Recap: Set Associative Cache

- **N-way set associative**: N entries for each Cache Index
  - N direct mapped caches operate 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

![Diagram of a two-way set associative cache]

Recap: Cache Performance

\[
\text{Execution Time} = \text{Instruction Count} \times \text{Cycle Time} \times (\text{Ideal CPI} + \text{Memory Stalls/Inst} + \text{Other Stalls/Inst})
\]

\[
\text{Memory Stalls/Inst} = \text{Instruction Miss Rate} \times \text{Instruction Miss Penalty} + \text{Loads/Inst} \times \text{Load Miss Rate} \times \text{Load Miss Penalty} + \text{Stores/Inst} \times \text{Store Miss Rate} \times \text{Store Miss Penalty}
\]

\[
\text{Average Memory Access time (AMAT)} = \text{Hit Time}_L + (\text{Miss Rate}_L \times \text{Miss Penalty}_L) = (\text{Hit Rate}_L \times \text{Hit Time}_L) + (\text{Miss Rate}_L \times \text{Miss Time}_L)
\]

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
- **Coherence** (Invalidation): other process (e.g., I/O) updates memory
The Five Classic Components of a Computer

Today’s Topics:
- Recap last lecture
- Virtual Memory
- Protection
- TLB
- Buses

Recap: Four Questions for Caches and Memory Hierarchy

Q1: Where can a block be placed in the upper level? (Block placement)
Q2: How is a block found if it is in the upper level? (Block identification)
Q3: Which block should be replaced on a miss? (Block replacement)
Q4: What happens on a write? (Write strategy)

Recap: Q1: Where can a block be placed?
- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets

Recap: Q2: How is a block found?
- Direct indexing (using index and block offset), tag compares, or combination
- Increasing associativity shrinks index, expands tag
Recap: Q3: Which block should be replaced on a miss?

- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)
Associativity: 2-way 4-way 8-way
Size   LRU Random LRU Random LRU Random
16 KB  5.2% 5.7% 4.7% 5.3% 4.4% 5.0%
64 KB  1.9% 2.0% 1.5% 1.7% 1.4% 1.5%
256 KB 1.15% 1.17% 1.13% 1.13% 1.12% 1.12%

Recap: Q4: What happens on a write?

- **Write through**—The information is written to both the block in the cache and to the block in the lower-level memory.
- **Write back**—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
  - Pros and Cons of each?
    - WT: read misses cannot result in writes
    - WB: no writes of repeated writes
  - WT always combined with write buffers so that don’t wait for lower level memory

Write Buffer for Write Through

- A Write Buffer is needed between the Cache and Memory
  - 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
  - Must handle burst behavior as well!
- Memory system designer’s nightmare:
  - Store frequency (w.r.t. time) \(> 1 / DRAM write cycle
  - Write buffer saturation

Write-miss Policy: Write Allocate versus Not Allocate

- Assume: a 16-bit write to memory location 0x0 and causes a miss
  - Do we read in the block?
    - Yes: Write Allocate
    - No: Write Not Allocate
Impact of Memory Hierarchy on Algorithms

- CPU time is \( f(\text{ops}, \text{cache misses}) \) not just \( f(\text{ops}) \)!
  - What does this mean to Compilers, Data structures, Algorithms?

  - Quicksort: fastest comparison based sorting algorithm when all keys fit in memory
  - Radix sort: also called “linear time” sort because for keys of fixed length and fixed radix a constant number of passes over the data is sufficient independent of the number of keys

Quicksort vs. Radix as vary number keys:

- Instructions

- Time

- Cache misses

What is proper approach to fast algorithms?
**Administrivia**

- Second midterm coming up (Tuesday, March 1st)
  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

**How Do you Design a Memory System?**

- Set of Operations that must be supported
  - read: data <= Mem[Physical Address]
  - write: Mem[Physical Address] <= Data

- Determine the internal register transfers
- Design the Datapath
- Design the Cache Controller

**What happens on a Cache miss?**

- For in-order pipeline, 2 options:
  - Freeze pipeline in Mem stage (popular early on: Sparc, R4000)
    
    IF ID EX Mem stall stall stall stall ... stall Mem Wr
    IF ID EX stall stall stall ... stall stall Ex Wr
  
    - Use Full/Empty bits in registers + MSHR queue
    - MSHR = “Miss Status/Handler Registers” (Kroft)
    - Each entry in this queue keeps track of status of outstanding memory requests to one complete memory line.
      - Per cache-line: keep info about memory address.
      - For each word: register (if any) that is waiting for result.
      - Used to “merge” multiple requests to one memory line
    - New load creates MSHR entry and sets destination register to “Empty”. Load is “released” from pipeline.
    - Attempt to use register before result returns causes instruction to block in decode stage.
    - Limited “out-of-order” execution with respect to loads. Popular with in-order superscalar architectures.

- Out-of-order pipelines already have this functionality built in... (load queues, etc).
Improving Cache Performance: 3 general options

Time = IC x CT x (ideal CPI + memory stalls)

Average Memory Access time =
Hit Time + (Miss Rate x Miss Penalty) =

(Hit Rate x Hit Time) + (Miss Rate x Miss Time)

Options to reduce AMAT:
1. Reduce the miss rate,
2. Reduce the miss penalty, or
3. Reduce the time to hit in the cache.

Improving Cache Performance

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

3Cs Absolute Miss Rate (SPEC92)

2:1 Cache Rule

miss rate 1-way associative cache size X = miss rate 2-way associative cache size X/2
### 3Cs Relative Miss Rate

- **Miss Rate**
  - **Conflict**
  - **Compulsory**

**Cache Size (KB)**

- 0%
- 20%
- 40%
- 60%
- 80%
- 100%

- 1-way
- 2-way
- 4-way
- 8-way

- Capacity
- Compulsory

### 1. Reduce Misses via Larger Block Size

- **Block Size (bytes)**
- **Miss Rate**
  - 0%
  - 5%
  - 10%
  - 15%
  - 20%
  - 25%

- 16
- 32
- 64
- 128
- 256

- 1K
- 4K
- 16K
- 64K
- 256K

### 2. Reduce Misses via Higher Associativity

- **2:1 Cache Rule:**
  - Miss Rate DM cache size N - Miss Rate 2-way cache size N/2
  - Beware: Execution time is only final measure!
  - Will Clock Cycle time increase?
  - Hill [1988] suggested hit time for 2-way vs. 1-way external cache +10%, internal + 2%

### Example: Avg. Memory Access Time vs. Miss Rate

- Assume CCT = 1.10 for 2-way, 1.12 for 4-way, 1.14 for 8-way vs. CCT direct mapped

<table>
<thead>
<tr>
<th>Cache Size (KB)</th>
<th>1-way</th>
<th>2-way</th>
<th>4-way</th>
<th>8-way</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.33</td>
<td>2.15</td>
<td>2.07</td>
<td>2.01</td>
</tr>
<tr>
<td>2</td>
<td>1.98</td>
<td>1.86</td>
<td>1.76</td>
<td>1.68</td>
</tr>
<tr>
<td>4</td>
<td>1.72</td>
<td>1.67</td>
<td>1.61</td>
<td>1.53</td>
</tr>
<tr>
<td>8</td>
<td>1.46</td>
<td>1.48</td>
<td>1.47</td>
<td>1.43</td>
</tr>
<tr>
<td>16</td>
<td>1.29</td>
<td>1.32</td>
<td>1.32</td>
<td>1.32</td>
</tr>
<tr>
<td>32</td>
<td>1.20</td>
<td>1.24</td>
<td>1.25</td>
<td>1.27</td>
</tr>
<tr>
<td>64</td>
<td>1.14</td>
<td>1.20</td>
<td>1.21</td>
<td>1.23</td>
</tr>
<tr>
<td>128</td>
<td>1.10</td>
<td>1.17</td>
<td>1.18</td>
<td>1.20</td>
</tr>
</tbody>
</table>

(Red means A.M.A.T. not improved by more associativity)
3. Reducing Misses via a “Victim Cache”

- How to combine fast hit time of direct mapped yet still avoid conflict misses?
- Add buffer to place data discarded from cache
- Jouppi [1990]: 4-entry victim cache removed 20% to 95% of conflicts for a 4 KB direct mapped data cache
- Used in Alpha, HP machines

4. Reducing Misses by Hardware Prefetching

- E.g., Instruction Prefetching
  - Alpha 21064 fetches 2 blocks on a miss
  - Extra block placed in “stream buffer”
  - On miss check stream buffer
- 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

5. 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
- Issuing Prefetch Instructions takes time
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth

6. Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- Instructions
  - Reorder procedures in memory so as to reduce conflict misses
  - Profiling to look at conflicts(using tools they developed)
- Data
  - Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays
  - Loop Interchange: change nesting of loops to access data in order stored in memory
  - Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap
  - Blocking: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows
Improving Cache Performance (Continued)

1. Reduce the miss rate,
2. 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) \( \ll \) 1 / DRAM write cycle
° Memory system designer’s nightmare:
  • Store frequency (w.r.t. time) \( \gg \) 1 / DRAM write cycle
  • Write buffer saturation

1. Reducing Penalty: Read Priority over Write on Miss

° Write-Buffer Issues:
  • 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

4. Reduce Penalty: Second-Level Cache

- L2 Equations
  - AMAT = Hit Time$_L1$ + Miss Rate$_L1$ x Miss Penalty$_L1$
  - Miss Penalty$_L1$ = Hit Time$_L2$ + Miss Rate$_L2$ x Miss Penalty$_L2$
  - AMAT = Hit Time$_L1$ + Miss Rate$_L1$ x (Hit Time$_L2$ + Miss Rate$_L2$ x Miss Penalty$_L2$)

- Definitions:
  - Local miss rate—misses in this cache divided by the total number of memory accesses to this cache (Miss rate$_L2$)
  - Global miss rate—misses in this cache divided by the total number of memory accesses generated by the CPU (Miss Rate$_L1$ x Miss Rate$_L2$)
  - Global Miss Rate is what matters

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: 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.

Relative CPU Time

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

Improving Cache Performance (Continued)

1. Reduce the miss rate,
2. **Reduce the miss penalty**, or
3. Reduce the time to hit in the cache:

- Lower Associativity (+victim caching or 2nd-level cache)?
- Multiple cycle Cache access (e.g. R4000)
- Harvard Architecture
- Careful Virtual Memory Design (rest of lecture!)

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 33% loads/store, hit time=1, miss time=50
  - Note: data hit has 1 stall for unified cache (only one port)

\[
\text{AMAT}_{\text{Harvard}} = (1/1.33)x(1+0.64\%x50)+(0.33/1.33)x(1+6.47\%x50) = 2.05 \\
\text{AMAT}_{\text{Unified}} = (1/1.33)x(1+1.99\%x50)+(0.33/1.33)x(1+1.99\%x50) = 2.24
\]
Recall: Levels of the Memory Hierarchy

<table>
<thead>
<tr>
<th>Component</th>
<th>Capacity</th>
<th>Access Time</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Registers</td>
<td>100s Bytes</td>
<td>&lt;10ns</td>
<td>$.01-.001/bit</td>
</tr>
<tr>
<td>Cache</td>
<td>K Bytes</td>
<td>10-100ns</td>
<td>$.01-.001/bit</td>
</tr>
<tr>
<td>Main Memory</td>
<td>M Bytes</td>
<td>100ns-1us</td>
<td>$.01-.001</td>
</tr>
<tr>
<td>Disk</td>
<td>G Bytes</td>
<td>ms</td>
<td>10-4 cents</td>
</tr>
<tr>
<td>Tape</td>
<td>infinite seconds</td>
<td>10^-6 cents</td>
<td></td>
</tr>
</tbody>
</table>

Capacity: Access Time: Cost:

Upper Level

- Registers
- Cache
- Memory
- Disk
- Files

Lower Level

- Staging Xfer Unit
- Virtual Address Space
- Physical Address Space

Virtual Address

- V page no.
- offset

Address Map

V = {0, 1, ..., n - 1} virtual address space
M = {0, 1, ..., m - 1} physical address space
n > m

MAP: V -> M U {∅} address mapping function

MAP(a) = a' if data at virtual address a is present in physical address a' and a in M

= ∅ if data at virtual address a is not present in M or some sort of protection violation

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”)
Issues in Virtual Memory System Design

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

Which region of M is to hold the new block ⇒ placement policy

How do we find a page when we look for it? ⇒ 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 ⇒ replacement policy

What do we do on a write? ⇒ write policy

Missing item fetched from secondary memory only on the occurrence of a fault ⇒ demand load policy

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 Meg entries x 4 bytes = 4MB

- With 4K pages, 64-bit address space, need:
  \[ 2^{64}/4K = 2^{52} \] entries = BIG!

- Can’t keep whole page table in memory!

Large Address Spaces

Two-level Page Tables

32-bit address:

- 2 GB virtual address space
- 4 MB of PTE2
  - paged, holes
- 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

⇒ TLBs or virtually addressed caches are critical
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

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?
Constraints on TLB organization

MIPS 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>
</tbody>
</table>

TLB
- 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 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)

Tail pointer:
Mark pages as “not used recently”

Set of all pages in Memory

Freelist

Free Pages

Head pointer:
Place pages on free list if they are still marked as “not used”. Schedule dirty pages for writing to disk
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 used</th>
<th>page table entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>page entry</td>
</tr>
<tr>
<td>0</td>
<td>page entry</td>
</tr>
<tr>
<td>1</td>
<td>page entry</td>
</tr>
<tr>
<td>0</td>
<td>page entry</td>
</tr>
<tr>
<td>0</td>
<td>page 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 a 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:

Virtual Address

V page no.  offset

TLB Lookup

V Access Rights

PA

Physical Address

P page no.  offset

Overlapped TLB & Cache Access

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

What if cache size is increased to 8KB?

Machines with TLBs go one step further: they overlap
TLB lookup with cache access.

° Works because lower bits of result (offset) available early
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:

11 2
00
20 12
This bit is changed by VA translation, but is needed for cache lookup.

Solutions: go to 8K byte page sizes; go to 2 way set associative cache; or SW guarantee VA[13]=PA[13]

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)

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)
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!
° Lots of techniques people use to improve the miss rate of caches:

<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>Pseudo-Associative 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>3</td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Summary #2 / 2: Memory Hierarchy
° 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!
° 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