Last Time

- Discussed concepts of locality and stride
  - Spatial locality: programs tend to access values near values they have already accessed
  - Temporal locality: programs tend to access a particular value multiple times
  - A stride-\(k\) reference pattern means the program accesses values every \(k\) steps

- Covered basic cache designs and trade-offs
  - The cache holds blocks of memory of a specific size
  - Cache lines contain blocks, plus bookkeeping info
    - Unique tag associated with each block, and a valid flag
  - Cache sets allow us to simplify our cache designs
    - Each address maps to exactly one cache-set
**Direct-Mapped Caches**

- Direct-mapped caches have $S$ sets, but each set contains only one cache line ($E = 1$)

  - $m$-bit address: $t = m - (s + b)$ bits
  - $s$ bits
  - $b$ bits

- Example: direct-mapped cache with 4 sets
  - 2 bits in set index

![Direct-Mapped Cache with 4 Sets](image)
**Fully Associative Caches**

- Fully associative caches have only one set, which contains all cache lines
  - $S = 2^s = 1 \Rightarrow s = 0$. No bits used for set index!

$m$-bit address: $t = m - b$ bits $\quad$ $b$ bits

- Example: fully-associative cache

![Fully Associative Cache Diagram]
Set-Associative Caches (2)

- Set-associative caches blend strengths of direct-mapped caches and fully associative caches
  - Multiple cache sets
  - Multiple lines per set
- For an \( m \)-bit address:
  - \( t \) bits
  - \( s = 2 \) bits
  - \( b \) bits
  - \((4 \text{ cache-sets} \Rightarrow 2 \text{ bits for cache-set index})\)

Two-Way Set-Associative Cache

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
<td></td>
</tr>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
<td></td>
</tr>
</tbody>
</table>
WRITING TO CACHES

- So far, only discussed reading from the cache...
- Most programs also write to memory...
  - Need to handle writes to the cache as well!
  - Ideally, want to minimize performance penalties on writes, just like on reads
- If CPU writes a word that is already in the cache:
  - Called a write hit
  - Several choices of how to handle this situation
- Option 1: employ a write-through strategy
  - Every write to cache causes cache to write the entire block back to memory
  - Problem: every single write to the cache causes a write to main memory! Can’t exploit data locality!
Writing to Caches (2)

- Option 2: use a write-back strategy
  - Add a “dirty flag” to each cache line
  - When a cached block is written to, set the dirty flag
  - When a cache line is evicted, write the block back to main memory

- Benefit:
  - Can now exploit locality of writes as well as reads
  - Writes to adjacent values likely to hit cached blocks
  - When dirty blocks are finally evicted, have a much smaller number of writes to main memory

- Drawback: more complex than write-through
  - These days, most caches use a write-back strategy
WRITING TO CACHES (3)

If address being written to isn’t already in the cache:
  • Called a write miss
  • Again we have several options

Option 1: use a write-allocate strategy
  • When a write miss occurs, load the block into cache and then perform the write against the cache
  • Assumes the program will perform subsequent writes

Option 2: use a no-write-allocate strategy
  • When a write miss occurs, just perform the write against main memory

Write-back caches usually use write-allocate strategy
Write-through caches typically use no-write-allocate strategy
Cache performance modeling can be extremely complicated...
- Usually easiest to measure actual system behaviors

Basic ideas can be captured with only a few simple parameters

- Miss rate: the fraction of memory references that result in cache misses
  - Miss rate = # misses / # references (0 ≤ miss rate ≤ 1)
- Hit rate: the fraction of memory references that hit the cache
  - Hit rate = 1 – miss rate
CACHE PERFORMANCE ANALYSIS (2)

- **Hit time:** time to deliver a value from the cache to the CPU
  - Includes all necessary steps for delivering the value!
  - Time to select the appropriate cache set
  - Time to find the cache line using the block’s tag
  - Time to retrieve the specified word from block data

- **Miss penalty:** time required to handle cache miss
  - *(includes cache-set identification, checking tags, etc.)*
  - Need to fetch a block from main memory
  - May need to evict another cache line from the cache
  - *(Evicted line may be dirty and need written back…)*
  - Other associated bookkeeping for storing cache line
Simple example to see benefit of caching:
- Hit time = 1 clock (typical goal for L1 hits)
- Miss penalty = 100 clocks (main memory usu. 25-100)
If all reads were from cache, each read is 1 clock
Hit rate of 80%
  - 0.8 \times 1 \text{ clock} + 0.2 \times 100 \text{ clocks} = 20.8 \text{ clocks/access}
Hit rate of 90%
  - 0.9 \times 1 \text{ clock} + 0.1 \times 100 \text{ clocks} = 10.9 \text{ clocks/access}
Hit rate of 95%
  - 0.95 \times 1 \text{ clock} + 0.05 \times 100 \text{ clocks} = 5.95 \text{ clocks/access}
Hit rate is very important!
  - For programs with low miss-rate (good data locality), get a large memory at (nearly) the cost of a small one!
Cache Design Trade-Offs

- Previous example shows importance of designing programs to maximize cache usage
  - ...and compilers that know how to optimize well...
- Cache designers also have important trade-offs to consider
- Try to maximize hit rate, without increasing hit time
- Can increase overall size of cache, but this will probably increase the hit time
  - Idea is to increase number of lines, or size of each line
  - Physics: larger memories are slower than smaller ones
- Can increase block size, keeping cache size constant
  - (Don’t want to incur penalties due to physics)
  - Miss penalty will increase due to larger transfer times
For associative caches, what about increasing number of cache lines per set \((E)\) ?

As number of cache lines per set increases:
- Complexity of line-matching goes up, since more lines to compare. Will probably increase hit times.
- Must choose a line to evict when a cache miss occurs; now there are more choices.
  - Will probably increase miss penalty
  - Complexity of replacement-policy logic also increases!

Direct mapped caches avoid above penalties with *ultra-simple* mapping of blocks to cache lines...
- ...but *dramatically* increase likelihood of thrashing due to conflict misses!
INTEL PROCESSOR CACHES

- So far have only discussed caching data...
  - Reading instructions also frequently has good locality
- Processor can manage separate instruction caches (i-caches) and data caches (d-caches)
  - Allows parallel data-access paths within the CPU
  - Also, instruction caches usually only support reads 😊
**Intel Core 2 Caches**

- Intel Core 2 Duo/Quad cache information (typical):

<table>
<thead>
<tr>
<th>Cache</th>
<th>Associativity (E)</th>
<th>Block Size (B)</th>
<th>Sets (S)</th>
<th>Cache Size (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 i-cache</td>
<td>8</td>
<td>64 bytes (16 dwords)</td>
<td>64</td>
<td>32KB</td>
</tr>
<tr>
<td>L1 d-cache</td>
<td>8</td>
<td>64 bytes (16 dwords)</td>
<td>64</td>
<td>32KB</td>
</tr>
<tr>
<td>Unified L2 cache</td>
<td>8</td>
<td>64 bytes (16 dwords)</td>
<td>2048 - 16384</td>
<td>2 MB – 6MB+</td>
</tr>
</tbody>
</table>

- For Core-2 Quad, L2 is divided into two segments, each of which is shared by two cores

- Small number of cache lines per set, but large number of cache sets
  - Fast to determine if a block is in a cache set, and many cache sets to minimize thrashing issues
**Intel Core i7 Caches**

- **Intel Core i7 cache information:**

<table>
<thead>
<tr>
<th>Cache</th>
<th>Associativity (E)</th>
<th>Block Size (B)</th>
<th>Sets (S)</th>
<th>Cache Size (C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 i-cache</td>
<td>8</td>
<td>64 bytes (16 dwords)</td>
<td>64</td>
<td>32KB</td>
</tr>
<tr>
<td>L1 d-cache</td>
<td>8</td>
<td>64 bytes (16 dwords)</td>
<td>64</td>
<td>32KB</td>
</tr>
<tr>
<td>Unified L2 cache</td>
<td>8</td>
<td>64 bytes (16 dwords)</td>
<td>512</td>
<td>256KB</td>
</tr>
<tr>
<td>Unified L3 cache</td>
<td>16</td>
<td>64 bytes (16 dwords)</td>
<td>8192</td>
<td>8MB</td>
</tr>
</tbody>
</table>

- Each core has its own L2 cache
- All cores share the L3 cache

- With another level of caching, expect to see a smoother degradation in memory performance
Measuring Cache Behavior

- Can measure the performance of various caches in our computer by constructing a special program:
  - Allocate an array of size elements to scan through
  - Access the array with $k$-stride reference patterns
    - Vary $k$ from 1 to some large value
  - Scan through the memory for each stride value
- Measure memory access throughput as we vary both of these parameters
- Remember, two different kinds of locality!
  - Spatial locality:
    - Program accesses data items that are close to other data items that have been recently accessed
  - Temporal locality:
    - Program accesses the same data item multiple times
Can measure the performance of various caches in our computer by constructing a special program:

- Allocate an array of \textit{size} elements to scan through
- Access the array with \textit{k}-stride reference patterns
  - Vary \textit{k} from 1 to some large value
- Scan through the memory for each stride value

When our working set and stride are small:

- Should fit entirely into L1 cache, giving \textit{fast} access!

When working set doesn’t fit completely within L1 cache, and stride is increased:

- Should see performance taper off, as ratio of L1 misses to L1 hits increases
- When \textit{all} accesses miss L1, will see L2 cache performance

Should see similar behavior from L2-L3, L3-DRAM
**Intel Core i7 Memory Mountain**

- Produces a 3D surface that shows our caches!

Intel Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache

Ridges of temporal locality

Slopes of spatial locality

---

Stride (x8 bytes)

Size (bytes)
**Intel Core i7 Memory Mountain (2)**

- As total size increases, temporal locality decreases
  - Working set can no longer fit in a given cache...
  - See a significant dropoff as each cache level becomes ineffective

![Diagram of Intel Core i7 Memory Mountain with labels for L1, L2, L3, Mem, and annotations for Cache specifications and locality.]
Intel Core i7 Memory Mountain (3)

- As stride increases, spatial locality decreases
  - Cache miss rate increases as stride increases
  - Throughput tapers off to a minimum for each level
**Intel Core 2 Duo – Macbook**

- Different systems have very different statistics.
Intel Core 2 Duo – Macbook (2)

- Core 2 only has two caches before main memory
  - Sharper degradation on Core 2 than on Core i7
Intel Core 2 Duo – Macbook (3)

- Dip for very small working sets – at these sizes, cost of function invocation is overwhelming the measurements!
Small bump where stride allows previously loaded cache lines to be used again…
**Intel Core 2 Duo – Macbook (5)**

- Peak throughput is *much* lower than before...
  - Affected as much by motherboard chipset as by the processor itself
Intel Core 2 Quad – Desktop

- Peak throughput is much more similar to Core i7 shown before

Intel Core 2 Quad
2.66 GHz
32 KB L1 d-cache
4 MB Unified L2 cache
SUMMARY: “CACHE IS KING”

- Caches are **essential** components in computers
  - Performance gap between CPU and memory is large and increasing...
- Caches give us an opportunity to have large memories that are nearly as fast as small ones
  - As long as we keep our cache hit rates up, program performance will improve dramatically
- As programmers, it is extremely important to be aware of data locality and caching considerations
  - Sometimes, simple changes in a program can produce dramatic changes in memory throughput
  - Other times, must carefully design program to take advantage of processor caches
PROCESSOR PERFORMANCE

- Increases in CPU clock speeds have tapered off...
  - Issues: clock signal propagation, heat dissipation
- New focus: design CPUs that do more with less
  - Parallelize instruction execution
    - Pipelined instruction execution
    - Superscalar architectures – multiple ALUs, etc.
  - Out-of-order instruction execution
  - Disabling unused CPU components to reduce power
  - (See CS:APP chapters 4, 5 – very good discussion!)

- Although CPU speeds have remained in 1-3GHz ballpark, CPUs are still becoming more powerful
PROCESSOR PERFORMANCE (2)

- Another approach: multi-core processors
  - Put multiple independent CPUs onto a single chip
  - Intel Core 2/Core i7, IBM/Sony/Toshiba Cell, NVIDIA GeForce 9/GeForce 200 series GPUs, etc.

- Still suffers from the same processor-memory performance gap as before...
  - (And, the solution is still caching...)

- Explore the challenges and opportunities that multi-core introduces into cached memory access
  - Very rich topic, so our discussion will be a pretty high-level overview
**MULTICORE AND CACHING**

- Go ahead and stick multiple processors on a chip
- What happens to caching in these systems?
- For example:

Any problems with this approach?
- More accurately, “What new problems do we have?”
INDEPENDENT CACHES

Two independent caches of a common resource:

- CPU 1 reads data in block 23, starts using it...
  - Block 23 is loaded into CPU1’s cache
- CPU 2 issues a write on block 23...
  - Block 23 also loaded into CPU2’s cache
  - CPU 1 cache no longer consistent with memory state
COORDINATED CACHES

- Must coordinate state between separate caches

- CPU 1 reads data in block 23, starts using it...
  - Block 23 is loaded into CPU1’s cache

- CPU 2 issues a write on block 23...
  - Block 23 also loaded into CPU2’s cache
  - Somehow, tell CPU 1’s cache that block 23 has changed...
**Cache Coherence**

- **Cache coherence** constrains the behavior of reads and writes to caches of a shared resource.
- Need to define how coherent memory should behave:
  - Processor P reads a location X, then writes to X
    - No other processors modify X between read and write
    - Subsequent reads of X **must** return the value that P wrote
  - Processor P1 writes to a location X
    - Subsequently, another processor P2 reads X
    - P2 must read the value that P1 wrote
      - Specifically, P2 **must not** read the old value of X
  - Processor P1 writes value A to a location X
    - Then, processor P2 writes value B to the same location X
    - Reads of X must see A and then B, **but never** B and then A
Clearly need a mechanism to coordinate cache interactions

Approach:
- Previously had simple state info in each cache line...
- Introduce additional state and operations to coordinate cache lines between processors
Cache Coherence Protocol

- Mechanism is called a *cache coherence protocol*
- Many different protocols to choose from!
- Intel and AMD multi-core processors use variants of the **MSI protocol**
  - Letters in protocol name specify the states of cache lines
- Each CPU has its own cache, but caches coordinate read and write operations to maintain coherence
- In each cache, a cache line is in one of these states:
  - **Modified** – the line contains data modified by the CPU, and is thus inconsistent with main memory
  - **Shared** – the line contains unmodified data, and appears in at least one CPU’s cache
  - **Invalid** – the line’s data has been invalidated (e.g. because another CPU wrote to their cache), and must be reread
    - ...either from main memory, or from another cache
MSI Cache Coherence Protocol

- Basic principles:
  - A memory block can appear in multiple caches only if it is shared (S)
  - If modified (M), may only appear in one CPU cache!

- When a processor reads a value from a block:
  - If the block is in my cache, and marked either S or M, just return the value

- If the block is not in my cache (or marked I):
  - If no other cache has the block marked M, just read it from main memory
  - If another cache has the block marked M, that cache must write data back to main memory, then switch to S or I state
    - Then processor can read block from memory, and mark it S
MSI Cache Coherence Protocol (2)

- When a processor writes a value to a block:
  - If the block is in my cache and already marked $M$, perform the write
- If the block is marked $S$ (shared) in my cache:
  - Invalidate all other caches’ copies of the block!
  - Change my block’s status to $M$, then perform write
- If block isn’t in my cache, or is marked $I$ (invalid):
  - If block is marked $S$ in other caches, invalidate them
  - If block marked $M$ in another cache, that cache must write data back to main memory, then switch it to $I$
    - i.e. block is completely evicted from the other CPU’s cache
  - Then processor can read block from memory, mark it $M$, then perform the write
Next Time

- Finish discussion of cache coherence protocols
  - Variants of MSI used by Intel/AMD processors
  - New cache performance issues to consider...

- Begin discussing a new topic: virtualization