CSC2/452 Computer Organization
Caches and the Memory Hierarchy

Sreepathi Pai

URCS

October 23, 2019

Outline

Outline

Matrix Multiply – IJK

- Multiplying two matrices:
  - A $(m \times n)$
  - B $(n \times k)$
  - C $(m \times k)$ [result]
- Here: $m = n = k$

```
for(ii = 0; ii < m; ii++)
   for(jj = 0; jj < n; jj++)
      for(kk = 0; kk < k; kk++)
         C[ii * k + kk] += A[ii * n + jj] * B[jj * k + kk];
```

Matrix Multiply – IKJ

```
for(ii = 0; ii < m; ii++)
   for(kk = 0; kk < k; kk++)
      for(jj = 0; jj < n; jj++)
         C[ii * k + kk] += A[ii * n + jj] * B[jj * k + kk];
```

Performance of the two versions

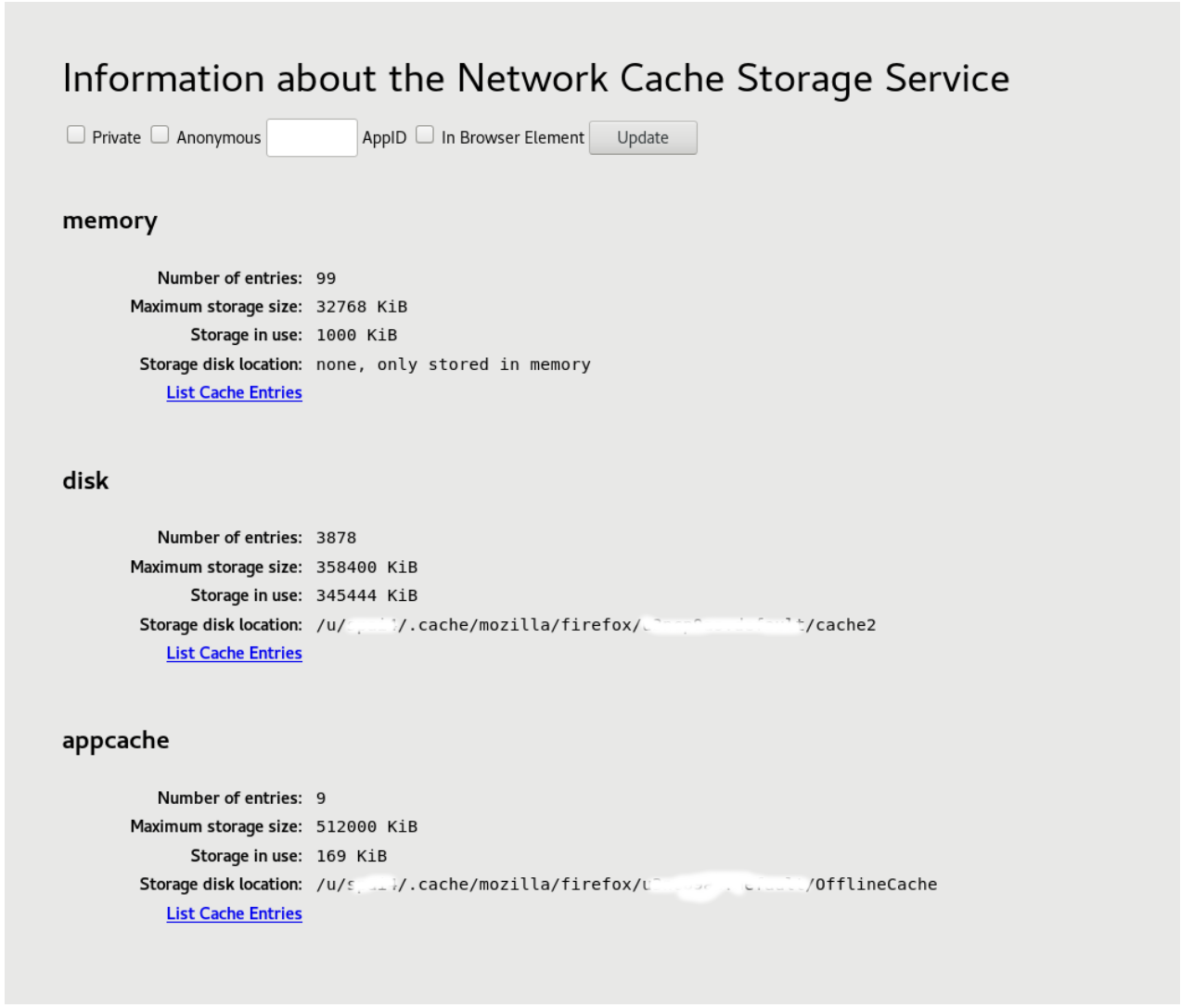
- on 1024x1024 matrices
- Time for IJK: 0.554 s $\pm$ 0.003s (95% CI)
- Time for IKJ: 6.618 s $\pm$ 0.032s (95% CI)

What caused the nearly 12X slowdown?

- Matrix Multiply has a large number of arithmetic operations
  - But the number of operations did not change
- Matrix Multiply also refers to a large number of array elements
  - Order in which they access elements changed
  - But why should this matter?

Outline

Browser Cache



Search Engine Cache

Buffer/Page/Disk Cache

```
$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 19497520 396644 10880272    0    0     0     0    0    0  0  0 99  0  0
```

Cache

(Roughly) a cache is a storage location that is faster to access than the original location.

- Cache type: Cache location, Original location
  - Browser: Disk, Website
  - Google: Google, Website
  - OS Disk Cache: RAM, Disk

Hardware Cache

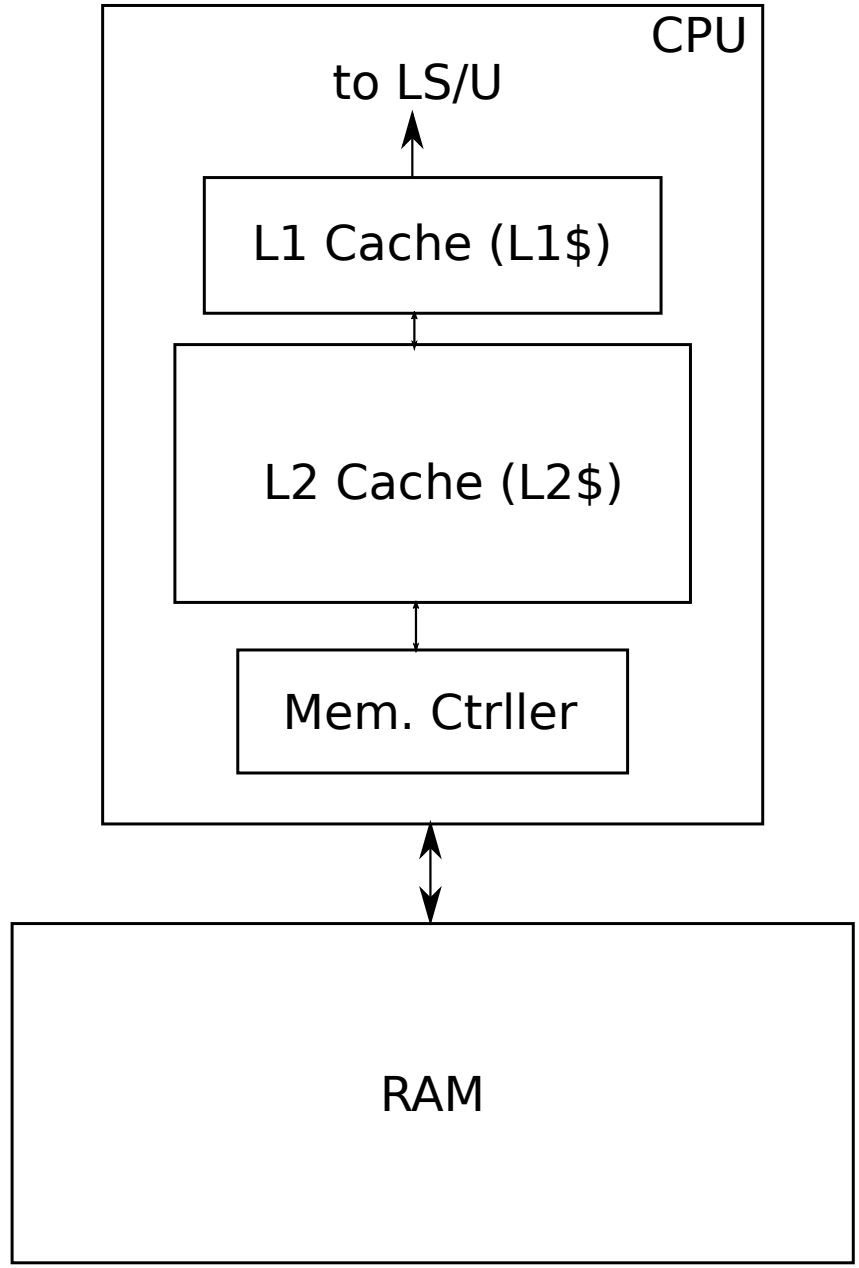
A hardware cache (specifically a CPU cache) is a small amount of fast (usually SRAM) memory in the CPU.
Q: Why not use this fast memory for building all of memory?

SRAM vs DRAM

- SRAM is a “flip-flop”, a type of circuit for memory
  - Uses at least 4 transistors
  - Compare to one transistor + one capacitor for DRAM
- SRAM has lower density
- SRAM is very power hungry

Using Hardware Caches

- The LSU asks the L1 cache for data at a specific address
  - if present, the L1 cache returns the data
- Otherwise, the L1 cache asks the L2 cache for the data
- And so on, until RAM is queried for the data
  - Actually on modern systems, disk is the last level (later lectures)
- Caches are transparent to the programmer
  - You don't have to do anything



Cache Performance Benefits

- Cache hit: Cache contains the data you're looking for
- Cache miss: Cache must query the next level of memory

$$T_{memavg} = H_{L1} \times T_{L1} + (1 - H_{L1}) \times (H_{L2} \times T_{L2} + (1 - H_{L2}) T_{RAM})$$

- Assume you have an average L1 hit rate of 100%
  - T_{L1} is 1 cycles[1]
  - T_{L2} is around 10 cycles
  - T_{RAM} is 100 cycles
- What is average time for a memory access?

[1] Latency Numbers Every Programmer Should Know

Locality

- Principle of Spatial Locality
  - Data that is near in space (in RAM) is accessed closer in time
- Principle of Temporal Locality
  - Data that is accessed now is likely to be accessed again in the near future

Code that shows locality

```
clock_gettime(CLOCK_MONOTONIC_RAW, &start);

for(i = 0; i < N; i++) {
  if(a[i] > max) max = a[i];
}

clock_gettime(CLOCK_MONOTONIC_RAW, &end);
```

About 13ms on my laptop, with `perf stat -e cache-misses,cache-references` showing:

```
Performance counter stats for './locality' (10 runs):

          622,559      cache-misses              #   87.861 % of all c
          708,570      cache-references
```

Code that may not show locality

```
clock_gettime(CLOCK_MONOTONIC_RAW, &start);

for(i = 0; i < N; i++) {
  if(a[b[i]] > max) max = a[i];
}

clock_gettime(CLOCK_MONOTONIC_RAW, &end);
```

- In the above code, `b[i] = i` (i.e. identity)
- About 14ms on my laptop, with `perf stat -e cache-misses,cache-references` showing:

```
Performance counter stats for './nolocality1' (10 runs):

         1,222,210      cache-misses              #   86.575 % of all c
         1,411,733      cache-references
```

Code that does not show locality

```
clock_gettime(CLOCK_MONOTONIC_RAW, &start);

// b[i] is now a random permutation of numbers from 0 to N-1
for(i = 0; i < N; i++) {
  if(a[b[i]] > max) max = a[i];
}

clock_gettime(CLOCK_MONOTONIC_RAW, &end);
```

- In the above code, `b` is a random permutation of the numbers 0 to N-1
- About 65ms on my laptop, with `perf stat -e cache-misses,cache-references` showing:

```
Performance counter stats for './nolocality2' (10 runs):

        10,857,423      cache-misses              #   70.655 % of all c
        15,366,835      cache-references
```

Locality in code

- Keep data you want to access together near each other (spatial locality)
  - e.g., use arrays
  - use a custom memory allocator for pointer-based structures if possible
- Reuse data as much as possible (temporal locality)
  - techniques called “blocking”

CPU Cache Design Problems

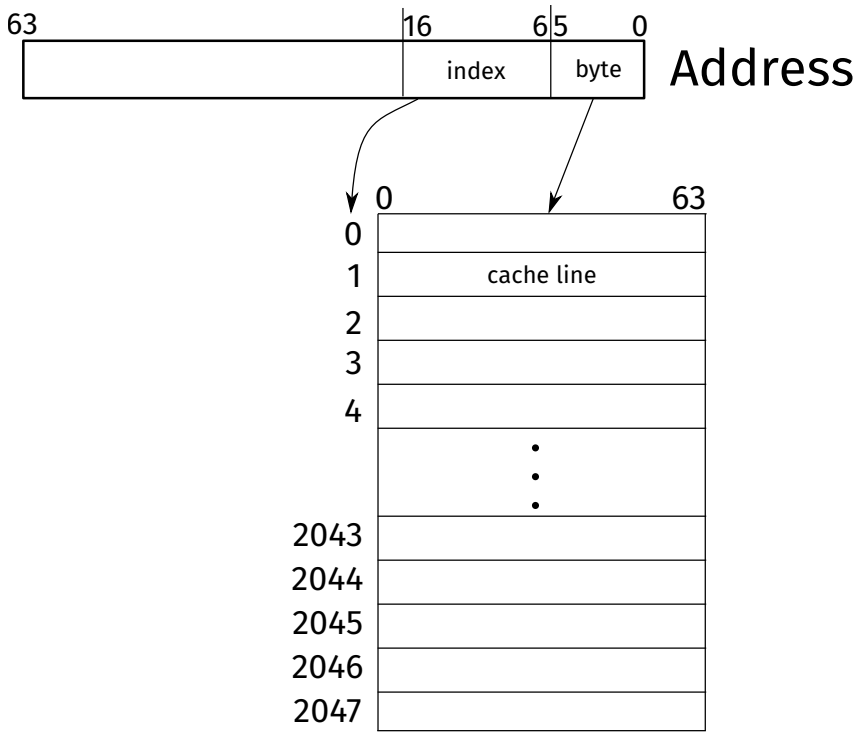
- Main memory is usually a few gigabytes
- The biggest caches are a few megabytes and fixed in size
  - How do you "fit" all data you want to access in the cache?
  - How do you address the cache?
  - And if you can't fit all the data you want, how do you decide what to keep and what to throw out?

Outline

Direct Addressing

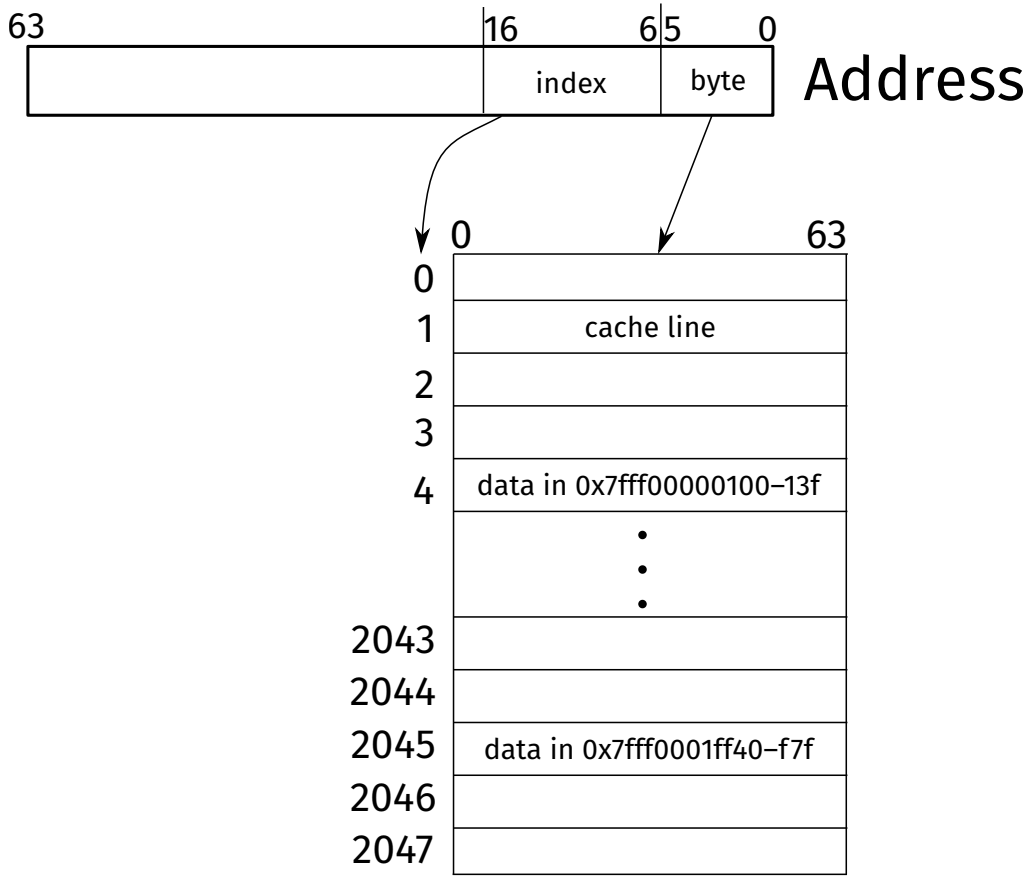
- Consider a 128KiB cache, with 64-byte cache lines
  - There are 2048 lines in total
  - Memory is divided into 64-byte chunks called “lines”
  - These lines do not overlap, and provide spatial locality
- How many bits to address a line?
- How many bits required to address a byte inside a cache line?

A Direct-mapped Cache: Try #1



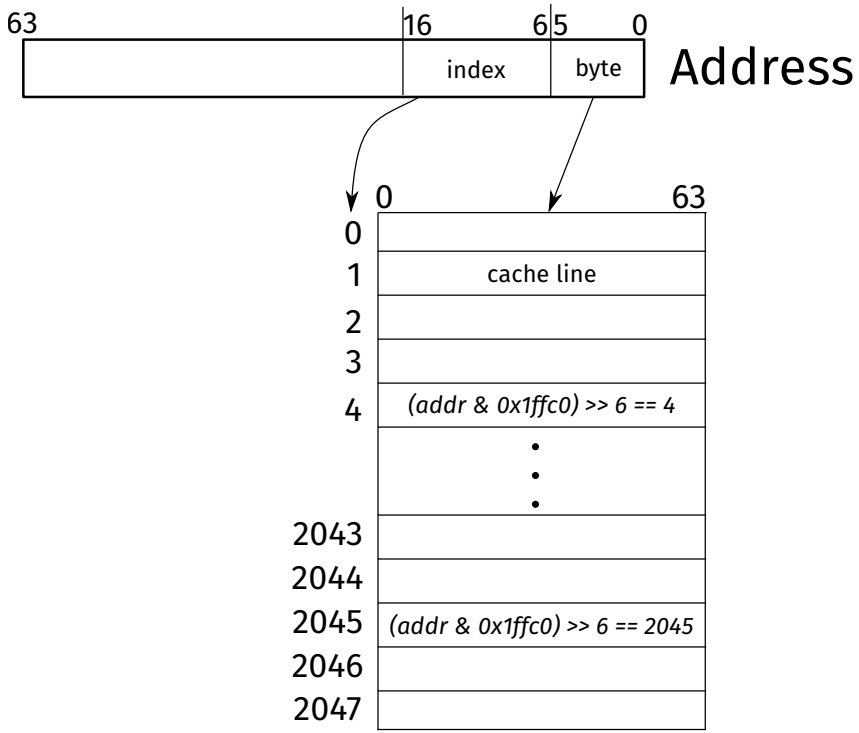
- We use bits 6 to 16 (both inclusive) to index into the cache
- We use bits 0 to 5 (both inclusive) to access individual bytes
- We ignore other bits

A Direct-mapped Cache: Addressing Example



- Each cache line contains 64 bytes of data from memory
- The data is placed in cache line indicated by the index field of the address

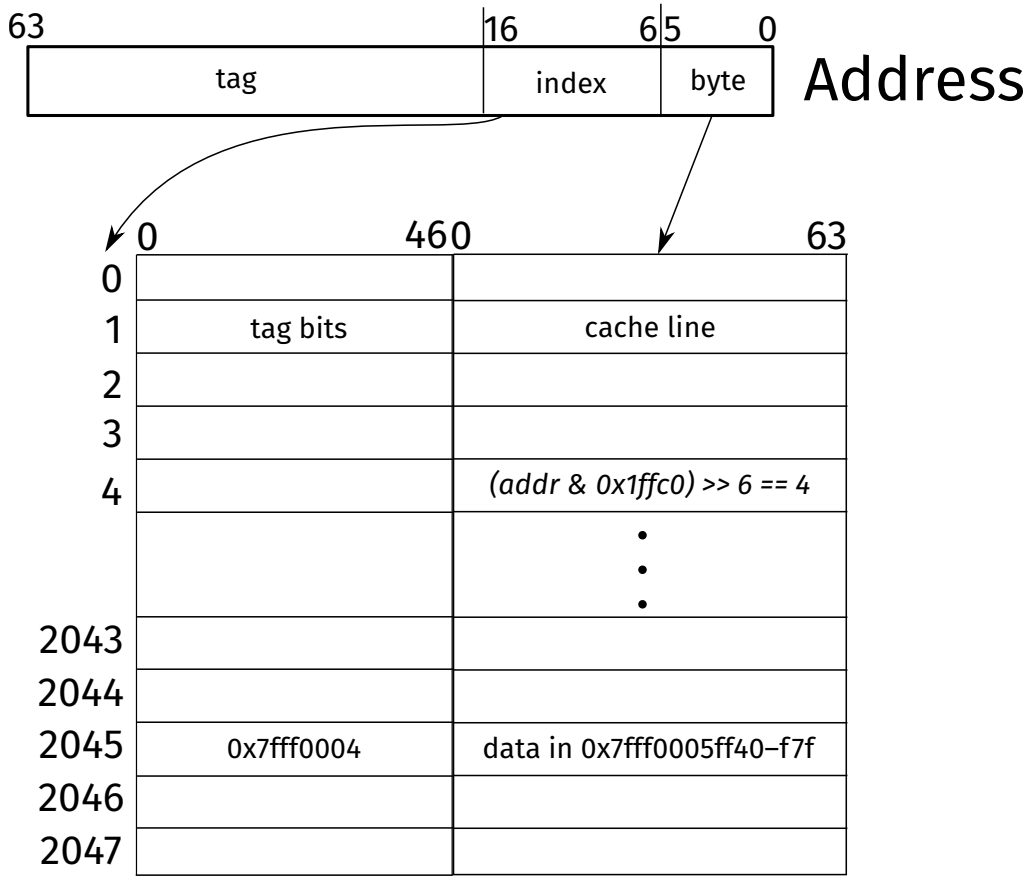
Conflicts in Direct-mapped Caches



There are multiple cache lines that can map to the same cache line!

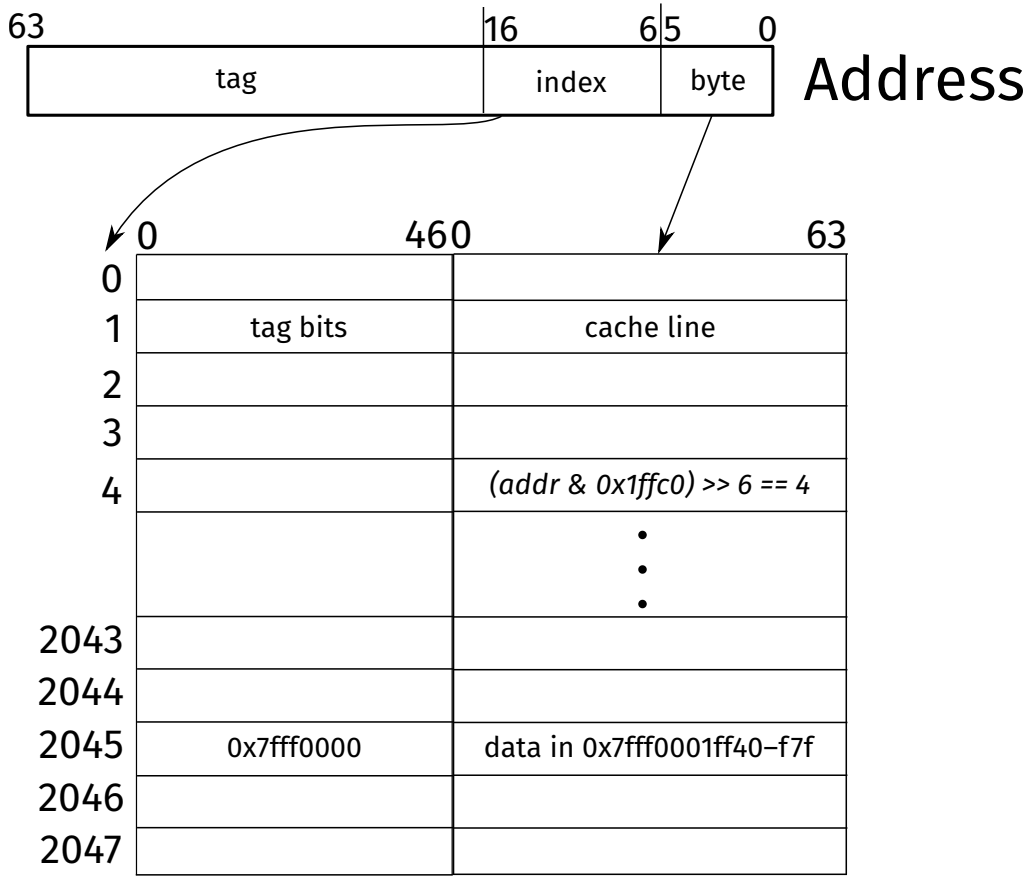
- E.g., `0x7fff0007ff40` and `0x7fff0005ff40` both map to line 2045
- How do we distinguish between different addresses that map to the same line?

Adding tag bits to disambiguate lines



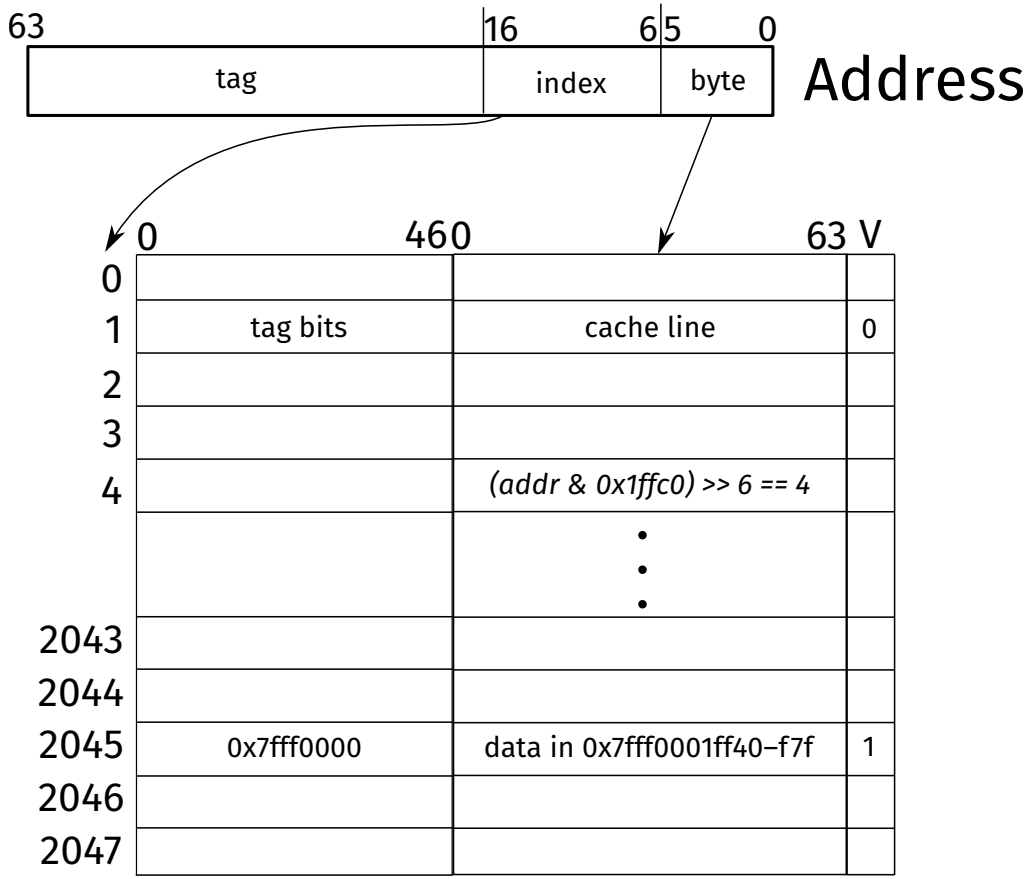
- Use bits from address not used so far as a *tag*
- Store tag with data
  - Always compare tags before reading/writing data
- In above figure: the tag is indexed from 0 to 46 *bits*

Different address → different tags



Although `0x7fff0001ff40` will occupy the same line as `0x7fff0005ff40`, they will have different tags.

Valid bits



- the valid bit is 1 if the cache line contain valid data
- initialized to zero when processor starts
- set to 1 whenever the cache line contains valid data

Direct-mapped Algorithm

```
in_cache(address) {
   index = address[6:16]
   tag = address[17:63]

   if(cache_lines[index].valid)
     if(cache_lines[index].tag == tag)
       return HIT;

   return MISS;
}
```

Cache operation

Suppose the program is accessing memory in the following order, and assume the cache starts out empty:

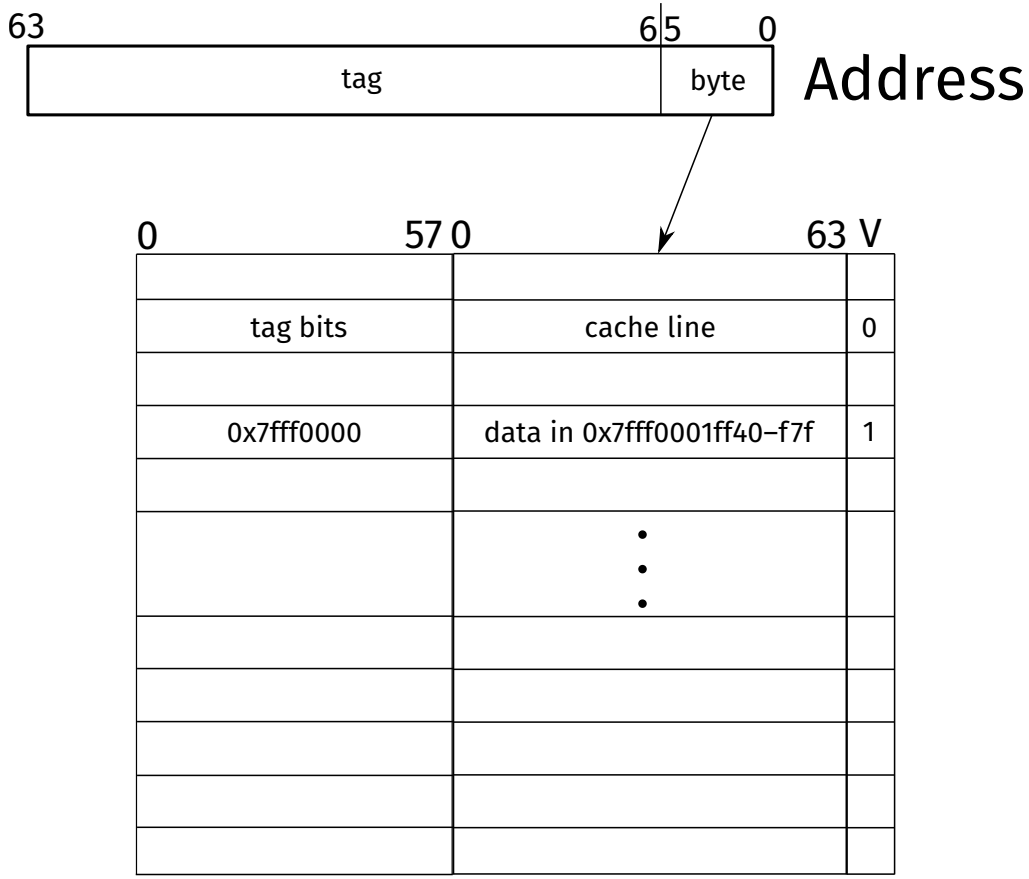
```
1: 0x7fff00000100    index=   4, MISS
2: 0x7fff0007ff40    index=2045, MISS
3: 0x7fff00000120    index=   4, HIT
4: 0x7fff0001ff40    index=2045, MISS
5: 0x7fff0007ff40    index=2045, MISS
```

- Access 3 is a cache hits, because Access 1 brought the data in
- Access 4 misses because the tag does not match, and the data brought in replaces 0x7fff0007ff40 (same index)
- Access 5 misses because it was replaced, and it replaces 0x7fff0001ff40 in turn (same index)
- Note all the misses happen despite the entire cache being empty, except for two lines!

Direct Mapped Caches

Direct-mapped caches are simple to build, and fast, but they may not utilize all cache lines for some patterns.

Associative Addressing



- Associative caches get rid of index bits
- They use all bits not used to address bytes in the cache line as tag bits
- In above figure: the tag is indexed from 0 to 57 *bits*

Associative Cache Algorithm

```
in_cache(address) {
   tag = address[6:63]

   foreach cache_line in cache
      if(cache_line.valid)
         if(cache_line.tag == tag)
            return HIT;

   return MISS;
}
```

- Note: in hardware, you can do the checks *in parallel*
  - i.e. it is not a serial loop!

Cache operation

Suppose the program is accessing memory in the following order, and assume the cache starts out empty:

```
1: 0x7fff00000100  tag=A  MISS
2: 0x7fff0007ff40  tag=B  MISS
3: 0x7fff00000120  tag=A  HIT
4: 0x7fff0001ff40  tag=C  MISS
5: 0x7fff0007ff40  tag=B  HIT
```

- Access 3 hits in the cache, because Access 1 brought the data in
- Access 4 misses
- Access 5 does not miss because it has a different tag than 0x7fff0001ff40
- There are now 3 lines in the cache

What if the cache can only hold two lines?

The same memory trace, but on an associative cache that can hold only two lines:

```
1: 0x7fff00000100    tag=A  MISS
2: 0x7fff0007ff40    tag=B  MISS
3: 0x7fff00000120    tag=A  HIT
4: 0x7fff0001ff40    tag=C  MISS --> what to do now?
5: 0x7fff0007ff40    tag=B
```

- Assuming we must replace an existing cache line
  - We can replace `0x7fff00000100`
  - Or we can replace `0x7fff0007ff40`
- Which one should we replace, if we wanted to maximize hit rate?

The OPT algorithm

```
1: 0x7fff00000100    tag=A  MISS
2: 0x7fff0007ff40    tag=B  MISS
3: 0x7fff00000120    tag=A  HIT
4: 0x7fff0001ff40    tag=C  MISS --> replace 0x7fff00000100
5: 0x7fff0007ff40    tag=B  HIT
```

- Replace the line that going to be used farthest in the future
  - `0x7fff00000100`, in our case (since we don't see it in our trace, assume next use is at ∞)
- Thus, using OPT will cause Access 5 to hit in the cache
- OPT is guaranteed optimal – it will produce the lowest miss rate
- What is the problem implementing OPT?

The Least-Recently Used (LRU) Algorithm

```
1: 0x7fff00000100    tag=A  MISS
2: 0x7fff0007ff40    tag=B  MISS
3: 0x7fff00000120    tag=A  HIT
4: 0x7fff0001ff40    tag=C  MISS --> replace 0x7fff0007ff40
5: 0x7fff0007ff40    tag=B  MISS --> replace 0x7fff00000100
```

- The LRU algorithm replaces the line whose last use was farthest in the past
  - "if it has not been used recently, maybe it will not be used again soon"
- Not necessarily as good as OPT (as this example shows)
  - But often better than other schemes (e.g., FIFO–first in, first out, LIFO–last in, first out)

Misses

- Compulsory Misses
  - The miss that occurs when the data is first brought in
  - Can't avoid this miss (?)
- Conflict Misses
  - A miss that occurs in a direct-mapped cache, but would not occur in a similarly-sized associative cache
- Capacity Miss
  - A miss that occurs in an associative cache

Latency issues in Associative caches

- Associative caches can be slower than direct-mapped caches
  - Especially for very large sizes
- They also consume a lot of power
  - Parallel search

Set-Associative Addressing

- Combines both direct-mapped and associative caches
- First, locate the set using index
- Then, search the set for tag
- In above figure: the tag is indexed from 0 to 47 *bits*

Set-Associative Algorithm

```
in_cache(address) {
   index = address[6:15]
   tag = address[16:63]

   foreach cache_line in cache_sets[index]
     if(cache_line.valid)
       if(cache_lines.tag == tag)
          return HIT;

   return MISS;
}
```

Set-associative Cache operation

```
1: 0x7fff00000100    index=   4, MISS, placed in set 1
2: 0x7fff0007ff40    index=2045, MISS, placed in set 1
3: 0x7fff00000120    index=   4, HIT
4: 0x7fff0001ff40    index=2045, MISS, placed in set 2
5: 0x7fff0007ff40    index=2045, HIT
```

- Most hardware caches use 4 cache lines per set
- Some go up to 8
  - Diminishing returns after that point

Real Hardware Caches don't use LRU

- Implementing real LRU would require tracking time each cache line was accessed
- Most hardware implements pseudo-LRU
  - Approximates LRU, different for each manufacturer
  - Only thing we can be sure of is that it is not LRU
- Lots of sophisticated replacement policies dreamt up over the years
  - Still an open area of research

Getting rid of compulsory misses

- Can get rid of compulsory misses if we can fetch data into cache just before it is required
- This is the task of the *prefetch unit*
  - Snoops on all memory accesses, trying to identify patterns
  - Once a pattern is detected, starts fetching cache lines ahead of time to reduce misses
- Again, an open area of research
  - How to prefetch useful lines before they will be used? [timeliness]
  - How to not displace existing useful lines? [cache pollution]

References

- Chapter 6: The Memory Hierarchy