LAST TIME

- Examined several memory technologies:
  - SRAM – volatile memory cells built from transistors
    - Fast to use, larger memory cells (6+ transistors per cell)
  - DRAM – volatile memory cells built from capacitors
    - Slower to use, smaller memory cells, can make very large
  - Magnetic disk storage – nonvolatile memory
    - Very slow to use, compared to SRAM/DRAM/CPU!
    - Can make extremely large disks

- Also discovered two important principles:
  - Small memories tend to be faster than large ones, due to physical limitations
  - Can make larger memories denser (although addressing and accessing become more expensive)
We want the performance of small memories, but we also want the convenience of large memories.

We will never get it with this design:

The processor-memory gap:
- Not only are processors much faster than large memories...
- The gap itself has been growing by > 50% per year!

Solution: use caching between CPU and memory
CACHING

- When CPU reads a value from main memory:
  - Read an entire block of data from main memory
  - As long as subsequent accesses are within this block, just use the cache!
  - If an access is outside the cached data, need to retrieve and cache a new block of data from memory

Today:
- What program behaviors maximize benefit of caching?
- How are these caches designed? What are the trade-offs?
Caching and Data Access Patterns

- When CPU reads a value from main memory:
  - Read neighboring values from main memory as well

- Not all programs will benefit from this approach!
- Only programs that access data in specific patterns will benefit from the cache!
  - Most accesses end up being served from the cache
- Programs that usually access data in a widely varied manner will not benefit from cache at all.
CACHING AND LOCALITY

- Programs must exhibit good **locality** if they are going to utilize the cache effectively
- 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
- Well-written programs will exhibit good locality
  - (“well-written” in terms of cache-friendliness)
  - ...and the computer hardware can run them faster!
- Poorly-written programs have poor locality
  - Program can’t take advantage of system caches
**Locality: Examples**

- Frequently can achieve good locality very easily: programs tend to access data in regular patterns
- Vector-add code from before:
  ```c
  int * vector_add(int *a, int *b, int length) {
    int i;
    int *result =
      (int *) malloc(length * sizeof(int));
    for (i = 0; i < length; i++)
      result[i] = a[i] + b[i];
    return result;
  }
  ```
- Elements of input and output arrays are accessed in sequential order – works well with caching
Still *extremely* valuable to understand locality as a programmer!
- Simple choices can have a profound impact on program performance

Molecular dynamics example from lecture 1
```c
#define N_ATOMS 10000
#define DIM 2
/* Array of data for each atom being simulated. */
double atoms[N_ATOMS][DIM][DIM];
```

Version 1:
```c
for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
        for (n = 0; n < N_ATOMS; n++)
            atoms[n][i][j] = ... ;
```

This code has poor data locality, so it runs slower
**Locality: Examples (3)**

- Memory layout of our atoms array:
  ```c
  double atoms[N_ATOMS][DIM][DIM];
  ```

<table>
<thead>
<tr>
<th>[0][0][0]</th>
<th>[0][0][1]</th>
<th>[0][1][0]</th>
<th>[0][1][1]</th>
<th>[1][0][0]</th>
<th>[1][0][1]</th>
<th>[1][1][0]</th>
<th>[1][1][1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2][0][0]</td>
<td>[2][0][1]</td>
<td>[2][1][0]</td>
<td>[2][1][1]</td>
<td>[3][0][0]</td>
<td>[3][0][1]</td>
<td>[3][1][0]</td>
<td>[3][1][1]</td>
</tr>
</tbody>
</table>

... … … … … … … … … … ...

- **Version 1:**
  ```c
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      for (n = 0; n < N_ATOMS; n++)
        atoms[n][i][j] = ... ;
  ```

  - Code accesses every 4th array element, because of the way the loops are arranged
  - Rearrange loops to access each element in sequence
**Locality: Examples (4)**

- Memory layout of our atoms array:

  ```
  double atoms[N_ATOMS][DIM][DIM];
  ```

<table>
<thead>
<tr>
<th>[0][0][0]</th>
<th>[0][0][1]</th>
<th>[0][1][0]</th>
<th>[0][1][1]</th>
<th>[1][0][0]</th>
<th>[1][0][1]</th>
<th>[1][1][0]</th>
<th>[1][1][1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2][0][0]</td>
<td>[2][0][1]</td>
<td>[2][1][0]</td>
<td>[2][1][1]</td>
<td>[3][0][0]</td>
<td>[3][0][1]</td>
<td>[3][1][0]</td>
<td>[3][1][1]</td>
</tr>
</tbody>
</table>

- Version 2:

  ```
  for (n = 0; n < N_ATOMS; n++)
      for (i = 0; i < DIM; i++)
          for (j = 0; j < DIM; j++)
              atoms[n][i][j] = ... ;
  ```

  - This code accesses each array element in sequence
  - A simple change, and results in a significant performance improvement!
LOCALITY AND STRIDE

- When a program accesses array elements sequentially, called a stride-1 reference pattern
  - (Consider stride in terms of processor word-size)
  - Vector-sum program has a stride-1 reference pattern
  - Updated version of molecular dynamics program also has a stride-1 reference pattern

- When a program accesses every $k^{th}$ element in sequence, called a stride-$k$ reference pattern
  - Original version of molecular dynamics program has a stride-4 reference pattern

- Generally, as stride increases, program locality decreases

- With 1D arrays, pretty easy to achieve stride-1
- With multidimensional arrays, it can become much trickier to achieve stride-1
EXTENDING OUR CACHING IDEA

- Added an external SRAM cache between the CPU and main memory (DRAM)
- Accessing the SRAM cache is still slower than using registers...
  - Can access a register in 0 clocks (i.e. same clock that instruction is executed in)
  - SRAM cache can take e.g. 1-30 clocks, depending on size
- Also, disk access is horribly slow!
  - 20 million clocks or more!
  - Accessing DRAM is much faster than accessing the disk...
  - Could exploit data locality with disk accesses as well, by using DRAM as a cache for the disk

Why not apply our caching technique in other places?
The Memory Hierarchy

L0: Registers

L1: On-Chip SRAM Cache

L2/L3: Off-Chip SRAM Cache

L4: Main Memory (DRAM)

L5: Local Secondary Storage (Local Disks)


Each level provides caching for the levels below it.

CPU registers hold and manipulate data values retrieved from L1 cache.

L1 cache holds blocks of data retrieved from L2/L3 cache.

L2/L3 caches hold blocks of data retrieved from main memory.

Main memory holds disk blocks read from local disks.

Local disks hold files retrieved from remote servers.

Smaller, faster, and more expensive per byte of storage

Larger, slower, and cheaper per byte of storage
THE MEMORY HIERARCHY (2)

- This is a very typical memory hierarchy used in modern computers, but by no means the only one!
- Many places where caching is employed within levels of the hierarchy
- Example: hard disks also employ caches
  - Disk buffer, often 2MB-64MB
    - Caches data prefetched from disk, pending writes to disk
  - Non-volatile RAM buffer (less common)
    - Disk writes are stored to this buffer until it fills up, then written to the disk itself
    - If power fails, this non-volatile memory retains its state
  - Finally, the magnetic disk storage itself
- Some systems use solid-state drives (SSDs) to cache data from traditional spinning hard-disks
CACHE MANAGEMENT

- Each level $k$ provides caching for level $k + 1$
- Some caches are managed entirely by hardware
  - L1 (on-chip SRAM), L2/L3 (off-chip SRAM) caches
  - Performance is absolutely critical, so hardware is designed to manage these caches
- Other caches are managed entirely by software
  - L4 (DRAM main memory) is managed extensively by the operating system
  - e.g. the operating system caches disk blocks from L5 into L4 to improve disk IO performance
  - CPU registers (L0 cache) are manually assigned by compiler to minimize need to access memory (L1+)
CACHES AND MEMORY BLOCKS

- Level $k$ caches data from level $k + 1$:
  - Memory at level $k + 1$ is partitioned into fixed-size blocks
  - Level $k$ stores these blocks in its cache
  - Anytime data needs to be transferred between levels $k$ and $k + 1$, this block size is used

- Actual block size depends on characteristics of levels $k$ and $k + 1$
  - Between L0 and L1, block size is 1 word
  - Between L1 – L4, block is 8 to 16 words (64B)
  - Between L4 and L5, block is up to several KB

- Lower levels have longer access times...
  - Also usually designed to read/write larger blocks of data from storage in one shot...

- Amortize read/write cost over a larger amount of data
Cache Lines

- Hardware caches manage blocks of data from the next level...
  - Clearly need more details than just the data itself!
- Cache lines hold:
  - A flag indicating whether the line currently holds valid data
  - A tag that uniquely identifies the block
    - Taken from the address where the block is actually stored
  - The block of cached data itself
- Pictorially:
**Cache Operation**

- CPU makes requests to main memory

- Main memory divided into blocks of $B$ bytes ($B = 2^b$)
- Cache lines are slightly larger than $B$ bytes
  - Line also includes the tag and a “valid” flag, as well as block

**Example:** CPU requests a word in block 10
- Cache returns value directly, since block 10 is cached
- This is called a **cache hit** – the requested item was contained within the cache
**Cache Misses**

- Next, CPU requests a word in block 6
  - Cache doesn’t contain block 6
    - This is called a cache miss
    - Cache must load block 6 from main memory before providing requested value to the CPU
    - CPU incurs performance cost of accessing main memory

- What else must the cache do?
  - Figure out where to store block 6...
  - Need to *replace* (or *evict*) an existing block in the cache
CPU requested a value in block 6

Caches have a replacement policy:
- When loading a new block into a full cache, which existing block should be replaced?

Example replacement policies:
- Least Recently Used (LRU) policy: evict the block that was accessed furthest in the past
- Random replacement policy: randomly pick a block to evict
COLD CACHES

- Will there be a cache miss in this situation?

- All cache lines have valid flag = 0
- Duh, of course!

- Caches have different kinds of cache misses
- This is called a *cold cache*
  - No accesses yet, so cache isn’t populated with data!
  - Cache is “warmed up” by accessing memory and populating the cache lines
  - Misses during this phase are called *cold misses*
  - Also called “compulsory misses” since they are unavoidable
**Cache Placement Policy**

- CPU requests a word in block 5

- The biggest part of a cache’s work?
  - Determining whether the requested block actually appears in the cache, and if so, where is it?!

- Caches implement a placement policy, specifying where new blocks are placed in the cache

- Most flexible placement policy is random
  - Blocks from level $k + 1$ can be stored anywhere in level $k$

- Also the most expensive placement policy!
  - Very costly to track down blocks in the cache
  - Hardware caches usually cannot implement this policy
Cache Placement Policy (2)

- Hardware caches use a much stricter placement policy
  - Blocks from level $k + 1$ can only be stored into a subset of locations in level $k$
  - Makes it much faster and easier to determine if a block is already in the cache
- Cache lines are grouped into cache sets
  - Every block from level $k + 1$ maps to exactly one set in the cache at level $k$
  - If cache set contains multiple cache lines, a block may be stored into any cache line in the set
- Two extremes for our cache organization:
  - $S$ cache sets, each of which contains exactly one line
  - One cache set which contains all $E$ cache lines
Cache Placement Policy (3)

- Cache lines are grouped into cache sets
  - Every block from level $k + 1$ maps to exactly one set in the cache at level $k$

- Direct-mapped caches:
  - $S$ cache sets, each of which contains exactly one line
  - Fast and easy to determine if a block is in the cache
  - Find the cache set associated with the block, and look at the one cache line in the set

- Fully associative caches:
  - One cache set which contains all $E$ lines
  - Much more expensive to find if a block is in the cache
  - Need to look at the tags from all cache lines, to see if the block is in the cache
**Direct-Mapped Caches and Misses**

- Direct-mapped caches map each memory block to a single cache line
  - Each block only appears in one cache set, and each cache set only contains one cache line
- Can lead to new kinds of cache misses
- Example: our cache from before
  - Four cache sets, each of which contains one line
  - A block \( i \) is mapped to cache-set \((i \mod 4)\)

What if a program alternately accesses data only in blocks 9 and 13?
DIRECT-MAPPED CACHES AND MISSES (2)

Example: direct-mapped cache
- A block $i$ is mapped to cache-set $(i \mod 4)$
- Program alternately accesses data only in blocks 9 and 13

Each time CPU makes a request, the cache doesn’t contain the associated block!

These are called **conflict misses**
- Cache is large enough to hold the requested blocks...
- ...but due to placement policy constraints, cache has to keep reloading blocks from main memory
**Direct-Mapped Caches and Misses (3)**

- Example: direct-mapped cache
  - A block $i$ is mapped to cache-set $(i \mod 4)$
  - Program alternately accesses data only in blocks 9 and 13

- If a program generates a large number of conflict misses like this, it is called **thrashing**
  - For our example, program thrashes between blocks 9 and 13

- **A major issue!**
  - Program can have great locality, but still runs horribly slow!
  - If a program thrashes, the correct adjustments to make are often very subtle
CACHE ORGANIZATION

- Cache is designed to work against a main memory with $M$ bytes
  - $m = \log_2(M)$ bits in addresses to main memory
- Caches have several important parameters
  - $B = 2^b$ bytes to store the block in each cache line
  - $S = 2^s$ cache sets
  - $E$ cache lines per set
  - Both $S$ and $B$ are powers of 2
- The cache stores $B \times E \times S$ bytes of data from main memory
  - (Don’t forget: cache lines also include a tag and a valid flag, which require additional space)
Mapping Cache Blocks

- Main memory with $M$ bytes:
  - $m = \log_2(M)$ bits in addresses to main memory

- Cache parameters:
  - $B = 2^b$ bytes to store the block in each cache line
  - $S = 2^s$ cache sets
  - $E$ cache lines per set

- Given a specific memory address, the cache must:
  - Map the address to a memory block
    - *(the cache works with blocks, not individual values)*
  - Figure out which cache set the block would live in
  - Figure out the tag that uniquely identifies the block
  - Figure out the offset of the address within the block

- Take $m$ address bits, map them to these things
Mapping Cache Blocks (2)

- Relevant parameters for main memory and cache:
  - $m = \log_2(M)$ bits in addresses to main memory
  - $B = 2^b$ bytes to store the block in each cache line
  - $S = 2^s$ cache sets in the cache

- When CPU accesses a data value:
  - Bottom-most $b$ bits of the address specify offset of data value within the block
  - Middle $s$ bits specify the cache set where the block resides
  - Remaining topmost bits constitute the block’s tag
    - (Must be able to uniquely identify all blocks that can be cached)
Why use middle bits for set index?

- Why not topmost bits?

If topmost bits identify the cache set, will cause long runs of addresses to map to same cache set.

Example: $M = 16$ bytes, $S = 4$ cache sets, $s = 2$

If topmost bits select cache set, programs with good locality won’t map accesses to different sets.

If middle bits select cache set, programs with good locality use all cache sets much more evenly.

<table>
<thead>
<tr>
<th>tag</th>
<th>set index</th>
<th>block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = m - (s + b)$ bits</td>
<td>$s$ bits</td>
<td>$b$ bits</td>
</tr>
</tbody>
</table>
CACHE ACCESS OPERATIONS

- When cache receives a memory access, it must:
  - Figure out the cache set where the block goes
  - Figure out which cache line matches the block (*if any*)
  - Access the specific value within the cached block

- Given our mapping from memory addresses to cache details, these steps become pretty easy:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Set Index</th>
<th>Block Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t = m - (s + b)$ bits</td>
<td>$s$ bits</td>
<td>$b$ bits</td>
</tr>
</tbody>
</table>

  - Cache set’s index is the middle $s$ bits
  - Use tag bits to identify the cache line within the set
  - Bottom-most $b$ bits are the access’ offset within the block
DIRECT-MAPPED CACHES

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

$m$-bit address: 

<table>
<thead>
<tr>
<th>tag</th>
<th>set index</th>
<th>block offset</th>
</tr>
</thead>
</table>

$t = m - (s + b)$ bits

Example: direct-mapped cache with 4 sets
- 2 bits in set index
- Very fast to map an address to a cache set
- Very fast to determine if a block is in the cache
  - If the tag doesn’t match, block isn’t in the cache!

Direct-Mapped Cache with 4 Sets

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

Example: fully-associative cache

- Still very fast to map an address to a cache set
- More complicated to find if a block is in the cache
  - Need to examine all cache-line tags
  - Also, the tag is larger than in a direct-mapped cache
**Set-Associative Caches**

- Direct-mapped caches can easily cause a large number of conflict misses.
- Fully associative caches must have complex logic to identify blocks by their tags.
- **Set-associative caches** combine capabilities of both approaches into a single cache:
  - Employ $S$ cache sets, where $S > 1$.
  - Each cache set contains $E$ cache lines, where $E > 1$.
- Achieves benefits of both techniques:
  - Greatly reduces potential for conflict misses.
  - Limits the complexity of the logic that has to match block tags.
**Set-Associative Caches (2)**

- **Example:** a two-way set-associative cache
  - $E$ is number of cache lines in each set...
  - Cache is called “$E$-way set-associative cache” when $S > 1$
  - $E = 2$ for this example

- For an $m$-bit address:
  - $t$ bits
  - $s = 2$ bits
  - $b$ bits
  - | tag | set idx | blk off |

- Use set index to find cache set to examine
- Only have to check tags on small number of lines

---

**Two-Way Set-Associative Cache**

<table>
<thead>
<tr>
<th></th>
<th>Valid</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set 0</strong></td>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
</tr>
<tr>
<td></td>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Valid</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set 1</strong></td>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
</tr>
<tr>
<td></td>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Valid</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set 2</strong></td>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
</tr>
<tr>
<td></td>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Valid</th>
<th>Tag</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Set 3</strong></td>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
</tr>
<tr>
<td></td>
<td>Valid</td>
<td>Tag</td>
<td>Block</td>
</tr>
</tbody>
</table>
**Line Replacement Policies**

- When a cache miss occurs, must load a new block into the cache
  - Store in some cache line
- For direct-mapped caches, this is easy
  - Only one line per set
- For set-associative and fully associative caches, we get to choose a line!
- Replacement policy controls which cache line to evict when new block is loaded into cache

---

### 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>Valid</td>
<td>Valid</td>
<td>Valid</td>
</tr>
<tr>
<td>Tag</td>
<td>Tag</td>
<td>Tag</td>
<td>Tag</td>
</tr>
<tr>
<td>Block</td>
<td>Block</td>
<td>Block</td>
<td>Block</td>
</tr>
</tbody>
</table>
LINE REPLACEMENT POLICIES (2)

- Least Recently Used (LRU) policy
  - Evict cache line that was accessed furthest in past

- Least Frequently Used (LFU) policy
  - Evict cache line that was accessed the least frequently, over some time window

- These policies take extra time and hardware to implement
  - Not used as much in caches close to the CPU, where performance is critical
  - Used very often in caches further from the CPU, where cache misses are extremely costly

- For example, disk-block caches benefit greatly from more sophisticated replacement policies
  - ...when a cache miss costs 20 million clocks, spend a few thousand clocks to figure out what to keep in the cache...
**ASSOCIATIVE CACHES**

- Where does the “associative” come from in set-associative caches and fully-associated caches?
- Each cache set has $E$ cache lines in it...
  - Need to look up cache line using only the block’s tag
  - The cache set is an *associative memory*
- Associative memory:
  - Not accessed with an address, like normal memories!
  - Associative memory stores (key, value) pairs
  - Key is the input to the associative memory
  - Memory returns value

![Fully Associative Cache Diagram]
**Associative Caches (2)**

- Associative caches must effectively implement associative memories for their cache sets
  - Keys are a concatenation of the tag, *plus* the valid flag
    - *No reason to look at the cache line if it isn’t valid…*
  - Value is the block of data in the cache

- Set-associative caches:
  - Each cache set is an associative memory
  - Number of cache lines in each set is small, so logic is easier to implement

- Fully associative caches:
  - Need to examine *many* cache lines in parallel
  - *Much* more expensive…