Lecture 12: Hardware Assisted Software ILP and IA64/Itanium Case Study

Lecture Outline

- Review of Global Scheduling, Hardware-Assisted Software ILP
- IA-64 Instruction Set Architecture
- Itanium, Itanium-2 Processor
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
- Similar steps for Superblock selection and compaction

What if branches are not statically predictable?

- Compiler Speculation is needed to solve this
  - Conditional/Predicated instructions “if-conversion”
    - Conditional MOVEs and Loads
    - Full-Predication (IA64)
  - Hardware support for exception/memory-dependence checks
    - Hardware/ISA support for exception checks
    - Memory-dependence hardware checks
### Predicated Loads Example

- 2-issue superscalar: (one mem + one alu) OR one branch

<table>
<thead>
<tr>
<th>First instruction slot</th>
<th>Second instruction slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW R1,40(R2)</td>
<td>ADD R3,R4,R5</td>
</tr>
<tr>
<td></td>
<td>ADD R6,R3,R7</td>
</tr>
<tr>
<td>BEQZ R10,L</td>
<td></td>
</tr>
<tr>
<td>LW R8,0(R10)</td>
<td></td>
</tr>
<tr>
<td>LW R9,0(R8)</td>
<td></td>
</tr>
</tbody>
</table>

- Waste slot since 3rd LW dependent on result of 2nd LW

---

### Predicated Loads

- Use predicated version load word (LWC)?
  - load occurs unless the third operand is 0
- First instruction slot    Second instruction slot
  | LW   | R1,40(R2) | ADD R3,R4,R5 |
  | LWC  | R8,20(R10),R10 | ADD R6,R3,R7 |
  | BEQZ | R10,L    |              |
  | LW   | R9,0(R8) |              |

- When would we totally eliminate the branch?
Full Predication

- Full Predication works better for long streams of code
- Set-Predicate Instructions, e.g. seqzp
- Instructions converted to predicated versions
- If predicate is true perform op, otherwise ignore it

<table>
<thead>
<tr>
<th>Normal Code</th>
<th>Predicated Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQZ R2, L</td>
<td>SEQZP P1,R2</td>
</tr>
<tr>
<td>ADD R4, R6, R5</td>
<td>ADD.np R4,R6,R5,p1</td>
</tr>
<tr>
<td>JUMP L2</td>
<td>ADD.p R4,R5,R6,p1</td>
</tr>
<tr>
<td>L: ADD R4,R5,R6</td>
<td></td>
</tr>
<tr>
<td>L2:</td>
<td></td>
</tr>
</tbody>
</table>

Limits of Conditional Instructions

- When to annul? Early in the pipe or late?
- Annulled predicated instruction use resources
  - Fetch, Decode, Functional Units
  - Ok for a CMOV, but what about a long sequence?
- Predicate must be evaluated early to be useful
  - May convert a control-hazard stall to a data-hazard stall
- Moving across multiple branches is really tough
  - Requires two conditions to be specified/compute predicate
- CMOVs may complicate implementations
Hardware Assisted Software Speculation

- Compiler must find instructions that can be speculatively moved without impacting program data flow
- Move instructions ahead of the condition evaluation
  - Predication cannot do this!
- Must be able to
  - Ignore exceptions in speculated instructions until they are non-speculative
  - Speculatively interchange loads/stores and stores/stores that may have address conflicts

Hardware Exception Behavior Support

- Several mechanisms to ensure that speculation by compiler does not violate exception behavior
  - Ignore exceptions for speculative instructions
  - Only use speculative instructions that do not raise exceptions
  - “Poison” bits attached to the result registers
  - Hardware buffers for speculative work

```assembly
If (A==0) A=B; else A=A+4
LD R1,0(R3) ;Load A
BNEZ R1,L1 ;test A
LD R1,0(R2) ; then clause
J L2 ; skip else
L1:ADDI R1,R1,#4 ; else clause
L2:SD R1,0(R3) ; store A
```
Ignore Speculative Exceptions

- Handle all resumable exceptions when they occur (spec. too)
- Ignore terminating exceptions
  - Speculative exceptions are ignored – don’t matter anyway
  - Non-speculative exceptions cause programs to be in error (generate incorrect results)
- Correct programs do not fail no matter how much speculation
- Incorrect programs (should terminate) do not

```assembly
LD R1,0(R3) ;Load A
LD R14,0(R2) ;speculative Load B
BEQZ R1,L3 ;other branch of the if
DADDI R14, R1, #4 ;else clause
L3:SD R14,0(R3) ;store A
```

Special Instructions

- Speculative loads (sLD) and Speculative Checks (SPECCK)
- sLD will not generate exceptions
- SPECCK will generate exceptions

```assembly
LD R1,0(R3) ;Load A
sLD R14,0(R2) ;speculative, no exceptions
BNEZ R1,L1 ;test A
SPECCK 0(R2) ;Perform Speculation Check
J L2 ;skip else
L1:ADDI R14,R1,#4 ;else clause
L2:SD R14,0(R3) ;store A
```
Poison Bits

- Track exceptions as they occur
- Postpone terminating exceptions until a value is used
- Poison bits are added to every register
  - Set when speculative instruction causes a terminating fault
- Non-Speculative instructions that access a poisoned register cause a fault

```
LD     R1,0(R3)    ;Load A
sLD    R14,0(R2)  ;speculative Load B
BEQ2   R1,L3      ;
DADDI  R14,R1,#4  ;
L3:SD  R14,0(R3)  ;exception for speculative LW
```

Reorder buffer approaches

- Similar to hardware speculation
  - No register renaming/dynamic scheduling/branch prediction used
- Each speculative instruction has 1-bit saying whether it came from the taken or not-taken path
- Reorder buffer tracks when instructions are ready to commit
  - Delays “write-back” for speculative instructions
  - Only commit when no longer speculative
Hardware Support for Memory Reference Speculation

- Problem: Compiler wants to move loads across stores
  - What if it cannot be absolutely certain that such a movement is correct?
- HW support for a special instruction to check for address conflicts
  - The special instruction is left at the original location of the load and the load is moved up across stores
  - When a speculated load is executed, the hardware saves the address of the accessed memory location
  - If a subsequent store changes the location before the check instruction, then the speculation has failed
- What should we do if the load mis-speculated?

Problems with First Generation VLIW

- Increase in code size
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- Operated in lock-step; no hazard detection HW
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict
- Binary code compatibility
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code
Advantages of HW (Tomasulo) vs. SW (VLIW) Speculation

- **HW advantages:**
  - HW better at memory disambiguation since knows actual addresses
  - HW better at branch prediction since lower overhead
  - HW maintains precise exception model
  - HW does not execute bookkeeping instructions
  - Same software works across multiple implementations
  - Smaller code size (not as many nops filling blank instructions)

- **SW advantages:**
  - Window of instructions that is examined for parallelism much higher
  - Much less hardware involved in VLIW
  - More involved types of speculation can be done more easily
  - Speculation can be based on large-scale program behavior, not just local information

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- **IA-64:** instruction set architecture; EPIC is type
  - EPIC = 2nd generation VLIW?
- **Itanium™** is name of first implementation (June 2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process
- Itanium2 – Sept 2002 (1GHz), Sept 2003 (1.5GHz)
- 128 64-bit integer registers + 128 82-bit floating point registers
- Hardware checks some dependencies (interlocks => binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags)
IA-64 Registers

- The integer registers designed to assist procedure calls using a register stack
  - Similar to SPARC’s register windows.
  - Registers 0-31 are always accessible and addressed as 0-31
  - Registers 32-128 are used as a register stack and each procedure is allocated a set of registers (from 0 to 96)
  - The new register stack frame is created for a called procedure by renaming the registers in hardware;
  - A special register called the current frame pointer (CFM) points to the set of registers to be used by a given procedure
- 8 64-bit Branch registers used to hold branch destination addresses for indirect branches
- 64 1-bit predict registers

- Both the integer and floating point registers support register rotation for registers 32-128.
- Register rotation eases the task of allocating registers in software pipelined loops
- Avoid the need for unrolling and for prologue and epilogue code for a software pipelined loop
  - Makes the SW-pipelining usable for loops with smaller numbers of iterations
“Explicitly Parallel Instruction Computer (EPIC)”

- **Instruction group**: a sequence of consecutive instructions with no register data dependences
  - All the instructions in a group could be executed in parallel (if no structural hazards and if any dependences through memory were preserved)
  - Instruction group can be arbitrarily long
  - Compiler must explicitly indicate the boundary between one instruction group and another by placing a stop between 2 instructions that belong to different groups
- IA-64 instructions are encoded in bundles, which are 128 bits wide.
  - Each bundle consists of a 5-bit template field and 3 instructions, each 41 bits in length
- 3 Instructions in 128 bit “groups”; field determines if instructions dependent or independent

---

### 5 Types of Execution in Bundle

<table>
<thead>
<tr>
<th>Execution Unit Slot</th>
<th>Instruction Unit Type</th>
<th>Instruction Description</th>
<th>Example Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>I-unit A</td>
<td>A</td>
<td>Integer ALU</td>
<td>add, subtract, and, or, cmp</td>
</tr>
<tr>
<td>I-unit I</td>
<td>I</td>
<td>Non-ALU Int</td>
<td>shifts, bit tests, moves</td>
</tr>
<tr>
<td>M-unit A</td>
<td>A</td>
<td>Integer ALU</td>
<td>add, subtract, and, or, cmp</td>
</tr>
<tr>
<td>M-unit M</td>
<td>M</td>
<td>Mem access</td>
<td>Loads, stores for int/FP regs</td>
</tr>
<tr>
<td>F-unit F</td>
<td>F</td>
<td>Floating point</td>
<td>Floating point instructions</td>
</tr>
<tr>
<td>B-unit B</td>
<td>B</td>
<td>Branches</td>
<td>Conditional branches, calls</td>
</tr>
<tr>
<td>L+X L+X</td>
<td>L+X</td>
<td>Extended</td>
<td>Extended immediates, stops</td>
</tr>
</tbody>
</table>

---

Computer Science 146
David Brooks
Template Examples

<table>
<thead>
<tr>
<th>Template</th>
<th>Slot 0</th>
<th>Slot 1</th>
<th>Slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>28</td>
<td>M</td>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>29</td>
<td>M</td>
<td>F</td>
<td>B</td>
</tr>
</tbody>
</table>

Predication Support

- Nearly all instructions are predicated
  - Conditional branches are predicated jumps!
- Compare/Test instructions set predicates
  - Ten different comparison tests + 2 predicate destinations
  - Written with result of comparison + complement
Speculation Support

- All INT registers have a 1-bit NaT (Not A Thing)
  - This is a poison bit (as discussed earlier)
  - Speculative loads generate these
  - All other instructions propagate them
- Deferred exceptions
  - Nonspeculative exceptions receive a NAT as a source operand there is an unrecoverable exception
  - Chk. s instructions can detect and branch to recovery code

Memory Reference Support

- Advanced Loads allow speculative memory references
  - Move loads ahead of potentially dependent stores
  - \textit{ALAT} table is allocated with register destination + memory address
  - Stores associatively lookup the table when they execute
    - Invalidate ALAT entries with same memory address
- Before using the value of the advanced load
  - Explicit check is needed to see if ALAT entry is valid
  - If it fails, can re-load the value or perform cleanup operation
Itanium™ Processor Silicon

Core Processor Die

4 x 1MB L3 cache

Itanium™ Machine Characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>800 MHz</td>
</tr>
<tr>
<td>Transistor Count</td>
<td>25.4M CPU; 295M L3</td>
</tr>
<tr>
<td>Process</td>
<td>0.18u CMOS, 6 metal layer</td>
</tr>
<tr>
<td>Package</td>
<td>Organic Land Grid Array</td>
</tr>
<tr>
<td>Machine Width</td>
<td>6 insts/clock: (4 ALU/MM, 2 Ld/St, 2 FP, 3 Br)</td>
</tr>
<tr>
<td>Registers</td>
<td>14 ported 128 GR &amp; 128 FR; 64 Predicates</td>
</tr>
<tr>
<td>Speculation</td>
<td>32 entry ALAT, Exception Deferral</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>Multilevel 4-stage Prediction Hierarchy</td>
</tr>
<tr>
<td>FP Compute Bandwidth</td>
<td>3.2 GFlops (DP/EP); 6.4 GFlops (SP)</td>
</tr>
<tr>
<td>Memory -&gt; FP Bandwidth</td>
<td>4 DP (8 SP) operands/clock</td>
</tr>
<tr>
<td>Virtual Memory Support</td>
<td>64 entry ITLB, 32/96 2-level DTLB, VHPT</td>
</tr>
<tr>
<td>L2/L1 Cache</td>
<td>Dual ported 96K Unified &amp; 16KD; 16K1</td>
</tr>
<tr>
<td>L2/L1 Latency</td>
<td>6 / 2 clocks</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>4MB, 4-way s.a., BW of 12.8 GB/sec;</td>
</tr>
<tr>
<td>System Bus</td>
<td>2.1 GB/sec; 4-way Glueless MP</td>
</tr>
<tr>
<td></td>
<td>Scalable to large (512+ proc) systems</td>
</tr>
</tbody>
</table>
10 Stage In-Order Core Pipeline

Front End
- Pre-fetch/Fetch of up to 6 instructions/cycle
- Hierarchy of branch predictors
- Decoupling buffer

Execution
- 4 single cycle ALUs, 2 lds/sts
- Advanced load control
- Predicate delivery & branch
- Nat/Exception/Retirement

Instruction Delivery
- Dispersal of up to 6 instructions on 9 ports
- Reg. remapping
- Reg. stack engine

Operand Delivery
- Reg read + Bypasses
- Register scoreboard
- Predicated dependencies
Itanium processor 10-stage pipeline

- Front-end (stages IPG, Fetch, and Rotate): prefetches up to 32 bytes per clock (2 bundles) into a prefetch buffer, which can hold up to 8 bundles (24 instructions)
  - Branch prediction is done using a multilevel adaptive predictor like P6 microarchitecture
- Instruction delivery (stages EXP and REN): distributes up to 6 instructions to the 9 functional units
  - Implements registers renaming for both rotation and register stacking.

Instruction Dispersal

- Dispersal Network
Itanium processor 10-stage pipeline

- Operand delivery (WLD and REG):
  - Accesses register file
  - Performs register bypassing
  - Accesses and updates a register scoreboard
    - Scoreboard used to detect when individual instructions can proceed, so that a stall of 1 instruction in a bundle need not cause the entire bundle to stall
  - Checks predicate dependences.

- Execution (EXE, DET, and WRB)
  - Executes instructions through ALUs and load/store units
  - Detects exceptions and posts NaTs
  - Retires instructions and performs write-back
  - Deferred exception handling via poison bits (NaTs)

- Predicate Delivery
  - Predicates generated in EXE delivered in DET and feed into retirement, branch execution, dependency detect
  - All instructions read operands and execute
  - Canceled at retirement
Performance of IA-64 Itanium?

- Itanium 800MHz (96KB)
- Itanium-2 1GHz (3MB)
- Itanium-2 1.5GHz (6MB)
- Pentium 4 3.4GHz (.5MB)
- Pentium 4 3.4GHz (2.5MB)
- IBM POWER4 1.7GHz (1.5MB)
- AMD Opteron 2.2GHz (1MB)

SPECint2K
SPECfp2K

Computer Science 146
David Brooks