Lecture 15: Instruction Level Parallelism
-- Introduction, Compiler Techniques, and Advanced Branch Prediction

CSE 564 Computer Architecture Summer 2017
Department of Computer Science and Engineering
Yonghong Yan
yan@oakland.edu
www.secs.oakland.edu/~yan
Topics for Instruction Level Parallelism

• ILP Introduction, Compiler Techniques and Branch Prediction
  – 3.1, 3.2, 3.3

• Dynamic Scheduling (OOO)
  – 3.4, 3.5

• Hardware Speculation and Static Superscalar/VLIW
  – 3.6, 3.7

• Dynamic Scheduling, Multiple Issue and Speculation
  – 3.8, 3.9

• ILP Limitations and SMT
  – 3.10, 3.11, 3.12
3.1 ILP: Concepts and Challenges

- **Instruction-Level Parallelism (ILP):** overlap the execution of instructions to improve performance.

- **2 approaches to exploit ILP**
  - **Rely on hardware** to help discover and exploit the parallelism dynamically (e.g., Pentium 4, AMD Opteron, IBM Power), and
  - **Rely on software technology** to find parallelism, statically at compile-time (e.g., Itanium 2)

- **Pipelining Review** (branch taken, wasted cycles in **RED**)

<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>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ld x5 -32(x4)</code></td>
<td>IF</td>
<td>ID</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>ld x6 -16(x4)</code></td>
<td>IF</td>
<td>ID</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add x6 x5 x6</code></td>
<td>IF</td>
<td>-</td>
<td>ID</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add x6 x6 x6</code></td>
<td>IF</td>
<td></td>
<td>ID</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>BNEZ x6 L1</code></td>
<td>IF</td>
<td>ID</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add x10 x4 #10</code></td>
<td>IF</td>
<td>ID</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>add x11 x5 #10</code></td>
<td>IF</td>
<td>ID</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1: <code>add x12 x5 #10</code></td>
<td>IF</td>
<td>ID</td>
<td>EXE</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Improving Instruction Level Parallelism (ILP)

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduces</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forwarding and bypassing</td>
<td>Potential data hazard stalls</td>
<td>C.2</td>
</tr>
<tr>
<td>Delayed branches and simple branch scheduling</td>
<td>Control hazard stalls</td>
<td>C.2</td>
</tr>
<tr>
<td>Basic compiler pipeline scheduling</td>
<td>Data hazard stalls</td>
<td>C.2, 3.2</td>
</tr>
<tr>
<td>Basic dynamic scheduling (scoreboarding)</td>
<td>Data hazard stalls from true dependences</td>
<td>C.7</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>Control hazard stalls</td>
<td>3.2</td>
</tr>
<tr>
<td>Branch prediction</td>
<td>Control stalls</td>
<td>3.3</td>
</tr>
<tr>
<td>Dynamic scheduling with renaming</td>
<td>Stalls from data hazards, output dependences, and antidependsences</td>
<td>3.4</td>
</tr>
<tr>
<td>Hardware speculation</td>
<td>Data hazard and control hazard stalls</td>
<td>3.6</td>
</tr>
<tr>
<td>Dynamic memory disambiguation</td>
<td>Data hazard stalls with memory</td>
<td>3.6</td>
</tr>
<tr>
<td>Issuing multiple instructions per cycle</td>
<td>Ideal CPI</td>
<td>3.7, 3.8</td>
</tr>
<tr>
<td>Compiler dependence analysis, software pipelining, trace scheduling</td>
<td>Ideal CPI, data hazard stalls</td>
<td>H.2, H.3</td>
</tr>
<tr>
<td>Hardware support for compiler speculation</td>
<td>Ideal CPI, data hazard stalls, branch hazard stalls</td>
<td>H.4, H.5</td>
</tr>
</tbody>
</table>

Figure 3.1: The major techniques examined in Appendix C, Chapter 3, and Appendix H are shown together with their effects and relevant sections.
Data Flow

• **Data flow**: actual flow of data values among instructions that produce results and those that consume them.
  – Branches make flow dynamic, determine which instruction is supplier of data.

• Example

  ```
  DADDU
  BEQZ R4, L
  DSUBU
  L: ...
  OR R7, R1, R8
  ```

• R1 of OR depends on DADDU or DSUBU?
  – Must preserve data flow on execution.
Instruction-Level Parallelism (ILP): Basic Blocks

- **BB**: a straight-line code sequence with **no branches** in except to the entry and, no branches out except at the exit;

```
Source Code

w = 0;
x = x + y;
y = 0;
if( x > z )
{
    y = x;
x++;
}
else
{
    y = z;
z++;
}
w = x + z;

Basic Blocks

w = 0;
x = x + y;
y = 0;
if( x > z )
{
    y = x;
x++;
}
else
{
    y = z;
z++;
}
w = x + z;

Basic Blocks

w = 0;
x = x + y;
y = 0;
if( x > z )
{
    y = x;
x++;
}
else
{
    y = z;
z++;
}
w = x + z;

Flow Graph

```

https://en.wikipedia.org/wiki/Basic_block
Instruction-Level Parallelism (ILP)

- Basic Block (BB) ILP is quite small
  - Average dynamic branch frequency 15% to 25%;
  - 3 to 6 instructions execute between a pair of branches.
  - Plus instructions in BB likely to depend on each other.

- To obtain substantial performance enhancements, we must exploit ILP across basic blocks. (ILP → LLP)
  - Loop-Level Parallelism: to exploit parallelism among iterations of a loop. E.g., add two matrixes.

```java
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```
Data Dependences and Hazards

• Three data dependence: *data dependences* (true data dependences), *name dependences*, and *control dependences*.  
  1. Instruction $i$ produces a result that may be used by instruction $j$ ($i \rightarrow j$), or  
  2. Instruction $j$ is data dependent on instruction $k$, and instruction $k$ is data dependent on instruction $i$ ($i \rightarrow k \rightarrow j$, dependence chain).

• For example, a code sequence

```
Loop:  L.D     F0, 0(x1)    ;F0=array element
       ADD.D   F4, F0, F2    ;add scalar in
       S.D     F4, 0(x1)    ;store result
       DADDUI  x1, x1, #-8  ;decrement pointer 8 bytes
       BNE     x1, x2, Loop ;branch x1!=x2
```
Data Dependence

• Floating-point data part

Loop:

- L.D F0, 0(x1); F0 = array element
- ADD.D F4, F0, F2; add scalar in
- S.D F4, 0(x1); store result

• Integer data part

- DADDUI x1, x1, #-8; decrement pointer
  ; 8 bytes (per DW)
- BNE x1, x2, Loop; branch x1! = x2

† This type is called a Read After Write (RAW) dependency.
Data Dependence and Hazards

- Instr\textsubscript{J} is data dependent (aka true dependence) on Instr\textsubscript{I}:
  1) Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it;

\[
\begin{align*}
\text{l:} & \quad \text{add} \ r1, r2, r3 \\
\text{j:} & \quad \text{sub} \ r4, r1, r3
\end{align*}
\]

  2) Or Instr\textsubscript{J} is data dependent on Instr\textsubscript{K} which is dependent on Instr\textsubscript{I}.

- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped.

- Data dependence in instruction sequence \rightarrow data dependence in source code \rightarrow effect of original data dependence must be preserved.

- If data dependence caused a hazard in pipeline, called a Read After Write (RAW) hazard.
Data Dependencies vs Hazards for ILP

- HW/SW must preserve program order: order instructions would execute in if executed sequentially as determined by original source program.
  - **Dependences are a property of programs.**

- Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is property of the pipeline.

- Importance of the data dependencies.
  1) Indicates the possibility of a hazard;
  2) Determines order in which results must be calculated;
  3) Sets an upper bound on how much parallelism can possibly be exploited.

- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program.
Data Dependency Detection

- Data value being dependent on between instructions either through *registers* or through *memory locations*.

- When the data flow occurs in a register
  - *Detecting the dependence is straightforward since the register names are fixed in the instrs within BB, interlock*
  - *More complicated between BB*
    - branches intervene and correctness concerns force a compiler or hardware to be conservative.

- Dependences that flow through memory locations are more difficult to detect,
  - 100(x4) and 20(x6) may be identical memory addresses.
  - The effective address of a load or store may change from one execution of the instruction to another
    - so that 20(x4) and 20(x4) may be different
Name Dependence #1: Anti-dependence

- **Name dependence**: when 2 instructions use same register or memory location, called a name, but **no flow of data** between the instructions associated with that name;
- 2 versions of name dependence (WAR and WAW).
- Instr\_\_J\ writes operand **before** Instr\_\_I\ reads it

```
I:       sub r4,r1,r3
J:       add r1,r2,r3
K:       mul r6,r1,r7
```

- Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”.
- If anti-dependence caused a hazard in the pipeline, called a **Write After Read (WAR) hazard**.
Name Dependence #2: Output dependence

- Instr$_j$ writes operand before Instr$_I$ writes it.

  \[ \begin{align*}
  &I: \quad \text{sub } r1, r4, r3 \\
  &J: \quad \text{add } r1, r2, r3 \\
  &K: \quad \text{mul } r6, r1, r7
  \end{align*} \]

- Called an “output dependence” by compiler writers. This also results from the reuse of name “r1”

- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard.

- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict.
  - Register renaming resolves name dependence for regs;
  - Either by compiler or by HW.
Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order.

```plaintext
if p1 {
    S1;
};

if p2 {
    S2;
}
```

- S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.
Control Dependence

• Two constrains imposed by control dependence
  1. An instruction that is dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch;
  2. An instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch.

• Control dependence need not be preserved
  – Willing to execute instructions that should not have been executed
  – violating the control dependences, ok if can do so without affecting correctness of the program

• Not just branch or jump
  – Exception
Exception Behavior

• Preserving exception behavior
  – Any changes in instruction execution order must not change how exceptions are raised in program (⇒ no new exceptions).

• Example

  DADDU X2,X3,X4
  BEQZ   X2,L1
  LW     X1,0(X2)

  L1:

  † Assume branches not delayed.

• Problem with moving LW before BEQZ even if branch is not taken?
  – LW may cause memory protection exception
3.2 Basic Compiler Techniques for Exposing ILP

• This code, add a scalar to a vector

\[
\text{for (i=1000; i>0; i=i-1)} \\
\quad x[i] = x[i] + s;
\]

• Assume following latencies for all examples
  – Ignore delayed branch in these examples

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.2 Latencies of FP operations used in this chapter.
FP Loop: Where are the Hazards?

• First translate into MIPS code
  – To simplify, assume 8 is lowest address
  – R1 stores the address of X[999] when the loop starts

for (i=999; i>=0; i=i−1)

\[ x[i] = x[i] + s; \]

<table>
<thead>
<tr>
<th>Loop:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0,0(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4,F0,F2</td>
</tr>
<tr>
<td>S.D</td>
<td>F4,0(R1)</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1,R1,#-8</td>
</tr>
<tr>
<td>BNE</td>
<td>R1,R2,Loop</td>
</tr>
</tbody>
</table>
FP Loop Showing Stalls: V1

- Example 3-1 (p.158) scheduled and unscheduled for delays from floating-point operations, including any stalls or idle clock cycles. Schedule for delays from floating-point operations, but remember that we are ignoring delayed branches.

- Answer

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
</tbody>
</table>

Clock cycle issued

Loop:

L.D      F0,0(R1)        1
stall
ADD.D     F4,F0,F2       2
stall
stall
S.D      F4,0(R1)        3
DADDUI    R1,R1,#-8      4
stall
BNE      R1,R2,Loop      5

9 clock cycles, 6 for useful work
Rewrite code to minimize stalls?
### Revised FP Loop Minimizing Stalls: V2

#### Loop:
- **L.D**  
  
- **ADD.D**  
  
- **S.D**  
  
- **DADDUI**  
  
- **BNE**  

#### Before:
- **F0,0(R1)**
- **F4,F0,F2**
- **F4,0(R1)**
- **R1,R1,#-8**
- **R1,R2,Loop**

#### After:
- **F0,0(R1)**
- **DADDUI**
- **ADD.D**
- **S.D**
- **BNE**

- **R1,R1,#-8**
- **F4,F0,F2**
- **F4,8(R1)**
- **R1,R2,Loop**

---

- **Swap DADDUI and S.D by changing address of S.D**

- **7 clock cycles**
  - **3 for execution (L.D, ADD.D, S.D)**
  - **4 for loop overhead; How make faster?**
## Unroll Loop Four Times: V3

<table>
<thead>
<tr>
<th></th>
<th>Loop:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>L.D</td>
<td>F0,0(R1)</td>
</tr>
<tr>
<td>3.</td>
<td>ADD.D</td>
<td>F4,F0,F2</td>
</tr>
<tr>
<td>6.</td>
<td>S.D</td>
<td>0(R1),F4</td>
</tr>
<tr>
<td>7.</td>
<td>L.D</td>
<td>F6, -8(R1)</td>
</tr>
<tr>
<td>9.</td>
<td>ADD.D</td>
<td>F8,F6,F2</td>
</tr>
<tr>
<td>12.</td>
<td>S.D</td>
<td>-8(R1),F8</td>
</tr>
<tr>
<td>13.</td>
<td>L.D</td>
<td>F10,-16(R1)</td>
</tr>
<tr>
<td>15.</td>
<td>ADD.D</td>
<td>F12,F10,F2</td>
</tr>
<tr>
<td>18.</td>
<td>S.D</td>
<td>-16(R1),F12</td>
</tr>
<tr>
<td>19.</td>
<td>L.D</td>
<td>F14,-24(R1)</td>
</tr>
<tr>
<td>21.</td>
<td>ADD.D</td>
<td>F16,F14,F2</td>
</tr>
<tr>
<td>24.</td>
<td>S.D</td>
<td>-24(R1),F16</td>
</tr>
<tr>
<td>25.</td>
<td>DADDUI</td>
<td>R1,R1,#-32</td>
</tr>
<tr>
<td>26.</td>
<td>BNEZ</td>
<td>R1,LOOP</td>
</tr>
</tbody>
</table>

- 27 clock cycles (6*4+3), or 6.75 per iteration (Assumes R1 is multiple of 4) compared with 9 for unrolled/unscheduled
Unroll Loop Four Times

• 27 clock cycles (6*4+3), or 6.75 per iteration (Assumes R1 is multiple of 4) compared with 9 for unrolled/unscheduled
  – Reducing instrs for branch and loop bound calculation
    • Reduce branch stall

• Code size increases
  – 5 instructions to 14 instructions
Unrolling Loop in Real Program

• Do not usually know upper bound of loop.
• Suppose it is \( n \), and we would like to unroll the loop to make \( k \) copies of the body.
• Instead of a single unrolled loop, we generate a pair of consecutive loops:
  – 1st executes \( (n \mod k) \) times and has a body that is the original loop;
  – 2nd is the unrolled body surrounded by an outer loop that iterates \( (n/k) \) times.
• For large values of \( n \), most of the execution time will be spent in the unrolled loop.
1. Loop: L.D F0, 0(R1)
2. L.D F6, -8(R1)
3. L.D F10, -16(R1)
4. L.D F14, -24(R1)
5. ADD.D F4, F0, F2
6. ADD.D F8, F6, F2
7. ADD.D F12, F10, F2
8. ADD.D F16, F14, F2
9. S.D 0(R1), F4
10. S.D -8(R1), F8
11. S.D -16(R1), F12
12. DSUBUI R1, R1, #32
13. S.D 8(R1), F16 ; 8-32 = -24
14. BNEZ R1, LOOP

† 14 clock cycles
## Four Versions Compared

<table>
<thead>
<tr>
<th></th>
<th>Total Cycles (1000 Iterations)</th>
<th>Cycles Per Iterations</th>
<th>Code Sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1: Original</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V2: Scheduled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V3: Unrolled</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V4: Scheduled and Unrolled</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5 Loop Unrolling Decisions

• Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:
  1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code);
  2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations;
  3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code;
  4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent;
  • Transformation requires analyzing memory addresses and finding that they do not refer to the same address.
  5. Schedule the code, preserving any dependences needed to yield the same result as the original code.
Limits to Loop Unrolling

• 3 Limits to Loop Unrolling
  1. Decrease in amount of overhead amortized with each extra unrolling.
     ♦ Reducing the ratio of the portion that can not be optimized in Amdahl’s Law.
  2. Growth in code size.
     ♦ For larger loops, concern it increases the instruction cache miss rate.
  3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling.
     ♦ If not be possible to allocate all live values to registers, may lose some or all of its advantage.

• Loop unrolling reduces impact of branches on pipeline; another way is branch prediction.
  – We discuss it in section 3.3: Reducing Branch Costs with Prediction.
3.3 Reducing Branch Costs with Prediction

- Because of the need to enforce control dependences through branch hazards and stall, branches will hurt pipeline performance.
  - Solution 1: loop unrolling $\rightarrow$ reduce branch instrs
  - Solution 2: by predicting how they will behave $\rightarrow$ reduce stalls

- SW/HW technology
  - SW: Static Branch Prediction, statically at compile time;
  - HW: Dynamic Branch Prediction, dynamically by the hardware at execution time.
Static Branch Prediction

• Appendix C showed scheduling code around *delayed branch*.
  – Reorder code around branches, need to predict branch statically when compile.
• Another and simplest scheme is to predict a branch as taken.
  – Average misprediction = untaken branch frequency = 34% SPEC. Unfortunately, from very accurate (59%) to highly accurate (9%).

<table>
<thead>
<tr>
<th>Loop</th>
<th>Instruction</th>
<th>Misprediction Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0,0(R1)</td>
<td>18%</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4,F0,F2</td>
<td>11%</td>
</tr>
<tr>
<td>S.D</td>
<td>F4,0(R1)</td>
<td>12%</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1,R1,#-8</td>
<td>4%</td>
</tr>
<tr>
<td>BNE</td>
<td>R1,R2,Loop</td>
<td>9%</td>
</tr>
</tbody>
</table>

999/1000 is correct for 1000 iterations

More accurate scheme predicts branches using *profile information* collected from earlier runs, and modify prediction based on last run.

![Misprediction Rate Graph](image)

Figure C.17 The result of predict-taken in SPEC92
How It Works In Compiler (GCC)

Built-in Function: long `__builtin_expect` (long exp, long c)

You may use `built-in_expect` to provide the compiler with branch prediction information. In general, you should prefer to use actual profile feedback for this (`-fprofile-arcs`), as programmers are notoriously bad at predicting how their programs actually perform. However, there are applications in which this data is hard to collect.

The return value is the value of `exp`, which should be an integral expression. The semantics of the built-in are that it is expected that `exp == c`. For example:

```c
if (__builtin_expect (x, 0))
  foo ();
```

indicates that we do not expect to call `foo`, since we expect `x` to be zero. Since you are limited to integral expressions for `exp`, you should use constructions such as

```c
if (__builtin_expect (ptr != NULL, 1))
  foo (*ptr);
```

when testing pointer or floating-point values.

https://gcc.gnu.org/onlinedocs/gcc/Other-Builtin.html
Collect Branch Statistics

```bash
gcc -Wall -fprofile-arcs -ftest-coverage cov.c
```

where `cov.c` is the name of the program file. This creates an instrumented `executable` which contains additional instructions that record the number of times each line of the program is executed. The option `-ftest-coverage` adds instructions for counting the number of times individual lines are executed, while `-fprofile-arcs` incorporates instrumentation code for each branch of the program. Branch instrumentation records how frequently different paths are taken through ‘if’ statements and other conditionals. The `executable` must then be run to create the coverage data. The data from the run is written to several files with the extensions `.bb` `.bbg` and `.da` respectively in the current `directory`. This data can be analyzed using the `gcov` command and the name of a source file:
Dynamic Branch Prediction

• Why does prediction work?
  – Underlying *algorithm* has regularities;
  – *Data* that is being operated on has regularities;
  – *Instruction sequence* has redundancies that are artifacts of way that humans/compilers think about problems.

• Is dynamic branch prediction better than static branch prediction?
  – Seems to be;
  – There are a small number of important branches in programs which have dynamic behavior.
Dynamic Branch Prediction

- Performance = \( f(\text{accuracy}, \text{cost of misprediction}) \)
- **Branch History Table** (also called **Branch Prediction Buffer**): lower bits of PC address index table of 1-bit values.
  - Says whether the branch was recently taken or not;
  - No address check.
- Problem: in a loop, 1-bit BHT will cause two mispredictions (average is 9 in 10 iterations before exit).
  - End of loop case, when it exits instead of looping as before;
  - First time through loop on *next* time through code, when it predicts exit instead of looping.
Basic Branch Prediction Buffers

- a.k.a. Branch History Table (BHT) - Small direct-mapped cache of T/NT bits.
Dynamic Branch Prediction

• Solution: 2-bit scheme where change prediction only if get misprediction twice.

- Red: stop, not taken;
- Blue: go, taken;
- Adds *hysteresis* to decision making process.
2-bit Scheme Accuracy

• Mispredict because either:
  – Wrong guess for that branch;
  – Got branch history of wrong branch when index the table.

• 4,096 entry table

Figure 2.5 The result of 2-bit scheme in SPEC89
2-bit Scheme Accuracy

• The accuracy of the predictors for integer programs, which typically also have higher branch frequencies, is lower than for the loop-intensive scientific programs.

• Two ways to attack this problem
  – Large buffer size;
  – Increasing the accuracy of the scheme we use for each prediction.

• However, simply increasing the number of bits per predictor without changing the predictor structure also has little impact.
  – Single branch predictor V.S. correlating branch predictors.
Accuracy of Different Schemes

4,096 Entries 2-bit BHT
Unlimited Entries 2-bit BHT
1,024 Entries (2, 2) BHT

Frequency of Mispredictions

4,096 entries: 2-bits per entry
Unlimited entries: 2-bits/entry
1,024 entries (2, 2)
Improve Prediction Strategy By Correlating Branches

• Consider the worst case for the 2-bit predictor

  \[
  \begin{aligned}
  \text{if } (aa==2) & \quad \text{if the first 2 fail then the 3rd will always be taken.} \\
  aa=0; \\
  \text{if } (bb==2) & \quad \text{if the first 2 fail then} \\
  bb=0; \\
  \text{if } (aa != bb) & \\
  \{ & \\
  & \quad \text{Single level predictors can never get this case.}
  \end{aligned}
  \]

• Correlating predictors or 2-level predictors
  – Correlation = what happened on the last branch
    • Note that the last correlator branch may not always be the same.
  – Predictor = which way to go
    • 4 possibilities: which way the last one went chooses the prediction.
      – \((\text{Last-taken, last-not-taken}) \times (\text{predict-taken, predict-not-taken})\)
Correlated Branch Prediction

- Idea: record \( m \) most recently executed branches as taken or not taken, and use that pattern to select the proper \( n \)-bit branch history table.
- In general, \((m, n)\) predictor means record last \( m \) branches to select between \( 2^m \) history tables, each with \( n \)-bit counters.
  - Thus, old 2-bit BHT is a \((0, 2)\) predictor.
- Global Branch History: \( m \)-bit shift register keeping T/NT status of last \( m \) branches.
- Each entry in table has \( m \) \( n \)-bit predictors.
- Total bits for the \((m, n)\) BHT prediction buffer:

\[
Total\_memory\_bits = 2^m \times n \times 2^p
\]
  - \( 2^m \) banks of memory selected by the global branch history (which is just a shift register) - e.g. a column address;
  - Use \( p \) bits of the branch address to select row;
  - Get the \( n \) predictor bits in the entry to make the decision.
Correlating Branches

- **(2, 2) predictor**
  - Behavior of recent 2 branches selects between four predictions of next branch, updating just that prediction.
Example of Correlating Branch Predictors

if (d==0)
    d = 1;
if (d==1)
    ...

BNEZ R1, L1 ;branch b1 (d!=0)
DADDIU R1, R0, #1 ;d==0, so d=1
L1: DADDIU R3, R1, #-1
    BNEZ R3, L2 ;branch b2 (d!=1)
    ...
L2:
Example: Multiple Consequent Branches

If $b_1$ is not taken, then $b_2$ will be not taken

if ($d == 0$) ;not taken
d=1;
else ;taken
if ($d == 1$) ;not taken
else ;taken

1-bit predictor: consider $d$ alternates between 2 and 0. All branches are mispredicted

<table>
<thead>
<tr>
<th>Initial value of $d$</th>
<th>$d == 0?$</th>
<th>$b_1$</th>
<th>Value of $d$ before $b_2$</th>
<th>$d == 1?$</th>
<th>$b_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>yes</td>
<td>not taken</td>
<td>1</td>
<td>yes</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>taken</td>
<td>1</td>
<td>yes</td>
<td>not taken</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>taken</td>
<td>2</td>
<td>no</td>
<td>taken</td>
</tr>
</tbody>
</table>

Figure 3.10 Possible execution sequences for a code fragment.

<table>
<thead>
<tr>
<th>$d =$?</th>
<th>$b_1$ prediction</th>
<th>$b_1$ action</th>
<th>New $b_1$ prediction</th>
<th>$b_2$ prediction</th>
<th>$b_2$ action</th>
<th>New $b_2$ prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>

Figure 3.11 Behavior of a 1-bit predictor initialized to not taken. T stands for taken, NT for not taken.
Example: Multiple Consequent Branches

2-bits prediction: prediction if last branch not taken/ and prediction if last branch taken

(1,1) predictor - 1-bit predictor with 1 bit of correlation: last branch (either taken or not taken) decides which prediction bit will be considered or updated
Branch Prediction with Neural Networks


[6] K. Aasaraai and A. Baniasadi Low-power Perceptrons

[7] A. Seznec. The O-GEometric History Length branch predictor


Summary

• Three kinds of data dependency
  – True data dependency
  – Name dependency
  – Control dependency

• Hazards from dependency
  – Stall the pipeline

• Compiler technology
  – Loop unrolling
  – Instruction Scheduling

• Branch Prediction
  – Static compiler-based prediction
  – Dynamic hardware-based prediction
    • Branch history table + Branch Target Buffer