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

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

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

![Diagram of software pipelining]

© 2003 Elsevier Science (USA). All rights reserved.
Software Pipelining versus Loop Unrolling

(a) Software pipelining

(b) Loop unrolling

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

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

```
LD A(i) ─── R1
LD B(i) ─── R2
LD C(i) ─── R3

+ ─── R4
    └── R5

ST B(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