Caches

• Monday lecture
  – Review of cache basics, direct-mapped, set-associative caches

• Today
  – More on cache performance, write strategies
Summary of Set Associativity

- **Direct Mapped**
  - One place in cache, One Comparator, No Muxes

- **Set Associative Caches**
  - Restricted set of places
  - N-way set associativity
  - Number of comparators = number of blocks per set
  - N:1 mux

- **Fully Associative**
  - Anywhere in cache
  - Number of comparators = number of blocks in cache
  - N:1 mux needed
More Detailed Questions

• Block placement policy?
  – Where does a block go when it is fetched?

• Block identification policy?
  – How do we find a block in the cache?

• Block replacement policy?
  – When fetching a block into a full cache, how do we decide what other block gets kicked out?

• Write strategy?
  – Does any of this differ for reads vs. writes?
Block Placement + ID

• Placement
  – Invariant: block always goes in exactly one set
  – Fully-Associative: Cache is one set, block goes anywhere
  – Direct-Mapped: Block goes in exactly one frame
  – Set-Associative: Block goes in one of a few frames

• Identification
  – Find Set
  – Search ways in parallel (compare tags, check valid bits)
Block Replacement

- Cache miss requires a replacement
- No decision needed in direct mapped cache
- More than one place for memory blocks in set-associative
- Replacement Strategies
  - Optimal
    - Replace Block used furthest ahead in time (oracle)
  - Least Recently Used (LRU)
    - Optimized for temporal locality
  - (Pseudo) Random
    - Nearly as good as LRU, simpler
Write Policies

- Writes are only about 21% of data cache traffic
- Optimize cache for reads, do writes “on the side”
  - Reads can do tag check/data read in parallel
  - Writes must be sure we are updating the correct data and the correct amount of data (1-8 byte writes)
  - Serial process => slow
- What to do on a write hit?
- What to do on a write miss?
Write Hit Policies

• **Q1**: When to propagate new values to memory?
• **Write back** – Information is only written to the cache.
  – Next lower level only updated when it is evicted (dirty bits say when data has been modified)
  – Can write at speed of cache
  – Caches become temporarily inconsistent with lower-levels of hierarchy.
  – Uses less memory bandwidth/power (multiple consecutive writes may require only 1 final write)
  – Multiple writes within a block can be merged into one write
  – Evictions are longer latency now (must write back)
Write Hit Policies

• Q1: When to propagate new values to memory?
• **Write through** – Information is written to cache and to the lower-level memory
  – Main memory is always “consistent/coherent”
  – Easier to implement – no dirty bits
  – Reads never result in writes to lower levels (cheaper)
  – Higher bandwidth needed
  – Write buffers used to avoid **write stalls**
Write buffers

- Small chunks of memory to buffer outgoing writes
- Processor can continue when data written to buffer
- Allows overlap of processor execution with memory update

- Write buffers are essential for write-through caches

Computer Science 246
David Brooks
Write buffers

- Writes can now be pipelined (rather than serial)
  - Check tag + Write store data into Write Buffer
  - Write data from Write buffer to L2 cache (tags ok)
- Loads must check write buffer for pending stores to same address
- Loads Check:
  - Write Buffer
  - Cache
  - Subsequent Levels of Memory
Write buffer policies:
Performance/Complexity Tradeoffs

- Allow merging of multiple stores? (“coalescing”)
- “Flush Policy” – How to do flushing of entries?
- “Load Servicing Policy” – What happens when a load occurs to data currently in write buffer?
Write misses?

• Write Allocate
  – Block is allocated on a write miss
  – Standard write hit actions follow the block allocation
  – Write misses = Read Misses
  – Goes well with write-back

• No-write Allocate
  – Write misses do not allocate a block
  – Only update lower-level memory
  – Blocks only allocate on Read misses!
  – Goes well with write-through
# Summary of Write Policies

<table>
<thead>
<tr>
<th>Write Policy</th>
<th>Hit/Miss</th>
<th>Writes to</th>
</tr>
</thead>
<tbody>
<tr>
<td>WriteBack/Allocate</td>
<td>Both</td>
<td>L1 Cache</td>
</tr>
<tr>
<td>WriteBack/NoAllocate</td>
<td>Hit</td>
<td>L1 Cache</td>
</tr>
<tr>
<td>WriteBack/NoAllocate</td>
<td>Miss</td>
<td>L2 Cache</td>
</tr>
<tr>
<td>WriteThrough/Allocate</td>
<td>Both</td>
<td>Both</td>
</tr>
<tr>
<td>WriteThrough/NoAllocate</td>
<td>Hit</td>
<td>Both</td>
</tr>
<tr>
<td>WriteThrough/NoAllocate</td>
<td>Miss</td>
<td>L2 Cache</td>
</tr>
</tbody>
</table>
Cache Performance

CPU time = (CPU execution cycles + Memory Stall Cycles)*Clock Cycle Time

AMAT = Hit Time + Miss Rate * Miss Penalty

• Reducing these three parameters can have a big impact on performance
• Out-of-order processors can hide some of the miss penalty
Reducing Miss Penalty

- Have already seen two examples of techniques to reduce miss penalty
  - Write buffers give priority to read misses over writes
  - Merging write buffers
    - Multiword writes are faster than many single word writes

- Now we consider several more
  - Victim Caches
  - Critical Word First/Early Restart
  - Multilevel caches
Reducing Miss Penalty: Victim Caches

- Direct mapped caches => many conflict misses
- Solution 1: More associativity (expensive)
- Solution 2: Victim Cache
- Victim Cache
  - Small (4 to 8-entry), fully-associative cache between L1 cache and refill path
  - Holds blocks discarded from cache because of evictions
  - Checked on a miss before going to L2 cache
  - Hit in victim cache => swap victim block with cache block
Reducing Miss Penalty: Victim Caches

- Even one entry helps some benchmarks!
- Helps more for smaller caches, larger block sizes
Reducing Miss Penalty: Critical Word First/Early Restart

• CPU normally just needs one word at a time
• Large cache blocks have long transfer times
• Don’t wait for the full block to be loaded before sending requested data word to the CPU

• Critical Word First
  – Request the missed word first from memory and send it to the CPU and continue execution

• Early Restart
  – Fetch in order, but as soon as the requested block arrives send it to the CPU and continue execution
Review: Improving Cache Performance

• How to improve cache performance?
  – Reducing Cache Miss Penalty
  – Reducing Miss Rate
  – Reducing Miss Penalty/Rate via parallelism
  – Reducing Hit Time
Non-blocking Caches to reduce stalls on misses

- **Non-blocking cache** or **lockup-free cache** allow data cache to continue to supply cache hits during a miss
  - requires out-of-order execution
  - requires multi-bank memories

- **“hit under miss”** reduces the effective miss penalty by working during miss vs. ignoring CPU requests

- **“hit under multiple miss”** or **“miss under miss”** may further lower the effective miss penalty by overlapping multiple misses
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses
  - Requires multiple memory banks (otherwise cannot support)
  - Penium Pro allows 4 outstanding memory misses
Value of Hit Under Miss for SPEC

- FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26
- Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19
- 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss
Reducing Misses by Hardware Prefetching of Instructions & Data

- Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in “stream buffer” not the cache
  - On Access: check both cache and stream buffer
  - On SB Hit: move line into cache
  - On SB Miss: Clear and refill SB with successive lines

- Works with data blocks too:
  - Jouppi [1990] 1 data stream buffer got 25% misses from 4KB cache; 4 streams got 43%
  - Palacharla & Kessler [1994] for scientific programs for 8 streams got 50% to 70% of misses from 2 64KB, 4-way set associative caches

- Prefetching relies on having extra memory bandwidth that can be used without penalty
Hardware Prefetching

- **What to prefetch?**
  - One block ahead (spatially)
    - What will this work well for?
  - Address prediction for non-sequential data
    - Correlated predictors (store miss, next_miss pairs in table)
    - Jump-pointers (augment data structures with prefetch pointers)

- **When to prefetch?**
  - On every reference
  - On a miss (basically doubles block size!)
  - When resident data becomes “dead” -- how do we know?
    - No one will use it anymore, so it can be kicked out
Reducing Misses by Software Prefetching Data

• Data Prefetch
  – Load data into register (HP PA-RISC loads)
  – Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
  – Special prefetching instructions cannot cause faults; a form of speculative execution

• Prefetching comes in two flavors:
  – Binding prefetch: Requests load directly into register.
    • Must be correct address and register!
  – Non-Binding prefetch: Load into cache.
    • Can be incorrect. Faults?

• Issuing Prefetch Instructions takes time
  – Is cost of prefetch issues < savings in reduced misses?
  – Higher superscalar reduces difficulty of issue bandwidth
Reducing Hit Times

• Some common techniques/trends
  – Small and simple caches
    • Pentium III – 16KB L1
    • Pentium 4 – 8KB L1
  – Pipelined Caches (actually bandwidth increase)
    • Pentium – 1 clock cycle I-Cache
    • Pentium III – 2 clock cycle I-Cache
    • Pentium 4 – 4 clock cycle I-Cache
  – Trace Caches
    • Beyond spatial locality
    • Dynamic sequences of instruction (including taken branches)
Cache Bandwidth

• Superscalars need multiple memory access per cycle
• Parallel cache access: more difficult than parallel ALUs
  – Caches have state so multiple accesses will affect each other
• “True Multiporting”
  – Multiple decoders, read/write wordlines per SRAM cell
  – Pipeline a single port by “double pumping” Alpha 21264
  – Multiple cache copies (like clustered register file) POWER4
• Interleaved Multiporting
  – Cache divides into banks – two accesses to same bank =>
    conflict