Caching

(get it??)
• Caching
• Cache implementations
• Cache policies
• Cache performance
• Performance improvement strategies
Motivation

- Caching is ubiquitous in modern computing:
  - L1-L3 memory
  - TLB and virtual memory (next week)
  - Disk controller buffers
  - Network controller buffers
  - Browser caches
  - Content delivery networks
Caching

- A **cache** is a small, fast memory that acts as a buffer or staging area for a larger, slower memory
  - Fundamental CS system design concept
  - Data is transferred in **blocks** or **lines**
  - Slower caches use larger block sizes
  - **Cache hit vs. cache miss**
  - **Hit ratio**: # hits / # memory accesses
Cache implementations

• What data structure can we use to implement caches?
  - Need **FAST** lookups and containment checks
  - From CS 240: use a hash table!
  - Cache slot = "real address" % CACHE_SIZE

Multiple addresses may map to the same cache slot! (this is called a **conflict**)

Here there are eight slots, so the slot ID is the last 3 bits of the memory address ($2^3 = 8$)
Question

- Suppose we have a sixteen-element cache, with slots labeled starting at zero. Which slot would we use to store a cached version of a data element stored at address 0x4d6?
  - Reminder: cache slot = "real address" % CACHE_SIZE
  - Hint: $2^4 = 16$, and four bits = one hex digit
Cache implementations

- A cache line is a block or sequence of bytes that is moved between memory levels in a single operation.
- A cache set is a collection of one or more cache lines.
  - Each cache line contains a tag to identify the source address and a valid flag/bit indicating whether the value is up-to-date.

\[
\begin{array}{c|c|c|c}
\text{Valid} & \text{Tag} & 0 & 1 \ldots B-1 \\
\hline
\text{Valid} & \text{Tag} & 0 & 1 \ldots B-1 \\
\hdots & \hdots & \hdots & \hdots \\
\text{Valid} & \text{Tag} & 0 & 1 \ldots B-1 \\
\end{array}
\]

\[S = 2^e \text{ sets}\]

\[1 \text{ valid bit per line} \quad t \text{ tag bits per line} \quad B = 2^b \text{ bytes per cache block}\]

\[C = B \times E \times S \text{ data bytes}\]
Cache implementations

- General cache organization:
  - \( S = \) # of cache sets = \( 2^s \)
    - \( s = \) # of bits for set index
  - \( E = \) # of lines per cache set
    - Level of associativity
  - \( B = \) block (cache line) size = \( 2^b \)
    - Essentially bytes per line
    - \( b = \) # of bits for block offset
  - \( m = \) # of bits for memory address
    - \( M = \) size of memory in bytes = \( 2^m \)
  - \( C = \) total cache capacity = \( S \times E \times B \)
    - sets x lines/set x bytes/line
  - \( t = \) # of tag bits = \( m - s - b \)

\[
\begin{align*}
\text{Cache size: } C &= B \times E \times S \text{ data bytes} \\
\text{Address: } &\quad &\text{Tag} &\quad \text{Set index} &\quad \text{Block offset}
\end{align*}
\]
Types of caches

- **Direct-mapped** \((E = 1)\)
  - One line per set

- **Set-associative** \((1 < E < C/B)\)
  - Multiple lines per set

- **Fully-associative** \((E = C/B)\)
  - All lines in one set

Here \(E = 2\)
Cache implementations

- **Direct-mapped** \((E = 1)\) caches

![Diagram of direct-mapped cache with set selection and block indexing](image)

- The valid bit must be set (1).
- The tag bits in the cache line must match the tag bits in the address (2).
- If (1) and (2), then cache hit, and block offset selects starting byte (3).
Question

- Suppose we have a direct-mapped cache (S=16, B=1), with sets labeled starting at zero. Which set would we use to store a cached version of a data element stored at address 0x4d6?
  - Hint: S=16 so the number of bits for the set index is four
  - Hint: B=1 so the number of bits for the block offset is zero
Cache implementations

- **Set-associative** \((1 < E < C/B)\) caches

  - Set-associative caches allow for \(E\) ways to access the same memory location, where \(E\) is a power of 2. For example, a two-way set associative cache, denoted as \(E=2\), can access the same memory location in two ways.

  - The diagram illustrates how the cache block is divided into sets, with each set containing \(E\) cache lines. The valid bit indicates whether a cache line is valid or not.

  - The set index is used to select a specific set out of \(S\) total sets, where \(S = C / |b|\), and \(|b|\) is the block size.

  - The selected set is used to determine which cache line is accessed.

  - The cache block diagram shows how the cache blocks are organized with the valid bit, tag, and cache block.

  - The diagram also includes conditions for cache hits and misses, and how the block offset selects the starting byte.
• Suppose we have a four-way set-associative cache (S=16, E=4, B=1), with sets labeled starting at zero. Which set would we use to store a cached version of a data element stored at address 0x4d6?
Cache implementations

- **Fully-associative** \((E = C/B)\) caches

  The entire cache is one set, so by default set 0 is always selected.

  **Set 0:**

  - \(t\) bits Tag
  - \(b\) bits Block offset

  \(E = C/B\) lines in the one and only set

  **Entire cache**

  - (1) The valid bit must be set
  - (2) The tag bits in one of the cache lines must match the tag bits in the address
  - (3) If (1) and (2), then cache hit, and block offset selects starting byte
Question

Suppose we have a fully-associative cache ($B=1$) with sets labeled starting at zero. Which set would we use to store a cached version of a data element stored at address $0x4d6$?
Cache implementations

- In general, we use the middle bits for the set index
  - Contiguous memory blocks should map to different cache sets
Cache misses (“Three C’s”)

- **Compulsory / cold miss**
  - First cache miss due to an “empty” cache
  - As the cache loads data, it is warmed up

- **Conflict miss**
  - Cache miss due to multiple lines in working set mapping to the same cache line
  - Repeated conflict misses for the same cache lines or blocks is called **thrashing**

- **Capacity miss**
  - The **working set** (amount of memory accessed in a given time interval) is too large to fit in cache
Cache policies

• If a cache set is full, a cache miss in that set requires lines to be replaced or evicted

• Policies:
  - Random replacement
  - Least recently used
  - Least frequently used

• These policies require additional overhead
  - More important for lower levels of the memory hierarchy
Cache policies

• How should we handle writes to a cached value?
  – **Write-through**: immediately update to lower level
    • Typically used for higher levels of memory hierarchy
  – **Write-back**: defer update until replacement/eviction
    • Typically used for lower levels of memory hierarchy

• How should we handle write misses?
  – **Write-allocate**: load then update
    • Typically used for write-back caches
  – **No-write-allocate**: update without loading
    • Typically used for write-through caches
Performance impact

- **Metrics**
  - **Hit rate/ratio**: # hits / # memory accesses (1 – miss rate)
    - **Hit time**: delay in accessing data for a cache hit
  - **Miss rate/ratio**: # misses / # memory accesses
    - **Miss penalty**: delay in loading data for a cache miss
  - **Read throughput** (or "bandwidth"): the rate that a program reads data from a memory system

- **General observations**:
  - Larger cache = higher hit rate but higher hit time
  - Lower miss rates = higher read throughput
Case study: matrix multiply

(a) Version ijk

```c
1 for (i = 0; i < n; i++)
2   for (j = 0; j < n; j++) {
3     sum = 0.0;
4     for (k = 0; k < n; k++)
5       sum += A[i][k]*B[k][j];
6     C[i][j] += sum;
7 }
```

(b) Version jik

```c
1 for (j = 0; j < n; j++)
2   for (i = 0; i < n; i++) {
3     sum = 0.0;
4     for (k = 0; k < n; k++)
5       sum += A[i][k]*B[k][j];
6     C[i][j] += sum;
7 }
```

(c) Version jki

```c
1 for (j = 0; j < n; j++)
2   for (k = 0; k < n; k++) {
3     r = B[k][j];
4     for (i = 0; i < n; i++)
5       C[i][j] += A[i][k]*r;
6 }
```

(d) Version kji

```c
1 for (k = 0; k < n; k++)
2   for (j = 0; j < n; j++) {
3     r = B[k][j];
4     for (i = 0; i < n; i++)
5       C[i][j] += A[i][k]*r;
6 }
```

(e) Version kij

```c
1 for (k = 0; k < n; k++)
2   for (i = 0; i < n; i++) {
3     r = A[i][k];
4     for (j = 0; j < n; j++)
5       C[i][j] += r*B[k][j];
6 }
```

(f) Version ikj

```c
1 for (i = 0; i < n; i++)
2   for (k = 0; k < n; k++) {
3     r = A[i][k];
4     for (j = 0; j < n; j++)
5       C[i][j] += r*B[k][j];
6 }
```

Figure 6.44  Six versions of matrix multiply. Each version is uniquely identified by the ordering of its loops.
Case study: matrix multiply
Optimization strategies

• Focus on the common cases
• Focus on the code regions that dominate runtime
• Focus on inner loops and minimize cache misses
• Favor repeated local accesses (temporal locality)
• Favor stride-1 access patterns (spatial locality)

Tip: You can use Valgrind to detect cache misses (look up a tool called cachegrind)
Core theme

- **Cache system design involves tradeoffs**
  - Larger caches => higher hit rate but higher hit time
    - Size vs. speed
  - Larger blocks => higher hit rate for programs with good spatial locality, but lower hit rate for others
    - Favor spatial vs. temporal locality
  - **Higher associativity** => lower chance of thrashing but expensive to implement w/ possibly increased hit time
    - Hit time vs. miss penalty
  - More writes => simpler to implement but lower performance
    - Write-through vs. write-back
Next time

- **Virtual memory**: an OS-level memory cache
  - Bridge between module 4 (machine architectures) and module 5 (operating systems)
Cache architecture

- **Example:** Intel Core i7
- **Per-core:**
  - Registers
  - L1 d-cache and i-cache
    - Data and instructions
  - L2 unified cache
- **Shared:**
  - L3 unified cache
  - Main memory
As the **working set size** of a loop decreases, what generally happens to the read throughput?

- A) It increases
- B) It decreases
- C) It remains the same
- D) There is no correlation
- E) Not enough information to determine
Temporal locality

- Working set size vs. throughput
As the **stride** of a loop increases, what generally happens to the read throughput?

- A) It increases
- B) It decreases
- C) It remains the same
- D) There is no correlation
- E) Not enough information to determine
Spatial locality

- Stride vs. throughput
Memory mountain (CS:APP)

- Stride and WSS vs. read throughput

![3D graph showing memory hierarchy and read throughput.](image)
Memory mountain (stu, 2017)

Output of `lscpu`:

- **Architecture:** x86_64
- **Byte Order:** Little Endian
- **CPU(s):** 24
- **Thread(s) per core:** 2
- **Core(s) per socket:** 6
- **Socket(s):** 2
- **Vendor ID:** Intel
- **Model name:** Intel(R) Xeon(R) CPU E5-2640
- **CPU max MHz:** 3000.0000
- **CPU min MHz:** 1200.0000
- **L1d cache:** 32K
- **L1i cache:** 32K
- **L2 cache:** 256K
- **L3 cache:** 15360K
Output of `lscpu`:

- **Architecture:** x86_64
- **Byte Order:** Little Endian
- **CPU(s):** 48
- **Thread(s) per core:** 2
- **Core(s) per socket:** 12
- **Socket(s):** 2
- **Vendor ID:** Intel
- **Model name:** Intel(R) Xeon(R) CPU E5-2680
- **CPU max MHz:** 3300.0000
- **CPU min MHz:** 1200.0000
- **L1d cache:** 32K
- **L1i cache:** 32K
- **L2 cache:** 256K
- **L3 cache:** 30720K
Output of `lscpu`:

Architecture:          x86_64
Byte Order:            Little Endian
CPU(s):                48
Thread(s) per core:    2
Core(s) per socket:    12
Socket(s):             2
Vendor ID:             Intel
Model name:            Intel(R) Xeon(R) CPU E5-2680 v3
CPU max MHz:           3300.0000
CPU min MHz:           1200.0000
L1d cache:             768K
L1i cache:             768K
L2 cache:              6M
L3 cache:              60M

Note: new per-user resource limits put in place Fall 2021 may be interfering
Question

- Assume the following cache:
  - \( S = 8 \) sets (so \( s=3 \) bits for set index)
  - \( E = 1 \) line per set (so direct-mapped)
  - \( B = 4 \) bytes per line (so \( b=2 \) bits for block offset)

- What is the set index, tag, and block offset for address 227?
  - Hint: 227 in binary is 11100011
Assume the following cache:
- \( S = 8 \) sets (so \( s = 3 \) bits for set index)
- \( E = 1 \) line per set (so direct-mapped)
- \( B = 4 \) bytes per line (so \( b = 2 \) bits for block offset)

Address 227 (binary: 11100011)
- Set index = 000\(_2\) (0)
- Tag = 111\(_2\) (7)
- Block offset = 11\(_2\) (3)

Is this a hit?  No! Need to load the line into cache:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (000)</td>
<td>1</td>
<td>111</td>
<td>m[224]</td>
<td>m[225]</td>
<td>m[226]</td>
<td>m[227]</td>
</tr>
</tbody>
</table>
• Assume the following cache:
  - \( S = 8 \) sets (so \( s = 3 \) bits for set index)
  - \( E = 1 \) line per set (so direct-mapped)
  - \( B = 4 \) bytes per line (so \( b = 2 \) bits for block offset)

• What is the set index, tag, and block offset for address 226? Is it a hit?
  - Hint: 226 in binary is 11100010