Peer Instruction Correction

- A direct-mapped cache will never out-perform a 2-way set-associative cache of the same size.
  - I said "TRUE ... increased associativity!"
  - Right Answer "FALSE ... consider the following"  
  We have 4 byte cache, block size = 1 byte. Compare a 2-way set-associative cache (2 sets using LRU replacement) with a direct mapped cache (four rows).

<table>
<thead>
<tr>
<th>Time</th>
<th>2-way set-assoc.</th>
<th>Direct mapped</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>Miss, Load 0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Miss, Load 2</td>
<td>Miss, Load 1</td>
</tr>
<tr>
<td>3</td>
<td>Miss, Load 4</td>
<td>Hit!</td>
</tr>
<tr>
<td>4</td>
<td>Miss, Load 2</td>
<td>Miss, Load 1</td>
</tr>
<tr>
<td>5</td>
<td>Hit!</td>
<td>Miss, Load 1</td>
</tr>
</tbody>
</table>

Address Translation & 3 Concept tests

Virtual Address

<table>
<thead>
<tr>
<th>VPN</th>
<th>INDEX</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN</td>
<td>P. P. N.</td>
<td>Physical Page Number</td>
</tr>
<tr>
<td>V. P. N.</td>
<td>P. P. N.</td>
<td>Virtual Page Number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Cache</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag Data</td>
<td>Physical Page Number</td>
</tr>
<tr>
<td>TAG</td>
<td>INDEX Offset</td>
</tr>
</tbody>
</table>

40-bit virtual address, 16 KB page

Virtual Page Number (18 bits) Page Offset (12 bits)

36-bit physical address

Physical Page Number (18 bits) Page Offset (18 bits)

40-bit VA, 36b PA

2-way set-assoc. TLB, 256 "slots"; 40b VA:

TLB Entry: Valid bit, Dirty bit, Access Control (say 2 bits), Virtual Page Number, Physical Page Number

Access (2 bits) TLB Tag (7 bits) Physical Page No. (12 bits)

Number of bits in TLB Tag / Index / Entry?

1: 12 / 14 / 38 (TLB Tag / Index / Entry)
2: 14 / 12 / 40
3: 18 / 8 / 44
4: 18 / 8 / 58
Peer Instruction (3/3)

° 2-way set-associative, 64KB data cache, 64B block

° Data Cache Entry: Valid bit, Dirty bit, Cache tag + ? bits of Data

° Number of bits in Data cache Tag / Index / Offset / Entry?
  1: 12 / 9 / 14 / 87 (Tag/Index/Offset/Entry)
  2: 20 / 10 / 6 / 86
  3: 20 / 10 / 6 / 534
  4: 21 / 9 / 6 / 87
  5: 21 / 9 / 6 / 535

4 Qs for any Memory Hierarchy

° Q1: Where can a block be placed?
  - One place (direct mapped)
  - A few places (set associative)
  - Any place (fully associative)

° Q2: How is a block found?
  - Indexing (as in a direct-mapped cache)
  - Limited search (as in a set-associative cache)
  - Full search (as in a fully associative cache)
  - Separate lookup table (as in a page table)

° Q3: Which block is replaced on a miss?
  - Least recently used (LRU)
  - Random

° Q4: How are writes handled?
  - Write through (Level never inconsistent w/ lower)
  - Write back (Could be “dirty”, must have dirty bit)

Q1: Where block placed in upper level?

° Block 12 placed in 8 block cache:
  - Fully associative
  - Direct mapped
  - 2-way set associative
    - Set Associative Mapping = Block # Mod # of Sets

Q2: How is a block found in upper level?

° Direct indexing (using index and block offset), tag compares, or combination
° Increasing associativity shrinks index, expands tag

Q3: Which block replaced on a miss?

° Easy for Direct Mapped
° Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

Miss Rates

<table>
<thead>
<tr>
<th>Associativity</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 KB</td>
<td>5.2%</td>
<td>5.7%</td>
<td>4.7%</td>
</tr>
<tr>
<td>64 KB</td>
<td>1.9%</td>
<td>2.0%</td>
<td>1.5%</td>
</tr>
<tr>
<td>256 KB</td>
<td>1.15%</td>
<td>1.17%</td>
<td>1.13%</td>
</tr>
</tbody>
</table>

Q4: What to do on a write hit?

° Write-through
  - update the word in cache block and corresponding word in memory
° Write-back
  - update word in cache block
  - allow memory word to be “stale”
    => add ‘dirty’ bit to each line indicating that memory be updated when block is replaced
    => OS flushes cache before I/O !!!
° Performance trade-offs?
  - WT: read misses cannot result in writes
  - WB: no writes of repeated writes
Three Advantages of Virtual Memory

1) Translation:
- Program can be given consistent view of memory, even though physical memory is scrambled
- Makes multiple processes reasonable
- Only the most important part of program ("Working Set") must be in physical memory
- Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later

2) Protection:
- Different processes protected from each other
- Different pages can be given special behavior (Read Only, Invisible to user programs, etc).
- Kernel data protected from User programs
- Very important for protection from malicious programs ⇒ Far more "viruses" under Microsoft Windows
- Special Mode in processor ("Kernel more") allows processor to change page table/TLB

3) Sharing:
- Can map same physical page to multiple users ("Shared memory")

Why Translation Lookaside Buffer (TLB)?
- Paging is most popular implementation of virtual memory (vs. base/bounds)
- Every paged virtual memory access must be checked against Entry of Page Table in memory to provide protection
- Cache of Page Table Entries (TLB) makes address translation possible without memory access in common case to make fast

Bonus slide: Virtual Memory Overview (1/4)
- User program view of memory:
  - Contiguous
  - Start from some set address
  - Infinitely large
  - Is the only running program
- Reality:
  - Non-contiguous
  - Start wherever available memory is
  - Finite size
  - Many programs running at a time

Bonus slide: Virtual Memory Overview (2/4)
- Virtual memory provides:
  - Illusion of contiguous memory
  - All programs starting at same set address
  - Illusion of ~ infinite memory (2^32 or 2^64 bytes)
  - Protection

Bonus slide: Virtual Memory Overview (3/4)
- Implementation:
  - Divide memory into "chunks" (pages)
  - Operating system controls page table that maps virtual addresses into physical addresses
  - Think of memory as a cache for disk
  - TLB is a cache for the page table
Bonus slide: Virtual Memory Overview (4/4)
° Let’s say we’re fetching some data:
  • Check TLB (input: VPN, output: PPN)
    - hit: fetch translation
    - miss: check page table (in memory)
      – Page table hit: fetch translation
      – Page table miss: page fault, fetch page from disk to memory, return translation to TLB
  • Check cache (input: PPN, output: data)
    - hit: return value
    - miss: fetch value from memory

Address Map, Mathematically
V = {0, 1, ..., n - 1}  virtual address space (n > m)
M = {0, 1, ..., m - 1}  physical address space
MAP: V -> M \theta  address mapping function
MAP(a) = a' if data at virtual address a is present in physical address a' and a' in M
= \theta if data at virtual address a is not present in M

And in Conclusion...
° Virtual memory to Physical Memory Translation too slow?
  • Add a cache of Virtual to Physical Address Translations, called a TLB
° Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well
° Virtual Memory allows protected sharing of memory between processes with less swapping to disk