Lecture 5: VLIW, Software Pipelining, and Limits to ILP

Professor David A. Patterson
Computer Science 252
Spring 1998
Review: Tomasulo

- Prevents Register as bottleneck
- Avoids WAR, WAW hazards of Scoreboard
- Allows loop unrolling in HW
- Not limited to basic blocks (provided branch prediction)

Lasting Contributions
- Dynamic scheduling
- Register renaming
- Load/store disambiguation

- 360/91 descendants are PowerPC 604, 620; MIPS R10000; HP-PA 8000; Intel Pentium Pro
Dynamic Branch Prediction

• Performance = $f$(accuracy, cost of misprediction)

• Branch History Table is simplest
  – Lower bits of PC address index table of 1-bit values
  – Says whether or not branch taken last time
  – No address check

• Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 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
Dynamic Branch Prediction

- Solution: 2-bit scheme where change prediction only if get misprediction *twice*: (Figure 4.13, p. 264)

- Red: stop, not taken
- Green: go, taken
BHT Accuracy

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

• 4096 entry table programs vary from 1% misprediction (nasa7, tomcatv) to 18% (eqntott), with spice at 9% and gcc at 12%

• 4096 about as good as infinite table (in Alpha 211164)
Correlating Branches

- Hypothesis: recent branches are correlated; that is, behavior of recently executed branches affects prediction of current branch
- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper 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
  - Old 2-bit BHT is then a (0,2) predictor
(2,2) predictor
- Then behavior of recent branches selects between, say, four predictions of next branch, updating just that prediction

Correlating Branches
Accuracy of Different Schemes
(Figure 4.21, p. 272)

4096 Entries 2-bit BHT
Unlimited Entries 2-bit BHT
1024 Entries (2,2) BHT

Frequency of Mispredictions

0% 2% 4% 6% 8% 10% 12% 14% 16% 18%

nasa7 matrix300 tomcatv doducd spice fppp gcc espresso eqntott li

1% 0% 1% 5% 6% 6% 11% 4% 6% 5%

4,096 entries: 2-bits per entry
Unlimited entries: 2-bits/entry
1,024 entries (2,2) BHT
Re-evaluating Correlation

• Several of the SPEC benchmarks have less than a dozen branches responsible for 90% of taken branches:

<table>
<thead>
<tr>
<th>program</th>
<th>branch %</th>
<th>static</th>
<th># = 90%</th>
</tr>
</thead>
<tbody>
<tr>
<td>compress</td>
<td>14%</td>
<td>236</td>
<td>13</td>
</tr>
<tr>
<td>eqntott</td>
<td>25%</td>
<td>494</td>
<td>5</td>
</tr>
<tr>
<td>gcc</td>
<td>15%</td>
<td>9531</td>
<td>2020</td>
</tr>
<tr>
<td>mpeg</td>
<td>10%</td>
<td>5598</td>
<td>532</td>
</tr>
<tr>
<td>real gcc</td>
<td>13%</td>
<td>17361</td>
<td>3214</td>
</tr>
</tbody>
</table>

• Real programs + OS more like gcc
• Small benefits beyond benchmarks for correlation? problems with branch aliases?
Need Address at Same Time as Prediction

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)
  - Note: must check for branch match now, since can’t use wrong branch address (Figure 4.22, p. 273)

- Return instruction addresses predicted with stack
HW support for More ILP

• Avoid branch prediction by turning branches into conditionally executed instructions:

\[
\text{if (x) then } A = B \text{ op } C \text{ else NOP}
\]

– If false, then neither store result nor cause exception
– Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.
– IA-64: 64 1-bit condition fields selected so conditional execution of any instruction

• Drawbacks to conditional instructions
  – Still takes a clock even if “annulled”
  – Stall if condition evaluated late
  – Complex conditions reduce effectiveness; condition becomes known late in pipeline
Dynamic Branch Prediction Summary

- Branch History Table: 2 bits for loop accuracy
- Correlation: Recently executed branches correlated with next branch
- Branch Target Buffer: include branch address & prediction
- Predicated Execution can reduce number of branches, number of mispredicted branches
HW support for More ILP

- **Speculation**: allow an instruction to issue that is dependent on branch predicted to be taken *without* any consequences (including exceptions) if branch is not actually taken ("HW undo"); called "boosting"

- Combine branch prediction with dynamic scheduling to execute before branches resolved

- Separate *speculative* bypassing of results from real bypassing of results
  - When instruction no longer speculative, write boosted results (*instruction commit*) or discard boosted results
  - execute out-of-order but *commit in-order* to prevent irrevocable action (update state or exception) until instruction commits
HW support for More ILP

- Need HW buffer for results of uncommitted instructions: **reorder buffer**
  - 3 fields: instr, destination, value
  - Reorder buffer can be operand source => more registers like RS
  - Use reorder buffer number instead of reservation station when execution completes
  - Supplies operands between execution complete & commit
  - Once operand commits, result is put into register
  - Instructions **commit in order**
  - As a result, it's easy to undo speculated instructions on mispredicted branches or on exceptions
Four Steps of Speculative Tomasulo Algorithm

1. **Issue**—get instruction from FP Op Queue
   
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called “dispatch”)

2. **Execution**—operate on operands (EX)
   
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called “issue”)

3. **Write result**—finish execution (WB)
   
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. **Commit**—update register with reorder result
   
   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called “graduation”)
Renaming Registers

• Common variation of speculative design
• Reorder buffer keeps instruction information but not the result
• Extend register file with extra renaming registers to hold speculative results
• Rename register allocated at issue; result into rename register on execution complete; rename register into real register on commit
• Operands read either from register file (real or speculative) or via Common Data Bus
• Advantage: operands are always from single source (extended register file)
CS 252 Administrivia

• Get your photo taken by Joe Gebis! (or give URL)
• Video in 201 McLaughlin, starting day of lecture
  Mon 9-11AM, 2 - 5 PM; Tue 9 AM - 5 PM;
  Wed 9-11AM, 2 - 10 PM; Thu 9 AM - 6 PM;
  Fri  9 - 5PM, 6 - 10 PM;
• Reading Assignments for Lectures 3 to 7
  – Computer Architecture: AQA, Chapter 4, Appendix B
• Exercises for Lectures 3 to 7
  – Due Thursday February 12 at 5PM homework box in 283 Soda (building is locked at 6:45 PM)
  – 4.2, 4.10, 4.19, B.2
  – 4.14 parts c) and d) only
  – Done in pairs, but both need to understand whole assignment
  – Study groups encouraged, but pairs do own work
Dynamic Scheduling in PowerPC 604 and Pentium Pro

- Both In-order Issue, Out-of-order execution, In-order Commit

Pentium Pro more like a scoreboard since central control vs. distributed
Dynamic Scheduling in PowerPC 604 and Pentium Pro

<table>
<thead>
<tr>
<th>Parameter</th>
<th>PPC</th>
<th>PPro</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. instructions issued/clock</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Max. instr. complete exec./clock</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Max. instr. commited/clock</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Window (Instrs in reorder buffer)</td>
<td>16</td>
<td>40</td>
</tr>
<tr>
<td>Number of reservations stations</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>Number of rename registers</td>
<td>8int/12FP</td>
<td>40</td>
</tr>
<tr>
<td>No. integer functional units (FUs)</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>No. floating point FUs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>No. branch FUs</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>No. complex integer FUs</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>No. memory FUs</td>
<td>1</td>
<td>1 load +1 store</td>
</tr>
</tbody>
</table>

Q: How pipeline 1 to 17 byte x86 instructions?
Dynamic Scheduling in Pentium Pro

• PPro doesn’t pipeline 80x86 instructions
• PPro decode unit translates the Intel instructions into 72-bit micro-operations (≈ DLX)
• Sends micro-operations to reorder buffer & reservation stations
• Takes 1 clock cycle to determine length of 80x86 instructions + 2 more to create the micro-operations
• 12-14 clocks in total pipeline (≈ 3 state machines)
• Many instructions translate to 1 to 4 micro-operations
• Complex 80x86 instructions are executed by a conventional microprogram (8K x 72 bits) that issues long sequences of micro-operations
Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Two variations
- **Superscalar**: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
- **(Very) Long Instruction Words (V)LIW**: fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates
  - Joint HP/Intel agreement in 1999/2000?
  - Intel Architecture-64 (IA-64) 64-bit address
  - Style: “Explicitly Parallel Instruction Computer (EPIC)”
- Anticipated success lead to use of **Instructions Per Clock** cycle (IPC) vs. CPI
Getting CPI < 1: Issuing Multiple Instructions/Cycle

- Superscalar DLX: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; Int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

<table>
<thead>
<tr>
<th>Type</th>
<th>PipeStages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int. instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Int. instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

- 1 cycle load delay expands to 3 instructions in SS
  - Instruction in right half can’t use it, nor instructions in next slot
Review: Unrolled Loop that Minimizes Stalls for Scalar

1 Loop:  
1. LD  F0, 0 (R1)  
2. LD  F6, -8 (R1)  
3. LD  F10, -16 (R1)  
4. LD  F14, -24 (R1)  
5. ADDD  F4, F0, F2  
6. ADDD  F8, F6, F2  
7. ADDD  F12, F10, F2  
8. ADDD  F16, F14, F2  
9. SD  0 (R1), F4  
10. SD  -8 (R1), F8  
11. SD  -16 (R1), F12  
12. SUBI  R1, R1, #32  
13. BNEZ  R1, LOOP  
14. SD  8 (R1), F16 ; 8-32 = -24

14 clock cycles, or 3.5 per iteration
Loop Unrolling in Superscalar

<table>
<thead>
<tr>
<th>Integer instruction</th>
<th>FP instruction</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>F0,0(R1)</td>
<td>1</td>
</tr>
<tr>
<td>LD</td>
<td>F6,-8(R1)</td>
<td>2</td>
</tr>
<tr>
<td>LD</td>
<td>F10,-16(R1)</td>
<td>3</td>
</tr>
<tr>
<td>LD</td>
<td>F14,-24(R1)</td>
<td>4</td>
</tr>
<tr>
<td>LD</td>
<td>F18,-32(R1)</td>
<td>5</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1),F4</td>
<td>6</td>
</tr>
<tr>
<td>SD</td>
<td>-8(R1),F8</td>
<td>7</td>
</tr>
<tr>
<td>SD</td>
<td>-16(R1),F12</td>
<td>8</td>
</tr>
<tr>
<td>SD</td>
<td>-24(R1),F16</td>
<td>9</td>
</tr>
<tr>
<td>SUBI</td>
<td>R1,R1,#40</td>
<td>10</td>
</tr>
<tr>
<td>BNEZ</td>
<td>R1,LOOP</td>
<td>11</td>
</tr>
<tr>
<td>SD</td>
<td>-32(R1),F20</td>
<td>12</td>
</tr>
</tbody>
</table>

- Unrolled 5 times to avoid delays (+1 due to SS)
- 12 clocks, or 2.4 clocks per iteration (1.5X)
Multiple Issue Challenges

• While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  – Exactly 50% FP operations
  – No hazards

• If more instructions issue at same time, greater difficulty of decode and issue
  – Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue

• VLIW: tradeoff instruction space for simple decoding
  – The long instruction word has room for many operations
  – By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  – E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    » 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  – Need compiling technique that schedules across several branches
## Loop Unrolling in VLIW

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0,0(R1)</td>
<td>LD F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>LD F10,-16(R1)</td>
<td>LD F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>LD F18,-32(R1)</td>
<td>LD F22,-40(R1)</td>
<td>ADDD F4,F0,F2</td>
<td>ADDD F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>LD F26,-48(R1)</td>
<td></td>
<td>ADDD F12,F10,F2</td>
<td>ADDD F16,F14,F2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ADDD F20,F18,F2</td>
<td>ADDD F24,F22,F2</td>
<td>5</td>
</tr>
<tr>
<td>SD 0(R1),F4</td>
<td>SD -8(R1),F8</td>
<td>ADDD F28,F26,F2</td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>SD -16(R1),F12</td>
<td>SD -24(R1),F16</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>SD -32(R1),F20</td>
<td>SD -40(R1),F24</td>
<td></td>
<td></td>
<td>SUBI R1,R1,#48</td>
<td>8</td>
</tr>
<tr>
<td>SD -0(R1),F28</td>
<td></td>
<td></td>
<td></td>
<td>BNEZ R1,LOOP</td>
<td>9</td>
</tr>
</tbody>
</table>

Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)
Average: 2.5 ops per clock, 50% efficiency

Note: Need more registers in VLIW (15 vs. 6 in SS)
Trace Scheduling

- Parallelism across IF branches vs. LOOP branches
- Two steps:
  - *Trace Selection*
    - Find likely sequence of basic blocks (*trace*) of (statically predicted or profile predicted) long sequence of straight-line code
  - *Trace Compaction*
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping code in case prediction is wrong
- Compiler undoes bad guess (discards values in registers)
- Subtle compiler bugs mean wrong answer vs. poorer performance; no hardware interlocks
Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

- HW determines address conflicts
- HW better branch prediction
- HW maintains precise exception model
- HW does not execute bookkeeping instructions
- Works across multiple implementations
- SW speculation is much easier for HW design
Superscalar v. VLIW

- Smaller code size
- Binary compatibility across generations of hardware
- Simplified Hardware for decoding, issuing instructions
- No Interlock Hardware (compiler checks?)
- More registers, but simplified Hardware for Register Ports (multiple independent register files?)
Intel/HP “Explicitly Parallel Instruction Computer (EPIC)”

- 3 Instructions in 128 bit “groups”; field determines if instructions dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show independence > 3 instr
- 64 integer registers + 64 floating point registers
  - Not separate files per functional unit as in old VLIW
- Hardware checks dependencies (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags)
  => 40% fewer mispredictions?
- **IA-64**: name of instruction set architecture; EPIC is type
- **Merced** is name of first implementation (1999/2000?)
- LIW = EPIC?
Dynamic Scheduling in Superscalar

- Dependencies stop instruction issue
- Code compiler for old version will run poorly on newest version
  - May want code to vary depending on how superscalar
Dynamic Scheduling in Superscalar

• How to issue two instructions and keep in-order instruction issue for Tomasulo?
  – Assume 1 integer + 1 floating point
  – 1 Tomasulo control for integer, 1 for floating point

• Issue 2X Clock Rate, so that issue remains in order

• Only FP loads might cause dependency between integer and FP issue:
  – Replace load reservation station with a load queue; operands must be read in the order they are fetched
  – Load checks addresses in Store Queue to avoid RAW violation
  – Store checks addresses in Load Queue to avoid WAR, WAW
  – Called “decoupled architecture”
## Performance of Dynamic SS

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Instructions</th>
<th>Issues</th>
<th>Executes</th>
<th>Writes result</th>
</tr>
</thead>
<tbody>
<tr>
<td>no.</td>
<td></td>
<td>clock-cycle number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>LD F0,0(R1)</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>ADDD F4,F0,F2</td>
<td>1</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>SD 0(R1),F4</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SUBI R1,R1,#8</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>BNEZ R1,LOOP</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>LD F0,0(R1)</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>ADDD F4,F0,F2</td>
<td>5</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>SD 0(R1),F4</td>
<td>6</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SUBI R1,R1,#8</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>BNEZ R1,LOOP</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

≈ 4 clocks per iteration; only 1 FP instr/iteration

Branches, Decrements issues still take 1 clock cycle

How get more performance?
Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations.
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (∼ Tomasulo in SW).
Software Pipelining Example

Before: Unrolled 3 times
1. LD F0,0(R1)
2. ADDD F4,F0,F2
3. SD 0(R1),F4
4. LD F6,-8(R1)
5. ADDD F8,F6,F2
6. SD -8(R1),F8
7. LD F10,-16(R1)
8. ADDD F12,F10,F2
9. SD -16(R1),F12
10. SUBI R1,R1,#24
11. BNEZ R1,LOOP

After: Software Pipelined
1. SD 0(R1),F4 ; Stores M[i]
2. ADDD F4,F0,F2 ; Adds to M[i-1]
3. LD F0,-16(R1); Loads M[i-2]
4. SUBI R1,R1,#8
5. BNEZ R1,LOOP

• Symbolic Loop Unrolling
  – Maximize result-use distance
  – Less code space than unrolling
  – Fill & drain pipe only once per loop vs. once per each unrolled iteration in loop unrolling
Limits to Multi-Issue Machines

• Inherent limitations of ILP
  – 1 branch in 5: How to keep a 5-way VLIW busy?
  – Latencies of units: many operations must be scheduled
  – Need about Pipeline Depth x No. Functional Units of independent operations to keep machines busy, e.g. 5 x 4 = 15–20 independent instructions?

• Difficulties in building HW
  – Easy: More instruction bandwidth
  – Easy: Duplicate FUs to get parallel execution
  – Hard: Increase ports to Register File (bandwidth)
    » VLIW example needs 7 read and 3 write for Int. Reg. & 5 read and 3 write for FP reg
  – Harder: Increase ports to memory (bandwidth)
  – Decoding Superscalar and impact on clock rate, pipeline depth?
Limits to Multi-Issue Machines

- Limitations specific to either Superscalar or VLIW implementation
  - Decode issue in Superscalar: how wide practical?
  - VLIW code size: unroll loops + wasted fields in VLIW
    » IA-64 compresses dependent instructions, but still larger
  - VLIW lock step => 1 hazard & all instructions stall
    » IA-64 not lock step? Dynamic pipeline?
  - VLIW & binary compatibility is practical weakness as vary number FU and latencies over time
    » IA-64 promises binary compatibility
Limits to ILP

• Conflicting studies of amount of parallelism available in late 1980s and early 1990s. Different assumptions about:
  – Benchmarks (vectorized Fortran FP vs. integer C programs)
  – Hardware sophistication
  – Compiler sophistication

• How much ILP is available using existing mechanisms with increasing HW budgets?

• Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
Limits to ILP

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

1. *Register renaming*—infinite virtual registers and all WAW & WAR hazards are avoided

2. *Branch prediction*—perfect; no mispredictions

3. *Jump prediction*—all jumps perfectly predicted => machine with perfect speculation & an unbounded buffer of instructions available

4. *Memory-address alias analysis*—addresses are known & a store can be moved before a load provided addresses not equal

1 cycle latency for all instructions; unlimited number of instructions issued per clock cycle
Upper Limit to ILP: Ideal Machine
(Figure 4.38, page 319)

Graph showing IPC (Instruction Per Cycle) for different programs:
- **Integer**: 18 - 60
- **FP**: 75 - 150

Programs:
- gcc: 54.8
- espresso: 62.6
- li: 17.9
- fpppp: 75.2
- doducd: 118.7
- tomcatv: 150.1

DAP.F96 40
More Realistic HW: Branch Impact

Figure 4.40, Page 323

Change from Infinite window to examine to 2000 and maximum issue of 64 instructions per clock cycle

FP: 15 - 45

Integer: 6 - 12

Program

<table>
<thead>
<tr>
<th>Program</th>
<th>Perfect</th>
<th>Selective predictor</th>
<th>Standard 2-bit</th>
<th>Static</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>espresso</td>
<td>12</td>
<td>7</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>li</td>
<td>16</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>fpppp</td>
<td>61</td>
<td>48</td>
<td>46</td>
<td>45</td>
<td>29</td>
</tr>
<tr>
<td>doducd</td>
<td>58</td>
<td>15</td>
<td>13</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>tomcatv</td>
<td>60</td>
<td>46</td>
<td>45</td>
<td>45</td>
<td>19</td>
</tr>
</tbody>
</table>

IPC

Instruction issues per cycle

Perfect  Pick Cor. or BHT  BHT (512)  Profile  No prediction
Selective History Predictor

- **Branch Addr**
- **8096 x 2 bits**
- **2 Global History**
- **2048 x 4 x 2 bits**
- **8K x 2 bit Selector**
- **Taken/Not Taken**
- **Choose Non-correlator**
- **Choose Correlator**

- 00
- 01
- 10
- 11

- 00
- 01
- 10
- 11

- 11 Taken
- 10
- 01 Not Taken
- 00
More Realistic HW: Register Impact

Figure 4.44, Page 328

Change 2000 instr window, 64 instr issue, 8K 2 level Prediction

Integer: 5 - 15

FP: 11 - 45

Infinite 256 128 64 32 None

Program

gcc espresso li fpppp doducd tomcatv

Instruction issues per cycle

IPC

DAP.F96 43
More Realistic HW: Alias Impact

Figure 4.46, Page 330

- Change 2000 instr window, 64 instr issue, 8K 2 level Prediction, 256 renaming registers
- Integer: 4 - 9
- FP: 4 - 45 (Fortran, no heap)

IPC

<table>
<thead>
<tr>
<th>Program</th>
<th>gcc</th>
<th>espresso</th>
<th>li</th>
<th>fpppp</th>
<th>doducd</th>
<th>tomcatv</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perfect</td>
<td>10</td>
<td>15</td>
<td>12</td>
<td>49</td>
<td>16</td>
<td>45</td>
</tr>
<tr>
<td>Global/Stack perf</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Inspection</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>None</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
Realistic HW for ‘9X: Window Impact
(Figure 4.48, Page 332)

Perfect disambiguation (HW), 1K Selective Prediction, 16 entry return, 64 registers, issue as many as window

Integer: 6 - 12

FP: 8 - 45

Infinite 256 128 64 32 16 8 4
Braniac vs. Speed Demon (1993)

- 8-scalar IBM Power-2 @ 71.5 MHz (5 stage pipe) vs. 2-scalar Alpha @ 200 MHz (7 stage pipe)
## 3 1996 Era Machines

<table>
<thead>
<tr>
<th></th>
<th>Alpha 21164</th>
<th>PPro</th>
<th>HP PA-8000</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year</strong></td>
<td>1995</td>
<td>1995</td>
<td>1996</td>
</tr>
<tr>
<td><strong>Clock</strong></td>
<td>400 MHz</td>
<td>200 MHz</td>
<td>180 MHz</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td>8K/8K/96K/2M</td>
<td>8K/8K/0.5M</td>
<td>0/0/2M</td>
</tr>
<tr>
<td><strong>Issue rate</strong></td>
<td>2int+2FP</td>
<td>3 instr (x86)</td>
<td>4 instr</td>
</tr>
<tr>
<td><strong>Pipe stages</strong></td>
<td>7-9</td>
<td>12-14</td>
<td>7-9</td>
</tr>
<tr>
<td><strong>Out-of-Order</strong></td>
<td>6 loads</td>
<td>40 instr (µop)</td>
<td>56 instr</td>
</tr>
<tr>
<td><strong>Rename regs</strong></td>
<td>none</td>
<td>40</td>
<td>56</td>
</tr>
</tbody>
</table>
SPECint95base Performance (July 1996)

Graph showing SPECint95base performance for various applications across different processors.
SPECfp95base Performance
(July 1996)
3 1997 Era Machines

<table>
<thead>
<tr>
<th></th>
<th>Alpha 21164</th>
<th>Pentium II</th>
<th>HP PA-8000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>1995</td>
<td>1996</td>
<td>1996</td>
</tr>
<tr>
<td>Clock</td>
<td>600 MHz (‘97)</td>
<td>300 MHz (‘97)</td>
<td>236 MHz (‘97)</td>
</tr>
<tr>
<td>Cache</td>
<td>8K/8K/96K/2M</td>
<td>16K/16K/0.5M</td>
<td>0/0/4M</td>
</tr>
<tr>
<td>Issue rate</td>
<td>2int+2FP</td>
<td>3 instr (x86)</td>
<td>4 instr</td>
</tr>
<tr>
<td>Pipe stages</td>
<td>7-9</td>
<td>12-14</td>
<td>7-9</td>
</tr>
<tr>
<td>Out-of-Order</td>
<td>6 loads</td>
<td>40 instr (μop)</td>
<td>56 instr</td>
</tr>
<tr>
<td>Rename regs</td>
<td>none</td>
<td>40</td>
<td>56</td>
</tr>
</tbody>
</table>
SPECfp95base Performance (Oct. 1997)
Summary

• Branch Prediction
  – Branch History Table: 2 bits for loop accuracy
  – Recently executed branches correlated with next branch?
  – Branch Target Buffer: include branch address & prediction
  – Predicated Execution can reduce number of branches, number of mispredicted branches

• Speculation: Out-of-order execution, In-order commit (reorder buffer)

• SW Pipelining
  – Symbolic Loop Unrolling to get most from pipeline with little code expansion, little overhead

• Superscalar and VLIW: CPI < 1 (IPC > 1)
  – Dynamic issue vs. Static issue
  – More instructions issue at same time => larger hazard penalty