Lecture 8: Compiling for ILP and Branch Prediction

Kunle Olukotun
Gates 302
kunle@ogun.stanford.edu

http://www-leland.stanford.edu/class/ee282h/

Advanced pipelining and instruction level parallelism

- Gcc
  - 17% control transfer
  - One branch every six instructions
  - Must look beyond basic block for ILP
- Loop level parallelism
  - Good opportunity
  - Software schemes
  - Hardware schemes
Loop level parallelism: A simple case

- double a[100], b[100], c = 10;
  for (i=0; i<100; i++)  a[i] = b[i] + c;
- The computations for each loop are independent: no references from i to i+1 or i-1 ("loop-carried dependencies")
- A "reasonably clever" compiler would produce:

```
loop:  ld   f0, 0(r1)  ;load vector element from b
       addd f4, f2, f0  ;add c from f2
       sd   f4, 0(r2)   ;store result in a
       addu r1, r1, #8  ;advance b pointer
       addu r2, r2, #8  ;advance a pointer
       bltu r2, r3, loop ;compare a pointer to end
```

Assume the operation latencies are as follows:
- LD:  2 cycles
- ADDD: 4 cycles
- Branch: 2 cycles
- Otherwise: 1 cycle

So that loop executes as follows:
```
loop:  ld   f0, 0(r1)  ;load vector element from b
       <stall>
       addd f4, f2, f0  ;add c from f2
       <stall>
       <stall>
       <stall>
       sd   f4, 0(r2)   ;store result in a
       addu r1, r1, #8  ;advance b pointer
       addu r2, r2, #8  ;advance a pointer
       bltu r2, r3, loop ;compare a pointer to end
       <stall>
```

A total of 11 cycles per iteration
Loop level parallelism:
Instruction scheduling

- First, by scheduling the store into the branch delay slot we get:

```
loop:   ld    f0, 0(r1) ; load vector element from b
       addd f4, f2, f0 ; add c from f2
       addu r1, r1, #8 ; advance b pointer
       addu r2, r2, #8 ; advance a pointer
       bltu r2, r3, loop ; compare a pointer to end
       sd    f4, -8(r2) ; store result in a
```

» Now 7 cycles per iteration

Compiling for ILP

```
max = a[0];
for(i=1;i<n;i++) {
    if(a[i] > max) max = a[i];
}
return max;
```

```
LOOP:   LW R1, R2;      // a[i]
       SGT R3, R1, R8;  // a[i] > max
       BEQZ R3, NOMAX
       ADDI R8, R1, #0;  // update max
       NOMAX:ADDI R2, R2, #4; // update a[i] ptr
       ADDI R4, R4, #1;  // update i
       SLT R5, R4, R9;   // i < n
       BNEZ R5, LOOP
```

Can only reschedule code inside a basic block.
Small basic blocks limit opportunities for scheduling
Compiling for ILP

```plaintext
max = a[0];
for(i=1;i<n;i++) {
    if(a[i] > max) max = a[i];
}
return max;
```

```
<table>
<thead>
<tr>
<th>Location</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP:</td>
<td>LW R1, R2;</td>
<td>a[i]</td>
</tr>
<tr>
<td></td>
<td>LW R11, 4(R2)</td>
<td>a[i+1]</td>
</tr>
<tr>
<td></td>
<td>SGT R3, R1, R8;</td>
<td>a[i] &gt; max</td>
</tr>
<tr>
<td>IF(R3)</td>
<td>ADDI R8, R1, #0;</td>
<td>update max</td>
</tr>
<tr>
<td></td>
<td>ADDI R2, R2, #4;</td>
<td>update a[i] ptr</td>
</tr>
<tr>
<td></td>
<td>ADDI R4, R4, #1;</td>
<td>update i</td>
</tr>
<tr>
<td></td>
<td>SLT R5, R4, R9;</td>
<td>i &lt; n</td>
</tr>
<tr>
<td></td>
<td>BNEZ R5, LOOP</td>
<td></td>
</tr>
</tbody>
</table>

LOOP: LW R1, R2;  // a[i]
ADDI R4, R4, #1; // update i
ADDI R2, R2, #4; // update a[i] ptr
SGT R3, R1, R8;  // a[i] > max
SLT R5, R4, R9;  // i < n
IF(R3) ADDI R8, R1, #0; // update max
BNEZ R5, LOOP | Reschedule to eliminate stalls
```

**Predicate conditional to make this all one basic block**

```
<table>
<thead>
<tr>
<th>Location</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP:</td>
<td>LW R1, R2;</td>
<td>a[i]</td>
</tr>
<tr>
<td></td>
<td>LW R11, 4(R2)</td>
<td>a[i+1]</td>
</tr>
<tr>
<td></td>
<td>SGT R3, R1, R8;</td>
<td>a[i] &gt; max</td>
</tr>
<tr>
<td>IF(R3)</td>
<td>ADDI R2, R2, #8;</td>
<td>update a[i] ptr</td>
</tr>
<tr>
<td></td>
<td>ADDI R8, R1, #0;</td>
<td>update max</td>
</tr>
<tr>
<td></td>
<td>SGT R12, R1, R8;</td>
<td>a[i+1] &gt; max</td>
</tr>
<tr>
<td>IF(R12)</td>
<td>ADDI R8, R11, #0;</td>
<td>update max</td>
</tr>
<tr>
<td></td>
<td>ADDI R4, R4, #2;</td>
<td>update i</td>
</tr>
<tr>
<td></td>
<td>SLT R5, R4, R9;</td>
<td>i &lt; n - 1</td>
</tr>
<tr>
<td></td>
<td>BNEZ R5, LOOP</td>
<td></td>
</tr>
</tbody>
</table>
```

**Unroll the loop to make an even bigger basic block**

- More opportunities for parallelism
- Lower loop overhead (6+4 vs 2(3+4))

This loop has a loop-carried dependence through `max`, more parallelism if loop is not serial.
Compiling for ILP

```c
for(i=0;i<n;i++) {
    d[i] = a[i]*b[i] + c;
}
```

No loop-carried dependencies
Can convert data (loop) parallelism into ILP

```
LOOP:    LD    F0, R1 ;  // a[i]
          LD    F2, R2 ;  // b[i]
          ADDI   R1, #8 ;
          ADDI   R2, #8 ;
          MULTD F4, F0, F2 ;  // a[i] * b[i]
          ADDD   F6, F4, F8 ;  // +c
          SD     F8, R3 ;  // d[i]
          ADDI   R3, #8 ;
          ADDI   R4, #1 ;  // increment i
          SLT    R5, R4, R6 ;  // i<n
          BNEZ   R5, LOOP ;
```

Unrolled Loop

```
LOOP:    LD    F0, R1 ;  // a[i]
          LD    F2, R2 ;  // b[i]
          LD    F4, 8(R1) ;  // a[i+1]
          LD    F6, 8(R2) ;  // b[i+1]
          ADDI   R1, #16 ;
          ADDI   R2, #16 ;
          MULTD F8, F0, F2 ;  // a[i] * b[i]
          MULTD F10, F4, F6 ;  // a[i+1] * b[i+1]
          ADDD   F12, F8, F16 ;  // +c
          ADDD   F14, F10, F16 ;  // +c
          SD     F12, R3 ;  // d[i]
          SD     F14, 8(R3) ;  // d[i]
          ADDI   R3, #16 ;
          ADDI   R4, #2 ;  // increment i
          SLT    R5, R4, R6 ;  // i<n- 1
          BNEZ   R5, LOOP ;
```

Reduced overhead
10+6 vs 2(5+6)
More parallelism and slack
More registers
Bigger basic blocks
Can unroll further
Compiler perspectives on code movement

- When is it safe for compiler to make changes?
- Compiler concerned about dependencies in program, whether or not hazards result depends on the pipeline
- (True) Data dependencies (potential for RAW hazards)
  - instruction i produces a result used by instruction j
  - instruction j is data dependent on instruction k and instruction k is dependent on instruction i
- Easy to determine for registers
  - fixed names
- Hard for memory
  - what is effective address at compile time?
  - does 100(R4) = 20(R6)?
  - does 20(R6) = 20(R6) for different loop iterations?

Named Dependencies

- Two instructions use same name but don’t exchange data
- Anti-dependence (potential for WAR hazards)
  - instruction i reads from a register or memory location that instruction j writes
- Output dependence (potential for WAW hazards)
  - instruction i and instruction j both write the same register or memory location
- Again easy for registers and hard for memory
- Example
  - How did we know that 8(r1) != -16(r2)?
Control Dependencies

if (c1) {s1;}
if (c2) {s2;}

- S1 is control dependent on c1 and s2 is control dependent on c2 but not on c1
- Constraints on control dependencies
  - an instruction that is control dependent on a branch can’t be moved before the branch so its execution is not controlled by the branch
  - an instruction that is not control dependent on a branch can’t be moved after the branch so its execution is controlled by the branch
- Can relax control dependencies to get more parallelism, must preserve exception behavior and dataflow

Advanced Branch Prediction

- Reducing branch penalties is more critical as:
  - effect of data hazards becomes smaller through scheduling
  - pipeline gets deeper
  - we move to multiple-issue instruction scheduling
  - non-numerical applications with small basic blocks (between branches)
- Static branch prediction techniques
  - Branch delay slot
    - Assume taken, or assume not-taken
      - Perhaps: backward taken (loops), forward not-taken (if-then-else)
  - Have the compiler indicate which is likely
  - Gather statistics from previous executions
- Dynamic branch prediction techniques
  - Branch history table
  - Branch target buffer
- Prediction performance = f(accuracy, cost of misprediction)
Branch History Table

- Dynamically record the taken/not-taken history of particular branches
- Simplest BHT scheme
  - lower bits of PC address index table of 1-bit values
  - says whether or not branch taken last time
- Problem:
  - in a loop 1-bit BHT will cause 2 mispredictions
  - when branch exits the loop: predicts taken
  - first time through loop on next time through code: predicts not-taken
- Solution:
  - 2-bit scheme where change prediction if mispredict twice

```
T  T  T  T
  0  1  2  3
```

```
Predict not taken Predict not taken
```

T=Taken  NT=Not Taken

Branch History Table Organization

- Tag
- Taken count
- low bits
- high bits
- Branch address
- hit/miss
- Counter
Accuracy of Branch History Tables

- Mispredict because:
  - wrong guess for that branch
  - got history of wrong branch
- 2-bit counters are almost as good as n>2 bit counters
- 4096 entry table is as good as infinity

Figure 4.15, p. 266

Elaborating branch history: Correlating predictors

- The basic idea: base the taken/not-taken decision not only on the branch itself, but also on the m previous branches.
- For an \([m,n]\) correlator: Record the taken/not-taken history of the past m branches in an m-bit shift register, and use that to choose an n-bit branch history counter for the current branch.
- Used in the IBM Power-1

```c
if (aa == 2)
    aa = 0;
if (bb == 2)
    bb = 0;
if (aa != bb) {
    // code
}
```
Correlating Predictor Example

Given the following C-code segment:

```c
for (i = 0; i < 6; i++) {
    x = Array[i];
    if (x < 4) {
        <Perform some processing which does not modify x.>
    }
    if (x > 2) {
        <Perform some processing which does not modify x.>
    }
}
```

The above code compiles to the following DLX-like code sequence of instructions:

```
li    r10, #0
loop:
    lw    r1, Array_BASE_(r10) ; only writer to r1 in the loop.
    slti  r2, r1, #4 ; r2 = (r1 < 4)
    beqz  r2, label1 ; Branch B1 branches if (x >= 4).
label1:
    sgti  r2, r1, #2 ; r2 = (r1 > 2)
    beqz  r2, label2 ; Branch B2 branches if (x <= 2).
label2:
    addi  r10, r10, #4 ; r10 = r10 + 4
    bne   r10, 24, loop ; Branch B3 = loop bound check.
```

[0,2] versus [1,2] Predictor for B2

<table>
<thead>
<tr>
<th>Iteration</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>3</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>5</td>
<td>T</td>
<td>NT</td>
<td>T</td>
</tr>
<tr>
<td>6</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
</tr>
</tbody>
</table>

- Initial value of counters is 1 = not taken
- Assume all branches are found in prediction buffer

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Counter1</th>
<th>Counter2</th>
<th>Predicted correctly?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[0, 2] after each iteration

[1, 2] after each iteration
Comparison of two-bit predictors

Figure 4.21, p. 272

Branch History Table

Pipeline delay calculation example

- Assume: control point in EX, no delay slot, predict not taken
- Compare: adding prediction in ID
- Lost cycles:

<table>
<thead>
<tr>
<th></th>
<th>Taken</th>
<th>Not Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>No prediction</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Predict right</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Predict wrong</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

- If %taken = 60%, then
  no_prediction_loss = .60(2) = 1.2 cycles
  prediction_loss = %right x (.60(1)) + (1-%right) x (2)

  prediction_loss < no_prediction_loss if %right > 57%
Branch History Table
Pipeline examples

Predict taken, correctly

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch+1</td>
<td>IF</td>
<td>bubble</td>
<td>bubble</td>
<td>bubble</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predict taken, incorrectly

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch+1</td>
<td>IF</td>
<td>bubble</td>
<td>bubble</td>
<td>bubble</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Target</td>
<td>IF</td>
<td>bubble</td>
<td>bubble</td>
<td>bubble</td>
<td>bubble</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Predict not taken, correctly

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch+1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch+2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch+3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch Pattern Tables (Two-Level Predictors)

- History gives a pattern of recent branches
  - e.g., TTNTNTTN
  - what comes next?
- Predict next branch by looking up history of branches for a particular pattern
- Two-level predictor
  - first level - find history (pattern)
  - 2nd level - predict branch for that pattern

BPT may be independent for each BHT entry or shared
Branch Target Buffers

- Add to branch history table: the target address of the branch
- Take branch at the end of IF if table entry says “taken”

<table>
<thead>
<tr>
<th>tag</th>
<th>Target address</th>
<th>Taken count</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branch address

Branch Target Buffer

--- Pros and Cons

- **Advantages**
  - Allows 1-cycle (no loss) correctly predicted taken or not-taken branches

- **Disadvantages**
  - Full tag must be stored, because instruction has not been decoded -- we don’t even know that it’s a branch!
  - Full target address must be stored -- lots of bits
  - Doesn’t work well for indirect branches/jumps
  - Not as good as BHT for small number of bits
  - But BHT performance saturates after about 256 or 512 entries

- **Variations**
  - Store target instructions in addition to addresses -- allows 0-cycle branches!

- **Used by:** PowerPC 601, Pentium, others
  - PowerPC 620 has 256-entry branch target cache, and a separate 2048-entry branch history table with 2-bit counters
Branch Target Prediction

- Use current PC to index a cache of next PCs
- Use a push-down stack to record subroutine return addresses

Diagram:
- Actual Target
- IP (from A)
- IM
- BHT
- BTB
- Stack
- Prediction (Taken/Not)
- Predicted Target PC