Recap: Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel
- Example: Two-way set associative cache
  - Cache Index selects a “set” from the cache
  - The two tags in the set are compared to the input in parallel
  - Data is selected based on the tag result

Recap: How is a block found if it is in the upper level?

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

Recap: Cache Performance

Execution Time = Instruction Count x Cycle Time x (ideal CPI + Memory Stalls/Inst + Other Stalls/Inst)

Memory Stalls/Inst = Instruction Miss Rate x Instruction Miss Penalty + Loads/Inst x Load Miss Rate x Load Miss Penalty + Stores/Inst x Store Miss Rate x Store Miss Penalty

Average Memory Access time (AMAT) =
  Hit Time L1 + (Miss Rate L1 x Miss Penalty L1) = (Hit Rate L1 x Hit Time L1) + (Miss Rate L1 x Miss Time L1)
Recap: A Summary on Sources of Cache Misses

- **Compulsory** (cold start or process migration, first reference): first access to a block
  - “Cold” fact of life: not a whole lot you can do about it
  - Note: If you are going to run “billions” of instruction, Compulsory Misses are insignificant

- **Conflict** (collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity

- **Capacity**:
  - Cache cannot contain all blocks access by the program
  - Solution: increase cache size

Recap: 3Cs Absolute Miss Rate (SPEC92)

Recap: 3Cs Relative Miss Rate

The Big Picture: Where are We Now?

- **The Five Classic Components of a Computer**
  - Processor
  - Control
  - Datapath
  - Memory
  - Output

- **Today’s Topics:**
  - Recap last lecture
  - Virtual Memory
  - Protection
  - TLB
  - Buses
### Recap: Miss-Rate Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>MR</th>
<th>MP</th>
<th>HT</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger Block Size</td>
<td>+</td>
<td>-</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>+</td>
<td>-</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Victim Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>HW Prefetching of Instr/Data</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

### Improving Cache Performance (Continued)

1. Reduce the miss rate,
2. \textit{Reduce the miss penalty}, or
3. Reduce the time to hit in the cache.

### 0. Reducing Penalty: Faster DRAM / Interface

- **New DRAM Technologies**
  - RAMBUS - same initial latency, but much higher bandwidth
  - Synchronous DRAM
  - TMJ-RAM from IBM??
  - Merged DRAM/Logic - IRAM project here at berkeley
- **Better BUS interfaces**
- **CRAY Technique: only use SRAM**

### 1. Reducing Penalty: Read Priority over Write on Miss

- **A Write Buffer Allows reads to bypass writes**
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- **Write buffer is just a FIFO:**
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) \(<\) 1 / DRAM write cycle
- **Memory system designer’s nightmare:**
  - Store frequency (w.r.t. time) \(>\) 1 / DRAM write cycle
  - Write buffer saturation
1. Reducing Penalty: Read Priority over Write on Miss

- Write through with write buffers offer RAW conflicts with main memory reads on cache misses
- If simply wait for write buffer to empty, might increase read miss penalty (old MIPS 1000 by 50%)
- Check write buffer contents before read; if no conflicts, let the memory access continue
- Write Back?
  - Read miss replacing dirty block
  - Normal: Write dirty block to memory, and then do the read
  - Instead copy the dirty block to a write buffer, then do the read, and then do the write
  - CPU stall less since restarts as soon as do read

2. Reduce Penalty: Early Restart and Critical Word First

- Don’t wait for full block to be loaded before restarting CPU
  - Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Also called wrapped fetch and requested word first
- Generally useful only in large blocks,
- Spatial locality a problem; tend to want next sequential word, so not clear if benefit by early restart

3. Reduce Penalty: Non-blocking Caches

- Non-blocking cache or lockup-free cache allow data cache to continue to supply cache hits during a miss
  - Requires F/E bits on registers or 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)
  - Pentium 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
4. Reduce Penalty: Second-Level Cache

- **L2 Equations**
  \[ \text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times \text{Miss Penalty}_{L1} \]
  \[ \text{Miss Penalty}_{L1} = \text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2} \]
  \[ \text{AMAT} = \text{Hit Time}_{L1} + \text{Miss Rate}_{L1} \times (\text{Hit Time}_{L2} + \text{Miss Rate}_{L2} \times \text{Miss Penalty}_{L2}) \]

- **Definitions:**
  - **Local miss rate**—misses in this cache divided by the total number of memory accesses to this cache \( (\text{Miss rate}_{L2}) \)
  - **Global miss rate**—misses in this cache divided by the total number of memory accesses generated by the CPU \( (\text{Miss Rate}_{L1} \times \text{Miss Rate}_{L2}) \)
  - **Global Miss Rate is what matters**

---

Reducing Misses: which apply to L2 Cache?

- **Reducing Miss Rate**
  1. Reduce Misses via Larger Block Size
  2. Reduce Conflict Misses via Higher Associativity
  3. Reducing Conflict Misses via Victim Cache
  4. Reducing Misses by HW Prefetching Instr, Data
  5. Reducing Misses by SW Prefetching Data
  6. Reducing Capacity/Conf. Misses by Compiler Optimizations

---

L2 cache block size & A.M.A.T.

![Graph showing Relative CPU Time vs Block Size]

- **32KB L1, 8 byte path to memory**

---

Reducing Miss Penalty Summary

- **Five techniques**
  - Faster Main Memory
  - Read priority over write on miss
  - Early Restart and Critical Word First on miss
  - Non-blocking Caches (Hit under Miss, Miss under Miss)
  - Second Level Cache

- **Can be applied recursively to Multilevel Caches**
  - Danger is that time to DRAM will grow with multiple levels in between
  - First attempts at L2 caches can make things worse, since increased worst case is worse
Second midterm coming up (Wed, November 17)
Will be 5:30 - 8:30 in 277 Cory. LaVal’s for pizza afterwards!
- Pipelining
  - Hazards, branches, forwarding, CPI calculations
  - (may include something on dynamic scheduling)
- Memory Hierarchy (including Caches, TLBs, DRAM)
- Simple Power issues
- Possibly I/O
- Review Session this Sunday at 7:00 in 306 Soda

Computers in the news: Microsoft is a Monopoly!
- (Surprised???)
- Not clear what the effect will be. Stock down about $3.

Example: Harvard Architecture?
- Unified vs Separate I&D (Harvard)
- Sample Statistics:
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%
- Which is better (ignore L2 cache)?
  - Assume 75% instructions, hit time=1, miss time=50
  - Note: data hit has 1 stall for unified cache (only one port)

\[
\text{AMAT}_{\text{Harvard}} = 75\% \times (1 + 0.64\% \times 50) + 25\% \times (1 + 6.47\% \times 50) = 2.05 \\
\text{AMAT}_{\text{Unified}} = 75\% \times (1 + 1.99\% \times 50) + 25\% \times (1 + 1 + 1.99\% \times 50) = 2.24
\]
What is virtual memory?

- Virtual memory => treat memory as a cache for the disk
- Terminology: blocks in this cache are called “Pages”
- Typical size of a page: 1K — 8K
- Page table maps virtual page numbers to physical frames

Virtual Address Space

<table>
<thead>
<tr>
<th>Physical Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>V page no.</td>
</tr>
<tr>
<td>P page no.</td>
</tr>
</tbody>
</table>

Page Table

Virtual Address

<table>
<thead>
<tr>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Physical Address

Issues in Virtual Memory System Design

What is the size of information blocks that are transferred from secondary to main storage (M)? \( \Rightarrow \) page size
(Contrast with physical block size on disk, i.e. sector size)

Which region of M is to hold the new block \( \Rightarrow \) placement policy

How do we find a page when we look for it? \( \Rightarrow \) block identification

Block of information brought into M, and M is full, then some region of M must be released to make room for the new block \( \Rightarrow \) replacement policy

What do we do on a write? \( \Rightarrow \) write policy

Missing item fetched from secondary memory only on the occurrence of a fault \( \Rightarrow \) demand load policy

Three Advantages of Virtual Memory

- Translation:
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Makes multithreading reasonable (now used a lot!)
  - 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.

- Protection:
  - Different threads (or 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

- Sharing:
  - Can map same physical page to multiple users (“Shared memory”)
How big is the translation (page) table?

- **Simplest way to implement “fully associative” lookup policy** is with large lookup table.
- **Each entry in table is some number of bytes, say 4**
- **With 4K pages, 32-bit address space, need:** $2^{32}/4K = 2^{20} = 1 \text{ Meg entries x 4 bytes} = 4 \text{MB}
- **With 4K pages, 64-bit address space, need:** $2^{64}/4K = 2^{52} \text{ entries} = \text{BIG!}
- **Can’t keep whole page table in memory!**

Large Address Spaces

Two-level Page Tables

- 32-bit address: P1 index, P2 index, page offset
- $4K$ PTEs
- **2 GB virtual address space**
- **4 MB of PTE2**
- **4 KB of PTE1**
- **What about a 48-64 bit address space?**

Inverted Page Tables

IBM System 38 (AS400) implements 64-bit addresses.
48 bits translated
start of object contains a 12-bit tag

Virtual Address and a Cache: Step backward???

- **Virtual memory seems to be really slow:**
  - we have to access memory on every access -- even cache hits!
  - Worse, if translation not completely in memory, may need to go to disk before hitting in cache!
- **Solution: Caching! (surprise!)**
  - Keep track of most common translations and place them in a “Translation Lookaside Buffer” (TLB)
Making address translation practical: TLB

- Virtual memory => memory acts like a cache for the disk
- Page table maps virtual page numbers to physical frames
- Translation Look-aside Buffer (TLB) is a cache of recent translations

Virtual Address Space  Physical Address Space

TLB organization: include protection

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Dirty</th>
<th>Ref</th>
<th>Valid</th>
<th>Access</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFA00</td>
<td>0x0003</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>R/W</td>
<td>34</td>
</tr>
<tr>
<td>0x0040</td>
<td>0x0010</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>0x0041</td>
<td>0x0011</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>R</td>
<td>0</td>
</tr>
</tbody>
</table>

- TLB usually organized as fully-associative cache
  - Lookup is by Virtual Address
  - Returns Physical Address + other info
- Dirty => Page modified (Y/N)?
- Ref => Page touched (Y/N)?
- Valid => TLB entry valid (Y/N)?
- Access => Read? Write?
- ASID => Which User?

R3000 TLB & CP0 (MMU)

Entry HI, Entry Lo

R3000 Pipeline

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Dcd/ Reg</th>
<th>ALU / E.A</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>WB</td>
</tr>
<tr>
<td>E.A.</td>
<td>TLB</td>
<td>D-Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Constraints on TLB organization

- 64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space

<table>
<thead>
<tr>
<th>ASID</th>
<th>V. Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among 64 user processes without TLB flush
What is the replacement policy for TLBs?

° On a TLB miss, we check the page table for an entry.

° Two architectural possibilities:
  - Hardware “table-walk” (Sparc, among others)
    - Structure of page table must be known to hardware
  - Software “table-walk” (MIPS was one of the first)
    - Lots of flexibility
    - Can be expensive with modern operating systems.

° What if missing virtual entry is not in page table?
  - This is called a “Page Fault”
  - A “Page Fault” means that requested virtual page is not in memory.
  - Operating system must take over.

° Note: possible that parts of page table are not even in memory (i.e. paged out!)
  - The root of the page table always “pegged” in memory

Page Fault: What happens when you miss?

° Page fault means that page is not resident in memory

° Hardware must detect situation

° Hardware cannot remedy the situation

° Therefore, hardware must trap to the operating system so that it can remedy the situation
  - pick a page to discard (possibly writing it to disk)
  - start loading the page in from disk
  - schedule some other process to run

Later (when page has come back from disk):
  - update the page table
  - resume to program so HW will retry and succeed!

° What is in the page fault handler?
  - see CS162

° What can HW do to help it do a good job?

Page Replacement: Not Recently Used (1-bit LRU, Clock)

Associated with each page is a reference flag such that
ref flag = 1 if the page has been referenced in recent past
= 0 otherwise

-- if replacement is necessary, choose any page frame such that its reference bit is 0. This is a page that has not been referenced in the recent past

Page table entry

<table>
<thead>
<tr>
<th>dirty</th>
<th>used</th>
<th>page table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>page table entry</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>page table entry</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>page table entry</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>page table entry</td>
</tr>
</tbody>
</table>

Page fault handler:

last replaced pointer (lrp)
if replacement is to take place,
advance lrp to next entry (mod table size) until one with a 0 bit is found; this is the target for replacement; As a side effect, all examined PTE’s have their reference bits set to zero.

Or search for the a page that is both not recently referenced AND not dirty.

Architecture part: support dirty and used bits in the page table
⇒ may need to update PTE on any instruction fetch, load, store

How does TLB affect this design problem? Software TLB miss?
Reducing Translation Time

Still have TLB translation time in serial with cache lookup!

Machines with TLBs go one step further to reduce # cycles/cache access

They overlap the cache access with the TLB access

Works because high order bits of the VA are used to look in the TLB while low order bits are used as index into cache.

Reducing translation time further

- As described, TLB lookup is in serial with cache lookup:

![Diagram showing TLB and cache lookup process]

- Machines with TLBs go one step further: they overlap TLB lookup with cache access.
  - Works because lower bits of result (offset) available early

Overlapped TLB & Cache Access

- If we do this in parallel, we have to be careful, however:

![Diagram showing overlapped access and cache setup]

What if cache size is increased to 8KB?

Problems With Overlapped TLB Access

Overlapped access only works as long as the address bits used to index into the cache do not change as the result of VA translation.

This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache.

Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:

![Diagram showing cache size increase and related issues]

Solutions:
- go to 8K byte page sizes;
- go to 2 way set associative cache; or
**Another option: Virtually Addressed Cache**

**Only require address translation on cache miss!**

**synonym problem:** two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address!

nightmare for update: must update all cache entries with same physical address or memory becomes inconsistent
determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits.
(usually disallowed by fiat)

---

**Optimal Page Size**

- Minimize wasted storage
  - small page minimizes internal fragmentation
  - small page increase size of page table
- Minimize transfer time
  - large pages (multiple disk sectors) amortize access cost
  - sometimes transfer unnecessary info
  - sometimes prefetch useful data
  - sometimes discards useless data early

General trend toward larger pages because
- big cheap RAM
- increasing mem / disk performance gap
- larger address spaces

---

**Survey**

- R4000
  - 32 bit virtual, 36 bit physical
  - variable page size (4KB to 16 MB)
  - 48 entries mapping page pairs (128 bit)
- MPC601 (32 bit implementation of 64 bit PowerPC arch)
  - 52 bit virtual, 32 bit physical, 16 segment registers
  - 4KB page, 256MB segment
  - 4 entry instruction TLB
  - 256 entry, 2-way TLB (and variable sized block xlate)
  - overlapped lookup into 8-way 32KB L1 cache
  - hardware table search through hashed page tables
- Alpha 21064
  - arch is 64 bit virtual, implementation subset: 43, 47,51,55 bit
  - 8,16,32, or 64KB pages (3 level page table)
  - 12 entry ITLB, 32 entry DTLB
  - 43 bit virtual, 28 bit physical octword address

---

**Alpha VM Mapping**

- “64-bit” address divided into 3 segments
  - seg0 (bit 63=0) user code/heap
  - seg1 (bit 63 = 1, 62 = 1) user stack
  - kseg (bit 63 = 1, 62 = 0) kernel segment for OS
- 3 level page table, each one page
  - Alpha only 43 unique bits of VA
  - (future min page size up to 64KB => 55 bits of VA)
- PTE bits; valid, kernel & user read & write enable
  (No reference, use, or dirty bit)
**Cache Optimization: Alpha 21064**

- Separate Instr & Data
- TLB & Caches
- TLBs fully associative
- TLB updates in SW ("Priv Arch Libr")
- Caches 8KB direct mapped, write thru
- Critical 8 bytes first
- Prefetch instr. stream buffer
- 2 MB L2 cache, direct mapped, WB (off-chip)
- 256 bit path to main memory, 4 x 64-bit modules
- Victim Buffer: to give read priority over write
- 4 entry write buffer between D$ & L2$

**Summary #1/2 : TLB, Virtual Memory**

- Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions:
  1) Where can block be placed?
  2) How is block found?
  3) What block is replaced on miss?
  4) How are writes handled?
- More cynical version of this:
  Everything in computer architecture is a cache!
- Page tables map virtual address to physical address
- TLBs are a cache on translation and are extremely important for good performance
- Special tricks necessary to keep TLB out of critical cache-access path
- TLB misses are significant in processor performance:
  - These are funny times: most systems can't access all of 2nd level cache without TLB misses!

**Summary #2 / 2: Memory Hierarchy**

- Virtual memory was controversial at the time: can SW automatically manage 64KB across many programs?
  - 1000X DRAM growth removed the controversy
- Today VM allows many processes to share single memory without having to swap all processes to disk; VM translation, protection, and sharing are more important than memory hierarchy