Lecture 19: Cache Performance

Spring 2024
Jason Tang

Topics

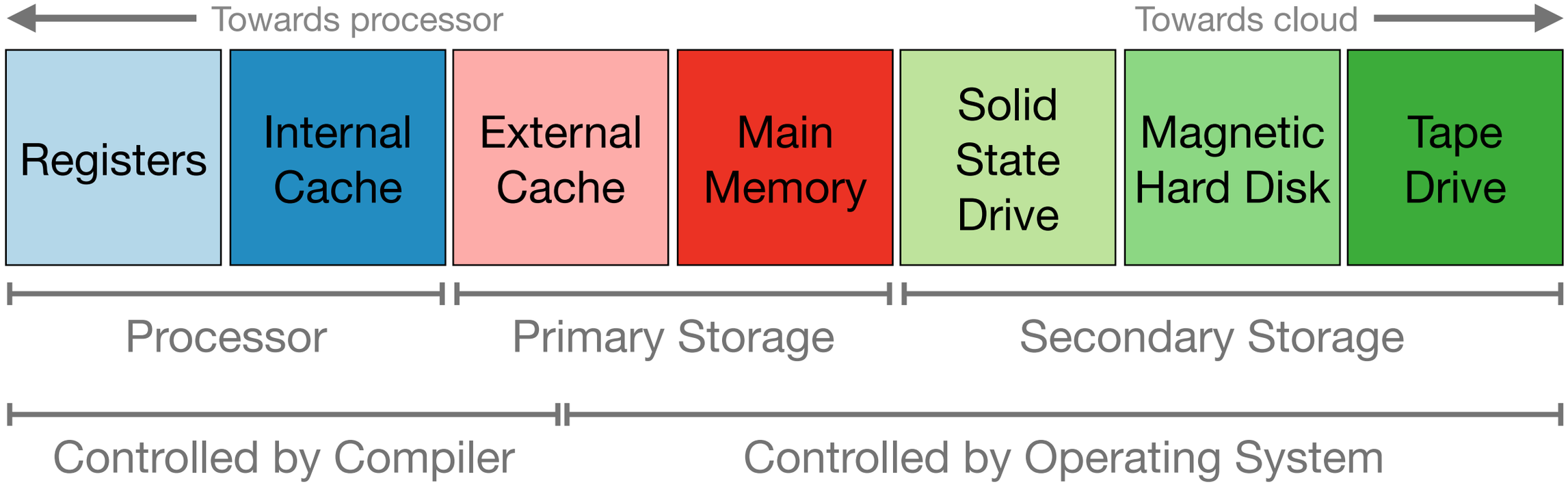
- Measuring cache performance
- Increasing hit rates
- Write policies

Importance of Caching

- Clever software writers can significantly decrease execution times by understanding how caches operate

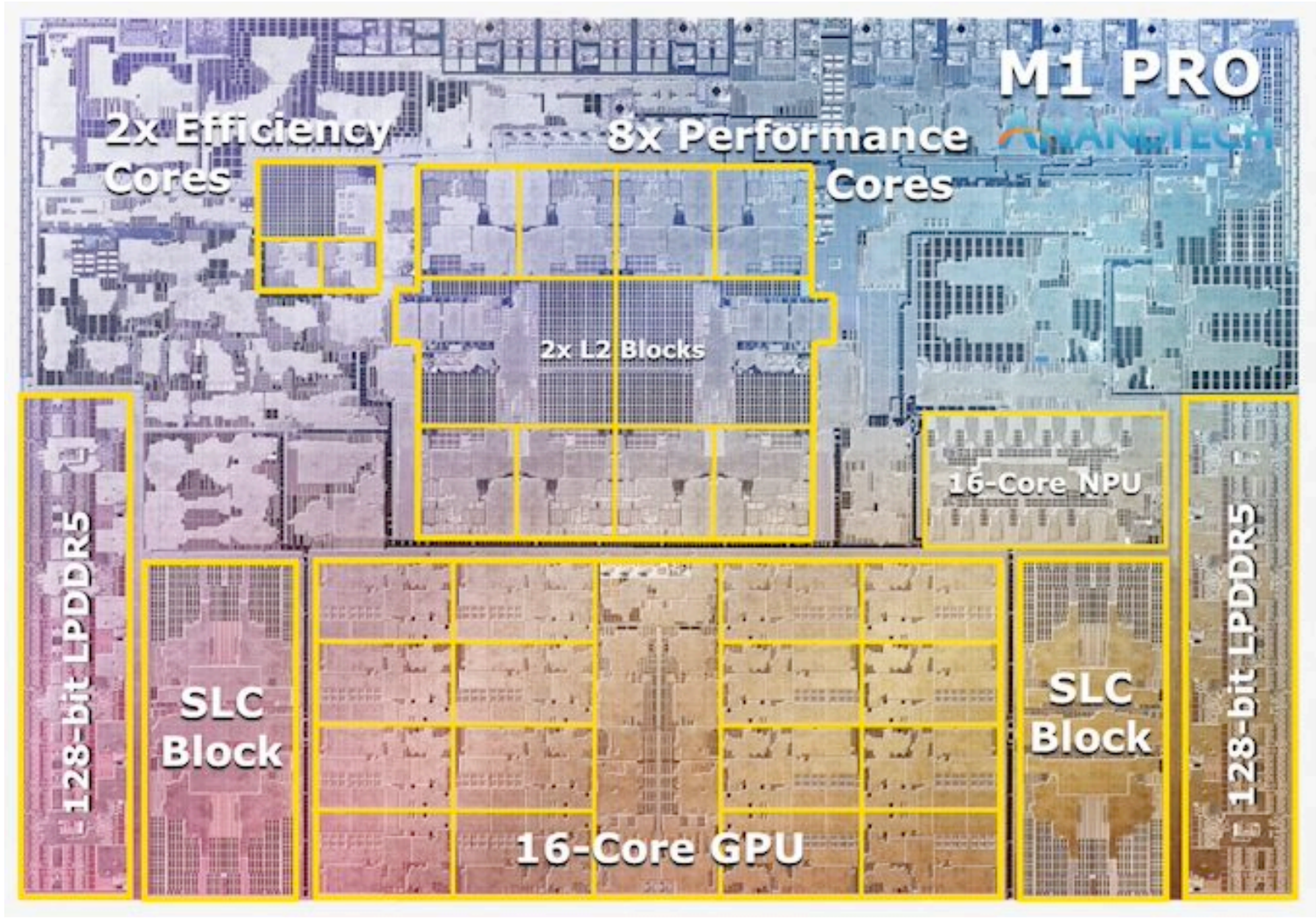
Faster
Smaller
More Expensive

Slower
Larger
Less Expensive



Importance of Caching

- System execution time includes not just CPU time, but also time spent on memory accesses
  - Time spent waiting for memory (a stall) is significant, measured in tens to hundreds of nanoseconds
- Modern processor designs have focused on decreasing stall penalty via improved cache designs



https://www.anandtech.com/show/17019/apple-announced-m1-pro-m1-max-giant-new-socs-with-allout-performance

Measuring Cache Performance

- CPUTime = (InstructionCycles + MemoryStallCycles) × ClockCycleTime
- MemoryStallCycles = ReadStallCycles + WriteStallCycles
- ReadStallCycles = %ReadInstructions × CacheReadMissRate × CacheReadMissPenalty
  - CacheReadMissRate = 1 - CacheReadHitRate
  - For most software, reads occur much more frequently than writes
- Calculating WriteStallCycles is harder
- Let average memory access time (**AMAT**) = Hit time + Miss rate × Miss penalty

Cache Performance Example

- Suppose that a particular system's cache takes 1 cycle to access, and the hit rate is 95%. Upon a cache miss, the penalty is 100 cycles to access main memory. What is the AMAT?

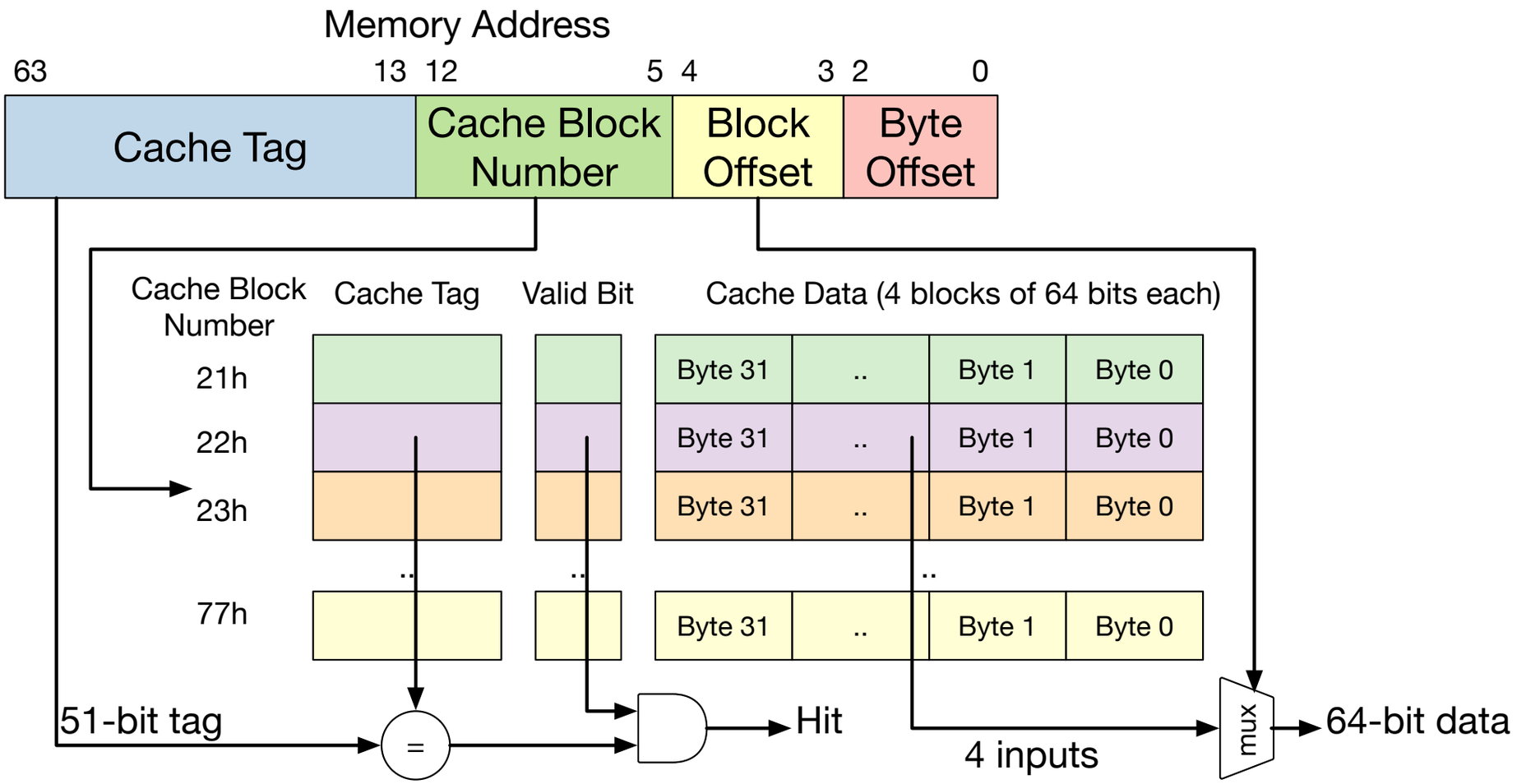
$$\begin{aligned} AMAT &= Hit\ Time + (Miss\ Rate \times Miss\ Penality) \\ &= 1 + (0.05 \times 100) \\ &= 6\ clock\ cycles \end{aligned}$$

- Suppose the hit rate increases to 98%. What is the new AMAT?
  - = 1 + (0.02 × 100) = 3 clock cycles
  - Thus a 3% increase in hit rate **halved** average memory access times

Decreasing AMAT

- Increase hit rate (or reduce miss rate)
  - Make memory reads more likely to be serviced by cache
  - Decrease unused items from being stored in cache
- Decrease miss penalty
  - Decrease time to find an available cache block
  - Decrease time to flush a used cached block

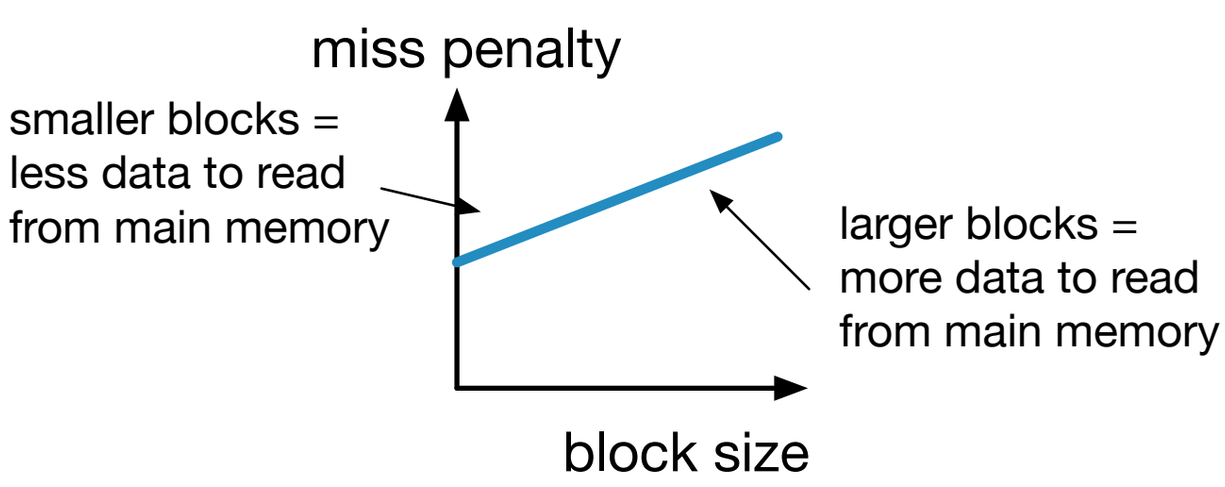
Multi-Word Direct Mapped Cache

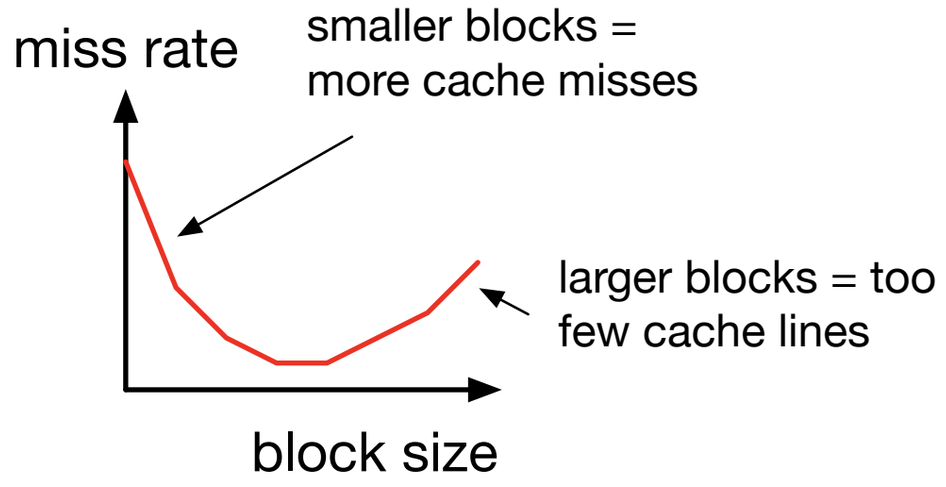


- Diminishing return when increasing internal cache, increasing manufacturing cost and increasing cache access time

- Given finite bits dedicated to cache, could increase the cache block size to increase hit rate, thus exploiting spatial locality

Multi-Word Direct Mapped Cache





- BlockAddress = ByteAddress / BytesPerBlock
- CacheBlockAddress = BlockAddress modulo NumberOfCacheBlocks
- Larger block increases read miss penalty, as that more memory needs to be transferred to fill up block
- Larger block also increases read miss rate, because too few cache blocks

Fully Associative Cache

- A given memory address is comprised of a block number and an offset within the block

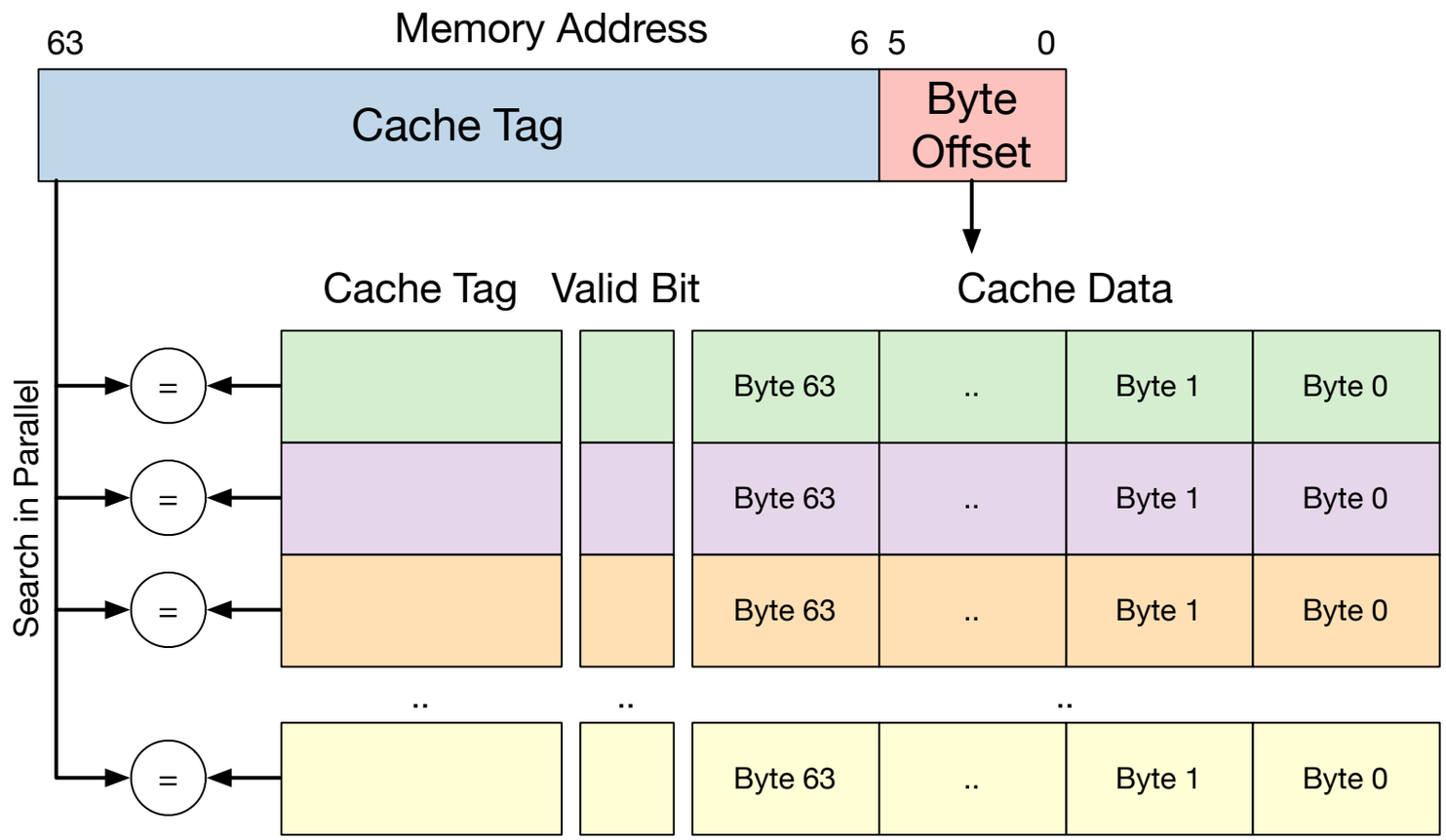
  - Example: a 64-bit address with a block size of 64 bytes has 58 bits for the tag and 6 bits for offset, and there are 2^{58} total blocks

Tag	Byte within a Block
58	6

- In a directly mapped cache each block corresponds to exactly one possible location within the cache, leading to higher conflict

- In a fully associative cache, any block can be stored in any location within the cache

Fully Associative Cache



- More flexible than direct mapped cache, stores blocks where it needs to be, higher cache hit rate

- Searching for a block takes longer (though done in parallel) and requires more hardware

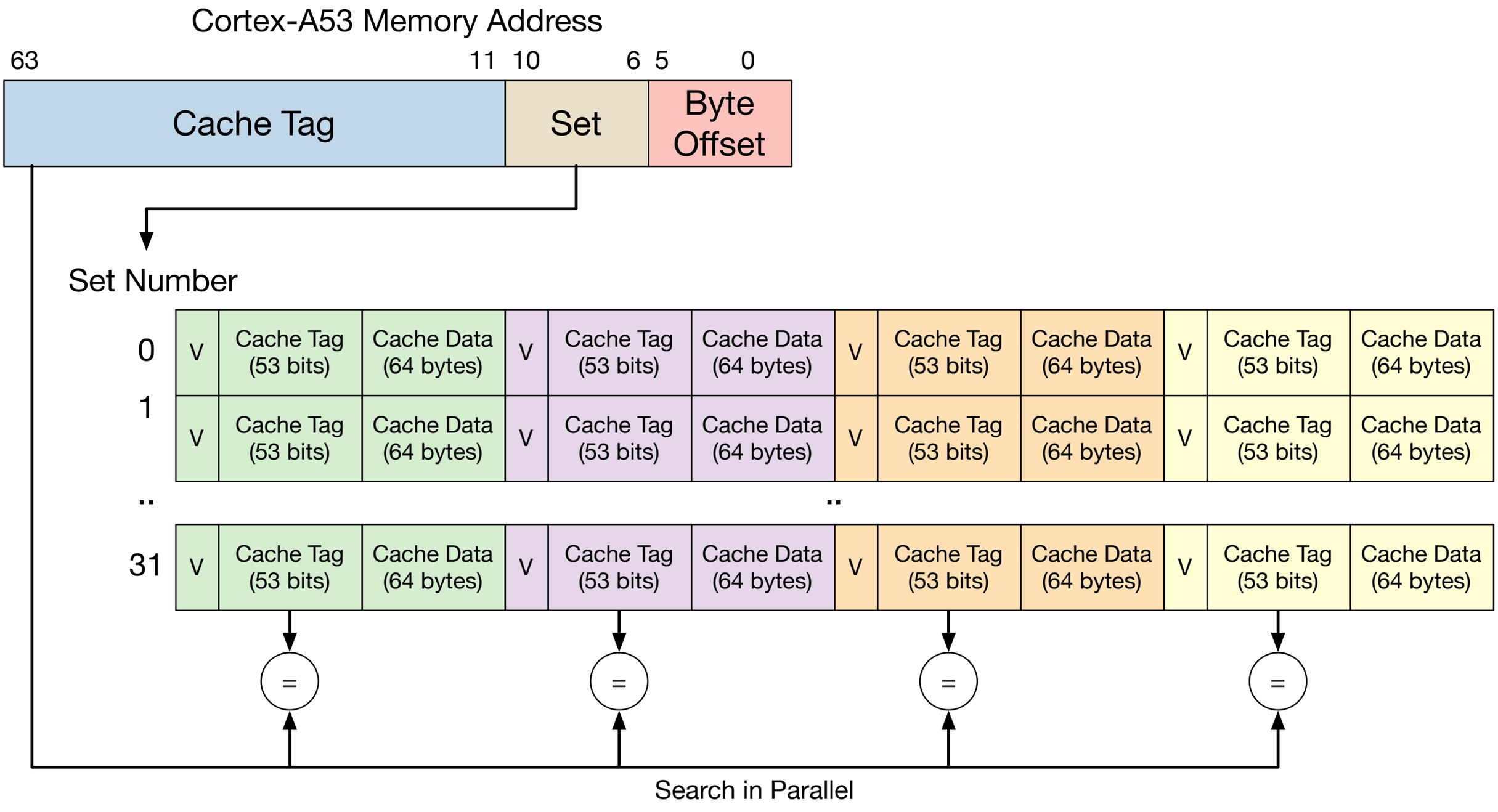
Set Associative Cache

- Compromise between direct mapped and fully associative cache
  - Map a block number to a set (direct mapped)
  - Then search within that set for an available block (associative mapped)

- Example: ARM Cortex-A53's data cache has a cache line of 64 bytes and is 4-way set associative. Supposing its total cache size is 8 KiB, then:
  - Total number of blocks = Cache Size / Cache Line Size = 8 KiB / 64 = 128
  - Total number of sets = Total Blocks / Set Associativity = 128 / 4 = 32

ARM Cortex-A53 Processor Technical Manual, §6.1

Cortex-A53 with 8 KiB Data Cache



Cache Alignment

- If a variable (`int`, `struct`, etc) spans across two cache lines, then there are two potential cache misses when using that variable
    - Be aware of wasted memory when ordering structures
        - C compiler will by default pad structure elements to be word aligned
- When working with large structures within a loop, reorganize data by splitting the large structure into smaller structures stored in separate arrays
    - Looping over a Row Major Ordered array optimally uses cache
- In C, dynamically allocate a piece of aligned memory via `memalign()` function

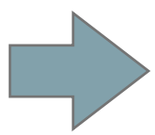
Alignment Example

```
#include <stddef.h>
#include <stdio.h>

struct s1 {
    char c1;  int i;   long l;  char c2;
};

struct s2 {
    long l;   int i;   char c1; char c2;
};

int main(void) {
    printf("Sizes: s1 = %zu, s2 = %zu\n",
       sizeof(struct s1), sizeof(struct s2));
    printf("Alignment of fields:\n");
    printf("  c1: %zu  %zu\n",
       offsetof(struct s1, c1), offsetof(struct s2, c1));
    printf("  c2: %zu  %zu\n",
       offsetof(struct s1, c2), offsetof(struct s2, c2));
    printf("  i: %zu  %zu\n",
       offsetof(struct s1, i), offsetof(struct s2, i));
    printf("  l: %zu  %zu\n",
       offsetof(struct s1, l), offsetof(struct s2, l));
    return 0;
}
```



```
Sizes: s1 = 24, s2 = 16
Alignment of fields:
  c1: 0  12
  c2: 16  13
  i: 4  8
  l: 8  0
```

Prefetching

- Another technique to increase hit rate is to prefetch memory

- Whenever cache is fetching a block from memory, initiate reads in anticipation of using the next block
  - Exploits spatial locality

- Example: PowerPC has a 32 byte cache line. When fetching instructions into cache, by default it also prefetches the next 96 bytes (3 cache lines), as that it assumes that program flow is linear.

PPC440 Processor User's Manual, §3.2.2

Write Policies

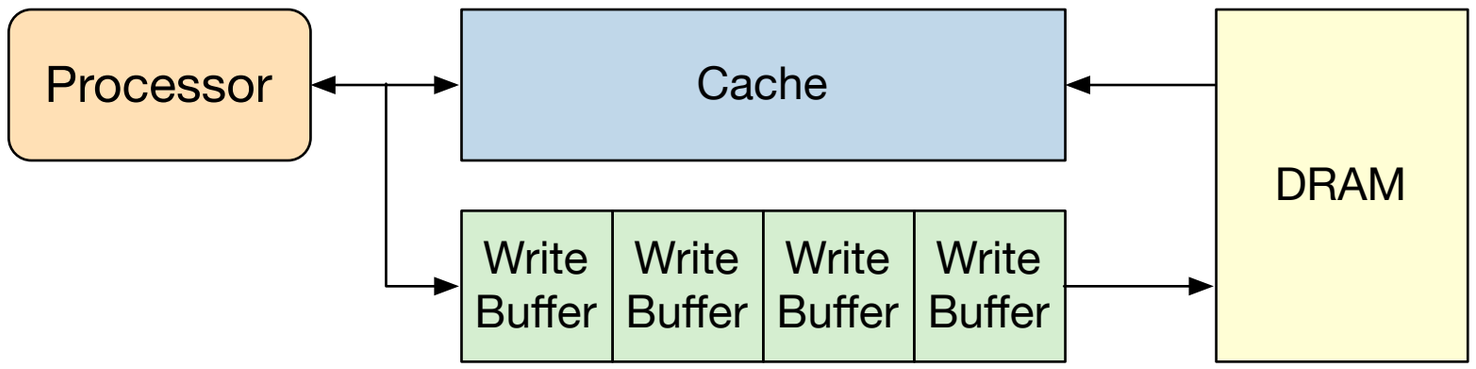
- In many software, a variable that is written will be read again soon (and thus the new value should be stored in cache)

- When writing to cache, should main memory also be updated?

- For some software, values are written into memory, but will not be read back for a long time

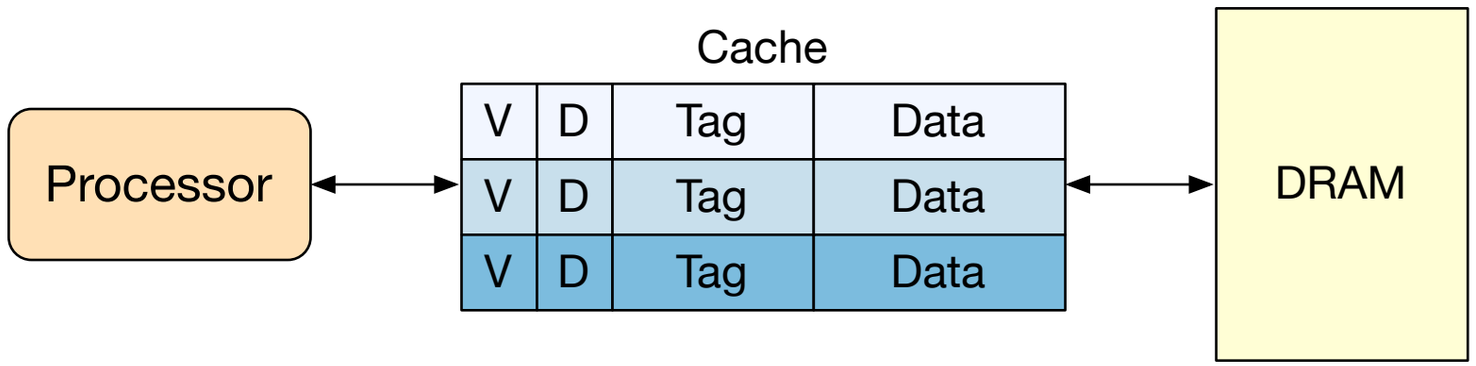
  - Storing it in cache prevents more useful things from being cached

Write Through Policy



- Writes data to both cache and main memory
  - Processor writes data to cache and a write buffer
  - Memory controller commits write buffer to memory, asynchronously
- Good if a block is rarely written, bad if block is constantly updated
  - Write buffer will fill, causing CPU to stall

Write Back Policy



- Writes data only to cache
- When cache needs to free up a block and if the dirty bit is set, then flush the cache line to memory (if not already invalidated)
- More complex to implement, but reduces time spent writing to DRAM
  - Reduces redundant writes to memory for repeated changes
  - Can lead to cache coherency

Write Combining

- Like a write-through system, but instead of writing to a write buffer instead store changed bits in a write combine buffer
    - Wait for write combine buffer to be filled before writing (a burst write)
        - Better than immediately writing many small chunks to memory
- Significantly decreases write times when processor is producing a lot of streaming data (like for a video card)
    - Caution: Reads from that cache line may return original cached data or from write combine buffer

Uncached Writes

- In uncached memory, read and write accesses bypass cache entirely
  - Write combining is a special type of uncached memory access
- Uncached accesses are **necessary** when accessing hardware
  - A read should force retrieving latest value from peripheral
  - A write should immediately effect hardware, instead of waiting upon a write buffer to be committed
- Using cache both wastes cache blocks and also leads to incorrect behavior

https://lwn.net/Articles/282250/