CMSC 411
Computer Systems Architecture
Lecture 13
Memory Hierarchy (cont.)

Alan Sussman
als@cs.umd.edu

Administrivia

- Homework #3 due today
- Finish reading Appendix C of H&P, start reading Ch. 5, but only up through 5.3 for now

Cache review

- Cache memory can be organized as direct mapped, set associative, or fully associative
- Can be write-through or write-back
- Extra bits such as valid and dirty bits help keep track of the status of the cache

Example: Alpha 21064
Write Buffers for Write-Through Caches

Holds data awaiting write-through to lower level memory

Q. Why a write buffer?  
A. So CPU doesn't stall

Q. Why a buffer, why not just one register?  
A. Bursts of writes are common.

Q. Are Read After Write (RAW) hazards an issue for write buffer?  
A. Yes! Drain buffer before next read, or send read 1st after check write buffers.

Memory stalls (cont.)
• To reduce the impact of cache misses, can reduce any of three parameters:
  – main memory access time (miss penalty)
  – cache access (hit) time
  – miss rate

Example: Apple iMac G5

Managed by compiler  
Managed by hardware  
Managed by OS, hardware, application

<table>
<thead>
<tr>
<th>Reg</th>
<th>L1 Inst</th>
<th>L1 Data</th>
<th>L2</th>
<th>DRAM</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>64K</td>
<td>32K</td>
<td>512K</td>
<td>256M</td>
<td>80G</td>
</tr>
<tr>
<td>Latency Cycles, Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1, 0.6 ns</td>
<td>3, 1.9 ns</td>
<td>3, 1.9 ns</td>
<td>11, 6.9 ns</td>
<td>88, 55 ns</td>
<td>10^7, 12 ms</td>
</tr>
</tbody>
</table>

Goal: Illusion of large, fast, cheap memory

Let programs address a memory space that scales to the disk size, at a speed that is usually as fast as register access

How much do stalls slow a machine?
• Suppose that on pipelined MIPS, each instruction takes, on average, 2 clock cycles, not counting cache faults/misses
• Suppose, on average, there are 1.33 memory references per instruction, memory access time is 50 cycles, and the miss rate is 2%
• Then each instruction takes, on average:

\[ 2 + (0 \times .98) + (1.33 \times .02 \times 50) = 3.33 \text{ clock cycles} \]
iMac’s PowerPC 970: All caches on-chip

L1 (64K Instruction)

L1 (32K Data)

512K L2

5 Basic Cache Optimizations

- Reducing Miss Rate
  1. Larger Block size (compulsory misses)
  2. Larger Cache size (capacity misses)
  3. Higher Associativity (conflict misses)

- Reducing Miss Penalty
  4. Multilevel Caches

- Reducing hit time
  5. Giving Reads Priority over Writes
    - E.g., Read completes before earlier writes in write buffer

More Terminology (for homework)

- ‘write-allocate’
  - ensure block in cache before performing a write operation
- ‘write-no-allocate’
  - don’t allocate block in cache if not already there

Another write buffer optimization

- Write buffer mechanics, with merging
  - An entry may contain multiple words (maybe even a whole cache block)
  - If there’s an empty entry, the data and address are written to the buffer, and the CPU is done with the write
  - If buffer contains other modified blocks, check to see if new address matches one already in the buffer – if so, combine the new data with that entry
  - If buffer full and no address match cache and CPU wait for an empty entry to appear (meaning some entry has been written to main memory)
  - Merging improves memory efficiency, since multi-word writes usually faster than one word at a time
5 Basic Cache Optimizations

- Reducing Miss Rate
  1. Larger Block size (compulsory misses)
  2. Larger Cache size (capacity misses)
  3. Higher Associativity (conflict misses)

- Reducing Miss Penalty
  4. Multilevel Caches

- Reducing hit time
  5. Giving Reads Priority over Writes
    - E.g., Read complete before earlier writes in write buffer

Don’t wait for the whole block on cache miss

- Two ways to do this – suppose need the 10th word in a block:
  - Early restart: access the required word as soon as it is fetched, instead of waiting for the whole block
  - Critical word first: start the fetch with word 10, and fill in the first few later

Use a nonblocking cache

- With this optimization, the cache doesn’t stop for a miss, but continues to process later requests if possible, even though an earlier one is not yet fulfilled
  - Introduces significant complexity into cache architecture – have to allow multiple outstanding cache requests (maybe even multiple misses)
  - but this is what’s done in modern processors

So Far....

- Fully associative cache
  - memory block can be stored in any cache block
- Write-through cache
  - write (store) changes both cache and main memory right away
  - reads only require getting block on cache miss
- Write-back cache
  - write changes only cache
  - read causes write of dirty block to memory on a replace
- Reads easy to make fast, writes harder
  - read data from cache in parallel with checking address against tag of cache block
  - write must verify address against tag before update
- Reducing memory stalls
  - reduce miss penalty, miss rate, cache hit time
  - Reducing miss penalty
    - give priority to read over write misses
    - don’t wait for the whole block
    - use a non-blocking cache
Multi-level cache

- For example, if cache takes 1 clock cycle, and memory takes 50, might be a good idea to add a larger (but necessarily slower) secondary cache in between, perhaps capable of 10 clock cycle access
- Complicates performance analysis (see H&P), but 2nd level cache captures many of 1st level cache misses, lowering effective miss penalty
  - and 3rd level cache has same benefits for 2nd level cache
- Most modern machines have separate 1st level instruction and data caches, shared 2nd level cache
  - and off processor chip shared 3rd level cache

Victim caches

- To remember a cache block that has recently been replaced (evicted)
  - use a small, fully associative cache between a cache and where it gets data from
  - check the victim cache on a cache miss, before going to next lower-level memory
    » if found, swap victim block and cache block
  - reduces conflict misses

Victim caches (cont.)

Figure from H&P 3ed

Cache miss terminology

- Sometimes cache misses are inevitable:
  - The first time a block is used, need to bring it into cache (a compulsory miss)
  - If need to use more blocks at once than can fit into cache, some will bounce in and out (capacity miss)
  - In direct mapped or set associative caches, there are certain combinations of addresses that cannot be in cache at the same time (conflict miss)
How to reduce the miss rate?

- Use larger blocks
- Use more associativity, to reduce conflict misses
- Victim cache
- Pseudo-associative caches (won’t talk about this)
- Prefetch (hardware controlled)
- Prefetch (compiler controlled)
- Compiler optimizations

Increasing block size

- Want the block size large so don’t have to stop so often to load blocks
- Want the block size small so that blocks load quickly

Increasing block size (cont.)

- So large block size may reduce miss rates, but …
- Example:
  - Suppose that loading a block takes 80 cycles (overhead) plus 2 clock cycles for each 16 bytes
  - A block of size 64 bytes can be loaded in 80 + 2*64/16 cycles = 88 cycles (miss penalty)
  - If the miss rate is 7%, then the average memory access time is
    \[
    1 + 0.07 * 88 = 7.16 \text{ cycles}
    \]
Memory Access Times

Higher associativity

- A direct-mapped cache of size $N$ has about the same miss rate as a 2-way set-associative cache of size $N/2$
  - 2:1 cache rule of thumb (seems to work up to 128KB caches)
- But associative cache is slower than direct-mapped, so the clock may need to run slower
- Example:
  - Suppose that the clock for 2-way cache needs to run at a factor of 1.1 times the clock for 1-way cache
    - the hit time increases with higher associativity
  - Then the average memory access time for 2-way is $1.10 + \text{miss rate} \times 50$ (assuming that the miss penalty is 50)

Memory access time

Pseudo-associative cache

- Uses the technique of chaining, with a series of cache locations to check if the block is not found in the first location
  - e.g., invert most significant bit of index part of address (as if it were a set associative cache)
- The idea:
  - Check the direct mapped address
  - Until the block is found or the chain of addresses ends, check the next alternate address
  - If the block has not been found, bring it in from memory
- Three different delays generated, depending on which step succeeds

- If cache big enough, slowdown in access time hurts mem performance