G22.2243-001
High Performance Computer Architecture

Lecture 9
Compiler Support for ILP (Review), ILP Limits,
Memory Hierarchy Design

October 30, 2002
Outline

• Announcements
  – Assignment 3 will be available tomorrow morning: Due November 13\textsuperscript{th}
    • All assignments must be handed in to receive a final grade
    • Do not plan for an “incomplete” (course is not offered on a regular basis)
  – Final exam on December 11\textsuperscript{th}: Room 402, WWH

• Last lecture: Architecture and compilation for VLIW Processors
• This lecture
  – Review of VLIW architectural support
  – Limits to ILP (cont’d)
  – Memory hierarchy design (caches)

[ Hennessy/Patterson CA:AQA (3rd Edition): Chapters 3, 4, 5]
(Review) Architectural Features in VLIW Processors

- VLIW processors rely on the compiler to identify a packet of instructions that can be issued in the same cycle
  - Compiler takes responsibility for scheduling instructions so that their dependences are satisfied
    - Question: How can a compiler predict memory-access latency?
    - Answer: It does not (it can guess), and needs some support from HW
      - Early days: Blocking caches, would cause all FUs to stall on a miss
      - Now: Hardware tracks dependences because of memory-access operations
  - Optimizations such as loop unrolling, software pipelining, software bubbling expose more ILP, allowing the compiler to build issue packets

- Architectural support helps compiler expose/exploit more ILP
(Review) Hardware Support for VLIW

- To expose more parallelism at compile time
  - Conditional or predicated instructions (see Lectures 6 and 7)
    - Predication registers in IA64
  - Allow the compiler to group instructions across branches

- To allow compiler to speculate, while ensuring program correctness
  - Issue: Speculative movement of instructions (before branches, reordering of loads/stores) must not cause exceptions
  - HW allows exceptions from speculative instructions to be ignored
    - Poison bits and Reorder Buffers (see Lectures 7 and 8)
  - HW tracks memory dependences between loads and stores
    - LDS (speculative load) and LDV (load verify) instructions
      - Check for intervening store
    - Variant: LDV instruction can point to fix-up code
(Review) Studies of the Limitations of ILP

- Start off with a hardware model of an ideal processor
  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 and an unbounded buffer of instructions available (predicts address)
  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
Fair bit of instruction-level parallelism, but how much of this stems from the ideal nature of assumptions

- Infinite registers, perfect jump/branch prediction, perfect alias analysis
More Realistic Hardware: Limiting the Instruction Window

- The dispatch unit typically only has access to a fixed number of instructions, which it can try to send to reservation stations
  - Limiting factor: operand checking
    - Scales as \#instr. completing/cycle \times window size \times \#operands/instr
More Realistic Hardware: Branch Impact
Instr. window = 2000, issue width = 64

Perfect  Tournament (Adaptive 2-bit and correlated)  2-bit  Static  None

Fortran Programs
More Realistic HW: Register Impact
Instr. window = 2000, issue width = 64, bpred = 8K adaptive

Number of renaming registers

Fortran Programs
More Realistic HW: Alias Impact
window = 2000, issue width = 64; bpred = 8k adaptive; 256 rename regs

Dynamic memory disambiguation (limited by size of load/store buffer)

IPC

Fortran Programs

Perfect  Global/stack perfect (assumes that all heap references conflict)
Compiler inspection  None
Realistic HW for 2001-2005: Window Impact
HW disambiguation, 1K Adaptive pred., 16-entry RAS, 64 rename regs

Issue as many as window size

IPC

Program

Fortran Programs

Infinite 256 128 64 32 16 8 4

gcc 10 10 10 9 8 6 4 3
expresso 15 15 13 10 8 6 4 2
li 12 12 11 11 9 6 4 3
fpppp 47 47 22 22 14 8 5 3
doduced 35 35 17 16 15 12 9 7
5 4 3
tomcatv 56 56 45 45 34 22 14 9
6 3
Beyond the Limits of the Study

- More aggressive optimizations
  - Address value prediction and speculation
    - Can help achieve results similar to near-perfect alias analysis
  - Speculating on multiple paths
    - Reduces recovery costs (hopefully some path is useful), and exposes more ILP

- Even perfect model has some limitations
  - WAR and WAW hazards through memory
    - Can arise due to reuse of stack locations
  - Unnecessary dependences (i.e., compilers can do better than assumed)
    - E.g., dependence on loop control variable can be eliminated by loop unrolling
  - Overcoming the data flow limit
    - Recent idea: **Value Prediction**
    - Speculate that a register will have a certain value, and then recover if this speculation turns out to be false
      - Can speculate both data values and address values (for alias elimination)
A Different Perspective: Multithreading

- So far: Parallelism among instructions in a single thread of control
- What if we interleave instructions from multiple threads of control?
  - These instructions are independent (modulo thread synchronization)
    - Different register sets per thread
  - Overall program finishes earlier
    - Note that behavior of a single thread has not been improved

- Do programs support this model?
  - Programming languages like Java
  - Loop-level parallelism (beyond software pipelining)

- Two alternative implementations being explored
  - Cycle-by-cycle multithreading (e.g., Tera)
    - Stalls because of hazards become less of an issue, but single-threaded programs take longer to run
  - Simultaneous multithreading (use up slots as they become available)
Memory Hierarchy Design
(Moving Outside the Processor)
Why Worry About the Memory Hierarchy?

- The course to this point has focused on processor performance issues
  - CPU cost/performance, ISA, Pipelined and dynamic execution
Processor-Memory Performance Gap “Tax”

- Fraction of processor area/transistors taken up by caches (~1997)

<table>
<thead>
<tr>
<th>Processor</th>
<th>% Area</th>
<th>% Transistors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(cost)</td>
<td>(power)</td>
</tr>
<tr>
<td>Alpha 21164</td>
<td>37%</td>
<td>77%</td>
</tr>
<tr>
<td>StrongArm SA110</td>
<td>61%</td>
<td>94%</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>64%</td>
<td>88%</td>
</tr>
</tbody>
</table>

2 dies per package: Proc/I$/$D$ + L2$

- Caches have no inherent value, only try to close performance gap
(Review) Cache Organization

- Caches is the name given to the first level of the memory hierarchy, encountered once the address leaves the CPU
  - It serves as a temporary place where frequently-used values can be stored
    - Retains the same name as in memory (different from registers)
  - To avoid having to go to memory every time this value is needed
    - Caches are faster (hence more expensive, limited in size) than DRAM

- Caches store values at the granularity of cache blocks (lines)
  - Larger than a single word: efficiency and spatial locality concerns
  - Cache hit if value in cache, else cache miss

- Effect of caches on CPU execution time

\[
\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{clock cycle time}
\]
\[
\text{Memory stall clock cycles} = (\text{Reads} \times \text{Read miss rate} \times \text{Read miss penalty} + \text{Writes} \times \text{Write miss rate} \times \text{Write miss penalty})
\]
\[
= \text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty}
\]
Four Questions for Memory Hierarchy Designers

Q1: Where can a block be placed in the upper level?  
(Block placement)  
– Fully Associative, Set Associative, Direct Mapped

Q2: How is a block found if it is in the upper level?  
(Block identification)  
– Tag per block

Q3: Which block should be replaced on a miss?  
(Block replacement)  
– Random, LRU

Q4: What happens on a write?  
(Write strategy)  
– Write Back or Write Through (with Write Buffer)
Question 1: Block Placement

Range of caches is really a continuum of levels of set associativity

Most caches today are direct-mapped (1-way), 2-way or 4-way associative
Question 2: Block Identification

- Caches have a **tag** on each block frame that gives the block address
  - All possible tags, where the block may be present, are checked in **parallel**
- Quick check of whether a block contains data: **Valid bit**
- Organization determines which (subset of) blocks need to be checked
  - View memory address as below

![Diagram of memory address]

- Selects “block” within set
- Selects the “set”

- Direct mapped caches: Only index
- Fully-associative caches: Only tag
Question 3: Block Replacement

- When a new block needs to be brought in (on demand), an existing cache block may need to be freed up
- Three commonly-used schemes
  (we only select a block within the appropriate “set”)
  - Random: Easiest to implement
  - Least-recently used (LRU)
  - First-in, first-out (FIFO): used as an approximation to LRU

- LRU outperforms Random and FIFO on smaller caches
  - FIFO outperforms Random
- Differences not as big for larger caches
  - Bigger benefit from avoiding misses in the first place
Question 4: Write Strategy

- When is memory updated with the contents of a store?
- **Issue**: Reads dominate cache traffic (writes typically 10% of accesses)
  - Optimization for read: Do tag checking and data transfer in parallel
  - Cannot do this for writes (also, only sub-portion of block needs update)

- Two write policies
  - **Write through**
    - Information written to both cache and memory
    - Simplifies replacement procedure (block is clean)
    - Also, simplifies data coherency (later in the course)
  - **Write back**
    - Information only written to the cache
    - Dirty bit keeps track of which blocks have data that needs to be sync-ed
    - Reduces memory bandwidth requirement (hence power)
  - Variants: With or without write-allocate

- Write stalls in write-through caches reduced using write buffers
The Alpha 21264 Data Cache

- 64KB cache, 64B blocks
- 2-way set associative, write-back, write allocate
- 44-bit physical address
  - 9-bit index
    - Identifies 2 blocks from 512 sets
  - 29-bit tag
    - Identifies which of 2 blocks
- Tag checking and data extraction proceed in parallel
Improving Cache Performance

CPU time = (CPU execution clock cycles + Memory stall clock cycles) x clock cycle time

Memory stall clock cycles = (Reads x Read miss rate x Read miss penalty + Writes x Write miss rate x Write miss penalty)

= Memory accesses x Miss rate x Miss penalty

- Above assumes 1-cycle to hit in cache
  - Hard to achieve in current-day processors (faster clocks, larger caches)
  - More reasonable to also include hit time in the performance equation

Average memory access time = Hit Time + Miss rate x Miss Penalty

Small/simple caches  Avoiding address translation Pipelined cache access Trace caches

Larger block size  Larger cache size  Higher associativity  Way prediction  Compiler optimizations

Multilevel caches  Critical word first  Read miss before write miss  Merging write buffers  Victim caches

Nonblocking caches  Hardware prefetching  Compiler prefetching
A. Reducing Cache Miss Penalty

- Miss penalty arises from having to go to memory to satisfy an access

Techniques minimize the time a processor needs to stall
- Multilevel caches
  - Defer access to larger, albeit slower caches
- Critical word first and early restart
- Read priority over write on miss
- Merging write buffer
- Victim caches
1. Reducing Miss Penalty via Multilevel Caches

- Average memory access time in a 2-level cache hierarchy

  Average memory access time = Hit time (L1) + Miss rate (L1) x Miss penalty (L1)
  Miss penalty (L1) = Hit time (L2) + Miss rate (L2) x Miss penalty (L2)

- Distinguish between two kinds of miss rates
  - **Local** miss rate = Miss rate (L1) or Miss rate (L2)
  - **Global** miss rate = Number of misses/total number of memory accesses
    = Miss rate (L1), but Miss rate (L1) x Miss rate(L2)

- Example: In 1000 memory references there are 40 misses in the L1 cache and 20 misses in the L2 cache
  - Local miss rates: 4% (L1), **50%** (L2) = 20/40
  - Global miss rates: 4% (L1), 2% (L2)
  - Avg. memory access time = 1 + 4% x (10 + 50% x 100)
    = 3.4 cycles
Multilevel Caches (cont’d)

- Doesn’t make much sense to have L2 caches smaller than L1 caches
- L2 needs to be significantly bigger to have reasonable miss rates
  - Cost of big L2 is smaller than big L1