Chapter 3

Instruction-Level Parallelism and Its Exploitation

Introduction

- Instruction level parallelism = ILP = (potential) overlap among instructions
- First universal ILP: pipelining (since 1985)
- Two approaches to ILP
  - Discover and exploit parallelism in hardware
    - Dominant in server and desktop market segments
    - Not used in PMD segment due to energy constraints
      - May be changing with Cortex-A9
  - Software-based discovery at compile time
    - Technical markets, scientific computing, HPC
    - Itanium is an example of aggressive software discovery
      - But mostly abandoned by the majority of server makers

Instruction-Level Parallelism Basics

- Goal: minimize CPI (maximize IPC)
- In a pipelined processor
  - CPI = ideal CPI + stalls (overheads):
    - Structural stalls
    - Data hazard stalls
    - Control stalls
- Fundamental unit for extracting parallelism is
  - Basic block (block of instructions between branches)
  - Branches disrupt analysis; add runtime dependence
- But typical basic blocks are small
  - 3-6 instruction (15%-25% branch frequency)
- Optimizing across branches is a must
  - Examples: loop-level parallelism, data parallelism (SIMD)

Instruction Parallelism Examples

- Loop-level parallelism
  - Loop unrolling (compiler)
  - Dynamic unrolling (superscalar scheduling)
- Data parallelism
  - Vector computers
    - Cray X1, X1E, X2; NEC SX-9
  - SIMT
    - GPUs
  - SIMD
    - Short SIMD (SSE, AVX, Intel Phi)

Types of Dependences

- Data dependences
- Name dependences
- Control dependences

Data Dependence: Basics

- Not all instructions can be executed in parallel
- Data dependent instructions have to be executed “in order”
  - Data dependence is a property of the code
- Instruction j is data dependent on instruction i if
  - Instruction i produces result used by instruction j
  - Instruction j depends on instruction k and k data depends on i
- Pipeline interlocks
  - With interlocks, data dependence causes a hazard and stall
  - Without interlocks, data dependence prohibits the compiler from scheduling instructions with overlap
- Data dependence conveys:
  - Possibility of a hazard (= negative side-effect if not “in order”)
  - The required order of instructions
  - Upper bound on achievable parallelism
Data Dependence Example

double F0, *R1

Loop:
  Load.D F0, 0(R1)  F0= array element  F0=R1[0]
  Add.D F4, F0, F2  add scalar in F2  F4=F0+F2
  Store.D F4, 0(R1)  store result  R1[0]=F4
  Add.I R1, R1, # -8  decrement by 8B  R1-=1
  Branch.NE R1, R2, Loop  if (R1!=R2)  goto Loop

Data Dependence Details

- Overcoming data dependence
  - Maintain the dependence by preventing the hazard
    - By compiler or by hardware scheduler
  - Eliminate dependence by transforming the code
- A separate topic
  - Dependences that flow through memory
    - Is R4[100] the same as R6[20]?  Is R4[20] the same as R6[20]?

Name Dependence

- Name dependence occurs when two instructions use the same register (or memory location) but there is no flow of data between them
  - Instruction i
  - Instruction j
- Two types of name dependence
  - Antidependence: Instruction i reads what instruction j writes
    - Store.D F4, 0(R1)
    - Add.I R1, R1, # -8
  - Output dependence: Instructions i and j both write
- Renaming is the common technique to deal with name dependence
  - Register renaming
  - Shadow registers

Data Hazards

- Hazards exists as a result of data or name dependence
  - Overlap of dependent (and nearby) instructions could change access order to instructions' operands
  - Avoiding hazards ensures program order
- Possible data hazards
  - RAW (read after write) ← true data dependence
    - Instruction i: write to x
    - Instruction j: read from x
  - WAW (write after write) ← output dependence
    - Instruction i: write to x
    - Instruction j: write to x
  - WAR (write after read) ← antidependence
    - Instruction i: read from x
    - Instruction j: write to x
  - RAR is not a hazard

Control Dependence

- Control dependence determines ordering of an instruction with respect to a branch instruction
  - The order must be preserved
  - The execution should be conditional
- Example:
  - If p1 { S1 }
  - If p2 { S2 }
  - S1 is control dependent on p1
  - S2 is control dependent on p2
  - S2 is not control dependent on p1
- Branches create barriers in code for potential code motion
  - It might be possible to violate control dependence but preserve correct execution with extra hardware
  - Speculative execution

Control Dependence Examples

- Exception handling
  - Add R2, R3, R4
  - Branch.equal0 R2, L1
  - Load R1, 0(R2)
  - L1: NoOp
  - No data dependence between Branch and Load
  - Load from wrong R2 could cause exception:
    - int *r2 = r3 + r4; y = r2[0] ? r2[0] : 0;
- Data flow
  - Add R1, R2, R3
  - Branch.equal0 R4, L
  - Subtract R1, R5, R6
  - L: NoOp
  - Or R7, R1, R8
Control Dependence: Software Speculation

- Ignoring control dependence may be possible after code analysis (liveness property)
  - Add R1, R2, R3
  - Branch.eq0 R12, Skip
  - Subtract R4, R5, R6
  - Add R5, R4, R9
  - Skip: Or R7, R8, R9
  - ; R4 is not used again (is dead)

Compiler Techniques for Exposing ILP

- Loop: Load F0, 0(R1)
  - Stall
  - Add F4, F0, F2
  - Stall
  - Stall
  - Store F4, 0(R1)
  - Add.i R1, R1, #-8
  - Branch.NoEq R1, R2, Loop

- Loop: Load F0, 0(R1)
  - Add.i R1, R1, #-8
  - Add F4, F0, F2
  - Stall
  - Stall
  - Store F4, 0(R1)
  - Branch.NoEq R1, R2, Loop

Instruction producing result | Instruction using result | Latency in clock cycles
------------------------------|--------------------------|-------------------------------
FP ALU op                     | Another FPU ALU op       | 3                             
FP ALU op                     | Store double             | 2                             
Load double                   | FP ALU op                | 1                             
Load double                   | Store double             | 0                             

Loop Unrolling Overview

- Loop unrolling simply copies the body of the loop multiple times, each copy operates on a new loop index

Benefits
  - Less branch instructions
  - Less pressure on branch predictor
  - Increased basic block size
  - Potential for more parallelism
  - Less instructions executed
  - For example: less increments of the loop counter

Downsides
  - Greater register pressure
  - Increased use of instruction cache
  - Could spill the instruction cache and cause cache thrashing

Original vs. Unrolled Loop

- Loop
  - F0 = R1[0]
  - F4 = F0 + F2
  - R1[0] = F4
  - R1 -= 1
  - if (R1 != R2) goto Loop

- Loop
  - F0 = R1[0]
  - F4 = F0 + F2
  - R1[0] = F4
  - R1 -= 1
  - if (R1 != R2) goto Loop

Unrolling with Generic Loops

- Given: \( k=0; k < N; ++k \)
  - Let's unroll 4 times
  - But what if \( N \) is not divisible by 4?

Solution:
  - First, unroll \( N \) times: for \( k=0; k < N; ++k \)
  - Then loop of group of 4:
    - for \( k = N \) times: \( k < N; k += 4 \)
    - // unroll 4 times for \( k, k+1, k+2, k+3 \)

- Refer to Chapter 4 and the technique called
  - Strip mining
Branch Prediction

- Instead of waiting for the branch to finish executing
  - Try to predict its behavior and act upon the prediction
- Requirements
  - Prediction must be cheaper than executing the branch instruction
    - Usually based on few bits of information
  - There has to be a way of dealing with wrong predictions
    - Beware of exceptions etc.
- Simple predictor
  - Keep a bit (or two) for a (fixed) number of branches
  - Every time a branch is taken increase the count
    - If N-consecutive executions resulted in "branch taken" then the next act as if the branch will be taken
    - If N-consecutive "branch not taken" then start predicting "not taken"

Correlating Branch Predictors

- Observation (based on existing codes)
  - Branches are correlated with each other
- Application: correlating branch predictors
- Instead of keeping of each branch individually, look also at the recent M branches
- (M, N) predictor
  - Uses behavior of last M branches
    - Total of 2^M branch decisions
  - Each predictor has N bits
- Advantages
  - Better prediction yield (always test on your own code!)
  - Little hardware require to implement it

Tournament Branch Predictors

- Problem
  - Branches might be badly mispredicted when moving between program scopes
  - The branch prediction information from the inner scope is inadequate for the outer scope
- Observation
  - There is locality in branching
    - Inner and outer loops, etc.
- Solution
  - Combine local and global information
- Typical predictors
  - Size: 8K-32K bits
  - Local predictors unchanged
- Examples: DEC Alpha, AMD Phenom and Opteron

Dynamic Scheduling Basics

- Simple techniques can only eliminate some data dependence stalls
  - Pipeline scheduling by compiler
  - Forwarding and bypassing
- Dynamic scheduling adds another level adding parallelism while maintaining the data flow
  - Some dependences are not known until runtime
  - The same binaries can run efficiently without recompilation
  - Compiler might not know the details of the micro-architecture
  - There could be unpredictable delays: multi-level caches
- Disadvantages
  - Substantial increase in hardware complexity
  - Exception handling (imprecise exceptions)

Dynamic Scheduling Details

- Dynamic scheduling breaks the “in order” execution
  - Out-of-order execution
    - Incoming instructions rearranged and unknown until runtime
  - Out-of-order completion
    - Retired instructions’ order depends on code, execution, delays
- New hazards to deal with
  - WAR
    - Possibility of overwriting a value that has not been read yet
      - Load F0, 0(R1) // a load from memory may be stalled for many cycles
      - Load R1, #1 // load of a constant takes only a few cycles
  - WAW
    - Writing twice to the same location
    - RAW hazards are still a problem
    - They always occur since they are called “true data dependence”

Dynamic Scheduling and Hazards

- \[ F_0 = F_2 / F_4 \]
- \[ F_6 = F_0 + F_8 \]
- \[ R_1[0] = F_6 \]
- \[ F_8 = F_10 - F_14 \]
- \[ F_6 = F_10 ^ * F_8 \]

- True data dependence (RAW)
- antidepenedence (WAR)
- Output dependence (WAW)
Register Renaming Example

- F0 = F2 / F4
- F6 = F0 + F8
- R1[0] = F6
- F8 = F10 – F14
- F6 = F10 * F8
- Only RAW hazards remain

Register Renaming Details
- Register renaming provided by reservation stations (RS)
- Each entry of RS contains
  - Instruction
  - Buffered operand values (when available)
  - References to instructions in RS that will provide values
- Operation
  - RS fetches and buffers an operand when available
    - Might bypass a register
  - Pending instructions indicate the RS where they send their output
  - Results broadcast on result bus (Common Data Bus)
  - Only the last output updates the register file
  - Upon instruction issue, registers are renamed with references to RS
- There may be more RS than registers!

Tomasulo Approach
- Introduced by Robert Tomasulo
  - Implemented in IBM 360/91 in its floating-point unit
  - IBM 360/91 had long memory and floating-point delays
    - Only 4 floating-point registers
    - Binary compatibility was important for IBM customers
- Modern processors use a variation of Tomasulo's approach
  - Also in use is a simpler algorithm called scoreboarding

Tomasulo's Algorithm
- Tomasulo's approach allows...
  - Out-of-order execution (as in scoreboarding in ARM A8)
    - Unlike scoreboarding, Tomasulo can handle anti- and output dependences by renaming (four-issue Intel i7)
    - Extension to handle speculation
- In Tomasulo, each instruction goes through three steps
  - Issue
    - FIFO queue maintains correct data flow
    - Transfer instruction to RS if available or structural stall
    - Rename registers to eliminate WAR and WAW (stall if no data)
  - Execute
    - Monitor bus for new data and distribute it to waiting RS (RAW)
    - Execute instructions in functional units when operands available
    - Write result (other RS, registers, store buffers)
    - Store buffer waits for address, value and memory unit(s)

Example Dynamic Execution: All Issued

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Execute</th>
<th>Write result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load f6, (r2)</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Load f2, (r3)</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult f0,f2,f4</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub f8,f2,f6</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Div f9,f0,f6</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add f6,f8,f12</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reservation stations
- Busy
- Op, Vj, Vk, Qj, Qk, A
- Load1
- Load2: yes, load, Reg[r3]
- Add1: yes, sub, Mem[r2]: Load2
- Add2: yes, add, Add1: Load2
- Add3: no
- Mul1: yes, mul, Reg[4]: Load2
- Mul2: yes, div, Mem[r2]: Mul1

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Issue</th>
<th>Execute</th>
<th>Write result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load1: Mul1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load2: f6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add1: f8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2: f8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mul1: f9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mul2: f9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Dynamic Execution: Mult Ready

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Issue</th>
<th>Execute</th>
<th>Write result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load f6, (r2)</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Load f2, (r3)</td>
<td>x</td>
<td>x</td>
<td>+</td>
</tr>
<tr>
<td>Mult f0, f2, f4</td>
<td>x</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Sub f8, f2, f6</td>
<td>x</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Div f9, f0, f6</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add f6, f8, f2</td>
<td>x</td>
<td>+</td>
<td></td>
</tr>
</tbody>
</table>

Reservation stations

<table>
<thead>
<tr>
<th>Busy</th>
<th>Op</th>
<th>Vj</th>
<th>Vk</th>
<th>Qj</th>
<th>Qk</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load1</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Load2</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add1</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add3</td>
<td>no</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td>yes</td>
<td>mul</td>
<td>Mem[r3]</td>
<td>Reg[f4]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td>yes</td>
<td>div</td>
<td>Mem[r2]</td>
<td>Mult1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reorder Buffer

- Another set of (invisible to programmer) registers for intermediate results
- ROB registers hold data after instruction completion but before instruction commit
- Each ROB entry (register) contains additional fields:
  - Instruction type
    - Branch (no destination), store (memory destination), register op (ALU or register destination)
  - Destination
    - Register number (for loads or ALU ops) or memory address (for stores)
  - Value
  - Ready
    - Indicates whether the supplying instruction completed its execution

Hardware Speculation Basics

- After dealing with data dependences, control dependences become an issue
  - Branch prediction is not as effective with multiple instructions in-flight
    - If predicted “taken”, conditional instructions are fetched and issued
  - Speculation allows to proceed almost as if branch was not there
    - Conditional instructions are fetched, issued, and executed
- Hardware speculation comprises
  - Dynamic branch prediction
  - (speculative) execution: instructions are executed and possibly undone
  - Dynamic scheduling
    - More basic blocks available after branches are speculated out of the instruction stream

Hardware Speculation Components

- Additional step in instruction execution
  - Issue, Execute, Write result, Commit
- Reorder buffer (ROB)
- Handling of...
  - Mispredictions
  - Mis-speculations
  - Exceptions

Reorder Buffer in Action

- Issue
  - Has to wait until ROB entry is available (in addition to RS entry)
- Execute
  - Results from Common Data Bus will have to end up on ROB
- Write result
  - Results have to be copied to ROB
- Commit (also called completion, graduation)
  - Normal commit: store result in destination, mark ROB entry as empty
  - Store commit: destination is a memory
  - Branch commit:
    - If correct prediction: no action needed
    - If incorrect prediction: ROB result is thrown away and instructions restarted at the correct branch point

Reorder Buffer Exception Handling

- Exceptions are not recognized until they are ready to commit
- ROB records exceptions
  - On mispredictions: flush the exception
  - Upon reaching the head of ROB: raise the exception
Speculation at Compile Time

- If (x==0) {
  a += 1
  b += 1
  c += 1
} else {
  a -= 1
}
- If (x==0) {
  a_copy = a+1
  b_copy = b+1
  c_copy = c+1
} if (x==0) {
  a = a_copy
  b = b_copy
  c = c_copy
}

Multiple Issue Execution

- All the techniques presented so far lead to ideal CPI=1
- For CPI to go below 1 there need to be multiple instructions retired most of the time
  - Too many stalls can quickly increase CPI above 1
  - See Amdahl law
- Most common flavors of multiple issue processors
  - Statically scheduled processors
    - In-order execution
    - Examples: MIPS, ARM
  - VLIW (very long instruction word) processors
    - Each cycle issues multiple (fixed number of) instructions
    - Examples: DSPs, Itaniums, some GPUs
  - Dynamically scheduled superscalar processors
    - Out-of-order execution
    - Examples: Intel i3-7, AMD Phenom, IBM POWER7

VLIW Disadvantages

- Static parallelism
  - Must be discovered and exploited early
  - Preferably by the compiler
  - Potential for intermediate representations, bytecodes
- Large code size
  - Parallelism relies on large basic blocks
  - Clever encoding or on-the-fly decompression may be needed
- Lack of hazard detection in lockstep execution
- Binary compatibility
  - Take code from 2-issue VLIW to (next-gen) 3-issue VLIW
  - Add a single ALU unit to the new processor and the old code will not take advantage of it
  - New (wider, with more functions) processors could change instruction encoding
    - ISA must provide for future hardware expansions

VLIW Processor Basics

- How many instructions per cycle?
  - Two-issue is common place
  - Four-issue is manageable
- Scheduling techniques
  - Local
    - Basic blocks
  - Global
    - Across branches
  - Trace
    - VLIW-specific
- Extensive loop unrolling to generate large basic blocks
- Disadvantages
  - Static parallelism, large code size, lack of hazard detection for lockstep execution, binary compatibility

Multiple Issue Taxonomy

<table>
<thead>
<tr>
<th>Common name</th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Distinguishing characteristic</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar (static)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Static</td>
<td>In-order execution</td>
<td>Mostly in the embedded space: MIPS and ARM (Cortex A8)</td>
</tr>
<tr>
<td>Superscalar (dynamic)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic</td>
<td>Some out-of-order execution but no speculation</td>
<td>None at present</td>
</tr>
<tr>
<td>Superscalar (speculative)</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic with speculation</td>
<td>Out-of-order execution with speculation</td>
<td>Intel Core i3-7; AMD Phenom; IBM POWER7</td>
</tr>
<tr>
<td>VLIW/LILW &amp; Static</td>
<td>Static</td>
<td>Primarily software</td>
<td>Static</td>
<td>All hazards determined and indicated by compiler (often implicitly)</td>
<td>Most examples are in signal processing, such as TI 6Cx. Also some GPUs</td>
</tr>
<tr>
<td>EPIC (Exp. Parallel Instruction Comp.)</td>
<td>Primarily static</td>
<td>Primarily software</td>
<td>Mostly static</td>
<td>All hazards determined and indicated explicitly by the compiler</td>
<td>Itanium</td>
</tr>
</tbody>
</table>
### VLIW Processors Basic Design

- Package multiple operations into one instruction
  - Instruction bundles
- Example VLIW processor
  - One integer instruction (or branch)
  - Two independent floating-point operations
  - Two independent memory references
  - Notice: there are restrictions on the instructions
- Must be enough parallelism in code to fill the available slots
  - Compiler: aggressive loop unrolling
  - Programmer: program restructuring

### Modern Microarchitectures

- Combine:
  - Dynamic scheduling
  - Multiple issue
  - Speculation
- Two approaches to dealing with dependences
  - Assign reservation stations and update pipeline control table in half clock cycles
    - Only supports 2 instructions/clock
  - Design logic to handle any possible dependencies between the instructions
    - Notice: design complexity
  - Hybrid approaches
- New bottleneck:
  - Issue logic

### Modern Multiple Issue

- Limit the complexity of a single instruction “bundle”
  - Limit bundle size
  - Limit classes of instruction in a bundle
    - One integer, two floating-point
  - With limited size, all dependences in a bundle can be examined
  - Dependences from a small bundle can also be fully encoded in RS
- Another bottleneck:
  - Completion/commit unit
  - Need multiple such units to keep up with incoming instructions

### Return Address Predictor

- Branch prediction deals with conditional branches
- Most unconditional branches come from function returns
- But the same function can be called from multiple sites
  - This may cause the branch prediction buffer to forget about return address from previous calls
- Solution
  - Create return address buffer organized as a stack