Optimizing Cache Performance
Today’s Lecture

- Use larger cache blocks to take advantage of spatial locality
- Use set associativity to resolve “hashing collisions”
For a byte-addressable machine with 16-bit addresses

A cache has the following characteristics:
- It is direct-mapped (as discussed last time)
- Each block holds one byte
- The cache index is the four least significant bits

Two questions:
- How many blocks does the cache hold?
- How many bits are stored at each cache block (e.g., for the data array, tags, etc.)?
1-byte cache blocks do not take advantage of spatial locality

Create larger cache blocks
Creating larger cache blocks moves adjacent blocks as a unit.

16-byte main memory

Two bytes (12 and 13) move as a unit.
Each cache block is an array indexed by a block offset

Create an “array of arrays”
The least-significant bit(s) is used to index the block offset “array”.

![Diagram showing memory address, 8-bit data, block offset, and block index relationships.](image)
Data gets moved in cache blocks and not bytes

Consider the following set of memory loads:
- load 0
- load 15
- load 10
- load 1
To increase cache block size, subdivide an \( m \)-bit address into tag, index, and block offset

- Suppose we have a cache with \( 2^k \) blocks, each containing \( 2^n \) bytes.
  - Lowest \( n \) bits are the **block offset** that decides which of the \( 2^n \) bytes in the cache block will store the data.
  - Next \( k \) bits of the address select one of the \( 2^k \) cache blocks.

\[
\begin{array}{c|c|c}
(m-k-n) & k & \text{n-bit Block Offset} \\
\hline
\text{m-bit Address} & \text{Tag} & \text{Index} \\
\end{array}
\]

- Example: \( 2^2 \)-block cache with \( 2^1 \) bytes per block. Memory address 13 (1101) would be stored in offset 1 of cache block 2.

\[
\begin{array}{c|c|c}
1-bit tag & 2 & \text{1-bit Block Offset} \\
\hline
\text{4-bit Address} & 1 & 10 \quad 1 \\
\end{array}
\]
Implement block offset with a multiplexer
For the addresses below, what byte is read from the cache (or is there a miss)?

- 1010
- 1110
- 0001
- 1101
Direct-mapped caches fall short when addresses collide

- Example: what happens if a program uses addresses 2, 6, 2, 6, 2, ...?
A **fully-associative cache** permits data to be stored in *any* cache block

- When data is fetched from memory, it can be placed in *any* unused block of the cache.
- Maximize temporal locality by keeping the most recently used data in the cache, replace the **least-recently used (LRU)** when cache is full.
A fully associative cache can map addresses anywhere, eliminating thrashing.

- Example: what happens if a program uses addresses 2, 6, 2, 6, 2, ...?
A fully-associative cache is expensive because we need to store the entire tag!

- Data could be anywhere in the cache, so we must check the tag of every cache block. That’s a lot of comparators!
A set-associative cache organizes cache blocks into groups called sets

- Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set.
- If each set has $2^x$ blocks, the cache is a $2^x$-way associative cache.
Blocks can still have multiple bytes

- Direct mapped:
  8 "sets", 1 block each

- 2-way associativity:
  4 sets, 2 blocks each

- 4-way associativity:
  2 sets, 4 blocks each
To find data, subdivide the address into tag, index, and block offset as before

- Memory has m-bit address
- Cache has $2^s$ sets and each block has $2^n$ bytes

Our arithmetic computations now compute a **set index**, to select a **set** within the cache instead of an individual block.

- **Block Offset** = Memory Address mod $2^n$
- **Set Index** = $(\text{Memory Address} / 2^n) \mod 2^s$
Example: Placement of address 6195 in caches with 8, 16-byte cache blocks

- 6195 in binary is 00...0110000 0 11 0011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.

The next three bits (011) are the set index.

The next two bits (11) are the set index.
Example: Placement of address 6195 in caches with 8, 16-byte cache blocks

- 6195 in binary is 00...011000001 10011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.

the next one bit (1) is the set index

The data may go in *any* block, shown in green
Place new data in empty cache blocks if possible, else, replace the least-recently used.

Approximate LRU for high associativity.
Given a fully-associative cache with two blocks, which of memory accesses miss in the cache?

Assume distinct addresses go to distinct blocks.
Direct-mapped and fully-associative caches are types of set-associative caches.

1-way, direct mapped
8 "sets", 1 block each

2-way associativity
4 sets, 2 blocks each
Way 0      Set 0
           Set 1
           Set 2
           Set 3

4-way associativity, 2 sets, 4 blocks each
Set 0
    Way 0
    Way 1
    Way 2
    Way 3

8-way associativity (fully associative), 1 set, 8 blocks each
Way 0       Way 1       Way 2       Way 3       Way 4       Way 5       Way 6       Way 7
Set associativity is implemented by using a multiplexer to choose from two identical caches.
Summary

- Larger block sizes can take advantage of spatial locality by loading data from not just one address, but also nearby addresses, into the cache.

- Associative caches assign each memory address to a particular set within the cache, but not to any specific block within that set.
  - Set sizes range from 1 (direct-mapped) to $2^k$ (fully associative).
  - Larger sets and higher associativity lead to fewer cache conflicts and lower miss rates, but they also increase the hardware cost.
  - In practice, 2-way through 16-way set-associative caches strike a good balance between lower miss rates and higher costs.

- Next time, we’ll talk more about measuring cache performance, and also discuss the issue of writing data to a cache.