The Memory Hierarchy
Memory

CPU

Memory
Memory

Abstraction: Big array of bytes
Main points for today

- What is a memory hierarchy?
- What is the CPU-DRAM gap?
- What is locality? What kinds are there?
- Learn a bunch of caching vocabulary.
Who Cares about Memory Hierarchy?

- Processor vs Memory Performance

1980: no cache in microprocessor;
1995 2-level cache
Memory’s impact

M = % mem ops
Mlat (cycles) = memory latency
BCPI = base CPI

CPI =
Memory’s impact

M = % mem ops
Mlat (cycles) = memory latency
BCPI = base CPI

CPI = BCPI + M*Mlat

BCPI = 1; M = 0.2; Mlat = 100 (somewhat aggressive today)

CPI = 21 -> 20x slow down

Remember!: Amdahl’s law does not bound the slowdown. Poor memory performance can make your program arbitrarily slow.
- **Memory Cache**

- **Memory cost**
  - >> capacity -> more $$
  - >> speed/bw -> more $$
  - >> speed -> larger (less dense)

- Build several memories with different trade-offs
- How do you use it? Build a “memory hierarchy”
- What should it mean for the memory abstraction?
Memory Cache

- Memory cost
  - >> capacity -> more $$$
  - >> speed/bw -> more $$$
  - >> speed -> larger (less dense)

- Build several memories with different trade-offs

- How do you use it? Build a “memory hierarchy”

- What should it mean for the memory abstraction?
Memory Cache

- Memory cost
  - \( \gg \) capacity \( \rightarrow \) more $$
  - \( \gg \) speed/bw \( \rightarrow \) more $$
  - \( \gg \) speed \( \rightarrow \) larger (less dense)

- Build several memories with different trade-offs
- How do you use it? Build a “memory hierarchy”
- What should it mean for the memory abstraction?
A typical memory hierarchy

- **on-chip cache**: KBs
- **off-chip cache**: MBs
- **main memory**: GBs
- **Disk**: TBs

<table>
<thead>
<tr>
<th>Level</th>
<th>Cost</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>on-chip cache</td>
<td>2.5 $/MB</td>
<td>&lt; 1ns</td>
</tr>
<tr>
<td>off-chip cache</td>
<td>0.07 $/MB</td>
<td>60ns</td>
</tr>
<tr>
<td>main memory</td>
<td>0.0004 $/MB</td>
<td>10,000,000ns</td>
</tr>
</tbody>
</table>
Why should we expect caching to work?
Why should we expect caching to work?

- Why did branch prediction work?
- Where is memory access predictable
  - Predictably accessing the same data
    - In loops: for(i = 0; i < 10; i++) {s += foo[i];}
    - foo = value + configuration_parameter;
    - bar = another_value + configuration_parameter;
  - Predictably accessing different data
    - In linked lists: while(l != NULL) {l = l->next;}
    - In arrays: for(i = 0; i < 10000; i++) {s += data[i];}
    - structure access: foo(some_struct.a, some_struct.b);
The Principle of Locality

• “Locality” is the tendency of data access to be predictable. There are two kinds:

• Spatial locality: The program is likely to access data that is close to data it has accessed recently

• Temporal locality: The program is likely to access the same data repeatedly.
Locality in Action

- Label each access with whether it has temporal or spatial locality or neither
  - 1
  - 2
  - 3
  - 10
  - 4
  - 1800
  - 11
  - 30
  - 1
  - 2
  - 3
  - 4
  - 10
  - 190
  - 11
  - 30
  - 12
  - 13
  - 182
  - 1004
Locality in Action

- Label each access with whether it has temporal or spatial locality or neither
  - 1 n
  - 2 s
  - 3 s
  - 10 n
  - 4 s
  - 1800 n
  - 11 s
  - 30 n
  - 1 t
  - 2 s, t

- 3 s,t
- 4 s,t
- 10 s,t
- 190 n
- 11 s,t
- 30 s
- 12 s
- 13 s
- 182 n?
- 1004 n
Locality in Action

- Label each access with whether it has temporal or spatial locality or neither
  - 1 n
  - 2 s
  - 3 s
  - 10 n
  - 4 s
  - 1800 n
  - 11 s
  - 30 n
  - 1 t
  - 2 s, t
  - 3 s,t
  - 4 s,t
  - 10 s,t
  - 190 n
  - 11 s,t
  - 30 s
  - 12 s
  - 13 s
  - 182 n?
  - 1004 n

There is no hard and fast rule here.
Typical Cache Hierarchy

- **Fetch/L1 Icache 16KB**
- **Decode**
- **EX**
- **Mem L1 Dcache 16KB**
- **Write back**
- **Unified L2 8MB**
- **Unified L3 32MB**
- **DRAM Many GBs**
The Memory Hierarchy

- There can be many caches stacked on top of each other.
- If you miss in one you try in the “lower level cache” Lower level, mean higher number.
- There can also be separate caches for data and instructions. Or the cache can be “unified”
- To wit:
  - The L1 data cache (d-cache) is the one nearest processor. It corresponds to the “data memory” block in our pipeline diagrams.
  - The L1 instruction cache (i-cache) corresponds to the “instruction memory” block in our pipeline diagrams.
- The L2 sits underneath the L1s.
- There is often an L3 in modern systems.
Cache Vocabulary

- Hit - The data was found in the cache
- Miss - The data was not found in the cache
- Hit rate - hits/total accesses
- Miss rate = 1 - Hit rate
- Locality - see previous slides
- Cache line - the basic unit of data in a cache. generally several words.
- Tag - the high order address bits stored along with the data to identify the actual address of the cache line.
- Hit time -- time to service a hit
- Miss time -- time to service a miss (this is a function of the lower level caches.)
Cache Vocabulary

• There can be many caches stacked on top of each other
• if you miss in one you try in the “lower level cache”  Lower level, mean higher number
• There can also be separate caches for data and instructions. Or the cache can be “unified”
• to whit:
  • the L1 data cache (d-cache) is the one nearest processor. It corresponds to the “data memory” block in our pipeline diagrams
  • the L1 instruction cache (i-cache) corresponds to the “instruction memory” block in our pipeline diagrams.
• The L2 sits underneath the L1s.
• There is often an L3 in modern systems.
Data vs Instruction Caches

• Why have different I and D caches?
  • Different areas of memory
  • Different access patterns
    • I-cache accesses have lots of spatial locality. Mostly sequential accesses.
    • I-cache accesses are also predictable to the extent that branches are predictable
    • D-cache accesses are typically less predictable
  • Not just different, but often across purposes.
    • Sequential I-cache accesses may interfere with the data the D-cache has collected.
    • This is “interference” just as we saw with branch predictors
  • At the L1 level it avoids a structural hazard in the pipeline
  • Writes to the I cache by the program are rare enough that they can be prohibited (i.e., self modifying code)
Key Point

- What are
  - Cache lines
  - Tags
  - Index
  - Offset
- How do we find data in the cache?
- How do we tell if it’s the right data?
- What decisions do we need to make in designing a cache?
- What are possible caching policies?
Typical Cache Hierarchy

- **Fetch/L1 Icache 16KB**
- **Decode**
- **EX**
- **Mem L1 Dcache 16KB**
- **Write back**

**Unified L2**
- 8MB

**Unified L3**
- 32MB

**DRAM**
- Many GBs
Data vs Instruction Caches

- Why have different I and D caches?
Data vs Instruction Caches

• Why have different I and D caches?
  • Different areas of memory
  • Different access patterns
    • I-cache accesses have lots of spatial locality. Mostly sequential accesses.
    • I-cache accesses are also predictable to the extent that branches are predictable
    • D-cache accesses are typically less predictable
  • Not just different, but often across purposes.
    • Sequential I-cache accesses may interfere with the data the D-cache has collected.
    • This is “interference” just as we saw with branch predictors
  • At the L1 level it avoids a structural hazard in the pipeline
  • Writes to the I cache by the program are rare enough that they can be prohibited (i.e., self modifying code)
The Cache Line

- Caches operate on “lines”
- Caches lines are a power of 2 in size
  - They contain multiple words of memory.
- Usually between 16 and 128 bytes
- The bit-width (i.e., 32 or 64 bits) does directly effect the cache configuration.
  - In fact almost all aspects of a cache and independent of the big-A architecture.
- Caches are completely transparent to the processor.
Basic Problems in Caching

- A cache holds a small fraction of all the cache lines, yet the cache itself may be quite large (i.e., it might contain 1000s of lines)
- Where do we look for our data?
- How do we tell if we’ve found it and whether it’s any good?
Basic Cache Organization

- **Anatomy of a cache line entry**
  - Dirty bit -- does this data match what is in memory
  - Valid -- does this mean anything at all?
  - Tag -- The high order bits of the address
  - Data -- The program’s data

- **Anatomy of an address**
  - Index -- bits that determine the lines possible location
  - Offset -- which byte within the line (low-order bits)
  - Tag -- everything else (the high-order bits)

- Note that the index of the line, combined with the tag, uniquely identify one cache line’s worth of memory
Cache line size

• How big should a cache line be?
• Why is bigger better?
  • Exploits more spatial locality.
  • Large cache lines effectively *prefetch* data that we have not explicitly asked for.
• Why is smaller better?
  • Focuses on temporal locality.
  • If there is little spatial locality, large cache lines waste space and bandwidth.
  • More space devoted to tags.
• In practice 32-64 bytes is good for L1 caches were space is scarce and latency is important.
• Lower levels use 128-256 bytes.
Cache Geometry Calculations

- Addresses break down into: tag, index, and offset.
- How they break down depends on the “cache geometry”

- Cache lines = L
- Cache line size = B
- Address length = A (32 bits in our case)

- Index bits = $\log_2(L)$
- Offset bits = $\log_2(B)$
- Tag bits = $A - (\text{index bits} + \text{offset bits})$
Reading from a cache

• Determine where in the cache, the data could be
• If the data is there (i.e., is it hit?), return it
• Otherwise (a miss)
  • Retrieve the data from the lower down the cache hierarchy.
  • Is there a cache line available for the new data?
    • If so, fill the the line, and return the value
    • Otherwise choose a line to evict
      • Is it dirty? Write it back.
      • Otherwise, just replace it, and return the value
Reading from a cache

• Determine where in the cache, the data could be
• If the data is there (i.e., is it hit?), return it
• Otherwise (a miss)
  • Retrieve the data from the lower down the cache hierarchy.
  • Is there a cache line available for the new data?
    • If so, fill the the line, and return the value
    • Otherwise choose a line to evict
      • Is it dirty? Write it back.
      • Otherwise, just replace it, and return the value
Hit or Miss?

- Use the index to determine where in the cache, the data might be
- Read the tag at that location, and compare it to the tag bits in the requested address
- If they match (and the data is valid), it’s a hit
- Otherwise, a miss.
On a Miss: Finding Room

• We need space in the cache to hold the data that is missing
• The cache entry at the required index might be invalid, if so, great! there is space.
• Otherwise, we need to \textit{evict} the cache line at this index.
  • If it’s dirty, we need to \textit{write it back}
  • Otherwise (it’s clean), we can just overwrite it.
Writing To the Cache (simple version)

- Determine where in the cache, the data could be
- If the data is there (i.e., is it hit?), update it
- Possibly forward the request down the hierarchy
- Otherwise
  - Retrieve the data from the lower down the cache hierarchy (why?)
  - Is there a cache line available for the new data?
    - If so, fill the the line, and update it
    - Otherwise choose a line to evict
      - Is it dirty? Write it back.
      - Otherwise, just replace it, and update it.
    - Alternate Otherwise
      - Forward the write request down the hierarchy
Writing To the Cache (simple version)

- Determine where in the cache, the data could be
- If the data is there (i.e., is it hit?), update it
- Possibly forward the request down the hierarchy
- Otherwise
  - Retrieve the data from the lower down the cache hierarchy (why?)
  - Is there a cache line available for the new data?
    - If so, fill the the line, and update it
    - Otherwise choose a line to evict
      - Is it dirty? Write it back.
      - Otherwise, just replace it, and update it.
  - Alternate Otherwise
    - Forward the write request down the hierarchy

<-- Replacement policy
Writing To the Cache (simple version)

• Determine where in the cache, the data could be
• If the data is there (i.e., is it hit?), update it
• Possibly forward the request down the hierarchy
• Otherwise
  • Retrieve the data from the lower down the cache hierarchy (why?)
  • Is there a cache line available for the new data?
    • If so, fill the the line, and update it
    • Otherwise choose a line to evict
      • Is it dirty? Write it back.
      • Otherwise, just replace it, and update it.
    • Alternate Otherwise
      • Forward the write request down the hierarchy
Writing To the Cache (simple version)

- Determine where in the cache, the data could be
- If the data is there (i.e., is it hit?), update it
- Possibly forward the request down the hierarchy  <-- Write back policy
- Otherwise
  - Retrieve the data from the lower down the cache hierarchy (why?)
  - Is there a cache line available for the new data?
    - If so, fill the the line, and update it
    - Otherwise choose a line to evict  <-- Replacement policy
      - Is it dirty? Write it back.
      - Otherwise, just replace it, and update it.
    - Alternate Otherwise  <-- Write allocation policy
      - Forward the write request down the hierarchy
Write Through vs. Write Back

• When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
• If we do not forward the write, the cache is “Write back”, since the data must be written back when it’s evicted (i.e., the line can be dirty)
• If we do forward the write, the cache is “write through.” In this case, a cache line is never dirty.
• Write back advantages

• Write through advantages
Write Through vs. Write Back

• When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
• If we do not forward the write, the cache is “Write back”, since the data must be written back when it’s evicted (i.e., the line can be dirty)
• If we do forward the write, the cache is “write through.” In this case, a cache line is never dirty.

Write back advantages

Write through advantages

No more write backs. Reads might be faster.
Write Through vs. Write Back

• When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
• If we do not forward the write, the cache is “Write back”, since the data must be written back when it’s evicted (i.e., the line can be dirty)
• If we do forward the write, the cache is “write through.” In this case, a cache line is never dirty.

• Write back advantages
  Fewer writes farther down the hierarchy. Less bandwidth. Faster writes

• Write through advantages
  No more write backs. Reads might be faster.
Write Allocate/No-write allocate

- On a write miss, we don’t actually need the data, we can just forward the write request
- If the cache allocates cache lines on a write miss, it is *write allocate*, otherwise, it is *no write allocate*.
- Write Allocate advantages

- No-write allocate advantages
  Fewer spurious evictions. If the data is not read for a long time, the eviction is a waste.
Write Allocate/No-write allocate

- On a write miss, we don’t actually need the data, we can just forward the write request
- If the cache allocates cache lines on a write miss, it is *write allocate*, otherwise, it is *no write allocate*.
- Write Allocate advantages
  Exploits temporal locality. Data written will likely be read soon, and that read will be faster.
- No-write allocate advantages
  Fewer spurious evictions. If the data is not read for a long time, the eviction is a waste.
Write Allocate Caveat

- Write allocate caches must fill the cache line before they can complete the store, since the line contains multiple words.
- If they do not do the fill first, only the portion of the line that was actually written to would be valid.
Dealing the Interference

• By bad luck or pathological happenstance a particular line in the cache may be highly contended.
• How can we deal with this?
Associativity

- (set) Associativity means providing more than one place for a cache line to live.
- The level of associativity is the number of possible locations
  - 2-way set associative
  - 4-way set associative
- One group of lines corresponds to each index
  - it is called a “set”
- Each line in a set is called a “way”
## Associativity

<table>
<thead>
<tr>
<th>Set 0</th>
<th>Set 1</th>
<th>Set 2</th>
<th>Set 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>dirty</td>
<td>valid</td>
<td>Tag</td>
<td>Data</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Way 0</th>
<th>Way 1</th>
</tr>
</thead>
</table>
New Cache Geometry Calculations

• Addresses break down into: tag, index, and offset.
• How they break down depends on the “cache geometry”

• Cache lines = L
• Cache line size = B
• Address length = A (32 bits in our case)
• Associativity = W

• Index bits = \( \log_2(L/W) \)
• Offset bits = \( \log_2(B) \)
• Tag bits = A - (index bits + offset bits)
Eviction in Associative caches

• We must choose which line in a set to evict if we have associativity
• How we make the choice is called *the cache eviction policy*
  • Random -- always a choice worth considering. Hard to implement true randomness.
  • Least recently used (LRU) -- evict the line that was last used the longest time ago.
  • Prefer clean -- try to evict clean lines to avoid the write back.
  • Farthest future use -- evict the line whose next access is farthest in the future. This is provably optimal. It is also impossible to implement.
The Cost of Associativity

- Increased associativity requires multiple tag checks
  - N-Way associativity requires N parallel comparators
  - This is expensive in hardware and potentially slow.
  - The fastest way is to use a “content addressable memory” They embed comparators in the memory array. -- try instantiating one in Xlinix.
- This limits associativity L1 caches to 2-4. In L2s to make 16 way.
Increasing Bandwidth

• A single, standard cache can service only one operation at time.
• We would like to have more bandwidth, especially in modern multi-issue processors
• There are two choices
  • Extra ports
  • Banking
Extra Ports

- **Pros**: Uniformly supports multiple accesses
  - Any N addresses can be accessed in parallel.
- **Costly in terms of area.**
  - Remember: SRAM size increases quadratically with the number of ports
Banking

- Multiple, independent caches, each assigned one part of the address space (use some bits of the address)
- Pros: Efficient in terms of area. Four banks of size $N/4$ are only a bit bigger than one cache of size $N$.
- Cons: Only one access per bank. If you are unlucky you don’t get the extra.