Caches and Virtual Memory
Caches

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

• Today
  – More on cache performance, write strategies
  – Virtual Memory
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
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
- There are many more
  - Victim Caches
  - Critical Word First/Early Restart
  - Multilevel caches
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
Virtual Memory

• Point we have been avoiding
  – Addresses generated by program are not the addresses that we use to access the memory (physical memory)
  – Why?
Virtual Memory: Motivation

• Original Motivation: Allow main memory to “act as a cache” for secondary storage (disks)
  – Physical memory expensive and not very dense (too small)
  – Programmers wrote “overlays” to load memory from disk
  – Programming nightmare, incompatible code across products

• Current Motivation: Use indirection of VM as a feature
  – Physical memories are quite large
  – Multiprogramming, sharing, relocation, protection
  – Fast program startup
  – Memory mapped files, networks
Virtual vs. Physical Memory
Virtual Memory: Cache Analogy

- Cache blocks/lines are called *pages or segments*
- Cache misses are *page faults or address faults*
- Processes use *virtual addresses (VA)*
- Physical memory uses *physical addresses (PA)*
- Addresses divided into page offset, page number
  - Virtual: Virtual Page Number (VPN)
  - Physical: Physical Page Number (PPN)
- *Addresses translation*: system maps VA to PA
  - E.g. 4KB pages, 32-bit machine, 64MB physical memory
  - 32-bit VA, 26-bit PA (\(\log_2 64\text{MB}\)), 12-bit page offset (\(\log_2 4\text{KB}\))
Virtual Memory: Cache Analogy

<table>
<thead>
<tr>
<th>Parameter</th>
<th>First-Level Cache</th>
<th>Virtual Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block (page) Size</td>
<td>16-128 Bytes</td>
<td>4KB – 64KB</td>
</tr>
<tr>
<td>Hit Time</td>
<td>1-3 clock cycles</td>
<td>50-150 clock cycles</td>
</tr>
<tr>
<td>Miss Penalty</td>
<td>8-150 clock cycles</td>
<td>1M-10M clock cycles</td>
</tr>
<tr>
<td>(access time)</td>
<td>(6-130 clock cycles)</td>
<td>(.8M – 8M clock cycles)</td>
</tr>
<tr>
<td>(transfer time)</td>
<td>(2-20 clock cycles)</td>
<td>(.2M – 2M clock cycles)</td>
</tr>
<tr>
<td>Miss Rate</td>
<td>0.1-10%</td>
<td>0.00001 –0.001%</td>
</tr>
<tr>
<td>Address Mapping</td>
<td>25-45bit PA to 14-20bit CacheAd</td>
<td>32-64 bit VA to 25-45 bit PA</td>
</tr>
<tr>
<td>Replacement Policy</td>
<td>Hardware Replacement</td>
<td>Software Replacement</td>
</tr>
<tr>
<td>Total Size</td>
<td>Independent of Address Space</td>
<td>Processor Address Space</td>
</tr>
<tr>
<td>Backing Store</td>
<td>Level 2 Cache</td>
<td>Physical Disk</td>
</tr>
</tbody>
</table>
System maps VA to PA

- Virtual Page Number (VPN) => Physical: Physical Page Number (PPN)
- OS/Hardware perform the mapping, *not* processes
- Same VPNs in different processes have different PPNs
  - **Protection**: processes cannot use each other’s PA
  - **Programming** Simplicity: Each process thinks its alone
  - **Relocation**: Program can be run anywhere in memory
    - Doesn’t have to be physically contiguous
    - Can be paged out, paged back to a different physical location
Virtual Memory: 4 Cache Questions

- Same four questions as caches
  - Page Placement: fully associative
    • Why?
  - Page Identification: address translation
    • Indirection through one or two page tables
  - Page Replacement: Sophisticated LRU + Working set
    • Why?
  - Write Strategy: Always write-back + write allocate
    • Why?
Why?

• Backing store to main memory is disk
  – Memory is 50-100x slower than processor
  – Disk is 20-100 thousand times slower than memory
    • Disk is 1 to 10 million times slower than processor

• VA miss (page fault) is expensive
  – Minimize at all costs
  – Fully associative + Software Replacement reduce miss rate
  – Write-back reduces disk traffic
  – Large page sizes (4KB – 16KB) amortize reads
Page ID: Address Translation

- OS performs address translation using page table
  - Each process has its own page table
    - OS knows address of each process’s page table
  - Page table is an array of Page Table Entries
    - One entry for each VPN of each process, indexed by VPN
  - Each PTE contains
    - Phys. Page Number
    - Permissions
    - Dirty bit
    - LRU
    - ~4 bytes total
Page Table Size

• Page Table Size
  – Example #1: 32-bit VA, 4KB pages, 4-byte PTE
    • 1M Pages, 4MB Page Table
  – Example #2: 64-bit VA, 4KB pages, 4-byte PTE
    • 4P Pages, 16PB page table

• Page table reduction techniques
  – Multi-level page tables
  – Inverted page tables
Multi-Level Page Tables

- Most processes use only a tiny portion of total VA space
- Tree of page tables
  - L1 table points to L2 tables (and more if needed)
    - Different VPN bits are offsets at different levels
  - Save space: not all tables at all levels need to exist
  - Slow: Multi-hop chains of translations (space savings outweigh)

PT Root → VPN → 1st Level PT → 2nd Level PTs
Multi Level Page Tables

- 32-bit address space, 4KB pages, 4 byte PTEs
- 2 level virtual page table
- 2nd-level tables are each the size of 1 data page
- Program uses only upper and lower 1MB of address space
- How much memory does page table take?
Multi Level Page Tables

- 32-bit address space, 4KB pages, 4 byte PTEs
- 2 level virtual page table
- 2\textsuperscript{nd}-level tables are each the size of 1 data page
- Program uses only upper and lower 1MB of address space
- How much memory does page table take?
  - 4GB VM / 4KB pages => 1M pages
  - 4KB pages / 4B PTEs => 1K pages per 2\textsuperscript{nd} level table
  - 1M pages / 1K pages per 2\textsuperscript{nd} level table => 1K 2\textsuperscript{nd}-level tables
  - 1K 2\textsuperscript{nd} level tables + virtual page table => 4KB first level table
  - 1MB VA space + 4KB pages => 256 PTEs => 1 2\textsuperscript{nd} level table
  - Memory = 1\textsuperscript{st} level table (4KB) + 2 * 2\textsuperscript{nd} level table (4KB) = 12KB
Address Translation

• How does address translation really work?
• Two-level mapped page tables
  – Several levels of indirection: 3 memory accesses for 1 virtual memory access (slow!)
  – Processes do not read page table + translate: system does
• Hardware involvement: Translation Lookaside Buffer
  – Cache dedicated to these translations
Fast Translation: Virtual Caches

- First-level caches are “virtually addressed”
- L2 and main memory are “physically addressed”
- Address translation only on a miss (not critical)
- Why not?
  - Protection: xlate checks page level protection
  - Context switch: Cache flush required (PID tags?)
  - I/O: typically uses PAs (would need conversion to access L1 cache)
  - Synonyms: 2 VAs => 1 PA (2 copies in cache)
Synonyms: Another problem with Virtual Caches

- VA => PA is not always unique (sharing among processes)
- Memory location could be fetched into the cache by two different virtual addresses: consistency problem
- Solutions
  - Eliminate/Restrict sharing
  - Restrict sharing within a process, flush on context switch
  - Search all possible synonymous sets in parallel
  - Restrict page placement in OS such that index(VA) = index(PA)
Fast Translation: Physical Caches with Translation Buffers

- Solution #2: First level caches are physical
  - Address translation before every cache access
  - Works fine with I/O, address space changes
  - SLOW

- Solution #2a: Cache recent translations in TB
  - Only go to page table on TB miss
    - Hit time problem: still 2 serial accesses
Fast Translation: Physical Caches with Translation Lookaside Buffers

- Solution #3: Address translation & L1 cache access in parallel
  - Translation lookaside buffer (TLB)
  - Fast (one step access)
  - No problems changing VA spaces
  - Keeps I/O coherent
Cache: Virtual Index, Physical Tag

- Physical cache with virtual address
  - Only cache index matters for access
  - Only part of virtual address changes during translation
  - Make sure index is in untranslated part
    - Index is within page offset
    - Virtual index == physical index
      - Fast
      - Restricts cache size (Block size * #sets <= page size)
      - Use associativity to increase size
Basic TLB Organization

- Fully Associative Structure
- Example: VA = 44bits, Page Size = 4MB, PA Space = 1GB
- VPN bits = \text{bits (VA)} - \log_2(\text{page size}) = 44 - 22 = 22 \text{ bits}
- Physical Addr. = \log_2(\text{PA Size}) = 30 \text{ bits} (8 PPN + 22 Page Offset)
Selecting Page Size

• Larger Page Size
  – Page table is smaller (inversely proportional to page size)
  – Larger page size may allow larger caches with virtually indexed, physically tagged caches (larger page offset)
  – Page transfers can be more efficient
  – More efficient TLB => reduces number of TLB misses

• Smaller Page Size
  – Internal fragmentation: contiguous region of virtual memory not a multiple of the page size
  – Process startup time (load in large pages for small processes)

• Multiple Page Sizes
  – Some processors support multiple choices => larger pages are powers of 2 times the smaller page sizes
Overall Memory System

Page Size: 8KB
256-entry TLB
8KB L1 Cache
4MB L2 Cache
VA 64 bits
PA 41 bits