Lecture 12: Memory Hierarchy Design Contd.

Kunle Olukotun
Gates 302
kunle@ogun.stanford.edu
http://www-leland.stanford.edu/class/ee282h/

Cache Problem

- Given the following data
  » 1 instruction reference per instruction
  » 0.27 loads/instruction
  » 0.13 stores/instruction
  » Memory access time = 4 cycles + # words/block
  » A 64KB cache with 4 W block size has a miss rate of 1.7%
  » Base CPI of 1.5

- Suppose the cache uses a write through, no write allocate, write around write strategy without a write buffer. How much faster would the machine be with a perfect write buffer?
No Write Buffer

Perfect Write Buffer
Improving Cache Performance

- AMAT = hit time + miss rate \times miss penalty
  - Reduce any of these to improve performance
- Start with miss rate
- Miss rate data needs to be interpreted carefully
  - Very dependent on application, language, operating system
  - Multitasking has a big effect
  - Miss rates for a given cache design tend to increase over time!

Understanding where misses come from: An intuitive model

- Compulsory: unavoidable first reference to a block
- Capacity: misses caused because the cache is too small
- Conflict: misses caused by mapping conflicts in the cache

<table>
<thead>
<tr>
<th>Type</th>
<th>Measure by</th>
<th>Reduce by</th>
<th>Impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compulsory (C1)</td>
<td>C1 = MR of infinite cache</td>
<td>increase block size</td>
<td>C2 and C3 can increase</td>
</tr>
<tr>
<td>Capacity (C2)</td>
<td>C2 = MR of fully associative cache - C1</td>
<td>increase cache size</td>
<td>increased cycle time and cost</td>
</tr>
<tr>
<td>Conflict (C3)</td>
<td>C3 = MR of set associative cache - C2 - C3</td>
<td>increase associativity</td>
<td>increased cycle time and cost</td>
</tr>
</tbody>
</table>

- Just a way to think about misses; reality is more complicated
  - Increasing cache size can eliminate some conflict misses.
Total Miss Rate

- Cache rules of thumb
  - cache size $N = 2$-way size $N/2$
  - double cache size removes 30% of misses

Miss rate vs. Cache size
Reducing misses:
Larger block size vs. Miss Penalty

- Increases miss penalty
- Decreases miss rate because data is pre-fetched
  - Effect reverses when a lot of prefetched data is not used (conflict misses)
- Reduces overhead (tag as % of data)
- Better for quick load of compulsory misses after task switch
- Large block sizes are better for I-caches than D-caches
Reducing misses: Larger block size vs. AMAT

- LA = latency (for 1st word)
- BW = transfer rate per byte
- AMAT = 1 + missrate \times (LA + blksize/BW)
- High LA and high BW favor large block sizes

Reducing misses: Victim cache

- Small, fully associative cache containing recently discarded blocks
- Lookup in parallel with main cache
- Think of it as higher associativity for a few of the cache lines
- Works best with small direct-mapped caches.
- Can remove 50% of conflict misses in 4KB DM with only 4 entries!
Reducing misses: Pseudo-associative caches

- Two-step cache check, to achieve:
  - the hit speed (almost) of a direct-mapped cache
  - the miss rate (almost) of a 2-way associative cache
- After a miss as a direct-mapped cache at location b(addr), make another check at a related location f(addr).
- Swap to make the most recent hit the first to be checked.

\[
\text{AMAT} = 1 + \text{direct-mapped miss rate} \times \text{2nd check cost} + \text{2-way miss rate} \times \text{miss penalty}
\]

Reducing misses: Hardware prefetching

- On a miss, fetch the requested block, and the next block(s).
- Store the next block “near” the cache, and prefetch again when used.
- Think of it as a bigger blocksize for one or a few of the cache lines.
- Can be used for instructions or data.
- Can remove 43% of the misses with 4 prefetch blocks in 4 KB DM!
Reducing misses: Software techniques

- Software prefetching
  - Load-to-register or load-to-cache instructions
  - Should be non-blocking to allow other cache references to proceed (more later)

  ```c
  for (i = 0; i < 1000; i++) {
      sum = sum + A[i];
  }
  for (i = 0; i < 1000; i++) {
      prefetch A[i+10];
      sum = sum + A[i];
  }
  ```

- Change the memory reference pattern
  - For code:
    - make the inline (non-branch) path the common case
    - isolate all exception-handling code together
  - For data: Improve locality
    - Merge arrays which are accessed in parallel
      ```c
      int a[100], b[100];
      ```
      becomes
      ```c
      struct {int a; int b;} ab[100];
      ```
    - Interchange loops for 2-dimensional arrays if access pattern is not sequential (row-wise vs. column-wise)
    - Fuse two consecutive loops through an array into one.
    - For large arrays, operate on sub-arrays rather than row-wise or column-wise (blocking)

Performance of Software Techniques

![Performance improvement graph](image)
Reducing the miss penalty:
Hardware tricks

- Give priority to reads over writes
  - Be careful to check for RAW hazards in the store buffer to memory
- Subblock allocation: only read part of a block
  - saves on tags and comparators compared to more associativity
  - works well with write-allocate no-fetch write miss policy
- Get the CPU what it needs as quickly as possible:
  - early restart: as soon as the needed word arrives, send it and continue fetching
  - critical word first (wrap-around fetch): ask for the needed word first, and continue fetching
- Nonblocking Cache
  - Don’t stall CPU for miss!

Nonblocking Cache

- First reference
  - Lockup free cache
  - Kroft, ISCA, 1981
- Hardware mechanisms that provide outstanding misses
  - One outstanding miss
    - “hit under miss”
    - hide miss latency with CPU execution and data references that hit
    - Used in PA7100, DEC Alpha 21064
  - Multiple outstanding misses
    - overlap multiple misses
    - requires more hardware than hit under miss
    - requires the next level up in the memory hierarchy to be pipelined
    - Used in UltraSPARC, R10000, DEC Alpha 21164
- Performance benefits from non-blocking loads depend on instruction scheduling or out-of-order execution and amount of ILP in application
Nonblocking Cache Potential

- Stall CPU on miss
- Hit under miss
- Stall only when result needed
- Multiple out-standing misses

Nonblocking Cache Hardware

- NBC terminology
  - *Primary miss* is the first to a main memory block
  - *Secondary misses* are the subsequent misses to a block
  - *Structural-stall misses* are the subsequent misses to a block that stall because of resource contention

- Miss Status Holding Registers (MSHRs)
  - Holds enough information on all outstanding misses to complete the load
  - Primary and secondary misses can use the same MSHR
Reducing the miss penalty: Two-level caches

- AMAT = L1 hit time +
  L1 miss rate x (L2 hit time + L2 miss rate x L2 miss penalty)
- This is based on local miss rate: misses/accesses to this cache
- The global miss rate: misses/ memory accesses by CPU
- The local miss rate of L2 caches is typically quite bad, since L1 has caught most locality.
  » Make it big, with big block size and high associativity to lower the miss rate, since the hit time doesn’t matter so much
- Want multi-level inclusion: everything at level i is also in level i+1
  » make analysis easier: solo miss rate of L2 = overall (global) miss rate = (local) miss rate of L1 x local miss rate of L2
  » I/O and multiprocessors only need to look at higher levels
  » Will be true if (not only if):
    – same blocksize and associativity
    – L2 is larger than L1
Reducing the hit time:  
**Hardware tricks**

- Use direct-mapped cache
  - no multiplexer in the path
  - data can be speculatively used while tag check is in progress

- Write hits take longer because tag must be checked before writing
  (read and write are in sequence). To fix, pipeline the writes:
  - WRITE1: readtag writecache
  - WRITE2: readtag writecache
  - (Ignore read hits, but watch for RAW hazards!)
  - Used by 21064, VAX 8800

- For write-through write-invalidate direct-mapped caches only:
  - always write regardless of tag!
    - use for subblock caches (multiple valid bits)
    - if the tag was right, it was the right block after all
    - if the tag was wrong, change the tag and invalidate other words
      (memory still has a good copy)
    - Do the write-through
Cache Improvement Summary

- Figure 5.29 from textbook