Static Scheduling Techniques

- Local scheduling (within a basic block)
- Loop unrolling
- Software pipelining (modulo scheduling)
- Trace scheduling
- Predication

Loop Unrolling (1 of 2)

Unroll the loop (e.g., four times)

Loop: L.D   F0, 0(R1)
ADD.D F4, F0, F2
S.D   F4, 0(R1)
ADDI  R1, R1, #8
BNE   R1, xxx , Loop
Loop: L.D   F0, 0(R1)
ADD.D F4, F0, F2
S.D   F4, 0(R1)
L.D   F6, 8(R1)
ADD.D F8, F6, F2
S.D   F8, 8(R1)
L.D   F10, 16(R1)
ADD.D F12, F10, F2
S.D   F12, 16(R1)
L.D   F14, 24(R1)
ADD.D F16, F14, F2
S.D   F16, 24(R1)
ADDI  R1, R1, #32
BNE   R1, xxx , Loop

More registers needed!

Benefits:
- Less dynamic instruction overhead (ADD/BNE)
- Fewer branches ⇒ larger basic block: can reschedule operations more easily

Loop Unrolling (2 of 2)

Positives/negatives

(+): Larger block for scheduling
(+): Reduces branch frequency
(+): Reduces dynamic instruction count (loop overhead)
(-): Expands code size
(-): Have to handle excess iterations ("strip-mining")
**Strip-mining**

- Suppose that our loop requires 10 iterations, and we can unroll only 4 of the iterations.
- If we go through our unrolled loop 3 times, how many iterations (of the original loop) would we be performing?
- If we go through our unrolled loop twice, how many iterations (of the original loop) would we be performing?
- So, we need to add some prelude code, e.g.,

```
<table>
<thead>
<tr>
<th>Iteration</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D F0, 0(R1)</td>
</tr>
<tr>
<td>2</td>
<td>ADD.D F4, F0, F2</td>
</tr>
<tr>
<td>3</td>
<td>S.D F4, 0(R1)</td>
</tr>
<tr>
<td>4</td>
<td>L.D F6, 8(R1)</td>
</tr>
<tr>
<td>5</td>
<td>ADD.D F8, F6, F2</td>
</tr>
<tr>
<td>6</td>
<td>S.D F8, 8(R1)</td>
</tr>
<tr>
<td>7</td>
<td>ADD R1, R1, 16</td>
</tr>
<tr>
<td>8</td>
<td>BNE R1, xxx, Prel</td>
</tr>
</tbody>
</table>
```

- So, we need to add some prelude code, e.g.,

**Static Scheduling Techniques**

- Local scheduling (within a basic block)
- Loop unrolling
- Software pipelining (modulo scheduling)
- Trace scheduling
- Predication

**Software Pipelining** (1 of 5)

- Symbolic loop unrolling
  - The instructions in a loop are taken from different iterations in the original loop.
  
- This slide borrowed from Per Stenström, Chalmers University, Stockholm, Sweden, Computer Architecture, Lecture 6.
Software Pipelining  (2 of 5)

- Treat dependent operations in a loop as a pipeline
  - LD[i] → ADDD[i] → SD[i]
  - Hide latencies by placing different stages in successive iterations
  - LD[i] goes in first iteration
  - ADDD[i] goes in second iteration
  - SD[i] goes in third iteration
  - Stated another way, LD[i], ADDD[i–1], SD[i–2] go into the same iteration

```
for (i=1; i<=N; i++) {
// a[i] = a[i] + k;
load a[i]
add   a[i]
store a[i]
}
```

```
START-UP-BLOCK
for (i=3; i<=N; i++) {
load  a[i]
add   a[i-1]
store a[i-2]
}
```

```
FINISH-UP-BLOCK
```

-Hide latencies by placing different stages in successive iterations

◊ LD[i] goes in first iteration
◊ ADDD[i] goes in second iteration
◊ SD[i] goes in third iteration
◊ Stated another way, LD[i], ADDD[i–1], SD[i–2] go into the same iteration

Software Pipelining  (3 of 5)

- Assembly-code version

```
Loop: L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
ADDI R1, R1, #8
BNE R1, xxx , Loop
```

```
START: L.D F0, 0(R1)
ADDI R1, R1, #8
```

```
FINISH: S.D F4, -16(R1)
ADD.D F4, F0, F2
ADD.D F4, F0, F2
L.D F0, 0(R1)
ADDI R1, R1, #8
BNE R1, xxx , Loop
```

Instructions from 3 consecutive iterations form the loop body:

- No data dependences within a loop iteration
- The dependence distance is 2 iterations
- WAR hazard elimination is needed
- Requires startup and finish code

This slide borrowed from Per Stenström, Chalmers University, Stockholm, Sweden, Computer Architecture, Lecture 6.
Software Pipelining (5 of 5)

- Positives/negatives
  - (+) No dependences in loop body
  - (+) Same effect as loop unrolling (hide latencies), but don’t need to replicate iterations (code size)
  - (-) Still have extra code for prologue/epilogue (pipeline fill/drain)
  - (-) Does not reduce branch frequency

Static Scheduling Techniques

- Local scheduling (within a basic block)
- Loop unrolling
- Software pipelining (modulo scheduling)
- Trace scheduling
- Predication

Trace Scheduling (1 of 3)

Creates a sequence of instructions that are likely to be executed—a trace.

Two steps:
- **Trace selection**: Find a likely sequence of basic blocks (trace) across statically predicted branches (e.g. if-then-else).
- **Trace compaction**: Schedule the trace to be as efficient as possible while preserving correctness in the case the prediction is wrong.

This slide and the next one have been borrowed from Per Stenström, Chalmers University, Stockholm, Sweden, Computer Architecture, Lecture 6.
Trace Scheduling

The leftmost sequence is chosen as the most likely trace.

The assignment to B is control-dependent on the if statement.

Trace compaction has to respect data dependencies.

The rightmost (less likely) trace has to be augmented with fixup code.

Trace Scheduling

Select most common path – a trace

- Use profiling to select a trace
- Allows global scheduling, i.e., scheduling across branches
- This is speculation because schedule assumes certain path through region
- If trace is wrong (other paths taken), execute repair code
- Efficient static branch prediction key to success
- Yields more instruction-level parallelism

Trace to be scheduled:

Original code:

\[ b[i] = \text{old value} \]
\[ a[i] = b[i] + c[i] \]
\[ \text{if} \ (a[i] \neq 0) \ \text{goto A} \]

Trace:

\[ b[i] = \text{new value} \]
\[ c[i] = \text{if} \ (a[i] = 0) \ \text{then} \]
\[ a[i] = b[i] + c[i] \]
\[ X \]

Repair code:

\[ b[i] = \text{old value} \]
\[ a[i] = b[i] + c[i] \]
\[ \text{if} \ (a[i] = 0) \ \text{then} \]
\[ b[i] = \text{new value} \]
\[ c[i] = \text{maybe calculate } c[i] \]
\[ \text{goto B} \]

Static Scheduling Techniques

- Local scheduling (within a basic block)
- Loop unrolling
- Software pipelining (modulo scheduling)
- Trace scheduling
- Predication
Predication (1 of 3)

- Conditional or *predicated* instructions
  - Executed only if a condition is satisfied
  - Control converted into data dependences

**Example:**

```plaintext
Normal code          Conditional
SNEZ     R1, L       CMOVZ R2, R3, R1
MOV      R2, R3
L:
```

- Useful for short if-then statements
- More complex; might slow down cycle time

This slide borrowed from Per Stenström, Chalmers University, Stockholm, Sweden, Computer Architecture, Lecture 6.

Predication (2 of 3)

- ISA role
  - Provide predicate registers
  - Provide predicate-setting instructions (e.g., compare)
  - Subset of opcodes can be guarded with predicates
- Compiler role
  - Replace branch with predicate computation
  - Guard alternate paths with (predicate) and (!predicate)
- Hardware role
  - Execute all predicated code, i.e., both paths after a branch
  - Do not commit either path until predicate is known
  - Conditionally commit or quash, depending on predicate

Original code        Predicated code
b[i] = old value;    b[i] = old value
b[i] = b[i] + c[i]    !b[i] = old value
if (b[i] = 0) then
  a[i] = b[i] + c[i] // remove case
else
  a[i] = b[i] + c[i] // remove case
end

Predicated code
b[i] = new value
!b[i] = new value
// common case

Predicated code
b[i] = new value
// common case

Predication (3 of 3)

- Positives/negatives
  - (+) Larger scheduling scope without speculation
  - (+) ISA extensions: opcode pressure, extra register specifier
  - (+) May degrade performance if over-commit fetch/execute resources
  - (+) Convert control dependence to data dependence
    - Does this really fix the branch problem?
    - Not predicting control flow delays resolving register dependencies
    - Can lengthen schedule w.r.t. trace scheduling
- Above discussion is simplified
  - Predication issues are much, much more complex ...
  - Complicating matters:
    - Software: Trace scheduling, predication, superblocks, hyperblocks, treegions, ...
    - Hardware: Selective multi-path execution, control independence, ...