CS356: Discussion #8
Memory Hierarchy, Caches, and Cache Lab
Illustrations from CS:APP3e textbook
The Memory Hierarchy

So far...
- We modeled the memory system as an abstract array of bytes.
- The CPU could access any location in constant time.

In practice, the memory system is a hierarchy of storage devices with different capacities, costs, and access times.
- Small, fast cache memories close to the CPU: staging area for data/instructions read from main memory.
- Main memory can be used as staging area for large, slow local disks.
- Local disks are often used as staging area for data from network devices.

“Well-written programs tend to access storage at any level more frequently than storage at the next lower level.”
The Memory Hierarchy

- **Registers** (~1 ns)
- **L1 Cache** (~10 ns)
- **Main Memory** (~100 ns)
- **Secondary Storage** (~1-10 ms)

**Higher Levels**

- **L1 Cache**
- **Main Memory**

**Lower Levels**

- **Registers**
- **L1 Cache**
- **Main Memory**
- **Secondary Storage**

**Unit of Transfer:**
- **Word or Byte**
- **Cache block/line (1-8 words)**
- **Page (4KB-64KB words)** (Take advantage of spatial locality)
Latency Numbers (2018)

- 1ns
  - Send 2,000 bytes over commodity network: 88ns
  - Packet roundtrip CA to Netherlands: 150,000,000ns = 150ms

- L1 cache reference: 1ns
  - SSD random read: 16,000ns ≈ 16μs

- Branch mispredict: 3ns
  - Read 1,000,000 bytes sequentially from memory: 5,000ns ≈ 5μs

- L2 cache reference: 4ns

- Mutex lock/unlock: 17ns
  - Round trip in same datacenter: 500,000ns ≈ 500μs

- Main memory reference: 100ns
  - Read 1,000,000 bytes sequentially from SSD: 78,000ns ≈ 78μs

- 1,000ns ≈ 1μs
  - Disk seek: 3,000,000ns ≈ 3ms

- Compress 1KB w/ Zippy: 2,000ns ≈ 2μs
  - Read 1,000,000 bytes sequentially from disk: 1,000,000ns ≈ 1ms

http://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html
Storage Technologies

Static RAM
- Used for cache memories (inside/outside CPU)
- **Faster**, more **expensive**: 6 transistors/bit
- Resistant to noise, **persistent** (bistable cells)
- About 10 MB on a desktop computer

Dynamic RAM
- Used for main memory and frame buffer of GPUs
- Very sensitive to noise, even light (array of capacitors)
- Must be periodically refreshed (recharge capacitors)
- **1000× cheaper**, **10× slower**
- About 16 GB on a desktop computer
Locality

Temporal Locality
“A memory location referenced once is likely to be referenced again multiple times in the near future.”

Spatial Locality
“Nearby memory locations are likely to be referenced in the near future.”

Locality is exploited at all levels:
- **Hardware level**: CPU cache memories for main memory access.
- **OS level**: use main memory as cache for virtual memory or disk blocks.
- **Application level**:
  - Web browsers caching page elements.
  - Web servers caching frequently accessed pages/images in memory.
The function `sum(int *array, int size)` has **good locality**: 
- Variable `sum` is referenced once in every loop cycle: good temporal locality.
- The elements of `array` are read sequentially: good spatial locality.

**Stride-k reference pattern**: visiting every k-th element of an array.
- The smaller the stride, the better the spatial locality.
# Sum by row or by column?

Function `sum_by_row(int matrix[N][N])` has better space locality.

- Multidimensional arrays are stored in row-major format in C.
- `sum_by_row` results in stride-1 accesses
- `sum_by_col` results in stride-N accesses

Loops also have great temporal and spatial locality with respect to instruction fetches.
A cache is a small, fast staging area for objects from a larger, slower device.

**Cache Hit.** When an object is found in the cache of a level.

**Cache Miss.** When an object is fetched from the next level.

- In turn, another cache miss can be caused in the next level.
- Once data is fetched, an **eviction policy** decides whether to store the data in the cache and, if the cache is full, which block to evict (LRU, FIFO, ...).
- Data is transferred between levels in **blocks** containing many objects:
  - If another object in the block is required, it will be already in the cache.
Types of Cache Misses

**Compulsory Miss (or Cold Miss)**
When the cache is initially empty (or “cold”).

**Conflict Miss**
When there are restrictive placement policies inside the cache, and two referenced data map to the same cache block.

**Capacity Miss**
When the cache is not large enough for the “working set” of a program phase.

Who takes care of cache misses?
- Compiler manages the register file.
- CPU L1, L2, L3 caches are managed by hardware logic.
- For virtual memory, DRAM is managed by the OS and by the address translation hardware of the CPU.
**Cache Organization**

**Memory**: addresses of $m$ bits

$\Rightarrow M = 2^m$ memory locations

**Cache**:
- $S = 2^s$ cache sets
- Each set has $K$ lines
- Each line has: **data block** of $B = 2^b$ bytes, **valid bit**, $t = m - (s+b)$ **tag bits**

How to check if the word at an address is in the cache?
Direct-Mapped Caches ($K = 1$)

Selected set

Set 0:

Set 1:

Set $S-1$:

(1) The valid bit must be set.

(2) The tag bits in the cache line must match the tag bits in the address.

(3) If (1) and (2), then cache hit, and block offset selects starting byte.
Conflict Misses in Direct-Mapped Caches

Multiple memory blocks can map to the same set/line!
- They are identified by different tags.
- They generate conflict misses.
- Eviction policy is trivial: must remove the only line in the set.
More than one line per set.

- Line-matching is more difficult: must check the tag of multiple lines.
- Requires a policy for cache eviction (when all lines in set are full).
  - Random, FIFO, least-recently used (LRU), least-frequently used (LFU).
Fully Associative Caches ($K = C/B$)

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

- Line-matching is very difficult: must check the tags of all lines.
- Appropriate for small caches (e.g., TLBs in virtual memory buffers).

A single set contains all the cache lines.
Memory Writes and Caching

**Write hit** (writing to a word in the cache). Two options:
- **Write-through**: immediately update the word in the next cache level.
- **Write-back**: wait until the word is evicted from this cache level.
  - Can significantly reduce traffic on the bus.
  - Additional complexity (needs a “dirty bit”).
  - More common at lower levels (e.g., virtual memory).
  - Also used for Intel L1.

**Write miss** (the word is not in the cache). Two options:
- **Write-allocate**: load the word from next cache level, then write.
- **No-write-allocate**: bypass the cache and write directly into the next level.

Write-through caches are typically no-write-allocate.
Write-back caches are typically write-allocate.
Performance Tuning of Caches

- Large caches decrease the miss ratio, but increase the hit time.
- Large blocks decrease the miss ratio with spatial locality, but having fewer lines per set can hurt programs where temporal locality dominates.
- Large blocks can also increase the miss penalty.
- Large associativity $K$ decreases the chance of conflict misses, but it is more expensive to implement and hard to make fast.
  - More tag bits per line.
  - Additional LRU state bits per line.
  - Additional control logic.
  - ... can increase both hit time and miss penalty.

Average Access Time = (Hit Time) + (Miss Ratio) \times (Miss Penalty)
Why hit ratio is important

Average Access Time = (Hit Time) + (Miss Ratio) × (Miss Penalty)

Hit Time = 1ns, Miss Penalty = 100ns

- When Hit Ratio = 99%
  Average Access Time = 1ns + 1% × 100ns = 2ns

- When Hit Ratio = 97%
  Average Access Time = 1ns + 3% × 100ns = 4ns
Problem
A processor has a 32-bit memory address space. The memory is broken into blocks of 32 bytes each. The cache is capable of storing 16 kB.

- How many blocks can the cache store?
- Break the address into tag, set, byte offset for direct-mapping cache.
- Break the address into tag, set, byte offset for a 4-way set-associative cache.

Solution
- 16 kB / 32 bytes per block = 512 blocks.
- Direct-mapping: 18-bit tag (rest), 9-bit set address, 5-bit block offset.
- 4-way set-associative: each set has 4 lines, so there are 512 / 4 = 128 sets.
  - 20-bit tag (rest)
  - 7-bit set address
  - 5-bit block offset
Exercise: Cache Size and Address

Problem
A processor has a 36-bit memory address space. The memory is broken into blocks of 64 bytes each. The cache is capable of storing 1 MB.

● How many blocks can the cache store?
● Break the address into tag, set, byte offset for direct-mapping cache.
● Break the address into tag, set, byte offset for a 8-way set-associative cache.

Solution
● 1 MB / 64 bytes per block = 2**(20-6) = 16k blocks.
● Direct-mapping: 16-bit tag (rest), 14-bit set address, 6-bit block offset.
● 8-way set-associative: each set has 8 lines, so there are 16k / 8 = 2k sets
  ○ 19-bit tag (rest)
  ○ 11-bit set address
  ○ 6-bit block offset
Goal

- To write a small C simulator of caching strategies.
- Expect about 200-300 lines of code.
- Starting point in your repository.

Traces

- The traces directory contains program traces generated by valgrind.
- The format of each line is: <operation> <address>,<size>

For example: “I 0400d7d4,8” “M 0421c7f0,4” “L 04f6b868,8”

- Operations
  - Instruction load: I (ignore these)
  - Data load: L (hit, miss, miss/eviction)
  - Data store: S (hit, miss, miss/eviction)
  - Data modify: M (load+store: hit/hit, miss/hit, miss/eviction/hit)

https://usc-cs356.github.io/assignments/cachelab.html
Reference Cache Simulator

```bash
./csim-ref [-hv] -S <S> -K <K> -B <B> -p <P> -t <tracefile>
```

- **-h** Optional help flag that prints usage information
- **-v** Optional verbose flag that displays trace information
- **-S <S>** Number of sets (s=log2(S) is the number of bits used for the set index)
- **-K <K>** Number of lines per set (associativity)
- **-B <B>** Number of block size (i.e., use $B = 2^b$ bytes / block)
- **-p <P>** Selects a policy, either LRU or FIFO
- **-t <tracefile>** Select a trace

$ ./csim-ref -S 16 -K 1 -B 16 -p LRU -t traces/yi.trace
hits:4 misses:5 evictions:3

$ ./csim-ref -S 16 -K 1 -B 16 -p LRU -v -t traces/yi.trace

L 10,1 miss
M 20,1 miss hit
...
M 12,1 miss eviction hit
hits:4 misses:5 evictions:3

(See [https://usc-cs356.github.io/assignments/cachelab.html](https://usc-cs356.github.io/assignments/cachelab.html))
LRU and FIFO

$ ./csim-ref -S 2 -K 2 -B 2 -p LRU -v -t traces/simple_policy.trac
e L 0,1 miss
L 1,1 hit
L 2,1 miss
L 6,1 miss
L 2,1 hit
L a,1 miss eviction
L 2,1 hit
hits:3 misses:4 evictions:1

$ ./csim-ref -S 2 -K 2 -B 2 -p FIFO -v -t traces/simple_policy.trac
e L 0,1 miss
L 1,1 hit
L 2,1 miss
L 6,1 miss
L 2,1 hit
L a,1 miss eviction
L 2,1 miss eviction
hits:2 misses:5 evictions:2
Memory accesses that cross a line boundary

$ ./csim-ref -S 2 -K 1 -B 4 -p LRU -v -t traces/simple_size.trace
  L 0,2 miss
  L 1,2 hit
  L 3,1 hit
  L 6,6 miss miss eviction
  L 3,1 miss eviction
hits:2 misses:4 evictions:2
Your Simulator

Not needed in this lab

$B = 2^b$ bytes per cache block

$K$ lines per set

$S = 2^s$ sets

1 valid bit per line
t tag bits per line

?? bits per line to support eviction policy

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>Tag</td>
<td>Metadata</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\ldots$</td>
<td>$B-1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\ldots$</td>
<td>$B-1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$0$</td>
<td>$1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\ldots$</td>
<td>$B-1$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
</tbody>
</table>
struct Line {
    // include valid bit, tag, and metadata
};

Example 1

Flat array for $S \times K$ struct Line

\[
\begin{array}{ccccccccccccc}
\hline
& & & & & & & & & & & \\
\hline
\end{array}
\]

struct Line *cache;
cache = (struct Line *)malloc(S * K * sizeof(struct Line));
Your Simulator

```c
struct Line {
    // include valid bit, tag, and metadata
};

Example 2

struct Line **
Array for $K$ struct Line per set

Array for $S$
struct Line *

struct Line **cache;
cache = (struct Line **)malloc(...);
for i = 0, 1, ..., S-1 do
    cache[i] = (struct Line *)malloc(...);
```
Your Simulator

Fill in the `csim.c` file to:
- Accept the same command-line options.
- Produce identical output.

Rules
- Include name and username in the header.
- Use only C code (must compile with `gcc -std=c11`)
- Use `malloc` to allocate data structures for arbitrary `S, K, B`
- Implement both LRU and FIFO policies.
- Ignore instruction cache accesses (starting with `I`).
- Memory accesses can cross block boundaries:
  ⇒ How to deal with this?
- At the end of your main function, call:
  `printSummary(hit_count, miss_count, eviction_count)`
Evaluation

3 test suites:
- Direct Mapped: $K = 1$; no need to implement an eviction policy
- Policy Tests: check that LRU and FIFO policies work correctly
- Size Tests: include memory accesses that cross a line boundary

You only need to output the correct number of cache hits, misses, evictions.
- You can run `csim-ref -v` to check the expected behavior.
- Start from small traces such as `traces/dave.traces`
- Use the `getopt` library to parse command-line arguments.
  - `int s = atoi(arg_str); int S = pow(2, s);`

You must pass all tests in a test suite to receive its points.
Problems

- How to parse the input traces?
  - `fopen` (open a file), `fgets` (read a line), `sscanf` (parse a line), `fclose`

- How to represent the cache? How to allocate memory for any $S$, $K$, $B$?
  - Cache = $S$ sets
  - Each set = $K$ cache lines
  - Use `malloc` and `free`

- What needs to be stored in a cache line?
  - Valid bit, tag, and what else?
  - How to keep track of statistics for LRU and FIFO policies?

- How to retrieve data at a memory address?
  - How to extract tag / set / block bits from an input address?
  - How to select the correct set? And how to look for a hit?
  - What to update in case of hit (in addition to hit counter)?
  - What to do in case of miss?

- Useful: Print the content of the cache after each request in a trace